UNIT - I

LESSON 1

DATABASE CONCEPTS

Contents

1.0 AIMS AND OBJECTIVES

To learn about basic database terminology.

• Relational database concepts are covered.

• The Database Management System (DBMS) and its functions are outlined.

1.1 A Relational Approach: Database:

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, birth date, 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 teacher.

1.2 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 l: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 V 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.

Many times, students find it difficult to determine the type of a relationship. You need to ask the following two questions to make the determination:

1. Does an entity in entity set X have more than one matching entity in entity set Y?
2. Does an entity in entity set V have more than one matching entity in entity set X?

If your answers to both questions are “No,” the relationship is a 1:1 relationship. If one of the answers is “Yes” and the other answer is “No,” it is a

1:M relationship. If both answers are "Yes," you have an M:N relationship. Later, you will see that the M:N relationship is not easy to implement and is decomposed into two 1 :M relationships.

Check your Progress 1:

What are the different relationships of data?

……
……
……
……

1.3 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 includes 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 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:

An example Database System is shown in the figure 1.1 below.

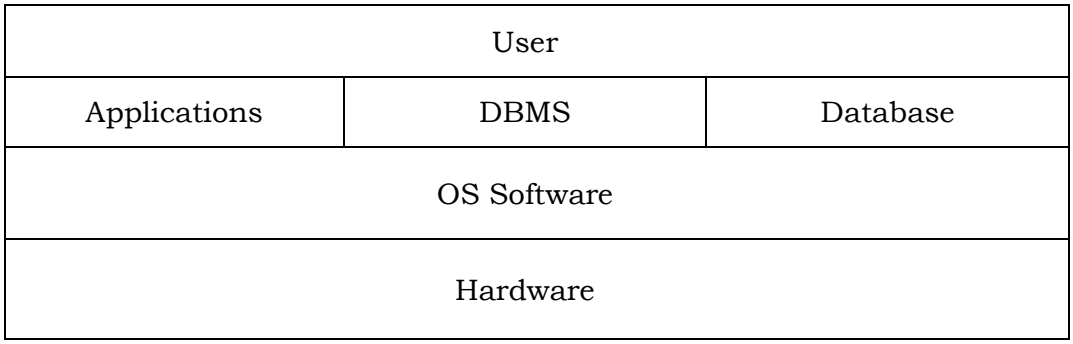


Fig. 1.1 Data Base System

- 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 Fare 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.

1.4 LET US SUM UP

Check your progress : Model Answers :

1. One-to-One, One-to-Many, Many-to-One Relationships. (Examples).

LESSON 2

RELATIONAL DATA MODEL

Contents

2.0 AIMS AND OBJECTIVES

To learn about basic database terminology.

- Integrity rules and types of relationships are explained.
- Two theoretical relational languages for data retrieval, relational algebra and relational calculus, are introduced.

2.1 Relational Data 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.

Figure 2-1 shows six relational tables-PR0J2002, PR0J2003, PRJPARTS, PARTS, DEPARTMENT, and EMPLOYEE. PROJ2002 has three columns and five entities. PR0J2003 contains three columns and four entities. PRJPARTS has three columns and five entities. In relational terminology, a row is also referred

to as a tuple. It rhymes with couple. In a relational database, it is easy to establish relationships between tables. For example, it is possible to find the name of the vendor who supplies parts for a project.

Each column in a relation or a table corresponds to a column of the relation, and each row corresponds to an entity. The number of columns in a table is called the degree of the relation. For example, if a table has four columns, then the table is of degree 4.

It is assumed that there is no predefined order to rows of a table and that no two rows have the exact same set of values. The order of columns is also immaterial, but correct order is used in the illustrations.

The set of all possible values that a column may have is called the domain of that column. Two domains are the same only if they have the same meaning and use. ProjNo, PartNo, DeptNo, and EmpNo are columns with numeric values, but their domains are different.

PROJ2002

projNo	Loc	Customer
1	Miami	Stocks
3	Trenton	Smith
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

PROJ2003

ProjNo	Loc	Customer
1	Miami	Stocks
2	Orlando	Allen
3	Trenton	Smith
4	Charlotte	Jones

PRJPARTS

ProjNo	PartNo	Qty
1	11	20
2	33	5
3	11	7
1	22	10
2	11	3

DEPARTMENT

DeptNo	DeptName
10	Production
20	Supplies
30	Marketing

PARTS

PartNo	PartDesc	Vendor	Cost
11	Nut	Richards	19.95
22	Bolt	Black	5.00
33	Washer	Mobley	55.99

EMPLOYEE

EmpNo	Ename	DeptNo	ProjNo	Salary
101	Carter	10	1	25000
102	Albert	20	3	37000
103	Breen	30	6	50500
104	Gould	20	5	23700
105	Barker	10	7	75000

Fig. 2.1 Relational Database Tables

Relational Terminology	File System Terminology
Entity Set or Table or Relation	File
Entity or Row or Tuple	Record
Attribute or Column	Field

Fig.2.2 Terminology Comparison

Terms like tuple and degree are used here because they are relational database terms, but in reality, these terms are not used in workplace.

Figure 2.2 shows a simple comparison between terminology used in relational databases and file systems. Many times, terms are borrowed from the file system for the relational field, and vice versa.

A key is a minimal set of columns used to uniquely define any row in a table. If a single column can be used to describe each row, there is no need to use two columns as a key. For example, in PROJ2002, ProjNo uniquely defines each row, and in PARTS, PartNo uniquely defines each row. In PRJPARTS, none of the columns defines each row uniquely by itself. The column ProjNo is not unique, and PartNo is not unique either. In such a table, a combination of columns can be used as a key. For example, ProjNo and PartNo together make a key for PRJPARTS table. When a single column is used as a unique identifier, it is known as a primary key. When a combination of columns is used as a unique identifier, it is known as a composite primary key or, simply, as a composite key.

Sometimes, a more human approach is used to identify or retrieve a row from a table because it is not possible to remember primary key values such as

the employee number, part number, department number, and so on. For example, a vendor's name, an employee's last name, a book's title, or an author's name can be used for the data retrieval. Such a key is known as a secondary key.

If none of the columns is a candidate for the primary key in a table, sometimes database designers use an extra column as a primary key instead of using a composite key. Such a key is known as a surrogate key. For example, columns such as customer identification number, term identification number, or vendor number can be added in a table to describe a customer, term, or vendor, respectively.

In a relational database, tables are related to each other through a common column. A column in a table that references a column in another table is known as a foreign key. For example, the PartNo column in PRJPARTS is a foreign key column that references the PartNo column in PARTS.

Figure 2.3 shows typical illustrations showing the notation used for tables in a relational database. The table name is followed by a list of columns within parentheses. The primary key or composite primary key columns are underlined. In Oracle, the primary key, composite key, or surrogate key is defined as a primary key only, and a foreign key in a table can reference a primary key column only.

Oracle uses key words PRIMARY KEY to define a primary, composite, or surrogate key. In Oracle tables, only primary and foreign keys are defined. Secondary key is not part of Oracle's table structure, but it is a column used in search operations. Later, you will learn to use Oracle's Data Dictionary to find table keys and other table information.

PROJ20 (ProjNo, Loc, Customer)

PROJ2003 (ProjNo, Loc, Customer)

PRJPARTS (ProjNo, PartNo, Qty)

PARTS (PartNo, PartDesc, Vendor, Cost) DEPARTMENT (DeptNo, DeptName)

EMPLOYEE (EmpNo, Ename, DeptNo, ProjNo, Salary)

Fig. 2.3 Notation used for tables.

2.2 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:

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.

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 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.

Check your progress 1 :

State the two integrity rules of the Relational Model?

2.3 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 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.

Suppose we want to see all the projects from years 2002 and 2003. We obtain it by performing a union (U) on the PR0J2002 and PROJ2003 tables as given in Figure 1-2.

If we call the resulting table TABLE_A, the operation can be denoted by

ProjNo	Loc	Customer
1	Miami	Stocks
.2	Orlando	Allen
3	Trenton	Smith
4	Charlotte	Jones
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

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.

If we use the same two tables that were used in the union operation, the intersection will give us the projects that appear in the year 2002 and in the year 2003. Let us call the resulting table, which is produced by the intersection (0) operation, TABLE_B:

ProjNo	Loc	Customer
1	Miami	Stocks
3	Trenton	Smith

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.

If we find the difference (-) of the same two tables used in the previous operations and create TABLE_C, it will have projects for the year 2002 that are not projects for the year 2003:

TABLE_C = PROJ2002 - PROJ2003

TABLE_C

ProjNo	Loc	Customer
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

Now, just as in mathematics, A - B is not equal to B - A. If we perform the same operation to find projects from the year 2003 that did not exist in year 2002, the resulting TABLE_D will look like this:

TABLE_D

ProjNo	Lac	Customer
2	Orlando	Allen
4	Charlotte	Jones

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:

TABLE_E = PARTS (PartDesc. Cost)

TABLE_E

PartDesc	Cost
Nut	19.95
Bolt	5.00
Washer	55.99

Selection. The selection operation selects rows from a table based on a condition or conditions. The conditional operators (=, < >, >, > =, <, < =) 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.

Let us apply the selection (Sel) operation to the PARTS table:

TABLE_F = Sel(PARTS:Cost >10.00)

TABLE_F

PartNo	PartDesc	Vendor	Cost
11	Nut	Richards	19.95
33	Washer	Mobley	55.99

The resulting table has the same number of columns as the original table but fewer rows. The rows that satisfy the given condition are returned.

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 . y rows. If the first table has m columns and the second table has n columns, the resulting product has m + n columns.

For simplicity, let us take two tables with one column each and perform the product (.) operation on them:

DEPARTMENT

DeptName
Production
Supplies

Marketing

EMPLOYEE

Ename
Carter
Albert

TABLE_G = EMPLOYEE.DEPARTMENT

TABLE_G

Ename	DeptName
Carter	Production
Carter	Supplies
Carter	Marketing
Albert	Production
Albert	Supplies
Albert	Marketing

In this example, EMPLOYEE has two rows and DEPARTMENT three rows, so TABLE_G has 2· 3 = 6 rows. EMPLOYEE has one column and DEPARTMENT one column, so TABLE_G has 1 + 1 = 2 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.

For example, if we are interested in employee information along with department information, a join can be carried out using the EMPLOYEE and DEPARTMENT tables shown in Figure 1.2. The DeptNo column is the common column in both tables and will be used for the join condition:

TABLE_H = join (EMPLOYEE, DEPARTMENT: DeptNo = DeptNo)

TABLE_H

EmpNo	Ename	DeptNo	ProjNo	Salary	DeptName
101	Carter	10	1	25000	Production
102	Albert	20	3	37000	Supplies
103	Breen	30	6	50500	Marketing
104	Gould	20	5	23700	Supplies
105	Barker	10	7	75000	Production

The expression is read as "join a row in the EMPLOYEE table with a row in the DEPARTMENT table, where the DeptNo value in the EMPLOYEE table is equal to the DeptNo value in the DEPARTMENT table."

The join operation is an overhead on the system, because it is accomplished using a series of operations. A product is performed first, which results in 5 . 3 = 15 rows. A selection is performed next to select rows where the DeptNo values are equal. Finally, a projection is performed to eliminate duplicate DeptNo columns.

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:

PROJ

ProjNo
1
2
3

PRJPARTS

ProjNo	PartNo
1	11
2	33
3	11
1	22
2	11

Suppose we want to find out which parts are used in every project. We have to divide (I) PRJPARTS by PROJ:

TABLE_I = PRJPARTS / PROJ

TABLE_I

PartNo
11

The columns of TABLE_I are tho e from the dividend PRJPARTS that are not in the divisor PROJ. The rows of TABLE_I are a subset of the projection PRJPARTS (PartNo). The row (PartNo) i in TABLE_I if and only if (ProjNo, PartNo) is in the dividend PRJPART for every value of (ProjNo) in the divisor PROJ.

The nine operations provide users with a sufficient set of operations to work with the relational databases. Some of the operations are combinations of other operations, as we saw in the case of the join operation, but such operations are very useful in actual practice. In later chapters on Oracle, you will find the actual query statements used to accomplish the different operations outlined here. You will learn to perform these operations using Oracle's SQL.

Check your progress 2:

What are the different types of relationships and give examples?

2.4 LET US SUM UP

Check your progress : Model Answers :

2. Entity Integrity and Referential Integrity.

LESSON 3

DATABASE DESIGN : DATA MODELING

Contents

3.0 AIMS AND OBJECTIVES

To understand the different pictorial methods, techniques, and concepts to create a "near-perfect" database.

The relational model is very popular because of its simplicity. It shows data to the user in a very simple, logical view as a two-dimensional table. Anyone can create tables, but the strongest characteristic of the relational model is its ability to establish relationships among tables, which helps to reduce redundancy. Your queries are as good as the database you create. The first and foremost step in database creation is database design, which involves a certain degree of common sense. If the given list of columns describes different entities, you would create a separate table for each entity type, You would use foreign keys to establish relationships. To join two tables, you need at least one common or redundant column in both tables. All situations are not the same. There are complex cases in which common sense does not do the job. Many proven modeling and design tools are available for a better database design.

3.1 Data Modeling

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.

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 relationships. 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.

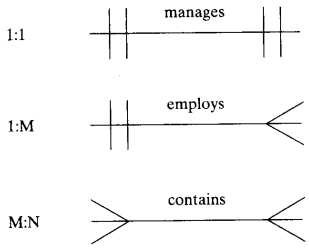
The E-R model uses E·R diagrams (ERD) for graphical representation of the database components. An entity (or an entity set) is represented by a rectangle. The name of the entity (set) is written within the rectangle. Some

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 which is given in Fig.3.1.

Entity representation in an E-R diagram.

Fig. 3.1

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. The representation is given in Fig. 3.2.



Representation of relationship in an E-R diagram.

Fig 3.2

The types of relationships (1:1, l:M, and M:N) between entities are called connectivity or multiplicity. The connectivity is shown with vertical or angled lines next to each entity, as shown in Figure 2-2. 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. Figure 3-3 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, mY, 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 (l,N) means a minimum of one and a maximum equal to any number (see Fig. 3-4). Some modern tools do not show cardinality in an E-R diagram.

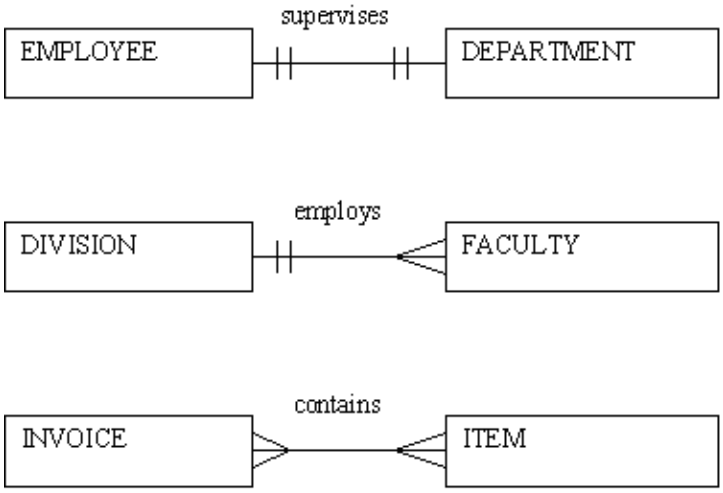


Fig.3.3 Entity Relationship and Connectivity

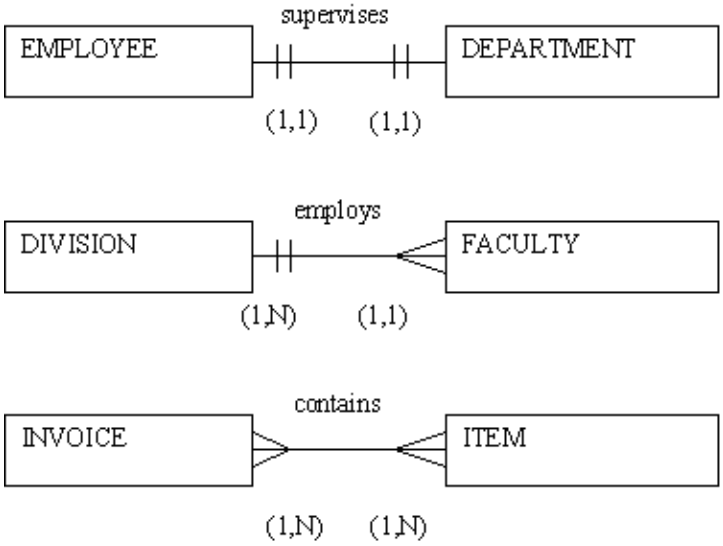


Fig.3.4 Cardinality

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.

Figure 3-4 shows the 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.

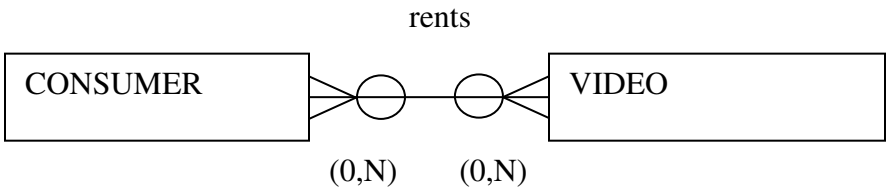
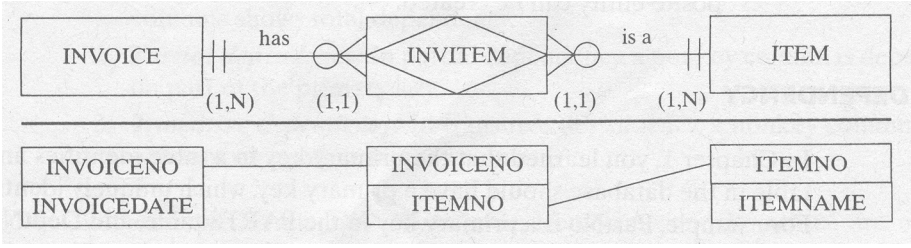


Fig. 3.5 Optional Relationships

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 in Figure 2-4. 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 l: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 (see Fig. 2-6). The composite 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. In the employee database, 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 (Fig. 3-7).

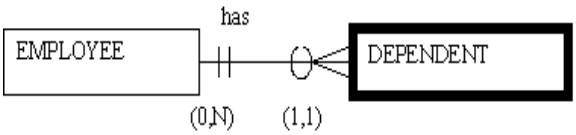


Fig.3.7 Weak Entity

•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.

Check your progress 1:

What do you mean by Data Modeling?

……
……

3.2 Dependency

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 Orade, if you create a table and do not define its primary key, Oracle do.es 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 in Figure 3-8. The INVOICE table in Figure 3-8 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.

1.Total or full dependency: A non key 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

3.3 Let us Sum Up

Check your progress : Answers

1. Modeling is a tool to represent the various components and their relationships. The entity-relationship (E-R) model is a modeling tool.

LESSON 4

DATABASE DESIGN : NORMALIZATION

Contents

4.0 AIMS AND OBJECTIVES

To understand the data base design, the normalization of the database using various normal forms.

4.1 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.

4.2 NORMAL FORMS

In Figure 4.1, 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, in Figure 4.1, 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

	InvDate	CustNo	ItemNo	CustName	Item Name	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

Fig. 4.1 INVOICE Table and its columns

Check your progress 1:

What is Normalization?

……

……

……

……

……

……

4.2.1 First Normal Form (1NF)

A table that is in INF may have redundant data. A table in INF 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.

4.2.2 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.

The partial dependency exists in a table in which nonkey columns are partially dependent on part of a composite key. Suppose a table is in INF 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.

4.2.3 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 domain key normal form (DKNF) are not covered in this text. In the following section, you will learn the normalization process by using dependency diagrams.

4.3 Let Us Sum Up

Check your Progress: Answer

1. Normalization is a decomposition process to reduce data redundancy and data anamolies.

LESSON 5

DATABASE DESIGN : DEPENDENCY DIAGRAMS

Contents

5.0 AIMS AND OBJECTIVES

To understand the dependency diagram and the process of denormalization.

5.1 Dependency Diagrams

A dependency diagram is used to show total (full), partial, and transitive dependencies in a table:

- The primary key components are highlighted. They are in bold letters and in boxes with a darker border. The primary key components are connected to each other using a bracket.
- The total and functional dependencies are shown with arrows drawn above the boxes.
- The partial and transitive dependencies are shown with arrows at the bottom of the diagram.

Conversion from 1NF to 2NF

In Figure 5-1, a composite key is in the table and INF-to-2NF conversion is required. In this conversion, you remove all partial dependencies:

- First, write each primary key component on a separate line, because they will become primary keys in two new tables. (Note: If a primary key component does not have partial dependency on it, there is no need to write it on a separate line. In other words, you don't create a new table with that primary key.)
- Write the composite key on the third line. It will be the composite key in the third table.

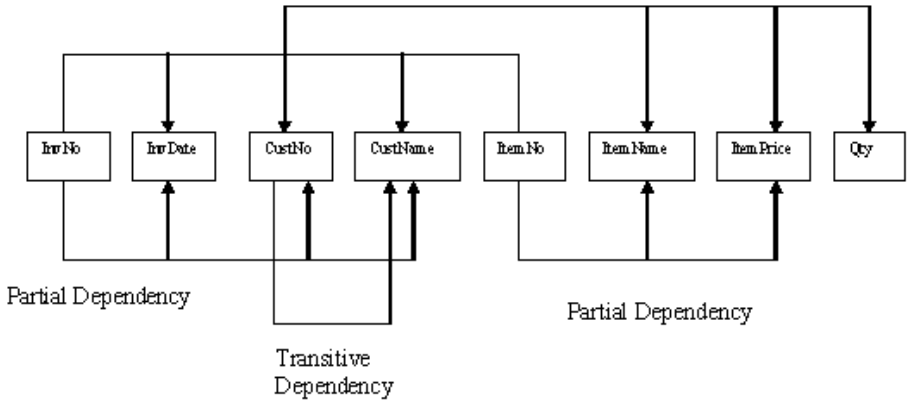


Fig.5-1 Dependency Diagram

Figure 5-2 shows the decomposition of one table in INF into three tables in 2NF.

The reason behind the decomposition is moving columns with partial dependency to the new table along with the primary key. If only one of the two primary key columns has nonkey columns dependent on it, you will create only one new table to remove the partial dependency.

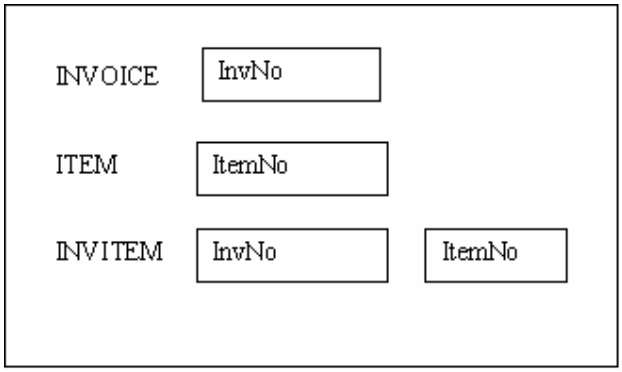


Figure 5-2 1NF-to-2NF decomposition

The InvNo, CustNo, and CustName columns will move to the INVOICE table, because they are partially dependent on InvNo. ItemName and ItemPrice will move to the ITEM table, because they are partially dependent on ItemNo in Figure 5-1. The Qty column stays in INVITEM, because it is totally dependent on the composite key. The database will look like the one shown in Figure 5-3.

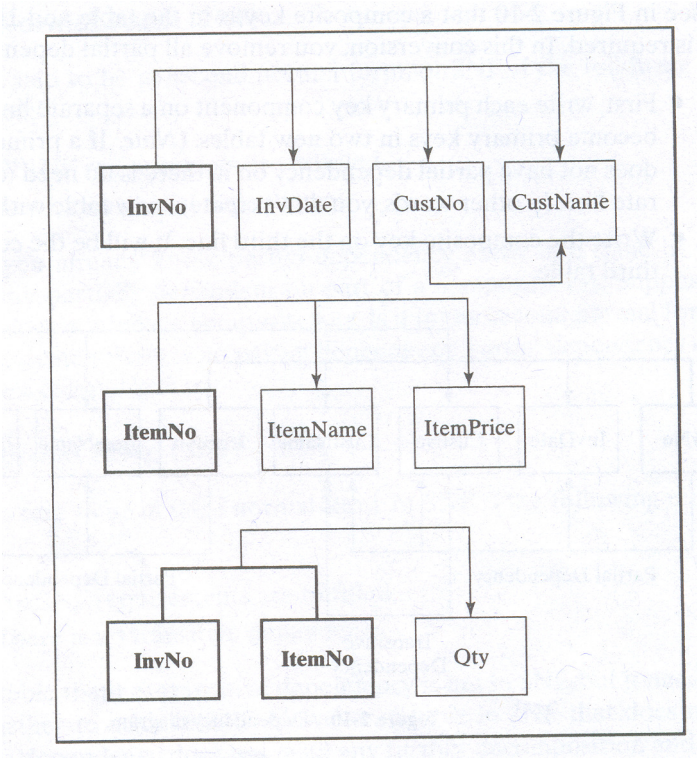


Fig.5-3 Tables in 2NF

Conversion from 2NF to 3NF

The database tables in 2NF (Fig. 5-3) have no partial dependency, but the INVOICE table still has transitive dependency:

- Move columns with the transitive dependency to a new table .
- Keep the primary key of the new table as a foreign key in the existing table.

In Figure 5-4, you see the decomposition from 2NF to 3NF to remove transitive dependency. A new CUSTOMER table is created with CustNo as its primary key. The CustNo column is kept in the INVOICE table as a foreign key to establish a relationship between INVOICE and CUSTOMER tables. The final database in 3NF looks like the one shown in Fig. 5.5

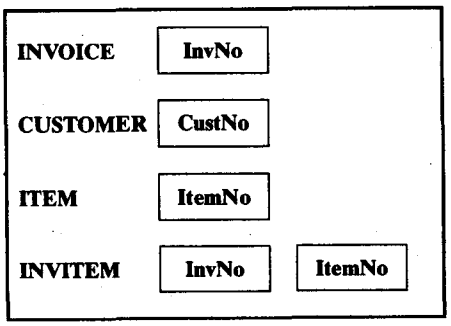


Fig. 5.4 2NF-to-3NF decomposition

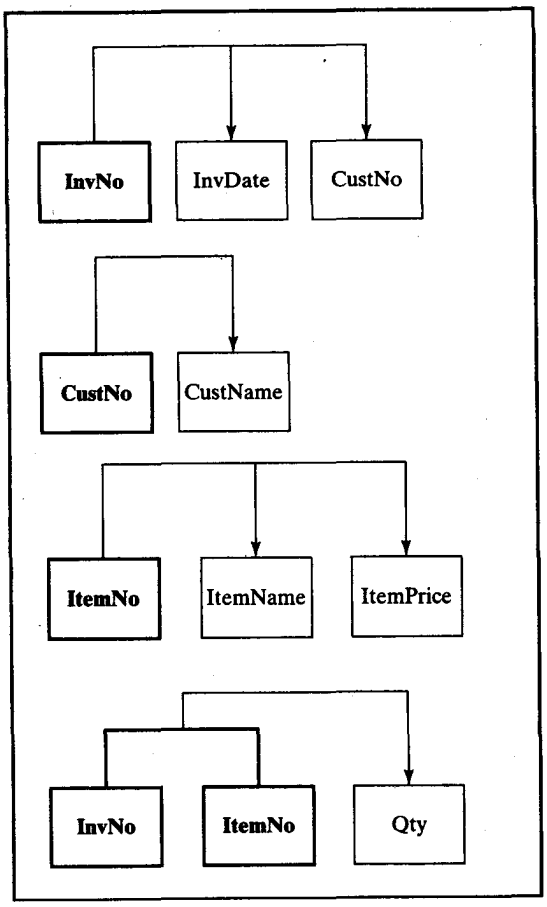


Fig.5.5 Tables in 3NF

5.2 Denormalization

The normalization process splits tables into smaller tables. These tables are joined through common columns to retrieve information from different tables. The more tables you have in a database, the more joins are needed to get the desired information. In a multiuser environment, it is a costly overhead, and system performance is affected. Denormalization is the reverse process. It reduces the normal form, and it increases data redundancy. With denormalization, the information is stored with duplicate data, more storage is required, and anomalies and inconsistent data exist. The designer has to weigh this against performance to come up with a good design and performance.

Check your progress 1:

Create a dependency diagram for the set of given columns for the EMP table with the following fields.

EMPID

FIRSTNAME

LASTNAME

DEPTID

DEPTNAME

DESIGNATION

DEPENDENTS

5.3 Another Example of Normalization

In Figure 5-6, a table is shown in 1NF. The table contains a composite key that is composed of two columns, PlayerId and Year. This table contains each player's yearly statistics as well as team information. A player may belong to different teams during different years (it is assumed that a player belongs to one team during a year).

Looking at the table, the following dependencies exist:

Total dependency-JerseyNum, PointsScoredIn Year, GamesPlayed, TeamId, TeamName, and TeamLoc columns are dependent on primary key columns PlayerId and Year. A player may wear a different jersey number with same team or with a different team. Player may play for different team every year.

Partial dependency-PlayerName and BirthDate columns are dependent on primary key column Piayerld only.

Transitive dependency- TeamName and TeamLoc columns are dependent on non-key column TeamId. Fig 2-16.

1NF to 2NF (Removing Partial Dependencies)

A new table is created with a primary key column that has partial dependency on it. A new table is created with the PlayerId column as its primary key. The original table stays as it is with columns showing total dependency.

2NF to 3NF (Removing Transitive Dependencies)

A new table is created with the TeamId column as its primary key. TeamName and TeamLoc move to this new table. TeamId column also stays in the previous table as a foreign key to reference the new table.

Playerid	Playername	Year	JerseyNum	BirthDate	PointsScoredinYear	GamesPlayed	Teamid	TeamName	TeamLoc
1	JOHNSON	2001	32	4/15/1980	150	5	1	MUSTANGS	BRONX
1	JOHNSON	2002	32	4/15/1980	174	6	1	MUSTANGS	BRONX
1	JOHNSON	2003	32	4/15/1980	115	5	1	MUSTANGS	BRONX
2	NAMAN	2001	10	12/2/1985	100	3	1	MUSTANGS	BRONX
2	NAMAN	2002	10	12/2/1985	149	6	2	DEVILS	PRINCETON
2	NAMAN	2003	10	12/2/1985	185	6	5	EAGLES	BRUNSWICK
3	SHAW	2001	11	5/10/1986	99	5	4	BEARCATS	FORDS
3	SHAW	2002	11	5/10/1986	97	6	4	BEARCATS	FORDS
3	SHAW	2003	3	5/10/1986	115	6	6	KINGS	MANHATTAN
4	ALBERT	2003	33	5/19/1983	29	3	3	BULLDOGS	MONROE
5	ANTHONY	2001	21	1/19/1979	110	6	3	BULLDOGS	MONROE
5	ANTHONY	2002	21	1/19/1979	78	4	3	BULLDOGS	MONROE
5	ANTHONY	2003	33	1/19/1979	111	5	1	MUSTANGS	BRONX
6	RICHARDS	2003	33	7/10/1977	63	6	2	DEVILS	PRINCETON
7	ROBERTS	2003	55	6/6/1981	44	6	5	EAGLES	BRUNSWICK
8	JONES	2001	2	12/31/1981	123	6	6	KINGS	MANHATTAN
8	JONES	2002	2	12/31/1981	100	6	4	BEARCATS	FORDS
9	JORDAN	2003	23	2/17/1986	101	2	1	MUSTANGS	BRONX

Fig. 5.6 Tables in 1NF

5.4 Let us Sum Up

Check your progress : Answer

Using the dependencies, classify the fields and draw the diagram

UNIT II

LESSON 6

PERSONAL DATABASES

Contents

6.0 Aims and Objectives

To understand the use of Personal databases, concept of client server databases and the differences between a client/server database.

6.1 Personal Databases

Personal databases, such as Microsoft Access and Visual Fox Pro, are usually stored on a user's desktop computer system or a **client computer.** These database packages are developed primarily for single-user applications. When such a package is used for a multi-user or a shared access environment, the database applications and the data are stored on a file server, or a **server,** and data are transmitted to the client computers over the network (Fig. 3-1). A server is a computer that accepts and services requests from other computers, such as client computers. A server also enables other computers to share resources. A server's **resources** could include the server's hard-disk drive space, application programs on a server's hard drive, data stored on the server's drive, or printers. A **network** is an infrastructure of hardware and software that enables computers to communicate with each other.

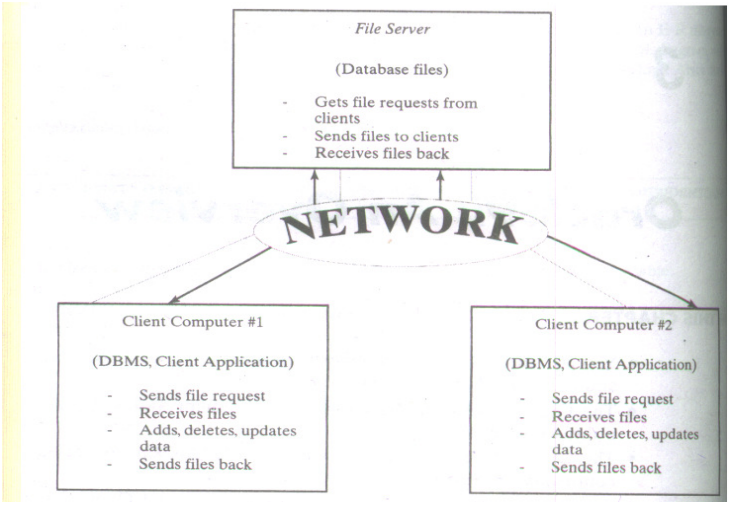


Fig. 6.1 A Personal Database System in a multiuser environment

Demand on Client and Network

In a network environment with a personal DBMS, the client computer must load the entire database application along with the client database application in its memory. If the client requires a small piece of data from the server's large database, the server has to transmit the entire database to the client over the network. In some database packages, only part of the database is transmitted. In any case, the client computer hardware must handle heavy demand, and the network must sustain heavy traffic in both directions. In the network environment, the system response to various client requests depends on the speed of the network and the amount of traffic over it.

Table Locking:

The personal database system assumes that no two transactions will happen at the same time on one table, which is known as **optimistic locking.** In optimistic locking, the tables are not locked by the database system. If one agent sells a seat for a basketball game and another agent tries to sell the same seat at the same time, the database system will notify the second agent about the update on the table after his or her read-but it will go ahead and let the second agent sell the seat anyway. Application programmers can write code to avoid such a situation, but that requires added effort on programmer's part. Personal database software does not lock tables automatically.

Client Failure:

When a client is performing record insertions, deletions, or updates, those records are locked by that client and are not available to the other clients. Now, if the client with all the record locks fails because of software or hardware malfunction or a power outage, the locked records stay locked. The transactions in progress at the time of failure are lost. The database can get corrupted and needs to be repaired. To repair the database, all users have to log off during the repair, which can take anywhere from a few minutes to a few hours! If the database is not repairable, data can be restored from the last backup, but the transactions since the last backup are lost and have to be reentered.

Transaction Processing:

Personal databases, such as Microsoft Access, do not have file-based transaction logging. Instead, transactions are logged in the client's memory. If the client fails in the middle of a batch of transactions, some transactions are written to the database and some are not. The **transaction log** is lost, because it is not stored in a file. If a client writes a check to transfer money from a savings account to a checking account, the first transaction debits money from the savings account. Now, suppose the client fails right after that. The checking account never gets credited with the amount because the second transaction is lost!

Check your progress 1:

What is a database server?

.....
.....
.....
.....
.....
.....

6.2 CLIENT SERVER DATABASES:

Client Server databases, such as Oracle, run the DBMS as a process on the server and run a client database application on each client. The client application sends a request for data over the network to the server. When the server receives the client request, the DBMS retrieves data from the database, performs the required processing on the data, and sends only the requested data (or query result) back to the client over the network (Fig. 6-2)

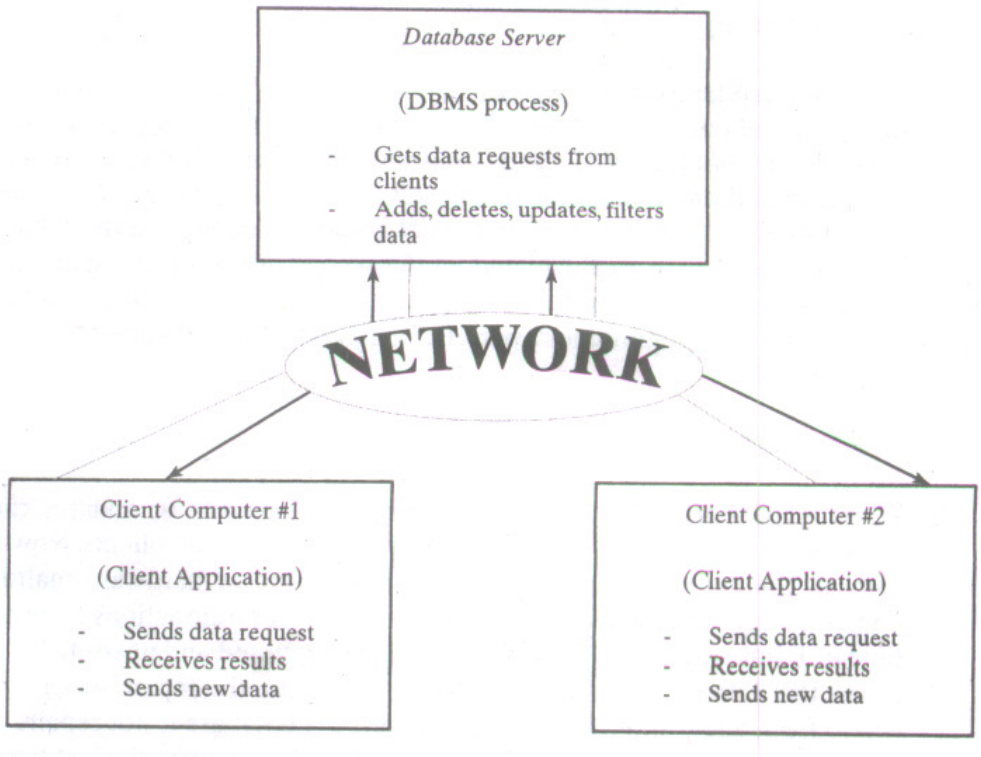


Fig. 6.2. A client/server database system in a multiuser environment

Demand on Client and Network:

The Client computer does not run the entire DBMS, only the client application that requests data from the server. The client does not store any database on its local drive; it receives the requested data from the server. Data Processing is performed on the server's side. The demand at the client's end is minimal. The clients request data from the server, ad only the requested data are sent back via the network, which results in less network traffic.

Table locking:

In a client/server system, such as Oracle, when an agent reads a table to sell a seat for a basketball game, for example, it is locked totally or partly by the DBMS. The second agent cannot read the part of the table with available seats. Once the first agent sells the seat and it is marked as sold, the lock is released for the next agent. The DBMS takes care of the locking automatically, and it involves no extra effort on an application programmer's part.

Client failure:

In case of a client failure, the client/server database is not affected. The other clients are not affected either. Only the failed client's transactions in progress are lost. If the server fails, a **central transaction log,** which keeps a log of all current database changes, allows the Database Administrator (DBA) or DBMS to complete or roll back unfinished transactions. The rolled-back transactions are not implemented in the database. The DBA (or DBMS) can notify clients to resubmit rolled-back transactions. Most client/server database packages have fast and powerful recovery utilities.

Transaction Processing:

If a grouped transaction or batch transaction fails in the middle, all transactions are rolled back. The DBMS will enable the bank, for example, to make sure that both accounts' balances are changed if the batch transaction goes through. If the batch transaction fails, the balance in none of the accounts is changed.

6.3 Let us Sum Up

A data base server is a one which gets file requests from the client and sends files to clients. The sending and receiving of data to the client application is taken care by the database server.

LESSON 7

ORACLE 9i : OVERVIEW

Contents

7.0 Aims and Objectives

To understand the Oracle 9i database server and the concepts of SQL*Plus environments, commands.

7.1 Oracle 9i: AN INTRODUCTION:

Oracle9i is a client/server DBMS that is based on the relational database model. Oracle9i is one of the most popular database-management software packages available today. The Oracle Corporation, incorporated in 1986, is the second-largest software company in the world. Its software product line includes Oracle9i Database, Oracle9i Application Server, Oracle9i Developer Suite, Oracle Collaboration Suite, and Oracle E-Business Suite. Oracle Corporation's revenue was $9.475 billion in the fiscal year ending May 2003, down 2 percent from the previous year. The net income for the fiscal year was $2.4 billion. Currently, the common stock trades at more than $14, with a company market capitalization of more than $75 billion. Oracle9i database is capable of supporting over 10,000 simultaneous users and a database size of up to 100 terabytes! It is preferred to the other PCbased RDBMS packages because its client/server database qualities, failure handling, recovery management, administrative tools to manage users and the database, object-oriented capabilities, graphical user interface (GUI) tools, and Web interface capabilities. It is widely used by corporations of all sizes to develop mission-critical applications. It is also used as a teaching tool by educational institutions to teach object-relational database technology, **Structured Query Language (SQL), PL/SQL** (Oracle's procedural language extension to SQL), and interfacing Web and Oracle databases. Oracle has an educational initiative program to form partnerships with educational institutions that enable these institutions to obtain Oracle database software at a nominal membership fee.

Oracle software is installed to work in three different environments. In a *stand-alone* environment, such as a laptop or desktop that is not on a network, Oracle Enterprise database software and SQL*Plus client software are installed on same machine. In a *Client/Server* environment, a two-tier architecture with a

client communicating with a server, Oracle Enterprise database software resides on the server side, and SQL"Plus client software resides on the client machine. In *Three- Tier* architecture, the client communicates with the Oracle database server through a middle tier iSQL*Plus, an interface through a Web browser.

Oracle9i components include Oracle9i Database, Oracle9i Application ServeJ and Oracle9i Developer Suite. Oracle9i Database introduces Oracle9i Real Application Cluster, which replaces Oracle Parallel Server, and features integrated s tern management, high availability, powerful disaster recovery, system fault recovery, planned downtime, and high security. Oracle9i Application Server is industry's preferred application server for database-driven Web sites, with an innm tive and comprehensive set of middle-tier services. Oracle9i Developer Suite, integrated product, provides a high-performance development environment \\i tools like Oracle Forms Developer, Oracle Designer, Oracle JDeveloper, Ora Reports Developer, and Oracle Discoverer. Some of the Oracle9i tools include:

- SQL'Plus-The SQL*Plus environment is for writing command-line queries to work with database objects such as tables, views, synonyms, an sequences.
- PLISQL-PL/SQL is Oracle's extension to SQL for creating procedure code to manipulate data.
- Developer Suite-This tool is used for developing database applications a includes:
- JDeveloper-a Java development tool.
- Designer-to model business processes and generate Enterprise applications.
- Forms Developer-a development tool for Internet and client/serve based environments.
- Oracle Reports-a report generation tool.
- Enterprise Manager-A tool for managing users and databases. Enterprise Manager uses the following tools:
- Storage Manager-to create and manage "tablespaces."
- Instance Manager-to start, stop, or tune databases.
- Security Manager-to create and manage users, profiles, and roles.
- Warehouse Manager-to manage data warehousing applications.
- XML Database Manager-to render traditional database data as XM for e-business support.
- SQL Worksheet-to enter, edit, and execute SQL 'Plus code or to run clien side scripts.
- iSQL'Plus-a Web-based environment to execute SQL*Plus code.

- Oracle Application Server (Oracle9iAS)-A tool for creating a Web site that allows users to access Oracle databases through Web pages. It includes:
- Web Server.

7.2 THE SQL *PLUS ENVIRONMENT :

When a user logs in to connect to the Oracle server, SQL*Plus provides the user with the SQL> prompt, where the user writes queries or commands. Features of SQL*Plus include:

- Accepts ad hoc entry of statements at the command line prompt (i.e., SQL>).
- Accepts SQL statements from files.
- Provides a line editor for modifying SQL queries.
- Provides environment, editor, format, execution, interaction, and file commands.
- Formats query results, and displays reports on the screen.
- Controls environmental settings.
- Accesses local and remote databases.

7.3 STRUCTURED QUERY LANGUAGE (SQL)

The standard query language for relational databases is SQL (Structured Query Language). It is standardized and accepted by ANSI (American National Standards Institute) and the ISO (International Organization for Standardization). Structured Query Language is a fourth-generation, high-level, nonprocedural language, unlike third generation compiler languages such as C, COBOL, or Visual Basic, which are procedural. Using a nonprocedural language query, a user requests data from the RDBMS. The SQL language uses English-like commands such as CREATE, INSERT, DELETE, UPDATE, and DROP. The SQL language is standardized, and its syntax is the same across most RDBMS packages. The different packages have minor variations, however, and they do support some additional commands. Oracle's SQL is different from the ANSI SQL. Oracle's SQL is referred to as SQL', but we will simply call it SQL throughout this text. Oracle9i also supports ANSI syntax for joining tables.

Oracle9i uses the following types of SQL statements for command-line queries to communicate with the Oracle server from any tool or application:

- Data retrieval-retrieves data from the database (e.g., SELECT).
- Data Manipulation Language (DML)-inserts new rows, changes existing rows, and removes unwanted rows (e.g., INSERT, UPDATE, and DELETE).
- Data Definition Language (DDL)--creates, changes, and removes a table's structure (e.g., CREATE, ALTER, DROP, RENAME, and TRUNCATE).
- Transaction control-manages and changes logical transactions.

Transactions are changes made to the data by DML statements that are grouped together (e.g., COMMIT, SAVEPOINT, and ROLLBACK).

•Data Control Language (DCL)-gives and removes rights to Oracle objects e.g., GRANT and REVOKE).

The SOL queries are typed at the SOL> prompt If a query exceeds one lm, the SOL 'Plus environment displays the next line number on the line editor. An SQL query is sent to the server by ending a query with a *semicolon* (;). A query can also be sent to the server by using a forward slash *(I)* on a new line instead of ending the query with a *semicolon.*

7.4 LOGGING IN TO SQL *PLUS

In the Windows environment click Start | Programs | Oracle – Orahome92 | Application Development | SQL Plus. A Logon window will pop up. Enter your username, password and the host string as provided by your Database Administrator. (Fig 7.1).

In a command-line environment such as DOS, type sqlplus [username [pass word[@host/database]]] to log in. If the entire command is typed, the password will be visible on the screen. If you enter your username only, a prompt will be displayed for your password. The password typed at this prompt will be masked to maintain its integrity (Fig 7.2).

There are a couple of common login problems. If you enter an incorrect username or password, you will receive the following error message from Oracle server:

ORA·01017: invalid username/password; logon denied

User should consult DBA (students should consult their instructor or academic computing personnel) to resolve username/password problems.

If there is a connectivity issue between your client PC and Oracle or the h string has an invalid entry, you will see the following error message:

ORA-12154: TNS: Could not resolve service name

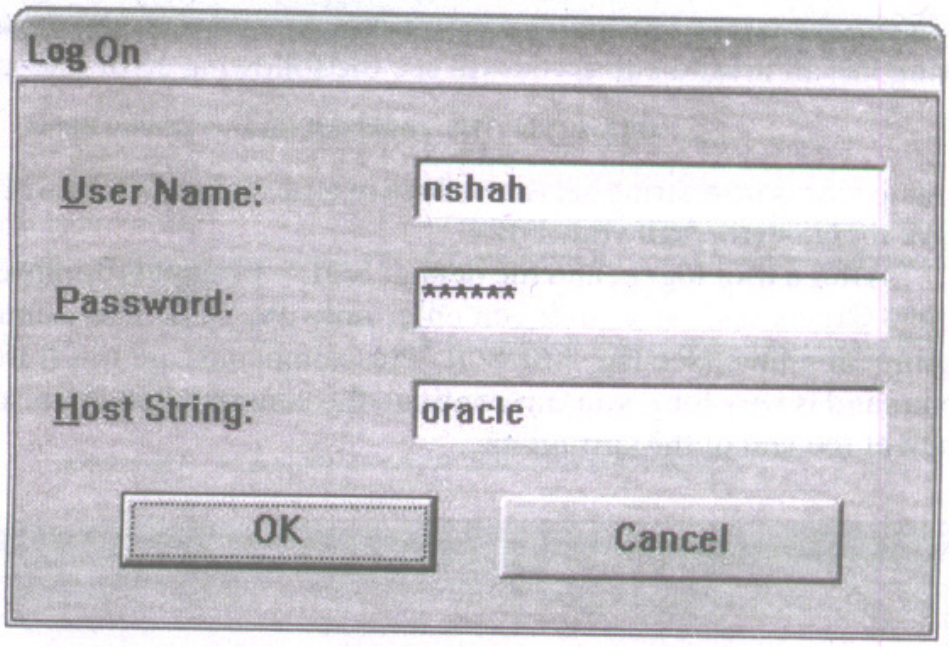


Fig. 7.1 SQL* Plus LOGON Window

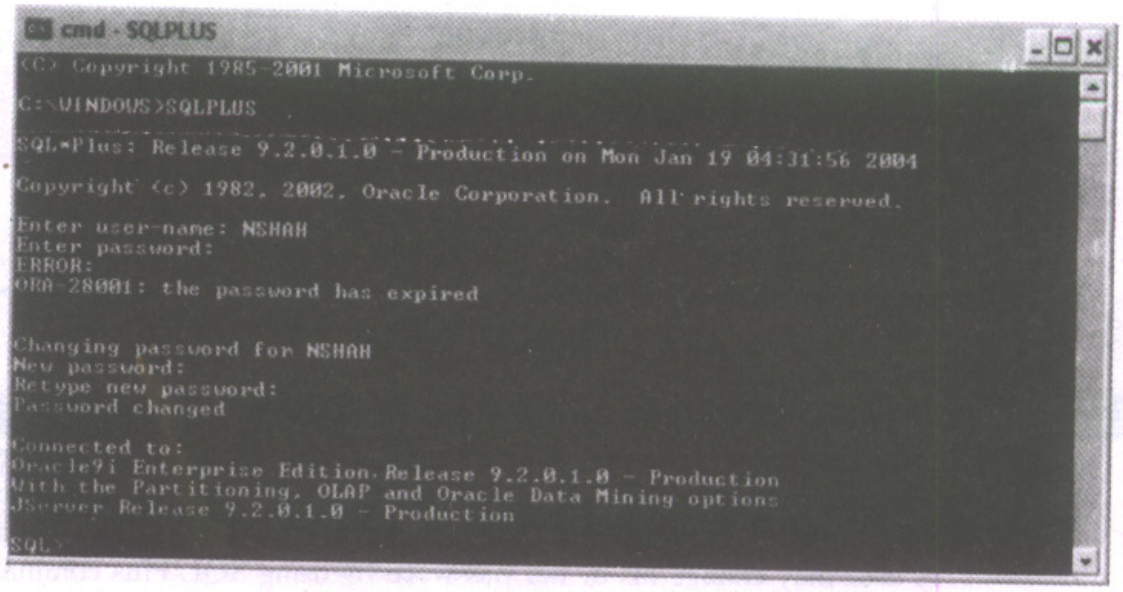


Fig. 7.2 Running SQL Plus from MSDOS Prompt.

Oracle stores host string/service values in a file called TNSNAMES.ORA.1f you receive a TNS error, call your DBA. After a user logs in and the default SOL> prompt is displayed, the user can start a new Oracle session. A user can enter only one SOL ·Plus command at the SQL> prompt at a time (Fig. 7.3). SOL*Plus commands are not stored in the buffer. If command is very long, you can continue the command on the next line by using a hyphen at the end of the current line

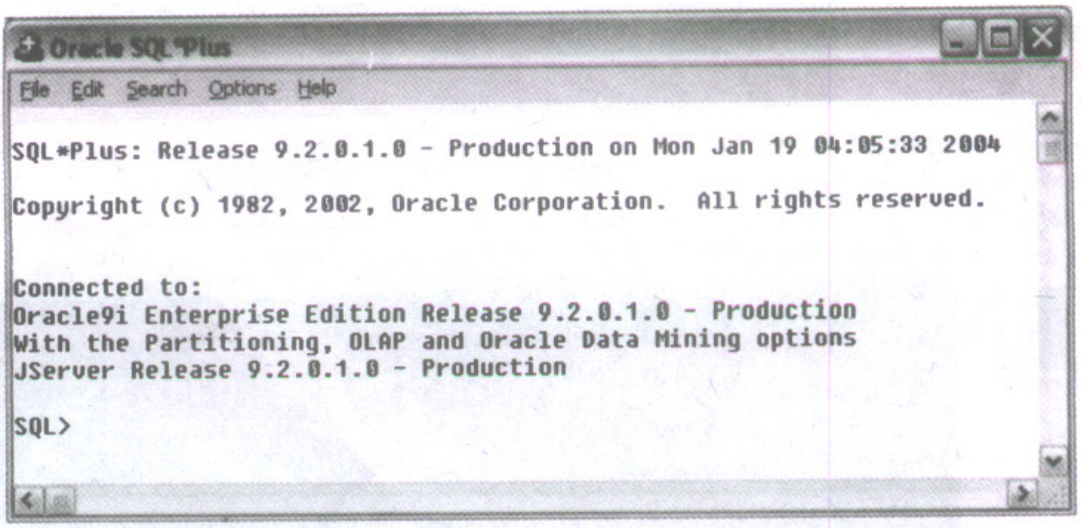


Fig. 7.3 - SQL*PLUS Environment – SQL Prompt

The SOL queries and SOL*Plus commands are typed at the SOL> prompt.

The SOL*Plus commands do not have a terminator, but SOL queries are terminated using a semicolon (;) at the end or by typing a forward slash (/) on a new line. Figure 7.4 shows the differences between SOL statements and SOL ·Plus commands.

SQL	SQL Plus
1. A nonprocedural language to communicate with the Oracle server.	1. An environment for executing SQL statements.
2. ANSI standard.	2. Oracle's proprietary environment.
3. Key words cannot be abbreviated.	3. Key words can be abbreviated.
4. Last statement is stored in the buffer.	4. Commands are not stored in the buffer.
5. Statements manipulate data and table structures in the database.	5. Commands do not allow manipulation of data in the database.
6. Uses a termination character to execute the command immediately.	6. Commands do not need a termination character.

Fig. 7.4 SQL Queries versus SQL *PLUS Commands

SQL	SQL Plus
1. A nonprocedural language to communicate with the Oracle server.	1. An environment for executing SQL statements.
2. ANSI standard.	2. Oracle's proprietary environment.
3. Key words cannot be abbreviated.	3. Key words can be abbreviated.
4. Last statement is stored in the buffer.	4. Commands are not stored in the buffer.
5. Statements manipulate data and table structures in the database.	5. Commands do not allow manipulation of data in the database.
6. Uses a termination character to execute the command immediately.	6. Commands do not need a termination character.

A user may change his or her password by using SOL*Plus command PASSWORD at the SOL> prompt. SOL*Plus prompts user to enter the old password first, then the new password, and then to confirm new password by retyping it (Fig. 7.5).

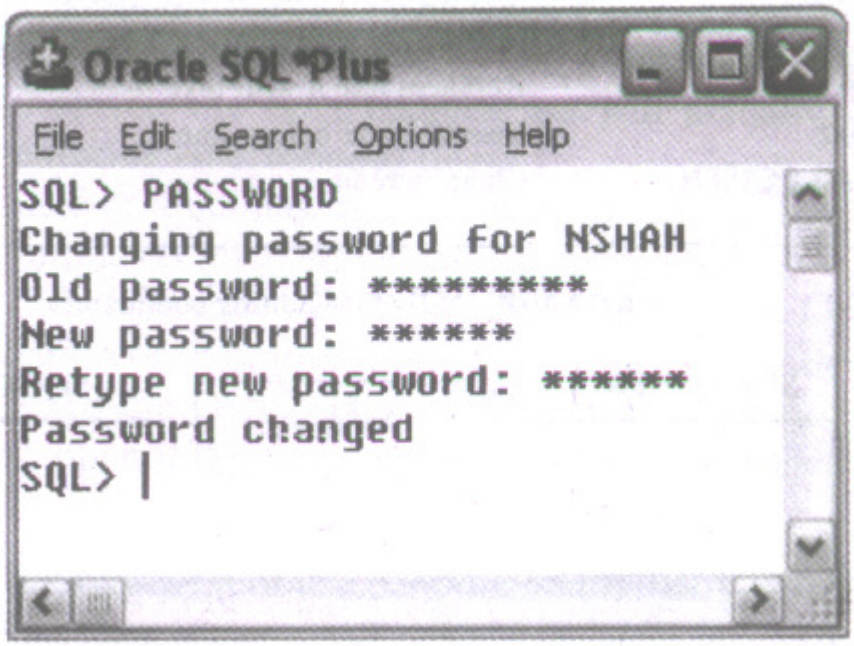


Fig. 7.5 The PASSWORD Command

Check your progress 1:

What are the different tools supported by Oracle 9i?

.....

.....

.....

.....

7.5 SQL *PLUS COMMANDS

In the tables of Figures 7.6 and 7.7, file-related (Fig. 7.6) and editor-related (Fig.7.7) commands are described. The command words are in bold letters, and user-supplied filenames and extensions are in lowercase. The abbreviations allowed for the SOL ·Plus commands are underlined. The optional parameters are enclosed within a pair of brackets ([]). Note that the filename in the file-related commands requires entire file path.

COMMAND	DESCRIPTION
GET *filename [.ext]*	Writes previously saved file to the buffer. The default extension is SQL. Writes SQL statements, not SQL*Plus commands.
START *filename [.ext]*	Runs a previously saved command from file.
@filename	Same as START.
EDIT	Invokes the default editor (e.g., Notepad), and saves buffer contents in a file called *afiedt.buf*.
EDIT [*filename [.ext]*]	Invokes editor with the command from a saved file.
SAVE *filename [.ext]* **REPLACE** | **APPEND**	Saves current buffer contents to a file with the option to replace or append.
SPOOL [*filename [.ext]* |**OFF** |**OUT**]	Stores query results in a file. OFF closes the file, and OUT sends the file to the system printer.
EXIT	Leaves SQL*Plus environment. Commits current transaction.

Fig. 7.6 – SQL *Plus File Related Commands

COMMAND	DESCRIPTION
APPEND *text*	Adds text to the end of the current line.
CHANGE / *old* / *new*	Changes old text to new text in the current line.
CHANGE / *text* /	Deletes text from the current line.
CLEAR BUFFER	Deletes all lines from the SQL buffer.
DEL	Deletes current line.
DEL *n*	Deletes line *n*.
DEL *m n*	Deletes lines *m* through *n*.
INPUT	Inserts an indefinite number of lines.
INPUT *text*	Inserts a line of text.
LIST	Lists all lines from the SQL buffer.
LIST *n*	Lists line *n*.
LIST *m n*	Lists lines *m* through *n*.
RUN	Displays and runs an SQL statement in the buffer.
N	Makes line *N* current.
n text	Replaces line *n* with text.
0 text	Inserts a line before line 1.
CLEAR SCREEN	Clears screen.

Fig. 7.7 – SQL Plus Editing Commands

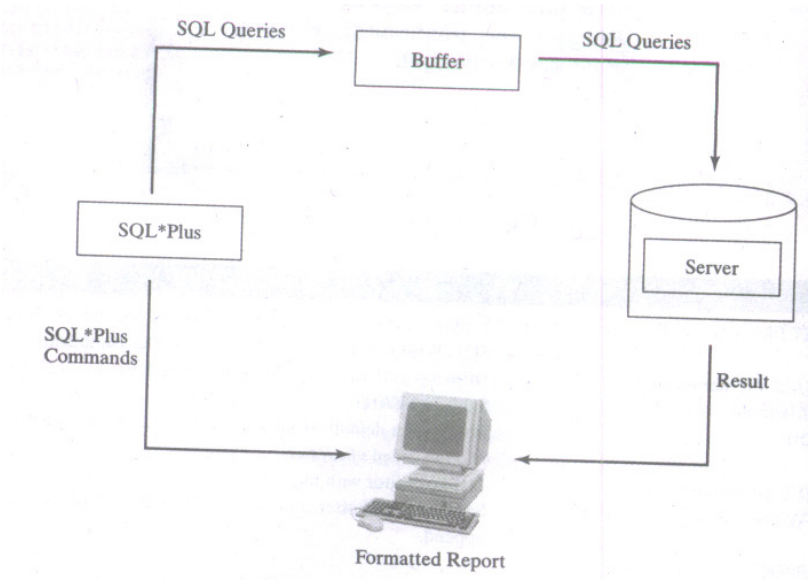


Fig. 7.8 – Interaction between SQL *Plus and SQL

A query typed at the prompt is loaded into the SQL *Plus buffer. When the query is sent to the server, the server processes data and sends back the result to the client computer, which can be formatted using SQL *Plus formatting commands. (Fig. 7-8)

7.6 LET US SUM UP

Check your progress Answers:

1. Oracle 9i Tools:

SQL*Plus, PL/SQL, Developer Suite, Enterprise Manager and Oracle Application Server.

LESSON 8

Oracle 9i: ERRORS and EDITORS

Contents

8.0 Aims and Objectives

To understand the Oracle 9i database server, the online errors and help and the various editors of Oracle 9i.

8.1 ORACLE ERRORS AND ONLINE HELP

If you make a syntax error, Oracle will display an error message showing the line number and the error location on that line. Oracle places an asterisk (*) at the location of the error and also displays an error code (e.g., ORAOOXXX), followed by a brief description. Just like any programming language compiler, some error messages are not user friendly. You will get used to some of the common error messages as you start experimenting. Some queries take up a few lines-and for a typist mistakes are bound to happen.

Some errors are easy to find; some are not. To fix an error, always start at the line where the error is shown, but keep in mind that the error might not be on that line. Cheek for common mistakes like misspelled keywords, missing commas, misplaced commas, missing parentheses, invalid user-defined names, or repeated user defined identifiers. In the next chapter, we will actually go through the entire procedure by entering an erroneous query.

Each Oracle product has its own specific help file. The help application opens up a window, similar to Microsoft Windows help, where you can click on the Index tab and type the error code (ORA-OOXXX) of interest. Oracle provides the user with an explanation of the cause of the error and the user action required to rectify it. If you don't see the Oracle Help option in the programs menu, search for the ora.hlp file and open it. You can also access Oracle9i Release 2 online help at the following URL:

http://download-west.oracle.com/docs/cd/B1050L01/mix.920/a96625/toc.htm

The URL is working at this time, but such URLs change frequently. Another way would be to go through Oracle's home page (www.oracle.com) and search for help on your Oracle product. You are required to register to Oracle Technology Network (otn.oracle.com) to access this page. Registration is free,

and the membership benefits include free software downloads and access to online documentation. Figure 8.1 shows the initial Web page with Oracle9i Release 2's master index.

SOL*Plus also provides the user with Help on its various commands and SOL language. You may use the HELP INDEX command at the SOL> prompt to list the SOL'Plus commands.

To obtain help on one of the topics listed in the help index, type INDEX [TOPIC]. For example, type HELP DESCRIBE as shown in Figure 8.2, and SOL*Plus returns a brief description and syntax on the topic.

8.2 ALTERNATE TEXT EDITORS

The SOL*Plus editor is a line editor similar to EDLIN in MS-DOS. It is not fun working with line editors. The user does not have control over the screen. Line editors do not allow a user to move the cursor up and down, and clicking with a mouse is definitely out of the question. You can use an alternate text editor such as Notepad or any other text editor in Windows to type your SQL Queries.

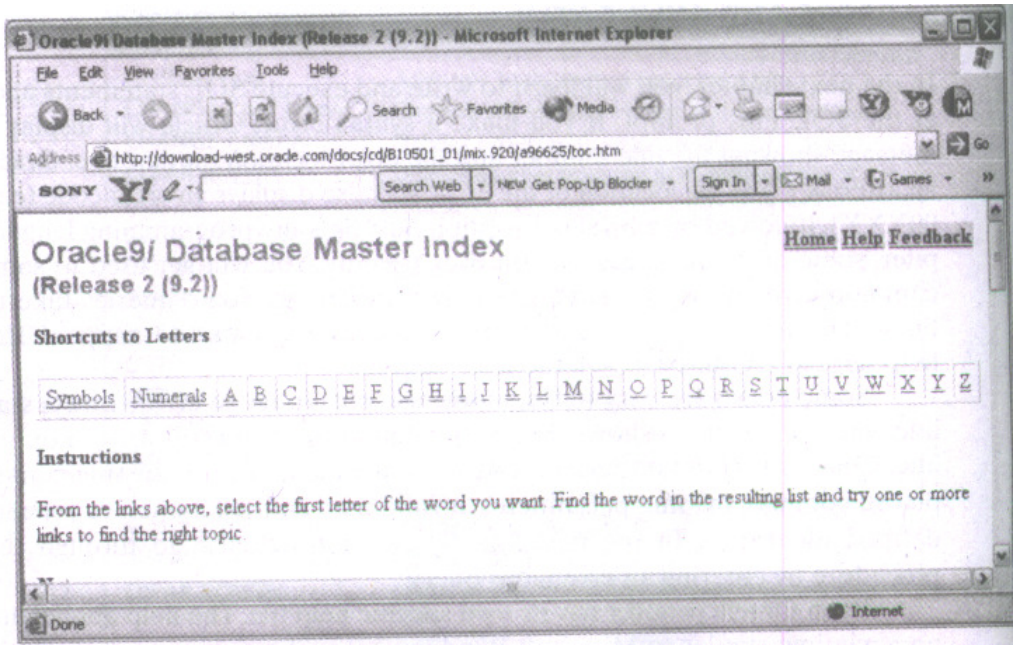


Fig. 8.1 Oracle 9i-Online Documentation- Index Page

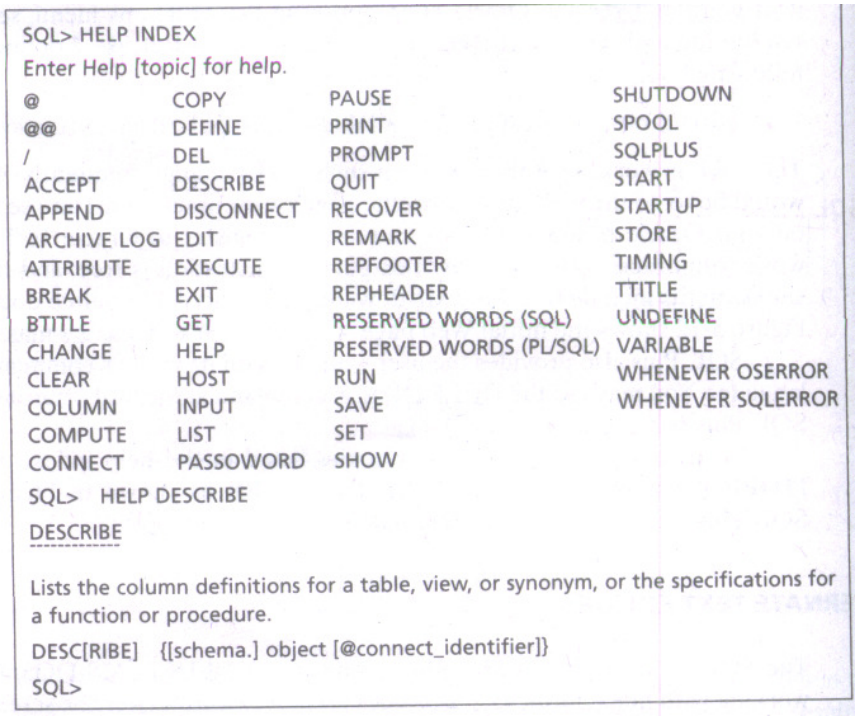


Fig.8.2 SQL*PLUS Help

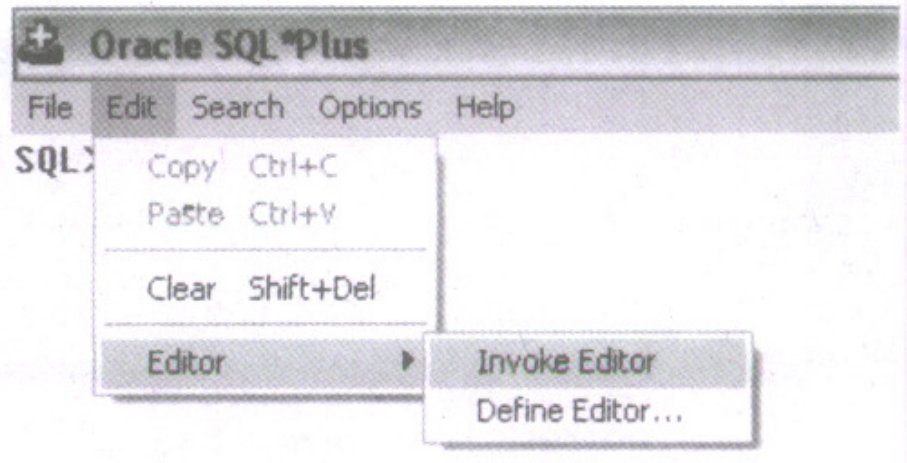


Fig. 8.3 Alternate Editor

The query typed in a full-screen text editor has to be saved in a file with an *.SQL extension* or copied to the clipboard. The query can be loaded from a file or pasted from the clipboard into SQL*Plus to execute it at the SQL>prompt.

8.3 SQL *PLUS WORKSHEET

The SQL*Plus Worksheet is another environment available with Oracle's Enterprise Manager. The SQL *Plus Worksheet enables you to enter, edit, and execute SQL*Plus code. You can also run client-side scripts. The SQL*Plus Worksheet maintains a history of the commands you have issued, so you can easily retrieve and execute previous commands. You can execute SQL*Plus Worksheet by double-clicking on the SQL*Plus Worksheet icon in the Windows

desktop or by selecting the following from the Windows START button:

START | All Programs | Oracle | OraHome92 | Application Development | SQL *Plus Worksheet

An Enterprise Manager login screen is then displayed. The user logs in with username, password, and host string, just like logging into SQL*Plus. On a successful login, the user enters the SQL*Plus Worksheet with database connection (Fig. 8-4). On the left side, a tool bar is displayed with connection, execute, command history, previous command, next command, and help icons from top to bottom, respectively. The user can select these same options from the File or Worksheet menu.

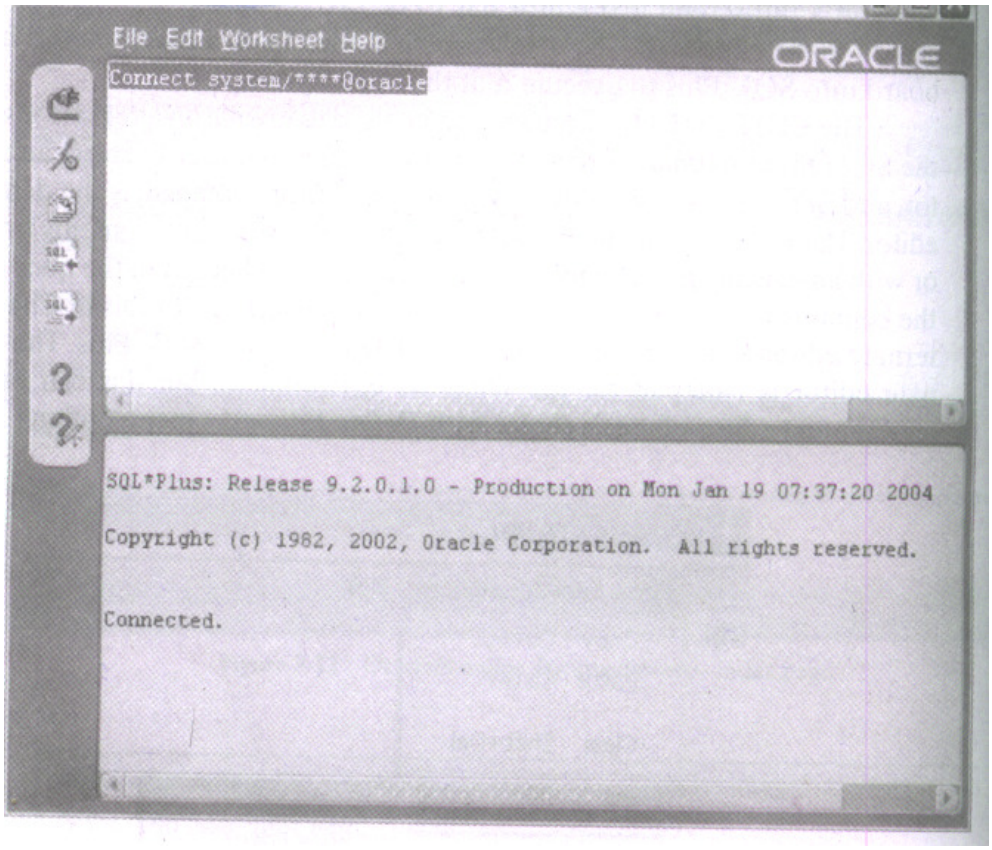


Fig. 8.4 SQL*PLUS Worksheet

Check your progress 1:

What are the various system variables used in SQL* plus worksheet?

.....

.....

.....

The SOL 'Plus Worksheet screen has two horizontal halves. The User issues an SOL query or SOL 'Plus command in the upper half and then clicks on the lightening-bolt icon to execute. SOL 'Plus Worksheet output is displayed in the lower half (Fig. 8.4). During a session, the User issues many commands and

statements. Unlike SOL 'Plus, SOL *Plus Worksheet keeps all commands and statements in history. The user can click on the command history icon to view them in reverse order, with the most recent command at the top. The user can then select a command/statement and click GET to load it again and execute it.

The user may save input and output in separate files with the FILE menu and its options Save Input As...and Save Output As... respectively. The input is stored in a file with the default extension .sql and output is stored with the default extension .txt.

There are a few differences between SQL*Plus and SQL*Plus Worksheet. The following settings have been set up by Oracle Enterprise Manager SQL'Plus Worksheet. It is recommended that users do not change them:

•The sqlplus system variable SQLPROMPT is disabled by default.

•The sqlplus system variable SQLNUMBER is set to OFF by default.

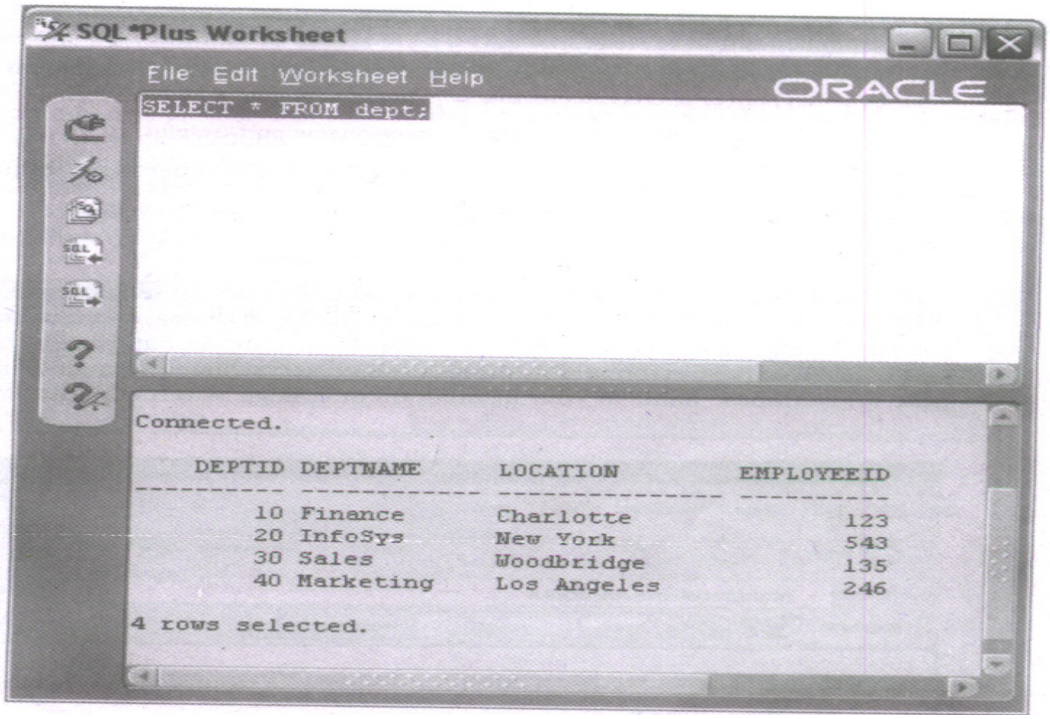


Fig. 8.5 Input / Output in SQL *Plus Worksheet

- The sqlplus system variable PAGESIZE is set to its maximum by default.
- The sqlplus system variable TAB is set to OFF by default.
- If there are any prompt characters "&" in a script, SOL*Plus treats it as a "define" prompt. (You will see more on DEFINE later.)
- The HOST command is not supported in SOL*Plus Worksheet.
- The SOL*Plus user variable _EDITOR is disabled by default. If you enable it by using "DEFINE EDITOR = 'editorname'," the specified editor launches on the server when you type the EDIT command. As a result, you are unable to access

normal SOL*Plus Worksheet functionalities.

- Remote execution is not supported.
- Remote load and save of script files is not supported.

8.4 iSQL *Plus

The third environment is Web based and is called iSQL*Plus. To access it through a web browser, enter a URL as follows:

http://machinename.domainname:port/isqlplus

In this URL, machinename is the machine, but port number is not required in all versions.

In Figure 8.6 the following URL is used:

http://nshah-monroe/isqlplus

where nshah-monroe is the machine name. The domain name is not used, because iSQL*plus is located in the local machine. In the login dialog, connection identifier (or host string) is optional if the URL points to the correct database instance.

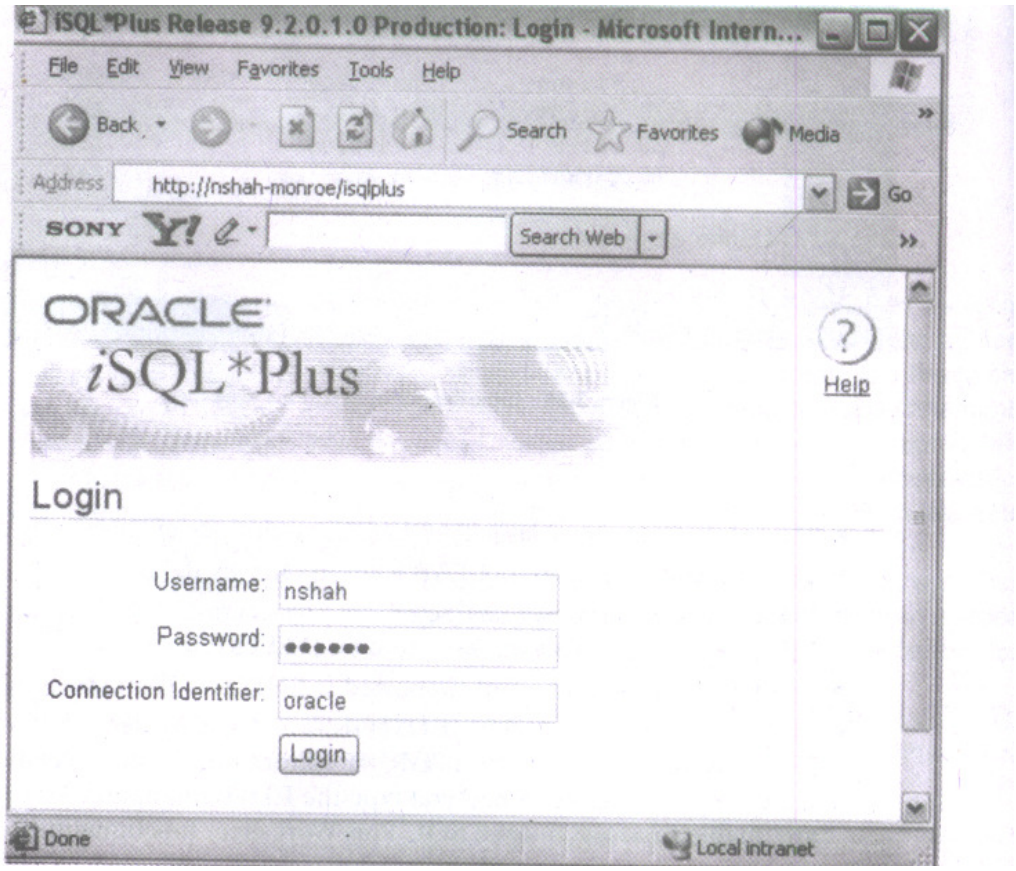


Fig. 8.6. iSQL *plus Login Screen

In Fig.8.7, a sample SQL query is executed in the iSQL *plus work screen. The output is at the bottom. This environment also provides the user with options to execute-commands, save or load scripts, check command history, and more. Another benefit of this environment is the buttons on the top right, such as the Help button to connect to Oracle's online help. History button to browse and select command history, and so on.

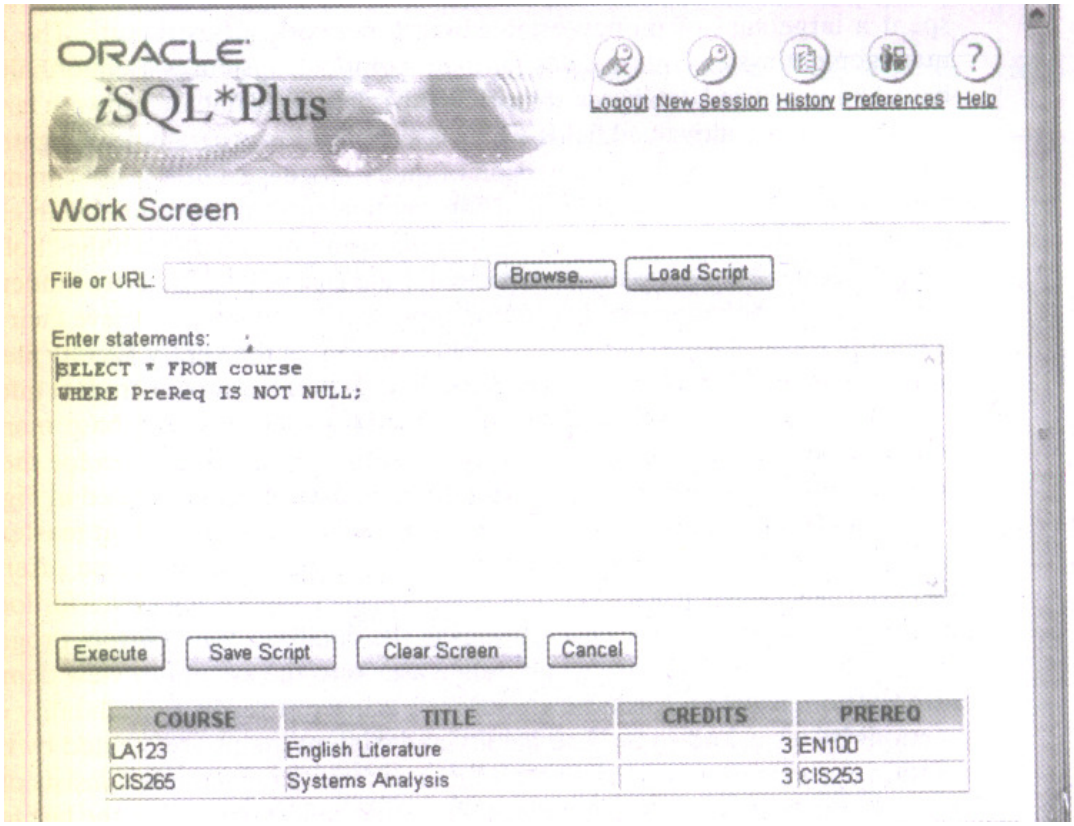


Fig.8.7 iSQL *Plus work screen with query and result.

In this text, most of the screen shots are from the SQL*Plus environment. The SQL *Plus Worksheet and iSQL *Plus environments are migrations to graphical and web based environments respectively, which are more popular environments than the command-based predecessors.

8.5 LET US SUM UP

Check your progress : Answers

1. SQLPROMPT, SQLNUMBER, PAGESIZE, TAB and _EDITOR.

LESSON 9

ORACLE TABLES : Data Definition Language

Contents

9.0 Aims and Objectives

To learn about the Data Definition Language (DDL) statements to work with the structure of an Oracle Database table, Create table and setting constraints for the fields.

9.1 ORACLE TABLES : Data Definition Language (DDL)

In Oracle9i, database tables are objects stored a user's account in an allocated table space (storage space) on the Oracle Server's disk. Each table under a user's account must have a unique table name. In the classroom environment, each student is a user with a unique login username. Each object including a table created by a user is stored under that user's schema.

9.2 NAMING RULES AND CONVENTIONS

A table is an object that can store data in an Oracle database. When you table, you must specify the table name, the name of each column, the data each column, and the size of each column. Oracle provides you with different constraints to specify a primary or a composite key for the table, to define a font in a table that references a primary key in another table, to set data validate for each column, to specify whether a column allows NULL values, and to S] a column should have unique values only.

The table and column names can be up to 30 characters long. It is pas have a table name that is only one character long. In naming tables and column (A-Z, a-z), numbers (0-9) and special characters-$ (dollar sign), _(undersea] # (pound or number sign)-are allowed. The table or column name, however begin with a letter. The names are not case sensitive, although Oracle stores all names in uppercase in its data dictionary. Spaces and hyphens are not allow table or a column name. An Oracle server-reserved word cannot be used as or a column name. Remember, the most common mistake is the use of a Sf naming a table or a column. It is always a good practice to create short but meaningful names. Also, remember that a table name must be unique in a schema (account; there

must not be another Oracle object with same name in a sc Figure 4-1 shows some valid and invalid table and column names. For invalid n the reasons are in parentheses.

Valid Names	Invalid Names
STUDENT	COURSE_REGISTRATION_ TABLE
	(more than 30 characters long)
MAJOR_CODE	MAJOR CODE (spaces not allowed)
X	CREATE
	(reserved word not allowed)
PROJEC2000	PROJEcr"'2000
	(special character' not allowed)
STUDENT#REG#TABLE	#STUDENT
	(must start with a letter)

9.3 DATA TYPES

When a table is created, each column in the table is assigned a data type. A data specifies the type of data that wil1 be stored in that column. By providing a data for a column, the wrong kinds of data are prevented from being stored in the column. For example, a name such as Smith cannot be stored in a column with a NUMBER data type. Similarly, a job title such as Manager cannot be stored in a column with a DATE data type. Data types also help to optimize storage space. Some of the Oracle data types are described below.

VARCHAR2

The VARCHAR2 type is a character data type to store variable-length alphanumeric data in a column. Currently, VARCHAR is synonymous with VARCHAR2, but it could be a separate data type with different semantics in the future. Users are advised to use VARCHAR2 only. A maximum size must be specified for this type. The default and minimum size is one character. The maximum allowable size is 4000 characters in Oracle9i. (The maximum size was 2000 characters in previous versions.) The size is specified within parentheses-for example, VARCHAR2(20). If the data are smaller than the specified size, only the data value is stored, and trailing spaces are not added to the value. For example; if a column NAME is assigned a data type VARCHAR2(25) and the name entered is Steve Jones, only 11 characters are stored. Fourteen spaces are not added to make its length equal to the size of the column. If a value longer than the specified size is entered, however, an error is generated. The longer values are not truncated. VARCHAR2 is the most appropriate type for a column whose values do not have a fixed length.

In Oracle9i, the VARCHAR2 data type can also take CHAR or BYTE parameters. For example, VARCHAR2(10 BYTE) is same as VARCHAR2(10) because byte is the default. If VARCHAR2(10 CHAR) is used, each CHAR may take up 1 to 4 bytes.

CHAR

The CHAR type is a character data type to store fixed-length alphanumeric data in a column. The default and minimum size is one character. The maximum allowable size is 2000 characters. (This was only 255 characters in previous versions.) If the value is smaller than the specified size is entered, trailing spaces are added to make its length equal to the specified length. If the value is longer than the specified size, an error occurs. The CHAR type is appropriate for fixed-length values. For example, PHONE, SOCIAL_SECURITY_NUMBER, or MIDDLE_INITIAL columns can use the CHAR type. The phone numbers and Soc;31 Security numbers have numeric values, but they also use special characters, such as hyphens and parentheses. Both use fixed-length values, however, so CHAR is the most appropriate type for them. The CHAR data type uses the storage more efficiently and processes data faster than the VARCHAR2 type.

In Oracle9i, the CHAR data type can also take CHAR or BYTE parameters. For example, CHAR (l0 BYTE) is same as CHAR(10) because byte is the default. If CHAR(10 CHAR) is used, each CHAR may take up 1 to 4 bytes. In this text, you will see the default semantic only.

NUMBER

The NUMBER data type is used to store negative, positive, integer, fixed-decimal floating-point numbers. The NUMBER data type is used for any column that is to be employed in mathematical calculations-for example, SALARY, COMMIS or PRICE. When a number type is used for a column, its precision and scale I specified. Precision is the total number of significant digits in the number, both left and to the right of the decimal point. The decimal point is not counted in special the precision. Scale is the total number of digits to the right of the decimal point precision can range from 1 to 38. The scale can range from -84 to 127.

An integer is a whole number without any decimal part. To define a c(with integer values, only the scale size is provided. For example, EmployeeId EMPLOYEE table has values of 111,246,123,433, and so on. The data type would be defined as NUMBER(3), where 3 represents the maximum number digits. Remember to provide room for future growth when defining the size. If a c ration has up to 999 employees, a size of 3 will work for now. With future growth corporation's number of employees may rise to 1000 or higher. By using a size you provide room for up to 9999 employees.

A fixed-point decimal number has a specific number of digits to the rig the decimal point. The PRICE column has values in dollars and cents, which requires two decimal places-for example, values like 2.95,3.99,24.99, and so on is defined as NUMBER(4,2), the first number specifies the precision and the se number the scale. Remember that the decimal place is not counted in the scale given definition will allow a maximum price of 99.99.

A floating-point decimal number has a variable number of decimal places decimal point may appear after any number of digits, and it may not appear at a define such a column, do not specify the scale or precision along with the NUMI type. For example, TAXRATE, INTEREST_RATE, and STUDENT _GPA

column are likely to have variable numbers of decimal places. By defining a column floating-point number, a value can be stored in it with very high precision.

DATE

The DATE data type is used for storing date and time values. The range of al able dates is between January 1,4712 B.c. and December 31, 9999 A.D. The month, century, hour, minute, and second are stored in the DATE-type column. There is no need to specify size for the DATE type. The default date format is MON-YY, where DD indicates the day of the month, MON represents the month's first three letters (capitalized), and YY represents the last two digits of the y These three values are separated by hyphens. The DD-MON-YYYY format works as the default in Oracle9i. To use any other format to enter a date value, you required to use the TO_DATE function. The default time format is HH:MM:SSA representing hours, minutes and seconds in a 12-hour time format. If only a date is entered, the time defaults to 12:00:00 A.M. If only a time is entered, the date defaults to the first day of the current month. For example, HIREDATE for EmployeeId 111 in the EMPLOYEE table in the N2 Corporation database is stored as 15-APR-60 12:00:00 A.M.

In a table, it is not advisable to use columns like AGE, because age not only changes for all entities but also changes at different times. A column like AGE can become a very high-maintenance column. It is advisable to use BIRTHDATE as a column instead. Oracle9i provides users with quite a few built-in date functions for date manipulation. Just simple date arithmetic is enough to calculate age from the birth date! The birth date never changes, so no maintenance on it is necessary.

The advanced data types are outlined here:

LONG. The LONG type is used for variable-length character data up to 2 gigabytes. There can be only one LONG-type column in a table. It is used to store a memo, invoice, or student transcript in the text format. When defining to LONG type, there is no need to specify its size.

NCHAR. The NCHAR type is similar to CHAR but uses 2-byte binary encoding for each character. The CHAR type uses I-byte ASCII encoding for each character, giving it the capability to represent 256 different characters. The NCHAR type is useful for character sets such as Japanese Kanji, which has thousands of different characters.

CLOB. The Character Large Object data type is used to store single-byte character data up to 4 gigabytes.

BLOB. The Binary Large Object data type is used to store binary data up to 4 gigabytes.

NCLOB. The character Large Object type uses 2-byte character codes. BFILE. The Binary File type stores references to a binary file that is external to the database and is maintained by the operating system's file system.

RAW(size) or LONG_RAW. These are used for raw binary data.

ROWID. For unique row address in hexadecimal format.

Many of the Large Object (LOB) data types are not supported by all versions of Oracle and its tools. These data types are used for storing digitized sounds, for images, or to reference binary files from Microsoft Excel spreadsheets or Microsoft Word documents. We will not use LOB data types in this book. The table below shows a brief summary of Oracle data types and their use in storing different types of data.

DATA TYPE	USE
VARCHAR2(SIZE)	Variable-length character data:1 to 4000Characters
CHAR(size)	Fixed-length character data: 1 to 2000 characters
	Integer values
NUMBER (p)	Integer Values
NUMBER (p, s)	Fixed-point decimal values
NUMBER	Floating-point decimal values
DATE	Date and time values
LONG	Variable-length character data up to gigabytes
NCHAR	Similar to CHAR; uses 2-byte encoding
BLOB	Binary data up to 4 gigabytes
CLOB	Single-byte character data up to 4 gigabytes
NCLOB	Similar to CLOB; supports 2-byte encoding
BFILE	Reference to an external binary file
RAW (size)	Raw binary data up to 2000 bytes
LONG_RAW	Same as RAW; stores up to 2 gigabytes
ROWID	Unique address of a row in a table

9.4 CONSTRAINTS

Constraints enforce rules on tables. An Oracle table can be created with the column names, data types, and column sizes, which are sufficient just to populate them with actual data. Without constraints, however, no rules are enforced. The constraints help you to make your database one with integrity. The constraints are used in Oracle to implement integrity rules of a relational database and to implement data integrity at the individual-column level. Whenever a row/record is inserted, updated, or deleted from the table, a constraint must be satisfied for the operation to succeed. A table cannot be deleted if there (dependencies from other tables in the form of foreign keys.

There are two types of constraints:

1.Integrity constraints: define both the primary key and the foreign k, with the table and primary key it references.

2.Value constraints: define if NULL values are disallowed, if UNIQUE Values are required, and if only certain set of values are allowed in a column.

NAMING A CONSTRAINT

Oracle identifies constraints with an internal or user-created name. For a user's account, each constraint name must be unique. A user cannot create constraints in two different tables with the same name. The general convention used for naming constraints is

<table name>_<column name>_<constraint type>

Here, table name is the name of the table where the constraint is being defined. Column name is the name of the column to which the constraint applies, and constraint type is an abbreviation used to identify the constraint's type. The constraint abbreviations are given below.

CONSTRAINT	ABBREVIATION
PRIMARY KEY	pk
FOREIGN KEY	fk
UNIQUE	uk
CHECK	ck or cc
NOT NULL	nn

For example, a constraint name emp_deptno_fk refers to a constraint in table EMP on column DeptNo of type foreign key. A constraint name dept _deptno_pk is for a primary key constraint in table DEPT on column DeptNo.

If the constraint is not named, Oracle server will generate a name for it by using SYS_Cn Format, where n is any unique number. For example, SYS_C00010 is an Oracle server-named constraint. These names are not user friendly like used named constraints.

Defining a Constraint

A constraint can be created at the same time the table is created, or it can be added to the table afterward. There are two levels where a constraint is defined:

1. Column Level: A column-level constraint references a single column and is defined along with the definition of the column. Any constraint can be defined at the column level except for a FOREIGN KEY and composite primary key constraints. The general syntax is

   Column datatype [CONSTRAINT constraint_name] constraint_type

2. Table level: A table-level constraint references one or more co is defined separately from the definitions of the columns. Nor written after all columns are defined. All constraints can be defi table level except for the NOT NULL constraint. The general syntax is

[CONSTRAINT constraint_name] constraint_type(Column,...)

The PRIMARY KEY Constraint: The PRIMARY KEY Constraint known as the entity integrity constraint. It creates a primary key for the table can have only one primary key constraint. A column or combination columns used as a primary key cannot have a null value, and it can only ha values. For example, the DEPT table in the N2 Corporation database used t] column as a primary key. At the column level, the constraint is defined by

DeptId NUMBER(2) CONSTRAINT dept_deptid_pk PRIMARY KEY,

At the table level; the constraint is defined by

CONSTRIANT dept_deptid_pk PRIMARY KEY(DeptId),

If a table uses more than one column as its primary key (i.e., a composite key), the key can only be declared at the table level. For example, the DEPENDENT table in the N2 database uses two columns for the composite primary key:

CONSTRAINT dependent_emp_dep_pk PRIMARY KEY(EmployeeId, DependentId)

The FOREIGN KEY Constraint : The FOREIGN KEY constraint known as the referential integrity constraint. It uses a column or columns as key, and it establishes a relationship with the primary key of the same 01 table. For example, FacultyId in the STUDENT table in the IU College references the primary key FacultyId in the FACULTY table. The STUDENT is known as the dependent or child table, and the FACULTY table is known as referenced or parent table.

To establish a foreign key in a table, the other referenced table and its key must already exist. Foreign key and referenced primary key columns l have the same name, but a foreign key value must match the value in the table's primary key value or be NULL. For example, the foreign key FacultyId cannot have value 999 in the STUDENT table, because it does not exist in the FA (parent) table's primary key FacultyId.

Oracle does not keep pointers for relationships, but they are based on constraints and data values within those columns. The relationship is purely logical and is not physical in Oracle. At the table level (in the STUDENT table),

CONSTRAINT student_facultyid_fk FOREIGN KEY(FacultyId)

Before ending a FOREIGN KEY constraint, ON DELETE CASCA be added to allow deletion of a record/row in the parent table and deletion dependent rows/records in the child table. Without the ON DELETE CASCADE clause, the row/record in the parent table cannot be deleted if the child table references it. For example, the row for Facultyld 111 cannot be deleted from the FACULTY table, because it is referenced by a row in the STUDENT table.

The NOT NULL Constraint. The NOT NULL constraint ensures that the column has a value and the value is not a null (unknown or blank) value. A space or a numeric zero is not a null value. There is no need to use the not null constraint for the primary key column, because it automatically gets the not null constraint. The foreign key is permitted to have null values, but a foreign key is sometimes given the not null constraint. This constraint cannot be entered at the table level. For example, the name column in FACULTY table is not a key column, but you don't want to leave it blank. At the column level, the constraint is defined by:

Name VARCHAR2(15) CONSTRAINT faculty_name_nn NOT NULL,

Or

Name VARCHAR2(15) NOT NULL,

In the second example, the user does not supply the constraint name, so Oracle will name it with SYS_Cn format.

The UNIQUE Constraint. The UNIQUE constraint requires that every value in a column or set of columns be unique. If it is applied to a single column, the column has unique values only. If it is applied to a set of columns, the group of columns has a unique value together. The unique constraint allows null values unless NOT NULL is also applied to the column. For example, the DeptName column in the DEPT table should not have duplicate values. At the table level, the constraint is defined by:

CONSTRAINT dept_dname_uk UNIQUE(DeptName),

At the column level, the constraint is defined by

DeptName VARCHAR2(12) CONSTRAINT dept_name_uk UNIQUE,

The composite unique key constraint can be defined only at the table level by specifying column names separated by a comma within parentheses. Oracle implicitly creates an index on the unique column to enforce the UNIQUE constraint.

The CHECK Constraint: The CHECK constraint defines a condition that every row must satisfy. There can be more than one CHECK constraint on a column, and the CHECK constraint can be defined at the column as well as the table level. At the column level, the constraint is defined by

DeptId NUMBER(2) CONSTRAINT dept_deptid_cc CHECK((DeptId>=10) and (DeptId <=99))

At the table level, the constraint is defined by

CONSTRAINT dept_deptid_cc CHECK((Deptld >= 10) and (Deptld <= 99»,

The NOT NULL CHECK Constraint. A NOT NULL constraint can clared as a CHECK constraint. Then, it can be defined at column or table level. For example,

Name VARCHAR2(15) CONSTRAINT faculty_name_ck CHECK(Name IS NOT NULL),

The DEFAULT Value (It's Not a Constraint). The DEFAULT val sures that a particular column will always have a value when a new row is in:

The default value gets overwritten if a user enters another value. The default value is used if a null value is inserted. For example, if most of the students live in New Jersey, "NJ" can be used as a default value for the State column in the STUDENT table. At the column level, the value is defined by:

State CHAR(2) DEFAULT 'NJ',

Check your progress 1:

What are the various data types available in Oracle?

.....

.....

.....

.....

.....

.....

9.5 CREATING ORACLE TABLE

A user creates an Oracle table in the SQL*Plus environment. The Oracle Client application is used to execute DDL statements. An Oracle table is created from the SQL> prompt in the SQL*'Plus environment. A Data Definition Language (DDL) SQL statement, CREATE TABLE, is used for table create table is created as soon as the CREATE statement is successfully executed Oracle server. The general syntax of CREATE TABLE statement is:

CREATE TABLE [schema.] tablename

(column 1 datatype [CONSTRAINT constraincname] constrainCtype ... ,

column2 datatype [CONSTRAINT constrainCname] constrainctype,

[CONSTRAINT constraint_name] constrainCtype (column, ...), ...);

In the Syntax,

Schema is optional, and it is same as the user's login name.

Tablename is the name of the table given by the user.

Column is the name of a single column.

Datatype is the column's data type and size.

Constraint_name is the name of constraint provided by the user

Constraint_type is the integrity or value constraint.

Each column may have zero, one, or more constraints defined at the column The table level constraints are normally declared after all column definitions.

SQL is not case sensitive. In this textbook, the reserved words are writ capitalized letters and user-defined names in lower or mixed-case letters. The spaces, tabs and carriage returns are ignored. Let us create the LOCATION table in the IU College database using the CREATE TABLE statement. When the statement is executed and there are no syntax errors, a "Table Created" message will be displayed on the screen.

```
SQL> CREATE TABLE location
2       (Roomld NUMBER(2),
3       Building VARCHAR2(7) CONSTRAINT location_building_nn NOT
NULL,
4       RoomNo CHAR(3) CONSTRAINT location_roomno_nn NOT NULL,
5       Capacity NUMBER(2)
6       CONSTRAINT location_capacity_ck CHECK(Capacity>O),
7       RoomType CHAR,
8       CONSTRAINT location_roomid_pk PRIMARY KEY{Roomld),
9       CONSTRAINT location_roomno_uk UNIQUE{RoomNo);

Table created.

SQL>
```

CREATE TABLE Statement

If there are errors in the CREATE TABLE statement, the statement does not return the "Table Created" message when executed. Oracle displays an error message instead. The error messages are not very user friendly.

```
SQL>   CREATE TABLE emplevel (LevelNo NUMBER(1),
          2 LowSalaray  Number(6),
         3 HighSalary     Number(6)-
           4  CONSRAINT empleveUevelno_pk PRIMARY KEY{LeveINo));
CREATE TABLE emplevel (LevelNo NUMBER(1),
ERROR at line 1:
ORA-00922: missing or invalid option SQL> 3
```

```
3*          HighSalary      Number(6)
SQL> A,
3*          HighSalary      Number(6),
SQL> /
CONSRAINT   emplevel_levelno_pk PRIMARY KEY(LeveINo))
ERROR at line 4:
ORA-00907: missing right parenthesis
SQL> C/CONSRAINT ICONSTRAINT /
4* CONSTRAINTemplevel_levelno_pk PRIMARY KEY(LeveINo))
SQL> /

Table Created.

SQL>
```

CREATE TABLE Statement with error.

In the statement shown, the column definition in line 3 is missing a comma-but the error message does not really tell us that! The debugging of the statement using SQL*Plus commands can be done by the following steps.

1. Go to line 3 (is displayed next to the current line number).
2. Replace the character) in line 3 with), or append a comma (,) to the line.
3. Execute the debugged statement using a slash (/).

As you see above, the statement has another error, this time in line 4. We use C/CONSRAINT/CONSTRAINT to change the incorrect spelling. Then, we execute the statement from buffer by entering a slash (/) again. The table is created. We can edit erroneous statement with the help of an alternate editor as Notepad. To load an erroneous statement in Notepad and modify it, we perform the following steps:

1. At the SQL> prompt, we type ED (or EDIT) to invoke Notepad.
2. We make required corrections to the script.
3. We save our statement on the disk using the Save option from the FILE menu in Notepad, and we name our statement A:\CREATE. Notepad adds the extension .txt to the filename. To suppress Notepad's dl .txt extension, type the file name in a pair of double quotes, and use extension .sql (e.g., "A:\CREATE.SQL").

4. We exit Notepad to go back to the SQL*Plus environment.

5. We can run the saved statement with @ or the RUN command.

The "Table Created" message is displayed when the statement is error-fn this point, the table is created, and its structure is saved. We created the LC TION and EMPLEVEL tables with PRIMARY KEY, UNIQUE, CHECK NOT NULL constraints. Once a table is created, more constraints can be added, more columns can be added, and existing columns' properties can be changed. We did not define any foreign key constraints with the LOCATION table, because table referenced by the foreign key must already exist!

STORAGE CLAUSE IN CREATE TABLE

A CREATE TABLE statement may have an optional STORAGE clause. This clause is used to allocate initial disk space for the table at the time of creation the INITIAL parameter and also to allocate additional space with the NEXT parameter in case the table runs out of allocated initial space. For example,

CREATE TABLE sample (ld NUMBER(3). Name VARCHAR2(25)) TABLESPACE CIS_DATA

STORAGE (INITIAL 1M NEXT 100K)

PCTFREE 20;

In the previous example, the TABLESPACE clause is used to specify the user's tablespace name. If it is not specified, Oracle uses the default permanent tablespace anyway. The STORAGE clause allocates 1 megabyte initially on tablespace CIS_DATA, and 100 kilobytes as additional space on the same tablespace. The INITIAL and NEXT parameters use values in K (kilobytes) or M (megabytes). The PCTFREE (percentage- free) clause is used to allow for future increment in row size. Oracle recommends the following formula in deciding initial extent size for a table:

AVG_ROW_LEN .Number of rows.(1+0.15).(1+PCTFREE/100)

The AVG_ROW_LEN is a column in USER_TABLES Data Dictionary table. The 0.15 (or 15%) is recommended for overhead.

Check your progress 2:

Using SQL Statements create a tables for STUDENT,FACULTY, COURSE, and REGISTRATION. Set necessary constraints (Primary Key, Foreign Key, Null)

.....
.....
.....
.....
.....

9.6 LET US SUM UP

Check your progress : Answers

1. Varchar2, Char, Number, Date, Long, Nchar, BLOB, CLOB, NCLOB, BFILE, RAW and ROWID.

2. Identify the fields for each tables.

   Identify the primary keys of each table.

   Relate the tables using the foreign key.

   Create the tables using the identified primary and foreign keys.

LESSON 10

Oracle Tables

Contents

10.1 DISPLAYING TABLE INFORMATION

When a user creates a table or many tables in his or her database, Oracle tracks them all using its own Data Dictionary. Oracle has SQL statements and SQL*Plus commands for the user to view that information from Oracle's Data Dictionary tables.

Viewing a User's Table Names

A user types an SQL statement to retrieve his or her table names. Often, you use it to review information, and often, you want to find out what has already been created and what is to be created. To find out all tables owned by you, type the following statement:

```
SELECT TABLE_NAME FROM USER_TABLES;
```

Oracle creates system tables to store information about users and user objects.

USER_TABLES is an Oracle system database table, and TABLE_NAME is one of its columns. The display will include all table names you have created and any other tables that belong to you. If you change USER_TABLES with ALL_TABLES, you can get listing of all tables you own as well as those you are granted privileges to by other users. The USER_ TABLES table has many other columns. To display all columns, type the following statement:

```
SELECT * FROM USER_TABLES;
```

You can get information about the STORAGE clauses' attributes by using the Data Dictionary view USER_SEGMENTS:

SELECT Segment_Name, Bytes, Blocks, Initial_Extent, NexCExtent FROM USER_SEGMENTS;

Viewing a Table's Structure

We can display the structure entered in a CREATE TABLE statement. If you have made any changes to the table's structure, the changes will also show in the structure's display. The command is DESCRIBE (or DESC), which does not need a semicolon at the end because it is not a SQL statement. Notice that the default display of column names, the NOT NULL constraint, and data type are in uppercase. You did not add a NOT NULL constraint for the primary key, but by default, Oracle adds it for all primary key columns. The SQL*Plus command to view a table's structure and the result is shown below.

```
SQL> DESCRIBE student Name

STUDENTID          NOT NULL          CHAR (5)

LAST          NOT NULL          VARCHAR2 (15)

FIRST              NOT NULL          VARCHAR2 (15)

STREET                               VARCHAR2 (25)

CITY                                 VARCHAR2( 15)

STATE                                CHAR (2)

 ZIP                                 CHAR (5)


STARITERM                            CHAR (4)

BIRTHDATE                            DATE

FACULTYID                            NUMBER (3)

MAJORID                              NUMBER (3)

PHONE                                CHAR (10)

SQL>
```

Viewing Constraint Information

Oracle's Data Dictionary table USER_CONSTRAINTS stores information about constraints you have entered for each column. When you type the statement, the table name must be typed in uppercase, because Oracle saves table names in uppercase. If you type the table name in lowercase, no constraint names will be displayed.

The constraints named by the user have more meaningful names than the ones named by Oracle. Constraints like NOT NULL are usually not named by the user. Oracle names them using the SYS_Cn format, where n is any number. Constraint type C is displayed for NOT NULL and CHECK constraints. Constraint type P is for primary key and type R for foreign key constraints.

The statement and the result, which include the constraint's name and type is given below.

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE
  2    FROM USER_CONSTRAINTS
3       WHERE TABLE_NAME = 'STUDENT';

CONSTRAINT_NAME                    C
------------------------------------------------------------
STUDENT_FIRST _NN                  C
STUDENT_STUDENTID_PK               P
STUDENT _FACULTYID_FK              R
STUDENT _MAJORID _FK               R
STUDENT _STARTIERMJK         R
STUDENT_LAST _NN                   C

6 Rows selected

SQL>
```

The ORDER BY clause is added to sort the constraint display by table name. For example,

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME
FROM USER_CONSTRAINTS
ORDER BY TABLE_NAME;
```

Another Data Dictionary table, USER_CONS_COLUMNS, stores columnrelated information about constraints.

To see the constraint names and associated column names, the following statement can be executed in SQL>.

```
SQL> COLUMN COLUMN_NAME FORMAT A15
SQL> SELECT CONSTRAINT_NAME, COLUMN_NAME
   2   FROM USER_CONS_COLUMNS
   3   WHERE TABLE_NAME = 'STUDENT';
```

```
CONSTRAINT_NAME              COLUMN_NAME
-------------------------------------------------------------------------
STUDENTJACULTYID_FK          FACULTYID
STUDENT_FIRST _NN                 FIRST
STIDENT _LAST _NN                 LAST
STUDENT _MAJORID _FK              MAJORID
STUDENT _STARTIERM_FK             STARTIERM
STUDENT _STUDENTID_PK             STUDENTID
6 rows selected.
SQL>
```

In this statement, only the table name within single quotes needs to be in uppercase, because Oracle stores table names in that case.

Viewing Tablespace Information

A tablespace consists of one or more physical data files. You can get information about all tablespaces available to you by using Data Dictionary view USER_TABLESPACES. You can use the DESCRIBE command and the SELECT statement with a view the same way you use with tables. For example,

DESCRIBE USER_TABLESPACES

SELECT * FROM USER_TABLESPACES;

Similarly, another Data Dictionary view, USER_USERS, gives user information about his or her account as well as permanent and temporary tablespaces. For example,

SELECT * FROM USER_USERS;

COMMENT on Tables and Columns

When you create a table, you can add comments to the table and its columns. You can do it for documentation purpose with a COMMENT statement. For example.

COMMENT ON TABLE student IS 'Table holds students for INDO-US College'

COMMENT ON COLUMN employee.Lname IS 'Employee"s last name'

You can view information about all comments on tables and columns by using Data Dictionary 'Views ALL_TAB_COMMENTS and ALL_COL_COMMENTS, respectively.

10.2 ALTERING AN EXISTING TABLE

In a perfect scenario, the table you create will not need any structural modifications. You must try to plan and design a database that is close to perfect in all respects. In reality, however, this is not the case. Even perfect

tables need changes. There are certain modifications that you can make to a table's structure. There are other modifications that you cannot make to an existing table's structure.

Modifications allowed without any restrictions include:

- Adding a new column to the table.
- Deleting a foreign key constraint from a table.
- Deleting a primary key constraint from a table, which also removes any references to it from other tables in the database with CASCADE clause.
- Increasing the size of a column. For example, VARCHAR2(IS) can be changed to VARCHAR2(20).
- Renaming columns (Oracle9i onward).
- Renaming constraints (Oracle9i onward).

Modifications allowed with restrictions include:

- Adding a foreign key constraint is allowed only if the current values are null or exist in the referenced table's primary key.
- Adding a primary key constraint is allowed if the current values are not null and are unique.
- Changing a column's data type and size is allowed only if there is no data in it (Oracle8i and earlier). In Oracle9i, column size may be decreased if existing data can be stored with the new column width.
- Adding a unique constraint is possible if the current data values are unique.
- Adding a check constraint is possible if the current data values comply with the new constraint.
- Adding a default value is possible if there is no data in the column.

Modifications not allowed include:

- Changing a column's name (Oracle8i and earlier).
- Changing a constraint's name (Oracle8i and earlier).
- Removing a column (Oracle8 and earlier).

In Oracle8i onward, you are allowed to remove/drop a column from a table or set it as unused. If you already have created a table and need to make a change that is not allowed, you may DROP the table and recreate it. A table can be created using another table with the use of a nested query.

Adding a New Column to an Existing Table

The general syntax to add a column to an existing table is

ALTER TABLE tablename

ADD columnname datatype;

For example, if the IU College decides to track a student's Social Security number along with the student's ID, a new column can be added to the STUDENT table, as shown in the query below. If the table already contained rows of data, you will have to use UPDATE statement for each row to add values in the newly added column.

```
SQL> ALTER TABLE student
2                ADD SocialSecurity CHAR(9);


  Table altered.


  SQL>
```

Modifying an Existing Column

The general syntax to modify an existing column is

ALTER TABLE tablename

MODIFY columnname newdatatype;

where newdatatype is the new data type or the new size for the column. For example, say the IU College wants to allow data-entry personnel to enter values with or without dashes in the Social Security column. The data type can be changed from CHAR(9) to VARCHAR2(1l) to accommodate this new format given below.

```
SQL> ALTER TABLE student
          2  MODIFY Social Security VARCHAR2(11);
```

Table altered.

SQL>

Adding a Constraint

In this section, we will try to add various constraints in a table using the ALTER TABLE statement. The EMPLOYEE table in the N2 corporation database has a PositionId column, which references the POSITION table's primary key PositionId. To add a constraint using ALTER TABLE, the syntax for table level constraint is used. The general syntax of ALTER TABLE is

ALTER TABLE tablename

ADD [CONSTRAINT constrainCnameJ constrainCtype (column, ...),

For example,

```
ALTER TABLE employee
ADD CONSTRAINT employee_positionid_fk FOREIGN KEY
    (Positionld) REFERENCES position (Positionld);
```

The below ALTER TABLE statement adds a new constraint to table COURSE. The foreign key column PreReq references primary key column CourseId of its own table. Such a reference is known as a circular reference.

```
SQL> ALTER TABLE course
  2 ADD CONSTRAINT COURSE_PREREQJK FOREIGN KEY(PREREQ)
  3  REFERENCES COURSE(COURSE_ID);
Table Altered.
SQL>
```

The TERM table in the IU College database contains two columns, StartDate and EndDate. The start date for a term must fall before the end date for the same term. Use of a CHECK constraint will guarantee the necessary data integrity. The problem is that during creation of the TERM table, defining a constraint that compares values in two columns of the same table is not possible. The constraint can be defined with the ALTER TABLE statement, however, as shown below.

```
SQL> ALTER TABLE term
2  ADD CONSTRAINT term_startdate_ck
3  CHECK(StartDate < EndDate);
Table Altered.
SQL>
```

Let us try to add another foreign key constraint in the STUDENT table as shown below. To create a foreign key constraint, the parent table, whose primary key column is referenced by the child table's foreign key column, must already exist in the database. Even the primary key column that is referenced must exist in the parent table defined as the primary key. Remember that the two columns, the foreign key and the primary key that it references, need not have the same name. The best solution in this situation would be to create all tables without any foreign key constraints first, then create tables using a CREATE TABLE statement with FOREIGN KEY constraints to the reference tables already created. An alternate solution is to create all tables with their constraints except for the foreign key constraint. Once all the tables are created, use the ALTER TABLE statement to add the FOREIGN KEY constraint.

```
SQL> ALTER TABLE student
  2 ADD CONSTRAINT studenCfacultyid_fk
  3   FOREIGN KEY (Facultyld) REFERENCES faculty(Facultyld);
```

REFERENCES faculty(Facultyld)

*ERROR at one 3:

ORA·00942: table or view does not exist

SQL>

In the query given below, we try to create a foreign key has failed. The problem with the query is the creation of a foreign key in the wrong table. The FacultyId column is common in the STUDENT and FACULTY tables, but remember the rule! A foreign key must reference a primary key. In our query, primary key Facultyld in the FACULTY table is trying to reference a non-key column Facultyld in the STUDENT table. It will definitely won't work in Oracle!

SQL> ALTER TABLE faculty

2 ADD CONSTRAINT faculty_facultyid_fk FOREIGN KEY(Facultyld)

3 REFERENCES student(Facultyld);

REFERENCES student(Facultyld);

ERROR at line 3:

ORA-00270; no matching unique or key for this column-list

SQL>

Once the parent table FACULTY is created, the foreign key is successfully created in the STUDENT table given below.

SQL> ALTER TABLE student

2 ADD CONSTRAINT student_facultyid_fk FOREIGN KEY(Facultyld)

3 REFERENCES Faculty(FacultyID);

Table altered.

SQL>

Now, let us add a NOT NULL constraint and a DEFAULT value to the StartTerm and State columns, respectively, in the STUDENT table. If a student does not have a start term, it is difficult for an academic department to track the student's class and projected date of graduation. If the college is located in the New Jersey area and most of the students are from in-state, it is a good idea to add a default value to minimize having to enter data. A user can always overwrite the default value, but if it is left blank or null, the default value is used by Oracle. To add such constraints, a MODIFY clause is used with an ALTER TABLE statement. For example,

ALTER TABLE student MODIFY StartTerm CHAR(4)

CONSTRAINT student_startterm_nn NOT NULL;

ALTER TABLE student MODIFY State CHAR(2) DEFAULT 'NJ';

Dropping a Column

Oracle8 and earlier versions do not allow you to remove a column from a table, but with Oracle8i onward, you can. Even so, only one column can be dropped at a time. The column mayor may not contain any data. When you drop a column, there must be at least one column left in the table. In other words, you can't remove the last remaining column from a table. It is not possible to recover a dropped column and its data. The general syntax is

ALTER TABLE tablename DROP COLUMN columnname;

Oracle 9i also allows a user to mark columns as unused by using

ALTER TABLE tablename SET UNUSED (columnname);

The unused columns are like dropped columns. This is not a very good feature, because the storage space used by unused columns is not released. They are not displayed with other columns or in the table's structure, and the user can drop all unused columns with the following statement. Setting a column to unused is quicker than dropping a column, however, and it requires fewer system resources. You can remove all unused columns when system resources are in less demand. The general syntax is

ALTER TABLE tablename DROP UNUSED COLUMNS

If no columns are marked as unused, this statement does not return any error messages. The statement given below shows setting a column as unused and then being dropped.

SQL> ALTER TABLE student

2 SET UNUSED(SociaISecurity);

Table altered.

SQL> ALTER TABLE student

2 DROP UNUSED COLUMNS;

Table altered.

SQL>

Dropping a Constraint

We can view constraint information from the USER_CONSTRAINTS table or the USER_CONS_COLUMNS table. A dropped constraint is no longer enforced by Oracle, and it does not show up in the list of USER_CONSTRAINTS or USER_CONS_COLUMNS. The general syntax is

ALTER TABLE tablename

DROP PRIMARY KEYlUNIQUE (columnname) | CONSTRAINT constraintname [CASCADE];

For example,

ALTER TABLE major

DROP PRIMARY KEY CASCADE;

This statement drops the primary key constraint from the MAJOR table. The CASCADE clause drops the dependent foreign key constraints, if any. You can drop a constraint by using its name, which is why it is important to name all constraints with a standard naming convention. For example,

ALTER TABLE employee

DROP CONSTRAINT employee_deptid_tk;

Enabling/ Disabling Constraints

The constraints may be enabled or disabled as needed. A newly created constraint is enabled automatically. A constraint verifies table data as they are added or updated. This verification slows down the process, so you may want to disable a constraint if you are going to add or update large volume of data. When you reenable the constraint, Oracle checks the validity of the data and for any violations.

You may disable multiple constraints with one ALTER TABLE statement, but you may only enable one constraint at a time. The general syntax for enabling or disabling constraint is

ALTER TABLE tablename

ENABLE I DISABLE CONSTRAINT constraintname;

You may enable or disable a primary key constraint with the following syntax that does not use constraint name:

ALTER TABLE tablename ENABLE | DISABLE PRIMARY KEY;

There is no CASCADE clause with ENABLE. The DISABLE and ENABLE clauses can also lie used in a CREATE TABLE statement.

Renaming a Column

We can rename a constraint with the following statement:

ALTER TABLE tablename RENAME CONSTRAINT oldname TO newname;

Modifying Storage of a Table

We can change storage attributes of a table, such as NEXT, PCTFREE, and so, with the following statement:

ALTER TABLE tablename STORAGE (NEXT nK);

10.3 DROPPING, RENAMING, TRUNCATING TABLE

Dropping a Table

When a table is not needed in the database, it can be dropped. Sometimes, the existing table structure has so many flaws it is advisable to drop it and recreate it. When a table is dropped, all data and the table structure are permanently deleted. The DROP operation cannot be reversed, and Oracle does not ask "Are You Sure?" You can drop a table only if you are the owner of the table or have the rights to do so. Many other objects based on the dropped table are affected. All associated indexes are removed. The table's views and synonyms become invalid. The general syntax is

DROP TABLE tablename [CASCADE CONSTRAINTS]:

Oracle displays a "Table dropped" message when a tabk is successfully dropped. If you add the optional CASCADE CONSTRAINTS clause, it removes foreign key references to the table as well.

Renaming a Table

Renaming of a table can be done only by the owner of the table. The general syntax is

RENAME oldtablename TO newtablename;

For example,

RENAME dept TO department;

Oracle will display a "Table renamed" message when this statement is executed. (We will not change the DEPT table's name and will still refer to it by its original name later in this textbook.) The RENAME statement can be used to change name of other Oracle objects, such as a view, synonym, or sequence.

Truncating a Table

Truncating a table is removing all records/rows from the table. The structure of the table, however, stays intact. You must be the owner of the table with the DELETE TABLE privilege to truncate a table. The SQL language has a DELETE statement that can be used to remove one or more (or all) rows from a table, and it is reversible as long as it is not committed. The TRUNCATE statement, on the other hand, is not reversible. Truncation releases storage space occupied by the table, but deletion does not. The syntax is

TRUNCATE TABLE tablename;

Oracle displays a "Table truncated" message on this statement's execution. The EMPLOYEE table is an integral part of the N2 Corporation's database, and you do not want to truncate it unless you would like to enter all the employees' data again!

The truncate operation releases all table storage except for the initially allocated extent. You can "keep" all storage used by table with the REUSE STORAGE clause. For example,

TRUNCATE TABLE tablename REUSE STORAGE;

Check your progress 1:

How do you view the constraint information of a table?

……

……

……

……

10.4 TABLE TYPES

Oracle9i uses various types of tables-permanent tables, temporary tables, indexorganized tables, and external tables. Permanent tables are used for storing data. Temporary tables are used during a session or a transaction. Temporary tables are like permanent tables, but they are not stored permanently. They store data during a session or a transaction. Temporary tables are created with the CREATE GLOBAL TEMPORARY TABLE statement. Index-organized tables are used for tables with primary key values that are looked up frequently.

Index-organized tables are created with a CREATE TABLE tablename ... ORGANIZATION INDEX statement. External tables are stored "outside" the database with CREATE TABLE tablename ... ORGANIZATION EXTERNAL statement. These tables are based on flat files, such as conima-delimited, double quotes-delimited or fixed-length files, whose directory path is made known to Oracle with a CREATE DIRECTORY statement. In most cases, the end user works with permanent data tables only.

10.5 SPOOLING

Spooling is a very handy feature. During a session, a user can redirect all statements, queries, commands, and results to a file for later review or printout. The spooling method creates a text file of all actions and their results. Everything you see on your screen is redirected to the file, which is saved with an .lst extension by default.

To start spooling, go to the File menu in the SQL*Plus window. Then, click on Spool and Spool File in subsequent menus You will be prompted to enter a file name, which will be created with an .lst extension.

To stop spooling at any point, use the same menu to click on Spool Off. When spooling is turned off, the file is saved to the disk and closed. The spooled file can be opened in any text editor, such as Notepad, for viewing or printing. In the classroom environment, I ask my students to spool all their work, which includes required queries and their results. The students can submit their disk or the printed hard copy.

10.6 ERROR CODES

If Oracle Error Help is installed on your system. you will be able to get to it by clicking on START -> Oracle -> OraHome92. Once the error help screen is displayed, click on the Index tab. Then, type the error code received from Oracle in the space provided. When you are done typing, click on the Display button to get an explanation of the error. The explanation of the error code is straightforward. The help function shows the cause of the error and gives hints for corrective actions.

You may use online help from Oracle's Web site by using the following URL:

http://otn.oracle.com/pls/db92/db92.error_search

To use the online help with error codes. follow three steps shown in Figures given below.

In step 1, type the error code in the text box

In step 2, select a result from Oracle's search results.

In step 3, view the cause of the error and the action required to rectify it.

This online help utility requires you to sign up with The Oracle Technology Network (OTN). The free membership to OTN has many benefits, including free downloads of Oracle software products.

10.7 LET US SUM UP

Check your progress Answers :

1.SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM
USER_CONSTRAINTS WHERE TABLE_NAME= table;

UNIT III

LESSON 11

WORKING WITH TABLE : DATA MANAGEMENT AND RETRIEVAL

Contents

11.0 AIMS AND OBJECTIVES

To learn the DML statements of Oracle 9i.

11.1 DML

DML changes data in an object. If you insert a row into a table, that is DML. You have manipulated the data. When you create, change or remove a database object, it is referred to as data definition language (DDL). All DDL statements issue an implicit commit, so they are a permanent change. All DML statements change data and must be committed before the change becomes permanent.

Each parallel execution server creates a different parallel process transaction. As a result, parallel DML requires more than one rollback segment for performance.

However, there are some restrictions as shown below:

A transaction can contain multiple parallel Oracle DML statements that modify different tables, but after parallel DML statements modify a table, no subsequent serial or parallel statement (DML or query) can access the same table again in that transaction.

Parallel DML operations cannot be done on tables with triggers. Relevant triggers must be disabled in order to parallel DML on the table.

A transaction involved in a parallel DML operation cannot be or become a distributed transaction. Clustered tables are not supported.

old_values - These are the old column values related to the change. These are the column values for the row before the DML change. If the type of the DML statement is UPDATE or DELETE, these old values include some or all of the columns in the changed row before the DML statement. If the type of the DML statement is INSERT, there are no old values.

11.2 Adding a New Row / Record

Basic Inserts	
Single Column Table Or View	INSERT INTO <table_name> (<column_name>) VALUES (<value>);
	CREATE TABLE state (state_abbrev VARCHAR2(2)); INSERT INTO state (state_abbrev) VALUES ('WA'); COMMIT; SELECT * FROM state;
Multiple Column Table Or View - All Columns	INSERT INTO <table_name> VALUES (<comma_separated_value_list>);
	ALTER TABLE state ADD (state_name VARCHAR2(30)); INSERT INTO state (state_abbrev, state_name) VALUES ('OR', 'Oregon'); COMMIT; SELECT * FROM state;
Multiple Column Table Or View - Not All Columns	INSERT INTO <table_name> (<comma_separated_column_name_list>) VALUES (<comma_separated_value_list>);
	RENAME state TO state_city; ALTER TABLE state_city ADD (city_name VARCHAR2(30));

	INSERT INTO state_city (state_abbrev, city_name) VALUES ('CA', 'San Francisco'); COMMIT; SELECT * FROM state_city;
Problem Not Specifying Column Names Demo	INSERT INTO <table_name> ~~(<comma_separated_column_name_list>)~~ VALUES (<comma_separated_value_list>);
	desc state_city INSERT INTO state_city VALUES ('NV', 'Nevada', 'Las Vegas'); desc state_city
INSERT SELECT	
Insert From SELECT statement	INSERT INTO <table_name> <SELECT Statement>;
	CREATE TABLE zip_new (zip_code VARCHAR2(5) NOT NULL, state_abbrev VARCHAR2(2) NOT NULL, city_name VARCHAR2(30)); INSERT INTO zip_new SELECT zip_code, state_abbrev, city_name FROM postal_code; SELECT * FROM zip_new;
RECORD INSERT	
	INSERT INTO <table_name>

Insert Using A Record	VALUES <record_name>;

	CREATE TABLE t AS
	SELECT table_name, tablespace_name
	FROM all_tables;
	
	SELECT COUNT(*)
	FROM t;
	
	DECLARE
	trec t%ROWTYPE;
	BEGIN
	trec.table_name := 'NEW';
	trec.tablespace_name := 'NEW_TBSP';
	
	INSERT INTO t
	VALUES trec;
	
	COMMIT;
	END;
	/
	
	SELECT COUNT(*) FROM t;

11.3 Updating and Deleting an Existing Row / Record

Basic Update Statements	
Update all records	UPDATE <table_name> SET <column_name> = <value>
	CREATE TABLE test AS SELECT object_name, object_type FROM all_objs; SELECT DISTINCT object_name FROM test; UPDATE test SET object_name = 'OOPS'; SELECT DISTINCT object_name FROM test; ROLLBACK;

Update a specific record	UPDATE <table_name> SET <column_name> = <value> WHERE <column_name> = <value>
	SELECT DISTINCT object_name FROM test; UPDATE test SET object_name = 'LOAD' WHERE object_name = 'DUAL'; COMMIT; SELECT DISTINCT object_name FROM test;

Basic Delete Statements	
Delete All Rows	DELETE <table_name> or DELETE FROM <table_name>;
	CREATE TABLE t AS SELECT * FROM all_tables; SELECT COUNT(*) FROM t; DELETE FROM t; COMMIT; SELECT COUNT(*) FROM t;
Delete Selective Rows	DELETE FROM <table_name> WHERE <condition>;

	CREATE TABLE t AS SELECT * FROM all_tables; SELECT COUNT(*) FROM t; DELETE FROM t WHERE table_name LIKE '%MAP'; COMMIT; SELECT COUNT(*) FROM t;

Delete From A SELECT Statement	DELETE FROM (<SELECT Statement>);
	CREATE TABLE t AS SELECT * FROM all_tables; SELECT COUNT(*) FROM t; DELETE FROM (SELECT * FROM t WHERE table_name LIKE '%MAP'); SELECT COUNT(*) FROM t;

Check your progress 1:

What is the use of DML Statements?

.....
.....
.....

11.4 Let Us Sum Up

Check Your Progress: Model Answers

1. The DML Statements are used for the Database Manipulation. All DML statements change data and must be committed before the change becomes permanent.

LESSON 12

WORKING WITH TABLE: DATA RETRIEVAL

Contents

12.0 AIMS AND OBJECTIVES

To learn how to retrieve data using the Select Statement and how to restrict the selection using the where clause.

12.1 Retrieving Data from Table - SELECT STATEMENT

The SELECT statement of DML is used to retrieve the records.

Basic Select Statements	
Select All Columns and All Records in a Single Table or View	SELECT * FROM <table_name>;
	SELECT * FROM all_tables;
Select Named Columns	SELECT <column_name, column_name, ..., <column_name> FROM <table_name>;
	SELECT table_name, tablespace_name, num_rows FROM all_tables;
Create Table As (CTAS) Note: Redo only created when in ARCHIVE LOG mode	CREATE TABLE <table_name> AS SELECT <column_name, column_name, ..., <column_name> FROM <table_name>;
	CREATE TABLE t AS SELECT *

	FROM all_tables;

Select Statement With SAMPLE Clause	
Sample Clause Returning 1% Of Records	SELECT * FROM <table_name> SAMPLE (1);
	CREATE TABLE t AS SELECT object_name FROM all_objects WHERE SUBSTR(object_name,1,1) BETWEEN 'A' AND 'W'; SELECT COUNT(*) FROM t; SELECT COUNT(*) * 0.1 FROM t; SELECT * FROM t SAMPLE(1); SELECT * FROM t SAMPLE(1); SELECT * FROM t SAMPLE(1);

Select Statement With WHERE Clause	
Sample Clause Returning 35% Of Records After Filtering With A WHERE	SELECT * FROM <table_name> SAMPLE (3.5) WHERE ....

Clause	SELECT COUNT(*) FROM t WHERE object_name LIKE '%J%'; SELECT COUNT(*) * 0.35 FROM t WHERE object_name LIKE '%J%'; SELECT * FROM t SAMPLE(35) WHERE object_name LIKE '%J%'; SELECT * FROM t SAMPLE(35) WHERE object_name LIKE '%J%'; SELECT * FROM t SAMPLE(35) WHERE object_name LIKE '%J%';

12.2 Arithmetic Operations

The arithmetic expressions are used to display mathematically calculated data. These expressions use columns, numeric values, and arithmetic operators.. When arithmetic operators are used with columns in the SELECT query, the underlying data are not changed. The calculations are for output purposes only.

*	Multiplication
/	Division
+	Addition

Subtraction

Whatever is in parentheses is done first.

Multiplication and division have higher precedence than addition and subtraction.

If more than one operator of the same precedence is present, the operators are performed from left to right.

It is optional to leave a space on both Sl arithmetic operator. One other peculiar thing is the total of Salary and COI When a Salary value is added to a null value in the Commission column, t a null value. To handle null values, the expression can be changed to salary (Commission, 0), where NVL is a function that replaces a NULL value with one argument in parentheses-in this case, a zero-for arithmetic operation. Any arithmetic open a null value returns a null value as result.

SQL> SELECT Lname, Fname, Salary+Commission FROM employee;

LNAME	FNAME	SALARY+COMISSION
Smith	John	300000
Houston	Larry	160000
Roberts	Sandi	
McCall	Alex	
Dev	Derek	100000
Shaw	Jinku	27500
Garner	Stanley	50000
Chen	Sunny	

12.3 Restricting Data with Where Clause

The WHERE clause allows you to filter the results from an SQL statement - select, insert, update, or delete statement.

It is difficult to explain the basic syntax for the WHERE clause, so instead, we'll take a look at some examples.

Example #1

```
SELECT *
FROM suppliers
WHERE supplier_name = 'IBM';
```

In this first example, we've used the WHERE clause to filter our results from the suppliers table. The SQL statement above would return all rows from the suppliers table where the supplier_name is IBM. Because the * is used in the select, all fields from the suppliers table would appear in the result set.

Example #2

```
SELECT supplier_id
FROM suppliers
WHERE supplier_name = 'IBM'
or supplier_city = 'Newark';
```

We can define a WHERE clause with multiple conditions. This SQL statement would return all supplier_id values where the supplier_name is IBM or the supplier_city is Newark.

Example #3

```
SELECT suppliers.suppler_name, orders.order_id
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id
and suppliers.supplier_city = 'Atlantic City';
```

We can also use the WHERE clause to join multiple tables together in a single SQL statement. This SQL statement would return all supplier names and order_ids where there is a matching record in the suppliers and orders tables based on supplier_id, and where the supplier_city is Atlantic City.

Check your progress 1:

What is the use of the WHERE clause?

.....
.....
.....
.....
.....

DISTINCT FUNCTION

The DISTINCT function is used to suppress duplicate values. The word DISTINCT is used right after the keyword SELECT and before the column name.

Let us see the difference in result from two SELECT queries, with and without the DISTINCT function. The SELECT statement below without DISTINCT, which outputs 11 rows with some duplicate values. The next SELECT statement with DISTINCT, which eliminates duplicate values and returns three unique values only.

```
SQL> SELECT Building

2 FROM location;

Gandhi

Gandhi

Kennedy

Kennedy

Nehru

Nehru

Gandhi

Kennedy

Kennedy

Gandhi

Gandhi
```

SQL> SELECT DISTINCT Building

2 FROM location;

Gandhi

Kennedy

Nehru

OR Condition in Where Clause

The OR condition allows you to create an SQL statement where records are returned when any one of the conditions are met. It can be used in any valid SQL statement - select, insert, update, or delete.

The syntax for the OR condition is:

```
SELECT columns
FROM tables
WHERE column1 = 'value1'
or column2 = 'value2';
```

The OR condition requires that any of the conditions be must be met for the record to be included in the result set. In this case, column1 has to equal 'value1' OR column2 has to equal 'value2'.

Example #1

The first example that we'll take a look at involves a very simple example using the OR condition.

```
SELECT *
FROM suppliers
WHERE city = 'New York'
or city = 'Newark';
```

This would return all suppliers that reside in either New York or Newark. Because the * is used in the select, all fields from the suppliers table would appear in the result set.

Example #2

The next example takes a look at three conditions. If any of these conditions is met, the record will be included in the result set.

```
SELECT supplier_id
FROM suppliers
WHERE name = 'IBM'
or name = 'Hewlett Packard'
or name = 'Gateway';
```

The AND Condition in Where Clause

The AND condition allows you to create an SQL statement based on 2 or more conditions being met. It can be used in any valid SQL statement - select, insert, update, or delete.

The syntax for the AND condition is:

```
SELECT columns
FROM tables
WHERE column1 = 'value1'
and column2 = 'value2';
```

The AND condition requires that each condition be must be met for the record to be included in the result set. In this case, column1 has to equal 'value1' and column2 has to equal 'value2'.

Example #1

The first example that we'll take a look at involves a very simple example using the AND condition.

```
SELECT *
FROM suppliers
WHERE city = 'New York'
and type = 'PC Manufacturer';
```

This would return all suppliers that reside in New York and are PC Manufacturers. Because the * is used in the select, all fields from the supplier table would appear in the result set.

Example #2

Our next example demonstrates how the AND condition can be used to "join" multiple tables in an SQL statement.

```
SELECT orders.order_id, suppliers.supplier_name
FROM suppliers, orders

WHERE suppliers.supplier_id = orders.supplier_id
and suppliers.supplier_name = 'IBM';
```

This would return all rows where the supplier_name is IBM. And the suppliers and orders tables are joined on supplier_id. You will notice that all of the fields are prefixed with the table names (ie: orders.order_id). This is required to eliminate any ambiguity as to which field is being referenced; as the same field name can exist in both the suppliers and orders tables.

12.4 Let Us Sum Up

Check your progress :Answers

1. The WHERE clause is used to restrict the data retrieval from the database. The filtering of records can be done by combining the conditions with OR and AND logical operators.

LESSON 13

WORKING WITH TABLE: SORTING

Contents

13.0 AIMS AND OBJECTIVES

To understand the concept of sorting, the use of the DEFINE command and the CASE structure.

13.1 SORTING

The ORDER BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements.

The syntax for the ORDER BY clause is:

```
SELECT columns
FROM tables
WHERE predicates
ORDER BY column ASC/DESC;
```

The ORDER BY clause sorts the result set based on the columns specified. If the ASC or DESC value is omitted, it is sorted by ASC.

ASC indicates ascending order. (default)
DESC indicates descending order.

Example #1

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'IBM'
ORDER BY supplier_city;
```

This would return all records sorted by the supplier_city field in ascending order.

Example #2

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'IBM'
ORDER BY supplier_city DESC;
```

This would return all records sorted by the supplier_city field in descending order.

Example #3

You can also sort by relative position in the result set, where the first field in the result set is 1. The next field is 2, and so on.

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'IBM'
ORDER BY 1 DESC;
```

This would return all records sorted by the supplier_city field in descending order, since the supplier_city field is in position #1 in the result set.

Example #4

```
SELECT supplier_city, supplier_state
FROM suppliers
WHERE supplier_name = 'IBM'
ORDER BY supplier_city DESC, supplier_state ASC;
```

This would return all records sorted by the supplier_city field in descending order, with a secondary sort by supplier_state in ascending order

Check your progress 1:

How do you do the sorting in Ascending order?

.....

.....

.....

.....

.....

13.2 Revisiting Substitution Variables

The substitution variables can be used in statements other than the INSERT statement. They can substitute for column names, table names, expressions, or text. Their use is to generalize queries by inserting them as follows:

In the SELECT statement in place of a column name.

In the FROM clause in place of a table name.

In the WHERE clause as a column expression or text.

As an entire SELECT statement.

If a variable is to be reused within a query without getting a prompt again for the same variable, the double-ampersand (&&) substitution variable is

used. The user gets only one prompt for the variable with &&, and the value of the variable is then used more

than one time. In example below, the variable columnname is used twice, once in the SELECT statement in the column list and then again as the sort column in the ORDER BY clause. The user, however, gets only one prompt for the variable.

```
SQL> SELECT Last, First, &&columnname
  2  FROM student
  3  ORDER BY &columnname;
Enter value for columnname: Majorld
old 1: SELECT Last, First, &&columnname
new 1: SELECT Last, First,  Majorld
old 3: ORDER BY &columnname
new 3: ORDER BY Majorld
LAST       FIRST          MAJORID
Diaz       Jose       100
Khan       Amir       200
Patel      Rajesh          400
Tyler      Mickey     500
Rickles    Deborah    500
Lee        Brian      600
6 rows selected. SQL>
```

13.3 DEFINE Command

A Variable can be defined at the SQL> prompt. The variable is assigned a value that is held until the user exits from SQL * Plus or undefines it. The general syntax is

```
DEFINE variable  [=value]
```

For example,

```
DEFINE Last= Shaw
```

The variable last gets the value Shaw, which can be used as a substitution variable in a query. For example,

The DEFINE Last command will return the value of the variable if it already has a value; otherwise, Oracle will display an "UNDEFINED" message.

```
UNDEFINE Last
```

The variable's value can be erased with the UNDEFINE command. For example, the variable is valid during a session only. If you want to use a variable every time you log in, it can be defined in your login script file (login.sql).

```
SQL> DEFINE Temp = 999

SQL> DEFINE

DEFINE _CONNECT_IDENTIFIER = "oracle" (CHAR)

DEFINE _SQLPLUS_RELEASE = "902000100" (CHAR)

DEFINE _EDITOR    = "Notepad" (CHAR)

DEFINE _O_VERSION       = "Oracle9i Enterprise Edition Release 9. 2 . 0 . 1 . 0
- Production

With the Partitioning. OLAP and Oracle Data Mining options JServer Release 9 .
2 . 0 . 1 . 0 - Production" (CHAR)

DEFINE _O_RELEASE       = "902000100" (CHAR)

DEFINE DEPT _10   = "SO" (CHAR)

DEFINE LOCATION         = "Monroe" (CHAR)

DEFINE DEPT_NAME        = "Accounting" (CHAR)

DEFINE MANAGER          = "NULL" (CHAR)

DEFINE COLUMNNAME    = "Majorld" (CHAR)

DEFINE TEMP         = "999" (CHAR)

SQL>
```

The above example shows use of DEFINE command to define variable Temp with a value of 999. The DEFINE command then displays all defined variables for the session, including the just-defined variable Temp. The variables are "to red with data-type CHAR.

13.4 CASE Structure

CASE structure is allowed anywhere expressions are allowed in SQL statements. SQL's CASE structure is similar to the SELECT ... CASE statement in Visual Basic language and the switch ... case statement in C++ and Java. The general syntax of CASE is

```
CASE WHEN condition 1 THEN Expression 1 WHEN condition2 THEN
Expression2
```

In the example below, CASE structure is illustrated with an UPDATE statement. The Salary column is updated with different increments based on employees' department Id. Employees in Department 10 get a 10% raise, employees in Department 20 get a 5% raise, and others do not get any raise.

```
SQL>UPDATE employee
2       SET Salary = CASE WHEN DeptId =  10 then
3                       Salary* 1.10
4               WHEN DeptId = 20 THEN
5                       Salary * 1.05
6               ELSE  Salary
7               END
8       /
8 rows updated.
SQL>
```

13.5 Let Us Sum Up

Check your progress: Answers.

The sorting can be done by giving with the keyword ASC.

LESSON 14

FUNCTIONS AND GROUPING

Contents

14.0 AIMS AND OBJECTIVES

To learn about the different types of Built-in-functions and the purpose of grouping the data.

14.1 Built-in-Functions

The built-in functions provide a powerful tool for the enhancement of a basic query. A function takes zero or more arguments and returns a single value. Just like other software and programming languages, the functions covered in this section are specific to Oracle. Functions are used for performing calculations on data, converting data, modifying individual data, manipulating a group of rows, and formatting columns. In Oracle's SQL, there are two types of functions:

1. Single-row functions, which work on columns from each row and return one result per row.

2. Group functions or aggregate functions, which manipulate data in a group of rows and return single result.

Single-Row Functions

The single-row functions take different types of arguments, work on a data item from each row, and return one value for each row. The arguments are in the form of a constant value, variable name, column, anchor expression. The value returned by a function may be of a different type than the argument(s) supplied. The general syntax is

Function (column I expression 1 argumenti, argument2,. . . D

where function is the name of the function, column is a column from a table, expression is a character string or a mathematical expression, and argument is any argument used by the function.

There are various types of single-row functions:

- Character functions take a character string or character-type column as an argument and return a character or numeric value.

- Number functions take a number or number-type column as an argument and return a numeric value.
- Date functions take a date value or date-type column as an argument and return date-type data. (Exception: The MONTHS_BETWEEN function returns a numeric value.)
- Conversion functions convert value from one data type to another.
- General functions perform different tasks.

Character Functions:

The character functions perform case conversion or character manipulation. The table below shows a list of character functions and their use. The case-conversion character functions change a string or character-type column data's case. For example,

UPPER('Oracle') — 'ORACLE'
LOWER('DaTaBaSe SyStEmS') —' 'database systems'
INITCAP('DaTaBaSe SyStEmS') — 'Database Systems'

For example, the use of character functions UPPER, LOWER, and INITCAP in the SELECT clause to display columns with different cases. Often, more than one data-entry person will populate a table. One person enters names in all uppercase, and the other uses proper case. This could become a nightmare for data retrieval query writers if not for functions. Functions are very useful in the WHERE clause's conditions as well.

For example, in Figure, a query is issued with the condition WHERE State = 'ny', and it resulted in a "no row selected" message. The table does contain students from New York state. The problem here is the case used in entering state values. The same query is rewritten with condition WHERE UPPER(State) = 'NY', and it returned two student names. The use of the UPPER function converted the value in the

Character Function	Use
UPPER(*column* | *expr*)	Converts each letter to uppercase.
LOWER(*column* | *expr*)	Converts each letter to lowercase.
INITCAP(*column* | *expr*)	Converts character value to the proper case (i.e., first character of each word is converted to uppercase and the rest to lowercase).
CONCAT(*column* | *expr, column* | *expr*)	Joins the first value to the second value. Similar to the ‖ operator discussed earlier.
SUBSTR(*column*| *expr, x, y*)	Returns a substring, starting at character position *x*, and returns *y* number of characters.
SUBSTR(*column* | *expr, z*)	Returns a substring, starting at character position *z* and going to the end of string.
INSTR(*column* | *expr, c*)	Returns the position of the supplied character.
LTRIM(*column* | *expr, c*)	Removes the leading supplied character.
RTRIM(*column* | *expr, c*)	Removes the trailing supplied character.
TRIM('*c*' *FROM column* | *expr*)	Removes the leading and trailing characters.
TRIM(*column*)	Removes the leading and trailing spaces only.
LENGTH(*column* | *expr*)	Returns the number of characters.
LPAD(*column* | *expr, n, 'str'*)	Pads the value with 'str' to the left to a total width of *n*.
RPAD(*column* | *expr, n, 'str'*)	Pads the value with 'str' to the right to a total width of *n*.
REPLACE(*column* | *expr, c, r*)	Replaces substring *c*, if present in the column or expression, with string *r*.

```
SQL> SELECT UPPER(Lname),  LOWER(Fname),
  2      INITCAP(Fname  ||  ' '  ||  Lname)
  3  FROM employee;

UPPER(LNAME)    LOWER(FNAME)    INITCAP(FNAME||''||LNAME)
--------------- --------------- ------------------------------
SMITH           john            John Smith
HOUSTON         larry           Larry Houston
ROBERTS         sandi           Sandi Roberts
MCCALL          alex            Alex Mccall
DEV             derek           Derek Dev
SHAW            jinku           Jinku Shaw
GARNER          stanley         Stanley Garner
CHEN            sunny           Sunny Chen

8 rows selected.

SQL>
```

```
SQL> SELECT Last, First FROM student
  2  WHERE State='ny';

no rows selected

SQL> SELECT Last, First FROM student
  2  WHERE UPPER(State) = 'NY';

LAST            FIRST
--------------- ---------------
Tyler           Mickey
Lee             Brian

SQL>
```

lumn to uppercase, and it was then compared to the value NY, which has same case. The condition can be written as WHERE LOWER (State) = 'fly' instead.

A character function is used in various SELECT clauses, including the ORDER BY clause. The LENGTH function returns the length of a character column or string literal. Suppose we want to see names in ascending order by length of names. The clause will use ORDER BYLENGTH(Last) instead of just ORDER BY Last, as shown in Figure below

```
SQL> SELECT Last, First FROM student
  2  ORDER BY LENGTH(Last);

LAST                 FIRST
-------------------- --------------------
Lee                  Brian
Diaz                 Jose
Khan                 Amir
Tyler                Mickey
Patel                Rajesh
Rickles              Deborah

6 rows selected.

SQL>
```

Character function in ORDER BY.

In Oracle9i, the LENGTH function is enhanced with other functions like LENGTHB (to get length in bytes instead of characters) and LENGTHC (to get length in unicode).

The character manipulation functions manipulate a character-type value to return another character- or numeric-type result. For example,

```
CONCAT('New', 'York')        → 'NewYork'
SUBSTR('HEATER', 2, 3)       → 'EAT'
```

INSTR('abcdefg', 'd') —' 4
LTRIM('00022345', '0') —' '22345'
RTRIM('0223455', '5') —* '02234'
TRIM(' 'FROM' Monroe') —' 'Monroe'
LENGTH('Oracle9i') —' 8
LPAD(265000, 9, '$') —' $$$265000
RPAD(265000, 9,*) 265000'
REPLACE('Basketball', 'ket', 'e') —' 'Baseball'

In Oracle9i, the INSTR function is enhanced to take more arguments:

SELECT INSTR('CORPORATE FLOOR DOOR', 'OR' 1, 2) FROM dual;

where we are looking for string 'OR' and function is asked to start at the first character from the left to look for the second occurrence of the string. The result is 5, as the second 'OR' starts at position 5. If argument 1 is changed to —3, the function will start at the third character from right and search in the reverse direction.

Numeric Functions. The numeric functions take numeric value(s) and return a numeric value. The ROUND function rounds the value, expression, or column

to n decimal places. If n is omitted, zero decimal place is assumed. If n is negative, rounding takes place to the left side of the decimal place.

For example,
ROUND(25.465, 2) = 2547
ROLJND(25.465, 0) =25
ROUND(25.465, —1) =30

The TRUNC function truncates the value, expression, or column to n decimal places. If n is not supplied, zero decimal place is assumed. If n is negative, truncation takes place to the left side of the decimal place. For example,

TRUNC(25.465, 2) = 25.46
TRUNC(25.465, 0) =25
TRUNC(25.465, —1) =20

The POWER function finds the power of a number (nP). For example,
POWER(2, 4) 16

POWER(5, 3) = 125

The ABS function returns the absolute value of a column, expression, or value. Fw exaxnfr,
ABS(—10)= 10

The MOD function finds the integer remainder of x divided by y.It ignores the quotient. For example,
MOD(5,2)=1
MOD(3, 5) = 3
MOD(8, 4) = 0

The SIGN function returns —1 for a negative number, 1 for a positive number, and 0 for a zero. For example,

SIGN(-50) = —1
SIGN(+43) = 1
SIGN(O) = 0

The FLOOR function is similar to the TRUNC function, and the CElL funcbar-tion is similar to the ROUND function. However, both take one argument instead
5, as of two. For example,
will
FLOOR(54.7) = 54
CEIL(54.7) = 55
There is a difference in CEIL function, because it always returns the next higher

integer value. For example,
ROUND (54.3) = 54
CEIL (54.3) = 55

The table is called DUAL, which is provided by Oracle. The DUAL table is owned by user SYS, and it is available to all users. The DUAL table is useful when you want to find the outcome of a function and the argument is not taken from any table. The DUAL table can also be used to perform arithmetic expressions. For example,

SELECT 25000 * 0.25 FROM DUAL;

The DUAL table contains a single column called DUMMY and a single row with
value X.

Date Functions. Oracle stores dates internally with day, month, year, century, hour, minute, and second information, The default date display format is DD-MON-YY. There is a very useful date function called SYSDATE that does value, not take any arguments. SYSDATE returns the system's current date. For example,

SELECT SYSDATE FROM DUAL;

```
SQL> SELECT ROUND(5.55, 1), TRUNC(5.5), SIGN(-5.5), MOD(5,2),
  2  ABS(-5), POWER(3, 4), FLOOR(5.5), CEIL(5.5)
  3  FROM DUAL;

ROUND(5.55,1) TRUNC(5.5) SIGN(-5.5) MOD(5,2) ABS(-5) POWER(3,4) FLOOR(5.5) CEIL(5.5)
------------- ---------- ---------- -------- ------- ---------- ---------- ---------
          5.6          5         -1        1       5         81          5         6

SQL>
```

Numeric Function	Use
ROUND(*column* | *expr, [n]*)	Rounds the column or expression to *n* decimal places.
TRUNC(*column* | *expr, [n]*)	Truncates the column or expression to *n* decimal places.
POWER(*n, p*)	Returns *n* raised to power $p(n^p)$.
ABS(*n*)	Returns the absolute value of *n*.
MOD(*x, y*)	Returns the integer remainder of *x*/*y*.
SIGN(*value*)	Returns 1 for positive, −1 for negative and 0 for a zero.
FLOOR(*value*)	Returns the largest integer less than or equal to value.
CEIL(*value*)	Returns the smallest integer greater than or equal to value.

```
SQL> DESCRIBE DUAL
Name                                      Null?    Type
----------------------------------------- -------- ----------
DUMMY                                              VARCHAR2(1)
SQL> SELECT * FROM dual;
D
-
X
SQL>
```

This query will display the current date. You can get more information about day, date and time by using a format mask with SYSDATE function.

SELECT TO_CHAR(SYSDATE, 'DY, MONTH 00, YYYY HH:MI:SS RM.')
FROM DUAL;

The Date Function

Date Expression	Result
Date + number	Adds a number of days to a date.
Date – number	Substracts a number of days from a date.
Date + number/24	Adds a number of hours to a date.
Date1 – Date2	Gives the number of days between two dates.

Age Calculation From Date:

```
SQL> SELECT Last, First, (SYSDATE – BirthDate) / 365 AGE
  2  FROM student;
LAST            FIRST           AGE
--------------- --------------- ----------
Diaz            Jose            20.8289038
Tyler           Mickey          19.7330134
Patel           Rajesh          17.9960271
Rickles         Deborah         33.1521915
Lee             Brian           18.0343832
Khan            Amir            19.4289038
6 rows selected.
SQL>
```

DATE Functions:

Date Function	Use
MONTHS_BETWEEN(date1, *date2*)	Finds the number of months between two dates.
ADD_MONTHS(date, *m*)	Adds calendar months to a date.
NEXT_DAY (date, 'day')	Finds the next occurrence of a day from the given date.
LAST_DAY(date)	Returns the last day of the month.
ROUND(date [, 'format'])	Rounds the date to the nearest day, month, or year.
TRUNC(date [, 'format'])	Truncates the date to the nearest day, month, or year.
EXTRACT(YEAR | MONTH | DAY FROM date)	Extracts the year, month, or day from a date value.
NEXT_TIME(*date, existing timezone, newtimezone*)	Returns the date in different time zone, such as EST or PST.

The function ADD_MONTHS adds the number of months supplied as a second argument. The number must be an integer value. It can be positive or negative. For example, if an item is shipped today and payment is due in three months, what is the payment date?

ADD_MONTHS('10-MAY-03', 3) —* '10-AUG-03'

The function NEXT_DAY returns the next occurrence of a day of the week following the date supplied. The second argument could be a number in quotes or a day of the week.

For example,
NEXT_DAY ('14-OCT-03', 'SUNDAY') —b '19-OCT-03'

NEXT_DAY('14-OCT-03', 'TUESDAY') '21-OCT-03'
The function LAST_DAY finds the last date of the month for the date supplied as an argument. If something is due by the end of this month, what is that date? For example,
LAST_DAY('05-FEB-04') —k '29-FEB-04' ;

The ROUND function rounds a date based on the format specified. If a format is missing, rounding is to the nearest day. For example,

ROUND(TO_DATE('07/20/03', 'MM/DD/YY'), 'MONTH') —* '01-AUG-03'

Here, the date is nearer to August 1 than to July 1. In the next example, the date is nearest to the first of next year:

ROUNO(TO_DATE('07!20103', 'MM/DD/YY'), 'YEAR') —* '01-JAN-04'

The TRUNC function truncates the date to the nearest format specified. Truncation to the nearest month returns the first day of the date's month, and truncation to the nearest year returns the January 1 of the date's year. For example,

TRUNC(TO_DATE('07/20/03', 'MMIDD/YY'), 'MONTH') —* '01-JUL-03'
TRUNC(TO_DATE('07120103', 'MMIDDIYY'), 'YEAR') -4 '01 -JAN-03'

The EXTRACT function extracts year, month, or day from a date value.For

SELECT EXTRACT(MONTH FROM sysdate),
EXTRACT(DAY FROM sysdate),
EXTRACT(YEAR FROM sysdate) FROM dual;

For example,

The following is a list of a few more date- and time-related functions introduced in Oracle9i:

CURRENT_DATE—returns the current date the in session's time zone.
CURRENT_TIMESTAMP—returns the current date and time in the session's time zone.
DBTIMEZONE—returns the value of the database's time zone.
SESSIONTIMEZONE—returns the current session's time zone.
SYSTIMESTAMP—returns the date and time in the time zone of the database

GROUP FUNCTIONS

Group Function	Use
SUM (*column*)	Finds the sum of all values in a column; ignores null values.
AVG (*column*)	Finds the average of all values in a column; ignores null values.
MAX(*column* | *expression*)	Finds the maximum value; ignores null values.
MIN(*column* | *expression*)	Finds the minimum value; ignores null values.
COUNT(* | *column* | *expression*)	Counts the number of rows, including nulls, for *; counts nonnull values if the column or expression is used as an argument.

While using the group functions, the key words DISTINCT or ALL can be used before listing the argument in parenthesis. The key word ALL which means use all values (including duplicate values), is the default. The key word DISTINCT tells the function to

use non duplicate values only. Let us write a query to find the total, average, highest, and lowest salaries from the EMPLOYEE table.

```
SQL> SELECT SUM(Salary), AVG(Salary), MAX(Salary), MIN(Salary)
  2  FROM EMPLOYEE;

SUM(SALARY)    AVG(SALARY)    MAX(SALARY)    MIN(SALARY)
-----------    -----------    -----------    -----------
     741000          92625         265000          24500

1 row selected.

SQL>
```

```
SQL> SELECT MAX(BirthDate) YOUNGEST,
  2         MIN(BirthDate) OLDEST
  3  FROM student;

YOUNGEST  OLDEST
---------  ---------
12-DEC-85  20-OCT-70

SQL>
```

Check your progress 1:

What are the different types of Built-in-functions?

.....

.....

.....

.....

.....

.....

14.2 Grouping Data

The rows in a table can be divided into different groups to treat each group separately. The group functions can be applied to individual groups in the same fashion they are applied to all rows. The GROUP BY clause is used for grouping data. The general syntax is

SELECT column, group function(column)
FROM tablename
(WHERE condition(s))
(GROUP BY column I expression)
(ORDER BY column I expression (ASC I DESC));

Important points to remember:

- When you include a group function and the GROUP BY clause in your
- query, the individual column(s) appearing in SELECT must also appear in GROUP BY.
- The WHERE clause can still be used to restrict data before grouping.
- The WHERE clause cannot be used to restrict groups.
- A column alias cannot be used in a GROUP BY clause.

- The GROUP BY column does not have to appear in a SELECT query.
- When a column is used in the GROUP BY clause, the result is sorted in ascending order by that column by default. In other words, GROUP BY has an implied ORDER BY. You can still use an ORDER BY clause explicitly to change the implied sort order.
- In Oracle9i. the order of the WHERE and GROUP BY clauses in the SELECT query does not matter. but traditionally, the WHERE clause is written before the GROUP BY clause.

```
SQL> SELECT DeptId, COUNT(*) "# of Emp"
  2  FROM employee
  3  GROUP BY DeptId
  4 /

   DEPTID          # of Emp
--------          -------
       10                3
       20                2
       30                2
       40                1

SQL>
```

```
SQL> SELECT DeptId, COUNT(*) "# of Emp"
  2  FROM employee
  3 /
SELECT DeptId, COUNT(*) "# of Emp"
       *

ERROR at line 1:
ORA-00937: not a single-group group function

SQL>
```

```
SQL> SELECT Building, COUNT(*)
  2  FROM location
  3  WHERE COUNT(*) >= 4
  4  GROUP BY Building;
WHERE COUNT(*) >= 4
      *

ERROR at line 3:
ORA-00934: group function is not allowed here

SQL>
```

The HAVING clause can restrict groups. The WHERE clause restricts rows, the GROUP BY clause groups remaining rows, the Group function works on each group, and the HAVING clause keeps the groups that match the group condition.

In the sample query, the WHERE clause filters out the building named Kennedy, the rest of the rows are grouped by the building names Gandhi and Nehru, the group function COUNT counts the number of rows in each group, and the HAVING clause keeps groups with four or more rows—that is, the Gandhi building with five rows/rooms.

The implied ascending sort with the GROUP BY clause can be overriden by adding an explicit ORDER BY clause to the query.

14.3 Let Us Sum Up

Check your progress: Answers:

1. String functions, Character Functions, Numeric Functions, Date Functions and Group Functions.

LESSON 15

MULTIPLE TABLES

Contents

15.0 AIMS AND OBJECTIVES

To learn about the Join and Set Operations.

15.1 Join Operation

A join is used to combine rows from multiple tables. A join is performed whenever two or more tables is listed in the FROM clause of an SQL statement. There are different kinds of joins. Let's take a look at a few examples.

Inner Join (simple join)

Chances are, you've already written an SQL statement that uses an inner join. It is the most common type of join. Inner joins return all rows from multiple tables where the join condition is met.

For example,

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id;
```

This SQL statement would return all rows from the suppliers and orders tables where there is a matching supplier_id value in both the suppliers and orders tables.

Let's look at some data to explain how inner joins work:

We have a table called suppliers with two fields (supplier_id and supplier_ name).
It contains the following data:

We have another table called orders with three fields (order_id, supplier_id, and order_date).

It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard

order_id	supplier_id	order_date
500125	10003	2003/05/12
500126	10001	2003/05/13

If we run the SQL statement below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13

The rows for Microsoft and NVIDIA from the supplier table would be omitted, since the supplier_id's 10002 and 10003 do not exist in both tables.

Outer Join

Another type of join is called an outer join. This type of join returns all rows from one table and only those rows from a secondary table where the joined fields are equal (join condition is met).

For example,

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
from suppliers, orders
where suppliers.supplier_id = orders.supplier_id(+);
```

This SQL statement would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.

The (+)after the orders.supplier_id field indicates that, if a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set.

The above SQL statement could also be written as follows:

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
from suppliers, orderswhere orders.supplier_id(+) = suppliers.supplier_id
```

We have a table called suppliers with two fields (supplier_id and name).

It contains the following data:

Supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called orders with three fields (order_id, supplier_id, and order_date).

It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13

If we run the SQL statement below:

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
from suppliers, orders
where suppliers.supplier_id = orders.supplier_id(+);
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13
10002	Microsoft	<null>
10003	NVIDIA	<null>

The rows for Microsoft and NVIDIA would be included because an outer join was used. However, you will notice that the order_date field for those records contains a <null> value.

15.2 Set Operation

There are situations when we need to combine the results from two or more SELECT statements. SQL enables us to handle these requirements by using set operations. The result of each SELECT statement can be treated as a set, and SQL set operations can be applied on those sets to arrive at a final result. Oracle SQL supports the following four set operations:

UNION ALL

UNION

MINUS

INTERSECT

SQL statements containing these set operators are referred to as compound queries, and each SELECT statement in a compound query is referred to as a component query. Two SELECTs can be combined into a compound query by a set operation only if they satisfy the following two conditions:

The result sets of both the queries must have the same number of columns. The datatype of each column in the second result set must match the datatype of its corresponding column in the first result set.

The datatypes do not need to be the same if those in the second result set can be automatically converted by Oracle (using implicit casting) to types compatible with those in the first result set.

These conditions are also referred to as union compatibility conditions. The term union compatibility is used even though these conditions apply to other set operations as well. Set operations are often called vertical joins, because the result combines data from two or more SELECTS based on columns instead of rows. The generic syntax of a query involving a set operation is:

<component query>

{UNION | UNION ALL | MINUS | INTERSECT}

<component query>

The keywords UNION, UNION ALL, MINUS, and INTERSECT are set operators. We can have more than two component queries in a composite query; we will always use one less set operator than the number of component queries.

The following sections discuss syntax, examples, rules, and restrictions for the four set operations.

Set Operators

The following list briefly describes the four set operations supported by Oracle SQL:

UNION ALL - Combines the results of two SELECT statements into one result set.

UNION - Combines the results of two SELECT statements into one result set, and then eliminates any duplicate rows from that result set.

MINUS - Takes the result set of one SELECT statement, and removes those rows that are also returned by a second SELECT statement.

INTERSECT - Returns only those rows that are returned by each of two SELECT statements.

Before moving on to the details on these set operators, let's look at the following two queries, which we'll use as component queries in our subsequent examples. The first query retrieves all the customers in region 5.

```
SELECT CUST_NBR, NAME

FROM CUSTOMER

WHERE REGION_ID = 5;
```

```
 CUST_NBR NAME
---------- ------------------------------
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
```

The second query retrieves all the customers with the sales representative is 'MARTIN'.

```
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
 CUST_NBR NAME
---------- ------------------------------
         4 Flowtech Inc.
         8 Zantech Inc.
```

If we look at the results returned by these two queries, we will notice that there is one common row (for Flowtech Inc.). The following sections discuss the effects of the various set operations between these two result sets.

UNION ALL

The UNION ALL operator merges the result sets of two component queries. This operation returns rows retrieved by either of the component queries. The following example illustrates the UNION ALL operation:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION ALL
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
```

```
                FROM CUST_ORDER O, EMPLOYEE E
                WHERE O.SALES_EMP_ID = E.EMP_ID
                AND E.LNAME = 'MARTIN');

 CUST_NBR NAME
---------- ------------------------------
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
         4 Flowtech Inc.
         8 Zantech Inc.
7 rows selected.
```

As we can see from the result set, there is one customer, which is retrieved by both the SELECTs, and therefore appears twice in the result set. The UNION ALL operator simply merges the output of its component queries, without caring about any duplicates in the final result set.

UNION

The UNION operator returns all distinct rows retrieved by two component queries. The UNION operation eliminates duplicates while merging rows retrieved by either of the component queries. The following example illustrates the UNION operation:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                FROM CUST_ORDER O, EMPLOYEE E
                WHERE O.SALES_EMP_ID = E.EMP_ID
                AND E.LNAME = 'MARTIN');
```

```
 CUST_NBR NAME
---------- ------------------------------
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
         8 Zantech Inc.

6 rows selected.
```

This query is a modification of the previous query; the keywords UNION ALL have been replaced with UNION. Notice that the result set contains only distinct rows (no duplicates). To eliminate duplicate rows, a UNION operation needs to do some extra tasks as compared to the UNION ALL operation. These extra tasks include sorting and filtering the result set. If we observe carefully, we will notice that the result set of the UNION ALL operation is not sorted, whereas the result set of the UNION operation is sorted. These extra tasks introduce a performance overhead to the UNION operation. A query involving UNION will take extra time compared to the same query with UNION ALL, even if there are no duplicates to remove. Therefore, unless we have a valid need to retrieve only distinct rows, we should use UNION ALL instead of UNION for better performance.

INTERSECT

INTERSECT returns only the rows retrieved by both component queries. Compare this with UNION, which returns the rows retrieved by any of the component queries. If UNION acts like 'OR', INTERSECT acts like 'AND'. For example:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
INTERSECT
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
```

```
                WHERE O.SALES_EMP_ID = E.EMP_ID
                AND E.LNAME = 'MARTIN');
CUST_NBR NAME
---------- ------------------------------
         4 Flowtech Inc.
```

"Flowtech Inc." was the only customer retrieved by both SELECT statements. Therefore, the INTERSECT operator returns just that one row.

MINUS

MINUS returns all rows from the first SELECT that are not also returned by the second SELECT. For example:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
MINUS
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                FROM CUST_ORDER O, EMPLOYEE E
                WHERE O.SALES_EMP_ID = E.EMP_ID
                AND E.LNAME = 'MARTIN');

 CUST_NBR NAME
---------- ------------------------------
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         5 Gentech Industries
```

You might wonder why we don't see "Zantech Inc." in the output. An important thing to note here is that the execution order of component queries in a set operation is from top to bottom. The results of UNION, UNION ALL, and INTERSECT will not change if we alter the ordering of component queries. However, the result of MINUS will be different if we alter the order of the component queries. If we rewrite the previous query by switching the positions of the two SELECTs, we get a completely different result:

```
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN')
MINUS
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5;


 CUST_NBR NAME
---------- ------------------------------
         8 Zantech Inc.
```

The row for "Flowtech Inc." is returned by both queries, so in our first MINUS example the first component query adds "Flowtech Inc." to the result set while the second component query removes it. The second example turns the MINUS operation around. The first component query adds "Flowtech Inc." and "Zantech Inc." to the result set. The

second component query specifies rows to subtract. One of the rows to subtract is "Flowtech Inc.", leaving "Zantech Inc." as the sole remaining row.

In a MINUS operation, rows may be returned by the second SELECT that are not also returned by the first. These rows are not included in the output.

Using Set Operations to Compare Two Tables

Developers, and even DBAs, occasionally need to compare the contents of two tables to determine whether the tables contain the same data. The need to do this is especially common in test environments, as developers may want to compare a set of data generated by a program under test with a set of "known good" data. Comparison of tables is also useful for automated testing purposes, when we have to compare actual results with a given set of expected results. SQL's set operations provide an interesting solution to this problem of comparing two tables.

The following query uses both MINUS and UNION ALL to compare two tables for equality. The query depends on each table having either a primary key or at least one unique index.

```
(SELECT * FROM CUSTOMER_KNOWN_GOOD
MINUS
SELECT * FROM CUSTOMER_TEST)
UNION ALL
(SELECT * FROM CUSTOMER_TEST
MINUS
SELECT * FROM CUSTOMER_KNOWN_GOOD);
```

We can look at it as the union of two compound queries. The parentheses ensure that both MINUS operations take place first before the UNION ALL operation is performed. The result of the first MINUS query will be those rows in CUSTOMER_KNOWN_GOOD that are not also in CUSTOMER_TEST. The result of the second MINUS query will be those rows in CUSTOMER_TEST that are not also in CUSTOMER_KNOWN_GOOD. The UNION ALL operator simply combines these two result sets for convenience. If no rows are returned by this query, then we know that both tables have identical rows. Any rows returned by this query represent differences between the CUSTOMER_TEST and CUSTOMER_KNOWN_GOOD tables.

If the possibility exists for one or both tables to contain duplicate rows, we must use a more general form of this query in order to test two tables for equality. This more general form uses row counts to detect duplicates:

```
(SELECT C1.*,COUNT(*)
 FROM CUSTOMER_KNOWN_GOOD
 GROUP BY C1.CUST_NBR, C1.NAME...
MINUS
 SELECT C2.*, COUNT(*)
 FROM CUSTOMER_TEST C2
 GROUP BY C2.CUST_NBR, C2.NAME...)
UNION ALL
(SELECT C3.*,COUNT(*)
 FROM CUSTOMER_TEST C3
 GROUP BY C3.CUST_NBR, C3.NAME...
MINUS
 SELECT C4.*, COUNT(*)
 FROM CUSTOMER_KNOWN_GOOD C4
 GROUP BY C4.CUST_NBR, C4.NAME...)
```

This query is getting complex! The GROUP BY clause (see Chapter 4) for each SELECT must list all columns for the table being selected. Any duplicate rows will be grouped together, and the count will reflect the number of duplicates. If the number of duplicates is the same in both tables, the MINUS operations will cancel those rows out. If any rows are different, or if any occurrence counts are different, the resulting rows will be reported by the query.

Let's look at an example to illustrate how this query works. We'll start with the following tables and data:

```
DESC CUSTOMER_KNOWN_GOOD
 Name                      Null?    Type
 ------------------------- -------- ----------------
 CUST_NBR                  NOT NULL NUMBER(5)
 NAME                      NOT NULL VARCHAR2(30)


SELECT * FROM CUSTOMER_KNOWN_GOOD;
 CUST_NBR NAME
---------- ------------------------------
         1 Sony
         1 Sony
         2 Samsung
         3 Panasonic
         3 Panasonic
         3 Panasonic

6 rows selected.


DESC CUSTOMER_TEST
Name                      Null?    Type
 ------------------------- -------- ----------------
 CUST_NBR                  NOT NULL NUMBER(5)
 NAME                      NOT NULL VARCHAR2(30)


SELECT * FROM CUSTOMER_TEST;
  CUST_NBR NAME
```

---------- ------------------------------

1 Sony

1 Sony

2 Samsung

2 Samsung

3 Panasonic

As we can see the CUSTOMER_KNOWN_GOOD and CUSTOMER_TEST tables have the same structure, but different data. Also notice that none of these tables has a primary or unique key; there are duplicate records in both. The following SQL will compare these two tables effectively:

```
(SELECT C1.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD C1
GROUP BY C1.CUST_NBR, C1.NAME
MINUS
SELECT C2.*, COUNT(*)
FROM CUSTOMER_TEST C2
GROUP BY C2.CUST_NBR, C2.NAME)
UNION ALL
(SELECT C3.*, COUNT(*)
FROM CUSTOMER_TEST C3
GROUP BY C3.CUST_NBR, C3.NAME
MINUS
SELECT C4.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD C4
GROUP BY C4.CUST_NBR, C4.NAME);
```

CUST_NBR	NAME	COUNT(*)
2	Samsung	1
3	Panasonic	3
2	Samsung	2
3	Panasonic	1

These results indicate that one table (CUSTOMER_KNOWN_GOOD) has one record for "Samsung", whereas the second table (CUSTOMER_TEST) has two records for the same customer. Also, one table (CUSTOMER_KNOWN_GOOD) has three records for "Panasonic", whereas the second table (CUSTOMER_TEST) has one record for the same customer. Both the tables have the same number of rows (two) for "Sony", and therefore "Sony" doesn't appear in the output.

Duplicate rows are not possible in tables that have a primary key or at least one unique index. Use the short form of the table comparison query for such tables.

Using NULLs in Compound Queries

We discussed union compatibility conditions at the beginning of this chapter. The union compatibility issue gets interesting when NULLs are involved. As we know, NULL doesn't have a datatype, and NULL can be used in place of a value of any datatype. If we purposely select NULL as a column value in a component query, Oracle no longer has two datatypes to compare in order to see whether the two component queries are compatible. For character columns, this is no problem. For example:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT 2 NUM, NULL STRING FROM DUAL;

       NUM STRING
---------- --------
         1 DEFINITE
         2
```

Oracle considers the character string 'DEFINITE' from the first component query to be compatible with the NULL value supplied for the corresponding column in the second component qery. However, if a NUMBER or a DATE column of a component query is set to NULL, we must explicitly tell Oracle what "flavor" of NULL to use. Otherwise, we'll encounter errors. For example:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;

SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
```

```
      *
ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression
```

Note that the use of NULL in the second component query causes a datatype mismatch between the first column of the first component query, and the first column of the second component query. Using NULL for a DATE column causes the same problem, as in the following example:

```
SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, NULL DATES FROM DUAL;
SELECT 1 NUM, SYSDATE DATES FROM DUAL
              *
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

In these cases, we need to cast the NULL to a suitable datatype to fix the problem, as in the following examples:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT TO_NUMBER(NULL) NUM, 'UNKNOWN' STRING FROM DUAL;

       NUM STRING
---------- --------
         1 DEFINITE
           UNKNOWN

SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, TO_DATE(NULL) DATES FROM DUAL;
```

```
       NUM DATES
---------- ---------
         1 06-JAN-02
         2
```

This problem of union compatibility when using NULLs is encountered in Oracle8i. However, there is no such problem in Oracle9i, as we can see in the following examples generated from an Oracle9i database:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;

       NUM STRING
---------- --------
         1 DEFINITE
           UNKNOWN

SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, NULL DATES FROM DUAL;

       NUM DATES
---------- ---------
         1 06-JAN-02
         2
```

Oracle9i is smart enough to know which flavor of NULL to use in a compound query.

Check your progress 1:

What are the various set operations?

……
……
……
……

Rules and Restrictions on Set Operations

These rules and restrictions are as follows:

Column names for the result set are derived from the first SELECT:

```
SELECT CUST_NBR "Customer ID", NAME "Customer Name"
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
```

```
 Customer ID Customer Name
----------- ----------------------
          1 Cooper Industries
          2 Emblazon Corp.
          3 Ditech Corp.
          4 Flowtech Inc.
          5 Gentech Industries
          8 Zantech Inc.


6 rows selected.
```

Although both SELECTs use column aliases, the result set takes the column names from the first SELECT. The same thing happens when we create a view based on a set operation. The column names in the view are taken from the first SELECT:

```
CREATE VIEW V_TEST_CUST AS
SELECT CUST_NBR "Customer ID", NAME "Customer Name"
FROM CUSTOMER
WHERE REGION_ID = 5
```

```
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                FROM CUST_ORDER O, EMPLOYEE E

                WHERE O.SALES_EMP_ID = E.EMP_ID
                AND E.LNAME = 'MARTIN');
 View created.

DESC V_TEST_CUST
 Name                      Null?    Type
 ------------------------------ -------- ----
 Customer_ID                       NUMBER
 Customer_Name                     VARCHAR2(45)
```

If we want to use ORDER BY in a query involving set operations, we must place the ORDER BY at the end of the entire statement. The ORDER BY clause can appear only once at the end of the compound query. The component queries can't have individual ORDER BY clauses. For example:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN'
ORDER BY CUST_NBR;

  CUST_NBR NAME
---------- --------------------
         1 Cooper Industries
         2 Emblazon Corp.
```

```
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
      7654 MARTIN


6 rows selected.
```

Note that the column name used in the ORDER BY clause of this query is taken from the first SELECT. We couldn't order these results by EMP_ID. If we attempt to ORDER BY EMP_ID, we will get an error, as in the following example:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN'ORDER BY EMP_ID;
ORDER BY EMP_ID
         *
ERROR at line 8:
ORA-00904: invalid column name
```

The ORDER BY clause doesn't recognize the column names of the second SELECT. To avoid confusion over column names, it is a common practice to ORDER BY column positions:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN'
ORDER BY 1;
```

```
  CUST_NBR NAME
---------- --------------------
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
      7654 MARTIN


6 rows selected.
```

Unlike ORDER BY, we can use GROUP BY and HAVING clauses in component queries.

Component queries are executed from top to bottom. If we want to alter the sequence of execution, use parentheses appropriately. For example:

```
SELECT * FROM SUPPLIER_GOOD
UNION
SELECT * FROM SUPPLIER_TEST
MINUS
SELECT * FROM SUPPLIER;


SUPPLIER_ID NAME
----------- -------------------------
          4 Toshiba
```

Oracle performs the UNION between SUPPLIER_GOOD and SUPPLIER_TEST first, and then performs the MINUS between the result of the UNION and the SUPPLIER table. If we want the MINUS between SUPPLIER_TEST and SUPPLIER to be performed first, and then the UNION between SUPPLIER_GOOD and the result of MINUS, we must use parentheses to indicate so:

```
SELECT * FROM SUPPLIER_GOOD
UNION
(SELECT * FROM SUPPLIER_TEST
MINUS
```

SELECT * FROM SUPPLIER);

SUPPLIER_ID NAME

---------- ------------------------

1 Sony

2 Samsung

3 Panasonic

4 Toshiba

The parentheses in this query forces the MINUS to be performed before the UNION. Notice the difference in the result as compared to the previous example.

The following list summarizes some simple rules, restrictions, and notes that don't require examples:

Set operations are not permitted on columns of type BLOB, CLOB, BFILE, and VARRAY, nor are set operations permitted on nested table columns.

Since UNION, INTERSECT, and MINUS operators involve sort operations, they are not allowed on LONG columns. However, UNION ALL is allowed on LONG columns.

Set operations are not allowed on SELECT statements containing TABLE collection expressions.

SELECT statements involved in set operations can't use the FOR UPDATE clause. The number and size of columns in the SELECT list of component queries are limited by the block size of the database. The total bytes of the columns SELECTed can't exceed one database block.

Check your progress 1:

What are the different SET operations?

.....
.....
.....
.....
.....
.....

15.3 Let Us Sum Up

Check your progress: Answers :

1. UNION, INTERSECT, MINUS, UNION ALL.

UNIT - IV

LESSON 16

PL/SQL: A PROGRAMMING LANGUAGE

Contents

16.0 AIMS AND OBJECTIVES

To learn the basics of the PL/SQL programming language, Variables, constants, data types, and declarations. The assignment statement and use of arithmetic operators, the scope and use of various types of variables are also discussed.

16.1 INTRODUCTION

SQL is a great query language, but it has its limitations. So, Oracle Corporation has added a procedural language extension to SQL known as Programming Language Extensions to Structured Query Language (PL/SQL). It is Oracle's proprietary language for access of relational table data. PL/SQL is like any other high-level compiler language.

PL/SQL also allows embedding of SQL statements and data manipulation in its blocks. SQL statements are used to retrieve data, and PL/SQL control statements are used to process data in a PL/SQL program. The data can be inserted, deleted, or updated through a PL/SQL block, which makes it an efficient transaction-processing language.

The Oracle Server has an engine to execute SQL statements. The server also has a separate engine for PL/SQL. Oracle Developer tools have a separate engine to execute PL/SQL as well. The SQL statements are sent one at a time to the server for execution, which results in individual calls to the server for each SQL statement. It may also result in heavy network traffic. On the other hand, all SQL statements within a single PL/SQL block are sent in a single call to the server, which reduces overhead and improves performance.

16.2 HISTORY OF PL/SQL

Before PL/SQL was developed, users embedded SQL statements into hot languages like C++ and Java. PL/SQL version 1.0 was introduced with Oracle 6.0 in 1991. Version 1.0 had very limited capabilities, however, and was far from being a full-fledged programming language. It was merely used for batch processing. With versions 2.0,2.1, and 2.2, the following new features were introduced:

- The transaction control statements SA VEPOINT, ROLLBACK, and COMMIT.
- The DML statements INSERT, DELETE, and UPDATE.
- The extended data types Boolean, BINARY_INTEGER, PL/SQL records,
- and PL/SQL tables.
- Built-in functions-character, numeric, conversion. and date functions.
- Built-in packages.
- The control structures sequence, selection, and looping.
- Database access through work areas called cursors.
- Error handling.
- Modular programming with procedures and functions.
- Stored procedures, functions, and packages.
- Programmer-defined subtypes.
- DDL support through the DBMS_SQL package.
- The PL/SQL wrapper.
- The DBMS_JOB job scheduler.
- -File I/O with the UTF _FILE package.

PL/SQL version 8.0, also known as PL/SQL8, came with Oracle8, the "object relational" database software. Oracle allows creation of objects that can be accessed with Java, C++, Object COBOL, and other languages. It also allows objects and relational tables to coexist. The external procedures in Oracle allow you to compile procedures and store them in the shared library of the operating system-for example, an .so file in UNIX or a .dll (Dynamic Linked Library) file in Windows. Oracle's library is written in the C language. It also supports LOB

(Large Object) data types.

16.3 FUNDAMENTALS OF PL/SQL

A PL/SQL program consists of statements. You may use upper- or lowercase letters in your program. In other words, PL/SQL is not case sensitive except for character string values enclosed in single quotes. Like any other programming language, PL/SQL statements consist of reserved words, identifiers, delimiters, literals, and comments.

Reserved Words

The reserved words, or key words, are words provided by the language that have a specific use in the language. For example, DECLARE, BEGIN, END, IF, WHILE, EXCEPTION, PROCEDURE, FUNCTION, PACKAGE, and TRIGGER are some of the reserved words in PL/SQL.

User-Defined identifiers

User-defined identifiers are used to name variables, constants, procedures, functions, cursors, tables, records, and exceptions. A user must obey the following rules in naming these identifiers:

- The name can be from 1 to 30 characters in length.
- The name must start with a letter.
- Letters (A-Z, a-z) , numbers, the dollar sign ($), the number sign (#) and the
- underscore C) are allowed.
- Spaces are not allowed.
- Other special characters are not allowed.
- Key words cannot be used as user-defined identifiers.
- Names must be unique within a block.
- A name should not be the same as the name of a column used in the block.

It is a good practice to create short and meaningful names. Figure 16-1 shows a list of valid user-defined identifiers. Figure 16-2 shows a list of invalid user-defined identifiers along with reasons why they are invalid.

User-Defined Identifiers

Rate_of_pay

Num

AU34567890

Dollars$_and_cents

SS#

Figure 16-1 Invalid User-Defined identifiers

Invalid User-Defined Identifiers	Reason
2Number	starts with a number
Employee-name	special character hyphen
END	Reserved word
Department number	spaces
Largest_yearly_salary_paid	Too Long
Taxrate%	Special character %

Figure 16-2 Invalid user-defined identifiers.

LITERALS

Literals are values that are not represented by user-defined identifiers. Literals are of three types: numeric, character, and boolean. For example:

Numeric	100,3.14, -55,5.25E7,or NULL
Character	'A','this is a string','0001','25-MAY-00','',or NULL
Boolean	TRUE,FALSE,or NULL

In this list of values, '25-MAY-OO' looks like a date value, but it is a character string. It can be converted to date format by using the TO_DATE function. The value" (two single quotes having nothing within) is another way of entering the NULL value.

PL/SQL is case sensitive regarding character values within single quotation marks. The values 'ORACLE', 'Oracle', and 'oracle' are three different values in PL/SQL. To embed a single quote in a string value, two single quote symbols are entered-for example, 'New Year"s Day'. Numeric values can be entered in scientific notation with the letter E or e. Boolean values are not enclosed in quotation marks.

16.4 PL/SQL BLOCK STRUCTURE

PL/SQL is a block-structured language. A program can be divided into logical blocks. The block structure gives modularity to a PL/SQL program, and each object within a block has "scope." Blocks are of two types:

1. An anonymous block is a block of code without a name. It can be used anywhere in a program and is sent to the server engine for execution at runtime.

BLOCK	USE
Anonymous Block	An unnamed block, that is independent or embedded within an application.
Procedure function	A named block that is stored on the Oracle server,

	can be called by its name, and can take arguments.
Package	A named PL/SQL module that is a group of functions, procedures, and identifiers.
Trigger	A block that is associated with a database table or a view. It is executed when automatically fired by a DML statement.

16.5 COMMENTS

Comments are used to document programs. They are written as part of a program, but they are not executed. In fact, comments are ignored by the PL/SQL engine. It is a good programming practice to add comments to a program, because this helps in readability and debugging of the program. There are two ways to write comments in PL/SQL:

1. To write a single-line comment, two dashes (--) are entered at the beginning of a new line. For example, - -This is a single-line comment.

2. To write a multiline comment, comment text is placed between /* and */.

A multiline comment can be written on a separate line by itself, or it can be used on a line of code as well. For example,

/* This is a multiline comment that ends here. */

A Programmer can use a comment anywhere in the program.

16.6 DATA TYPES

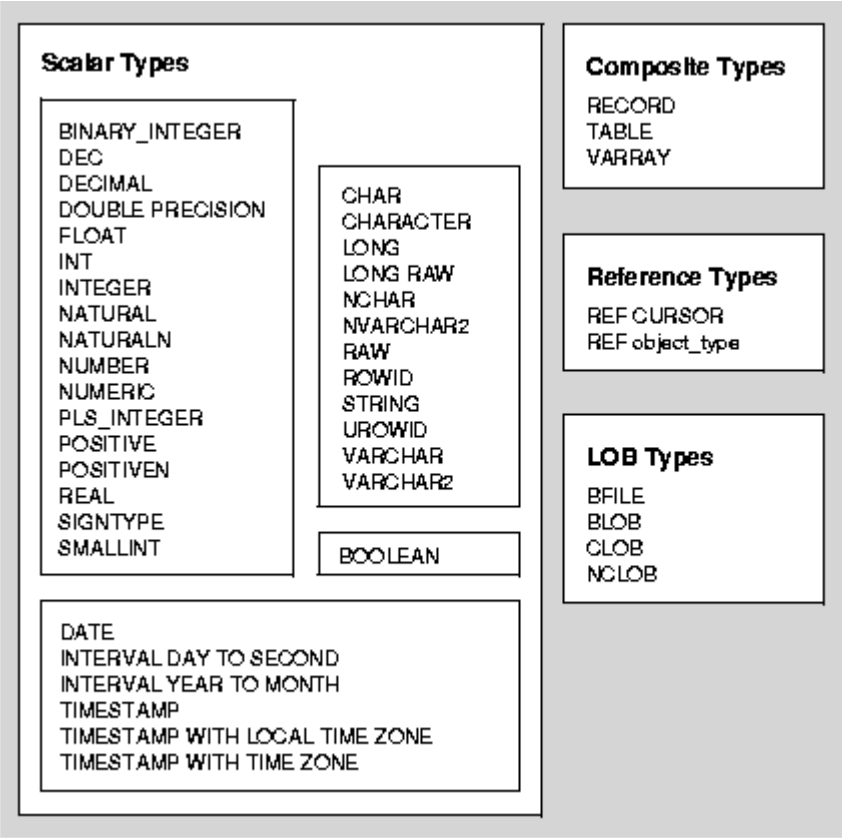
Each constant and variable in the program needs a data type. The data type decides the type of value that can be stored in a variable. PL/SQL has four data types:

1.Scalar.

2.Composite.

3.Reference.

4.LOB.



A scalar data type is not made up of a group of elements. It is atomic in nature. The \ composite data types are made up of elements or components.

There are four major categories of scalar data types:

1. Character.
2. Number.
3. Boolean.
4. Date.

Other scalar data type include raw, rowed, and trusted.

CHARACTER

Variables with a character data type can store text. The text may include letters, numbers, and special characters. The text in character-type variables can be manipulated with built-in character functions. Character data types include CHAR and VARCHAR2.

CHAR. The CHAR data type is used for fixed-length string values. The allowable string length is between 1 and 32,767. (If you remember, the allowable length in the Oracle database is only 2000.) If you do not specify a length for the variable, the default length is one. Get into the habit of specifying length along with the data type to avoid any errors.

If you are going to declare a variable of the CHAR type in PL/SQL code and that value is to be inserted into a table's column, the limitation on database size is only 2000 characters. You will have to find the substring of that character value to avoid the error message for inserting a character string longer than the length of the column

If you specify a length of 10 and the assigned value is only five characters long, the value is padded with trailing spaces because of the fixed-length nature of this data type. The CHAR type is not storage efficient, but it is performance efficient.

VARCHAR2. The VARCHAR2 type is used for variable-length string values. Again, the allowable length is between 1 and 32,767. A column in an Oracle database with a VARCHAR2 type, however, can only take 4000 characters, which is smaller than the allowable length for the variable.

Suppose you have two variables with data type of CHAR(20) and VARCHAR2(20), and both are assigned the same value, 'Oracle9i PL/SQL'. The string value is only 15 characters long. The first variable, with CHAR(20), is assigned a value padded with five spaces; the variable with VARCHAR2(20) does not get a string value padded with spaces. If the values of both variables are compared in a condition for equality, FALSE will be returned.

Other character data types are LONG (32,760 bytes, shorter than VARCHAR2), RAW (32,767 bytes), and LONG RAW (32,760 bytes, shorter than RAW). The RAW data values are neither interpreted nor converted by Oracle. VARCHAR2 is the most recommended character data type.

Number

PL/SQL has a variety of numeric data types. Whole numbers or integer values can be handled by following data types:

BINARY_INTEGER (approximately _231 + 1 to 231 - 1, or -2 billion to +2 billion)

INTEGER

INT

SMALLINT

POSITIVE (a subtype of BINARY_INTEGER-range, 0 to 2^{31}) NATURAL (a subtype of BINARY_INTEGER-range, 1 to 2^{31})

Similarly, there are various data types for decimal numbers :

NUMBER

DEC (fixed-point number) DECIMAL (fixed-point number) NUMERIC (fixed-point number) FLOAT (floating-point number) REAL (floating-point number)

DOUBLE PRECISION (floating-point number)

You are familiar with the NUMBER type from the Oracle table's column type. The NUMBER type can be used for fixed-point or floating-point decimal

numbers. It provides an accuracy of up to 38 decimal places. When using the NUMBER type, the precision and scale values are provided. The precision of a number is the total number of significant digits in that number, and the scale is the number of significant decimal places. The precision and scale values must be whole-number integers. For example,

NUMBER(p,s)

where p is precision and s is scale.

If scale has a value that is negative, positive, or zero, it specifies rounding of the number to the left of the decimal place, to the right of the decimal place, or to the nearest whole number, respectively. If a scale value is not used, no rounding occurs.

BOOLEAN

PL/SQL has a logical data type, Boolean, that is not available in SQL. It is used for Boolean data TRUE, FALSE, or NULL only. These values are not enclosed in single q·lotation marks like character and date values.

DATE

The date type is a special data type that stores date and time information. The date values have a specific format. A user can enter a date in many different formats with the TO_DATE function, but a date is always stored in standard 7-byte format. A date stores the following information:

Century Year Month Day Hour Minute Second

The valid date range is from January 1,4712 B.C., to December 31,9999 A.D. The time is stored as the number of seconds past midnight. If the user leaves out the time portion of the data, it defaults to midnight (12:00:00 A.M.). Various DATE functions are available for date calculations. For example, the SYSDATE function is used to return the system's current date.

Check your progress 1:

What are the main classification of data types in PL/SQL?

……

……

……

……

……

……

16.7 OTHER DATA TYPES

NLS

The National Language Support (NLS) data type is for character sets in which multiple bytes are used for character representation. NCHAR and NVARCHAR2

are examples of NLS data types.

LOB

•The NCLOB type contains a pointer to a large block of multibyte character data of fixed width .

•The BElLE type contains a pointer to large binary objects in an external operating system file. It would contain the directory name and the filename.

Oracle provides users with a built-in package, DBMS_LOB, to manipulate the contents of LOBs.

16.8 VARIABLE DECLARTION

A scalar variable or a constant is declared with a data type and an initial value assignment. The declarations are done in the DECLARE section of the program block. The initial value assignment for a variable is optional unless it has a NOT NULL constraint. The constants and NOT NULL type variables must be initialized. The general syntax is

DECLARE

identifiername [CONSTANT] data type [NOT NULL] [:= I DEFAULT expression]:

where identifiername is the name of a variable or constant. A CONSTANT is an identifier that must be initialized and the value of which cannot be changed in the program body. A NOT NULL constraint can be used for variables, and such variables must be initialized. The DEFAULT clause, or :=, can be used to initialize a constant or a variable to a value. An expression can be a literal, another variable, or an expression.

The identifiers are named based on rules given previously in this chapter. Different naming conventions can be used. You should declare one variable per line for good readability. For example,

```
DECLARE
 v_number          NUMBER(2);
 v_count           NUMBER(1) := 1;
 v_state           VARCHAR2(2) DEFAULT 'NJ';
 c_pi              CONSTANT NUMBER := 3.14;
        v _invoicedate   DATE DEFAULT SYSDATE;
```

In this example, you see a naming convention that uses prefix v_for variables and prefix c_ for constants.

ANCHORED DECLARATION

PL/SQL uses % TYPE attribute to anchor a variable's data type. Another variable or a column in a table can be used for anchoring. In anchoring, you tell PL/SQL to use a variable or a column's data type for another variable in the program. The general syntax is

Variablename typeattribute%TYPE [value assignment];

where typeattribute is another variable's name or table's column with a table qualifier(e.g. tablename.columnname).

16.9 ASSIGNMENT OPERATION

The assignment operation is one of the ways to assign a value to a variable. You have already learned that a variable can be initialized at the time of declaration by using the DEFAULT option or :=. The assignment operation is used in the executable section of the program block to assign a literal, another variable's value, or the result of an expression to a variable. The general syntax is

VariableName :=Literal\VeriableName\Expression:

For example,

v_num1:= 100;

v_num2 := v_num1;

v_sum := v_num1 + v_num2;

In these examples, the assumption is made that three variables have already been declared. The first example assigns 100 to the variable v _numl. The second example assigns the value of the variable v _numl to the variable v _num2. The third example assigns the result of an addition operation on v _numl and v _num2 to the variable v _sum.

The following statements are examples of invalid assignment operations and the reasons for their lack of validity.:

v_count = 10; /* Wrong assignment operator, = sign * /

v_count * 2 := v_double; /* Expression cannot be on the left. * /

v_num1 := v_num2 :=v_num3; /* Cannot use two assignments in one statement. */

16.10 BIND VARIABLES

Bind variables are also known as host variables. These variables are declared in the host SOL * Plus environment and are accessed by a PL/SOL block. Anonymous blocks do not take any arguments, but they can access host variables with a colon prefix (:) and the host variable name. Host variables can be passed to procedures and functions as arguments. A host variable is declared at the SOL> prompt with the SQL * Plus VARIABLE statement. The syntax of a host variable declaration is

VARIABLE variablename datatype

For example,

SQL>VARIABLE double NUMBER

When a numeric variable is declared with VARIABLE command, precision and scale values are not used. If a VARCHAR2-type variable is

declared, length is not used. A host variable's value can be printed in the SOL * Plus environment by using the PRINT command.

Let us put everything together in a program. The program contains a script that includes SOL * Plus statements and a PL/SOL block.

In Figure 16-3, two types of variables are used, a local variable v _num and a host variable g_double. The host variable g_double is declared in SOL * Plus with a VARIABLE statement, and the program block references it with a colon prefix (:). The local variable v _num is declared in the declaration section of a program block; there is no need to use the colon prefix with it.

```
SQL> VARIABLE g_double NUMBER

SQL> DECLARE
  2     v_num NUMBER(2);
  3 BEGIN
  4     v_num := 5;
  5     :g_double := v_num * 2;
  6 END;
  7 /

PL/SQL procedure successfully completed.

SQL> PRINT g_double

G_DOUBLE

        10

SQL>
```

Figure 16-3 Using a host variable in a PL/SQL block

The program assigns the value 5 to the local variable v _num, doubles it, and stores the result in the host variable g_double. Finally, the resulting variable is printed in the host environment with a PRINT statement ..

Question: How does a PL/SQL block end?

Answer: It ends with an END and a semicolon on the same line and a slash (/) on the next line.

The use of host variables should be minimized in a program block, because they affect performance. Every time a host variable is accessed within the block, the PL/SQL engine must stop to request the host environment for the value of the host variable. The variable's value can be assigned to a local variable to minimize calls to the host.

16.11 SUBSTITUTION VARIABLE IN PL/SQL

PL/SQL does not have any input capabilities in terms of having an input statement. There are no explicit input/output (I/O) statements, but

substitution variables of SQL are available in PL/SQL. Substitution variables have limitations, which become apparent in a loop.

Let us rewrite the program code in Figure 7-3 to the one in Figure 16-4. When the code in Figure 7-4 is executed, a standard prompt for p_num appears on the screen for users to type in a value for it.

```
SQI.> VARIABLE g_double NUMBER SQI.> DECLARE
                    2       v_num NUMBER(2);
                     3 BEGIN
                    4       v_num := &p_num;
                    5       :9_double := v_num * 2;
                     6 END;
                     7 /
Enter value for p_num: 10
PL/SQL procedure successfully completed.


SQL>PRINT g_double
        G_DOUBLE
            20
SQL>
```

Fig.16.4. Local, host and substitution variables

As you see in the example, there is no need to declare substitution variables in the program block. The value of the bind/host variable is printed with the PRINT command. The value of the local variable v _num \ cannot be printed with the PRINT command, however, because the scope of a local variable is within the block where it is declared.

When substitution variable is used in a program, the output contains lines showing how the substitution was done. You can suppress those lines by using the SET VERIFY OFF command before running the program.

16.12 PRINTING IN PL/SQL

There is no explicit output statement in PL/SQL. Oracle does have a built-in package called DBMS_OUTPUT with the procedure PUT_LINE to print. An environment variable named SERVEROUTPUT must be toggled ON to view output from it.

The DBMS_OUTPUT is the most frequently used package because of its capabilities to get lines from a file and to put lines into the buffer. The PUT_LINE procedure displays information passed to the buffer and puts a new-line marker at the end. For example,

DBMS_OUTPUT.PUT_LINE ('This line will be displayed');

The maximum size of the buffer is 1 megabyte. The following command enables you to view information from the buffer by using the DBMS_OUTPUT package and also sets the buffer size to the number of bytes specified:

SET SERVEROUTPUT ON [on size 10000);

The PL/SQL block in Figure 7-5 shows the use of DBMS_OUTPUT.PUT_ LINE.

```
SQL> VARIABLE NUM NUMBER
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2     DOUBLE NUMBER;
  3   BEGIN
  4     :NUM := 5;
  5     DOUBLE := :NUM * 2;
  6       DBMS_OUTPUT.PUT_LINE('DOUBLE OF '||
  7       TO_CHAR(:NUM) || 'IS' ||  TO_CHAR(DOUBLE);
8 END;
9 /

DOUBLE OF 5 IS 10
PL/SQL PROCEDURE SUCCESSFULLY COMPLETED
SQL>
```

Figure 16.5 DBMS_OUTPUT.PUT_LINE

Another procedure, DBMS_OUTPUT.PUT, also performs the same task of displaying information from the buffer, but it does not put a new-line marker at the end. If there is another DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUT_LINE statement following that statement, its output will be displayed on the same line.

16.13 ARIHMETIC OPEATORS

Five standard arithmetic operators are available in PL/SQL for calculations. If more than one operator exists in an arithmetic expression, the following order of precedence is used:

- Exponentiation is performed first, multiplication and division are performed next, and addition and subtraction are performed last.
- If more than one operator of the same priority is present, they are performed from left to right.

- Whatever is in parentheses is performed first.

Figure 16-6 shows arithmetic operators and their use.

+	Addition
-	Subtraction and negation
*	Multiplication
%	Division
**	Exponentiation

Question: What is the answer from the following expression ?

-2+3*(10-2*3)/6

Answer: 0 (The operations within the parentheses are performed first, with multiplication followed by subtraction. The result from within the parentheses is multiplied by 3, and that result is divided by 6. Finally, the result is added to -2)

16.4 LET US SUM UP

Check your progress : Answers

1. Scalar., Composite, Reference, LOB.

LESSON 17

PL/SQL: CONTROL STRUCTURES

Contents

17.0 AIMS AND OBJECTIVES

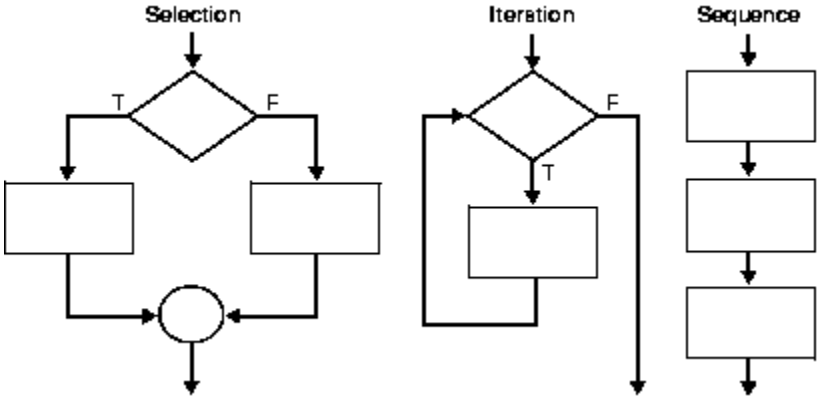
To learn the basics of the PL/SQL programming language, the various control structures used in PL/SQL and its uses.

17.1 CONTROL STRUCTURES

Overview of PL/SQL Control Structures

According to the structure theorem, any computer program can be written using the basic control structures shown in Figure 7-7. They can be combined in any way necessary to deal with a given problem.

Figure 7-7 Control Structures



The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements

repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF and CASE Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.

IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
  sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
  compute_bonus(empid);
  UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

You might want to place brief IF statements on a single line, as in

```
IF x > y THEN high := x; END IF;
```

IF-THEN-ELSE Statement

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
  sequence_of_statements1
ELSE
  sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
  UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
  IF new_balance >= minimum_balance THEN
    UPDATE accounts SET balance = balance - debit WHERE ...
  ELSE
    RAISE insufficient_funds;
  END IF;
END IF;
```

IF-THEN-ELSIF Statement

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example:

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

If the value of sales is larger than 50000, the first and second conditions are true. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the INSERT statement.

CASE Statement

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. (Recall from Chapter 2 that a selector is an expression whose value is used to select one of several alternatives.) To compare the IF and CASE statements, consider the following code that outputs descriptions of school grades:

```
IF grade = 'A' THEN
  dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
  dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
  dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
  dbms_output. put_line('Fair');
```

```
ELSIF grade = 'F' THEN
  dbms_output.put_line('Poor');
ELSE
  dbms_output.put_line('No such grade');
END IF;
```

Notice the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. Let us rewrite the preceding code using the CASE statement, as follows:

```
CASE grade
  WHEN 'A' THEN dbms_output.put_line('Excellent');
  WHEN 'B' THEN dbms_output.put_line('Very Good');
  WHEN 'C' THEN dbms_output.put_line('Good');
  WHEN 'D' THEN dbms_output.put_line('Fair');
  WHEN 'F' THEN dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('No such grade');
END CASE;
```

The CASE statement is more readable and more efficient. So, when possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The keyword is followed by a selector, which is the variable grade in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL datatype other than BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed. For instance, in the last example, if grade equals 'C', the program outputs 'Good'. Execution never falls through; if any WHEN clause is executed, control passes to the next statement.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHEN clause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

If the CASE statement selects the implicit ELSE clause, PL/SQL raises the predefined exception CASE_NOT_FOUND. So, there is always a default action, even when you omit the ELSE clause.

The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space. The CASE statement has the following form:

```
[<<label_name>>]
CASE selector
  WHEN expression1 THEN sequence_of_statements1;
  WHEN expression2 THEN sequence_of_statements2;
  …
  WHEN expressionN THEN sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

Like PL/SQL blocks, CASE statements can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the CASE statement. Optionally, the label name can also appear at the end of the CASE statement.

Exceptions raised during the execution of a CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

An alternative to the CASEstatement is the CASE expression, where each WHEN clause is an expression. For details, see "CASE Expressions".

Searched CASE Statement

PL/SQL also provides a searched CASE statement, which has the form:

```
[<<label_name>>]
CASE
  WHEN search_condition1 THEN sequence_of_statements1;
  WHEN search_condition2 THEN sequence_of_statements2;
  …
  WHEN search_conditionN THEN sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

The searched CASE statement has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type. An example follows:

```
CASE
  WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
  WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
  WHEN grade = 'C' THEN dbms_output.put_line('Good');
  WHEN grade = 'D' THEN dbms_output.put_line('Fair');
  WHEN grade = 'F' THEN dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('No such grade');
END CASE;
```

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which WHEN clause is executed. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

If none of the search conditions yields TRUE, the ELSE clause is executed. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

Exceptions raised during the execution of a searched CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

Guidelines for PL/SQL Conditional Statements

Avoid clumsy IF statements like those in the following example:

```
IF new_balance < minimum_balance THEN
  overdrawn := TRUE;
ELSE
  overdrawn := FALSE;
END IF;
...
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

This code disregards two useful facts. First, the value of a Boolean expression can be assigned directly to a Boolean variable. So, you can replace the first IF statement with a simple assignment, as follows:

```
overdrawn := new_balance < minimum_balance;
```

Second, a Boolean variable is itself either true or false. So, you can simplify the condition in the second IF statement, as follows:

```
IF overdrawn THEN ...
```

When possible, use the ELSIF clause instead of nested IF statements. That way, your code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN          |     IF condition1 THEN
  statement1;               |       statement1;
ELSE                        |     ELSIF condition2 THEN
  IF condition2 THEN        |       statement2;
    statement2;             |     ELSIF condition3 THEN
  ELSE                      |       statement3;
    IF condition3 THEN      |     END IF;
      statement3;           |
    END IF;                 |
  END IF;                   |
END IF;                     |
```

These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

If you are comparing a single expression to multiple values, you can simplify the logic by using a single CASE statement instead of an IF with several ELSIF clauses.

Iterative Control: LOOP and EXIT Statements

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and 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 END LOOP, as follows:

```
LOOP
  sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
  …
  IF credit_rating < 3 THEN
    …
    EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
  …
  IF credit_rating < 3 THEN
    …
    EXIT;  -- not allowed
  END IF;
END;
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement. For more information, see "Using the RETURN Statement".

EXIT-WHEN

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

```
LOOP
  FETCH c1 INTO ...
  EXIT WHEN c1%NOTFOUND;  -- exit loop if condition is true
  ...
END LOOP;
CLOSE c1;
```

Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.

The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:

```
IF count > 100 THEN     |     EXIT WHEN count > 100;
  EXIT;                 |
END IF;                 |
```

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

Loop Labels

Like PL/SQL blocks, loops 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 you nest labeled loops, use ending label names to improve readability.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement, as follows:

```
<<outer>>
LOOP
  ...
  LOOP
    ...
    EXIT outer WHEN ...  -- exit both loops
  END LOOP;
  ...
END LOOP outer;
```

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

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. An example follows:

```
WHILE total <= 25000 LOOP
  ...
```

```
  SELECT sal INTO salary FROM emp WHERE ...
  total := total + salary;
END LOOP;
```

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top. Therefore, the sequence of statements is executed at least once. PL/SQL has no such structure, but you can easily build one, as follows:

```
LOOP
  sequence_of_statements
  EXIT WHEN boolean_expression;
END LOOP;
```

To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:

```
done := FALSE;
WHILE NOT done LOOP
  sequence_of_statements
  done := boolean_expression;
END LOOP;
```

A statement inside the loop must assign a new value to the Boolean variable. Otherwise, you have an infinite loop. For example, the following LOOP statements are logically equivalent:

```
WHILE TRUE LOOP     |     LOOP
  ...               |       ...
END LOOP;           |     END LOOP;
```

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through 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 by the keywords FOR and LOOP. A double dot (..) serves as the range operator. 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.

As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP  -- assign the value 3 to i
  sequence_of_statements  -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

```
FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

Inside a FOR loop, the loop counter can be referenced like a constant but cannot be assigned values, as 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;  -- not allowed
  END IF;
```

```
END LOOP;
```

Using the EXIT Statement

The EXIT statement lets a FOR loop complete prematurely. For example, the following loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed:

```
FOR j IN 1..10 LOOP
  FETCH c1 INTO emp_rec;
  EXIT WHEN c1%NOTFOUND;
  ...
END LOOP;
```

Suppose you must exit from a nested FOR loop prematurely. You can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement to specify which FOR loop to exit, as follows:

```
<<outer>>
FOR i IN 1..5 LOOP
  ...
  FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT outer WHEN c1%NOTFOUND;  -- exit both FOR loops
    ...
  END LOOP;
END LOOP outer;
-- control passes here
```

Sequential Control: GOTO and NULL Statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in complex, unstructured code (sometimes called spaghetti code) that is hard to understand and maintain. So, use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
  ...
  GOTO insert_row;
  ...
  <<insert_row>>
  INSERT INTO emp VALUES ...
END;
```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```
BEGIN
  ...
  <<update_row>>
  BEGIN
    UPDATE emp SET ...
    ...
  END;
  ...
  GOTO update_row;
  ...
END;
```

The label end_loop in the following example is not allowed because it does not precede an executable statement:

```
DECLARE
  done  BOOLEAN;
BEGIN
  ...
  FOR i IN 1..50 LOOP
```

```
    IF done THEN
      GOTO end_loop;
    END IF;
    ...
  <<end_loop>>  -- not allowed
  END LOOP;  -- not an executable statement
END;
```

To debug the last example, just add the NULL statement, as follows:

```
FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
<<end_loop>>
NULL;  -- an executable statement
END LOOP;
```

As the following example shows, a GOTO statement can branch to an enclosing block from the current block:

```
DECLARE
  my_ename  CHAR(10);
BEGIN
  <<get_name>>
  SELECT ename INTO my_ename FROM emp WHERE ...
  BEGIN
    ...
    GOTO get_name;  -- branch to enclosing block
  END;
END;
```

The GOTO statement branches to the first enclosing block in which the referenced label appears.

Restrictions

Some possible destinations of a GOTO statement are not allowed. Specifically, a GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block. For example, the following GOTO statement is not allowed:

```
BEGIN
  ...
  GOTO update_row;  -- can't branch into IF statement
  ...
  IF valid THEN
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
```

As the example below shows, a GOTO statement cannot branch from one IF statement clause to another. Likewise, a GOTO statement cannot branch from one CASE statement WHEN clause to another.

```
BEGIN
  ...
  IF valid THEN
    ...
    GOTO update_row;  -- can't branch into ELSE clause
  ELSE
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
```

The next example shows that a GOTO statement cannot branch from an enclosing block into a sub-block:

```
BEGIN
  ...
```

```
  IF status = 'OBSOLETE' THEN
    GOTO delete_part;  -- can't branch into sub-block
  END IF;
  ...
  BEGIN
    ...
    <<delete_part>>
    DELETE FROM parts WHERE ...
  END;
END;
```

Also, a GOTO statement cannot branch out of a subprogram, as the following example shows:

```
DECLARE
  ...
  PROCEDURE compute_bonus (emp_id NUMBER) IS
  BEGIN
    ...
    GOTO update_row;  -- can't branch out of subprogram
  END;
BEGIN
  ...
  <<update_row>>
  UPDATE emp SET ...
END;
```

Finally, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is not allowed:

```
DECLARE
  ...
  pe_ratio  REAL;
BEGIN
  ...
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM ...
```

```
  <<insert_row>>
  INSERT INTO stats VALUES (pe_ratio, ...);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    pe_ratio := 0;
    GOTO insert_row;  -- can't branch into current block
END;
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

NULL Statement

The NULL statement does nothing other than pass control to the next statement. In a conditional construct, the NULL statement tells readers that a possibility has been considered, but no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    ROLLBACK;
  WHEN VALUE_ERROR THEN
    INSERT INTO errors VALUES ...
    COMMIT;
  WHEN OTHERS THEN
    NULL;
END;
```

In IF statements or other places that require at least one executable statement, the NULL statement to satisfy the syntax. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:

```
IF rating > 90 THEN
  compute_bonus(emp_id);
ELSE
  NULL;
END IF;
```

Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A stub is dummy subprogram that lets you

defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
BEGIN
  NULL;
END debit_account;
```

17.2 NESTED BLOCKS

PL/SQL block may contain another PL/SQL block; in other words, PL/SQL blocks can be nested. The execution starts with the outer block and continues with the inner block. The variables declared in the outer block are global to the inner block, and they are accessible in the inner block. The variables declared in the inner block, however, are not accessible in the outer block.

For example,

```
DECLARE                    /* Outer block starts here */
 Var1 NUMBER;              /*known to outer  and Inner */
BEGIN
   .....                   /* can use Var1 here */
DECLARE                    /* Inner block starts here */
Var2 NUMBER;               /* known to inner block*/
BEGIN    ...                     /* can use var1 and var2 here */
  END;                     /* Inner block ends here */
....                       /* can use Var1 here*/
     End;                  /* Outer block ends here.*/
```

Check your progress 1:

List the control structures in PL/SQL.

.....

.....

.....

.....

.....

17.3 SQL IN PL/SQL

The PL/SQL statements have control structures for calculations, decision making, and iterations. You need to use SQL to interface with the Oracle database. When changes are necessary in the database, SQL can be used to retrieve and change information. PL/SQL supports all Data Manipulation Language (DML) statements, such as INSERT, UPDATE, and DELETE. It also supports the Transaction Control Language statements ROLLBACK, COMMIT, and SAVEPOINT. You can retrieve data using the data retrieval statement SELECT. A row of data can be used to assign values to variables. More than one row can be retrieved and processed using cursors (covered in the next chapter). PL/SQL statements can use single-row functions, but group functions are not available for PL/SQL statements. SQL statements in the PL/SQL block, however, can still utilize those group functions.

PL/SQL does not support Data Definition Language (DOL) statements, such as CREATE, ALTER, and DROP. The Data Control Language (DCL) statements GRANT and REVOKE also are not available in PL/SQL.

SELECT STATEMENT IN PL/SQL

The SELECT statement retrieves data from Oracle tables. The syntax of SELECT is different in PL/SQL, however, because it is used to retrieve values from a row into a list of variables or into a PL/SQL record. The general syntax is

SELECT columnnames

INTO varlablenames / RecordName

FROM tablename

WHERE condition;

where columnnames must contain at least one column and may include arithmetic or string expressions, single-row functions, and group functions. Variablenames must contain a list of local or host variables to hold values retrieved by the SELECT clause. The variables are declared either at the SQL * Plus prompt or locally under the DECLARE section. The recordname is a PL/SQL record (covered in the next chapter). All the features of SELECT in SQL are available with an added mandatory INTO clause.

The INTO clause must contain one variable for each value retrieved from the table. The order and data type of the columns and variables must correspond. The SELECT ... INTO statement must return one and only one row. It is your responsibility to code a statement that returns one row of data. If no rows are returned, the standard exception (error condition) NO_DATA_FOUND occurs. If more than one row are retrieved, the TOO_MANY _ROWS exception occurs. You will learn more about exceptions and exception handling in the next chapter.

In Figure 17.1, a few columns from a row of the EMPLOYEE table are retrieved into a series of variables. The variables can be declared with data types, but more appropriately, attribute % TYPE is used to avoid any data-type mismatches.

The SQL statement in PL/SQL ends with a semicolon (;). The INTO clause is mandatory in a SELECT statement when used in a PL/SQL block.

```
SQL> DECLARE
  2       V_LAST          EMPLOYEE.LNAME%TYPE;
  3       V_FIRST         EMPLOYEE.FNAME%TYPE;
  4       V_SAL           EMPLOYEE.SALARY%TYPE;
  5  BEGIN
  6  SELECT LNAME, FNAME, SALARY
  7     INTO V_LAST, V_FIRST, V_SAL
  8     FROM EMPLOYEE
  9  WHERE EMPLOYEEID = 200;
 10  DBMS_OUTPUT.PUT_LINE
 11      ('EMPLOYEE NAME: ' || V_FIRST || ' ' || V_LAST );
 12  DBMS_OUTPUT.PUT_LINE
 13      ('SALARY:       ' || TO_CHAR(V_SAL));
 14  END;
 15  /
EMPLOYEE NAME: Jinku Shaw
SALARY:              24500

PL/SQL procedure successfully completed.

SQL>
```

Figure 17.1 SELECT-INTO in PL/SQL

```
SQL> DECLARE
  2       V_ID          EMPLOYEE.EMPLOYEEID%TYPE;
  3       V_DEPT        EMPLOYEE.DEPTID%TYPE := &DEPT_NUM;
  4  BEGIN
  5   SELECT EMPLOYEEID INTO V_ID
  6   FROM EMPLOYEE WHERE DEPTID = V_DEPT;
  7  END;
  8 /
Enter value for dept_num: 10
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5

SQL> /
Enter value for dept_num: 50
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5

SQL>
```

FIG.17.2 SELECT…INTO with Error

Figure 17.2 shows another example that results in exceptions because the SELECT. . .INTO statement returns either too many rows or no data.

17.4 DATA MANIPULATION IN PL/SQL

You can use all DML statements in PL/SQL with the same syntax you used in SQL. The three DML statements to manipulate data are:

The INSERT statement to add a new row in a table.

The DELETE statement to remove a row or rows.

The UPDATE statement to change values in a row or rows.

INSERT STATEMENT

We will use an INSERT statement to add a new employee in the EMPLOYEE table. The statement will use sequences created earlier. For simplicity, only a few columns are used in the statement in Figure 17-3. NEXTVAL uses the next value from the sequence as the new EmployeeId, and CURRVAL uses the current value of the department from that sequence. If you also decide to insert today's date as the hire date, you could use the SYSDATE function for the value.

```
Sql> BEGIN
  2      INSERT INTO EMPLOYEE
  3      (EMPlOYEEID, lNAME, FNAME, SALARY, DEPTID)
  4      VALUES
  5      (EMPlOYEE_EMPlOYEEID_SEQ.NEXTVAl, 'RAI',
  6      'AISH', 90000, DEPT_DEPTID_SEQ.CURRVAl);
  7      COMMIT;
  8 END;
  9 /
```

Figure 17.3 INSERT in PL/SQL

DELETE STATEMENT

We will show the use of the DELETE statement in the PL/SQL block to remove some rows. Suppose the NamanNavan (N2) Corporation decides to remove the IT Department. All the employees belonging to that department must be removed from the EMPLOYEE table. Figure 17.4 shows the DELETE statement in PL/SQL.

```
SQL> DECLARE
  2      V_DEPTID      DEPT.DEPTID%TYPE;
  3  BEGIN
  4      SELECT DEPTID
  5          INTO V_DEPTID
  6      FROM DEPT
  7      WHERE UPPER(DEPTNAME) = '&DEPT_NAME'
  8      DELETE FROM EMPLOYEE
  9          WHERE DEPTID = V_DEPTID;
 10      COMMIT;
 11 END;
 12 /
Enter value for dept_name: IT

PL/SQL procedure successfully completed.

SQL>
```

Figure 17.4 DELETE in PL/SQL

UPDATE STATEMENT

The UPDATE statement can be used in a PL/SQL block for modification of data. The company decides to give a bonus commission to all the employees who are entitled to commission. The bonus is 10% of the commission received.

Figure 17.5 shows an example of an UPDATE statement in PL/SQL block to modify commission.

```
SQL> DECLARE
  2      V_INCREASE   NUMBER := &DECIMAL_INCREASE;
  3  BEGIN
  4      UPDATE EMPLOYEE
  5            SET SALARY = SALARY * (1 + V_INCREASE)
  6      WHERE EMPLOYEEID = &EMP_ID;
  7      COMMIT;
  8  END;
  9  /
Enter value for decimal_increase: 0.15
Enter value for emp_id: 545

PL/SQL procedure successfully completed.

SQL>
```

Figure 17.5 UPDATE in PL/SQL

17.5 TRANSACTION CONTROL STATEMENTS

You know what a transaction is. You also know the control capabilities in Oracle. In Figures 17.3, 17.4 and 17.5, after performing a DML statement, the sample blocks have used a COMMIT statement. You do not have to commit within a PL/SQL block. If you do decide to use it, your data will be written to the disk right away, and the locks from those rows will be released. All transaction control statements are allowed in PL/SQL, and they are as follows:

- The COMMIT statement to commit the current transaction.
- The SAVEPOINT statement to mark a point in your transaction.
- The ROLLBACK [TO SAVEPOINT n] statement to discard all or part of the transaction.

17.6 LET US SUM UP

Check your progress Answers :

1. The control structures in PL/SQL are

   IF….THEN

   NESTED IF

   CASE STRUCTURE

FOR LOOP

LESSON 18

PL/SQL CURSORS

Contents

18.0 AIMS AND OBJECTIVES

To learn about a private work area for an SQL statement and its active set, called a cursor and the types of cursors.

18.1 INTRODUCTION

One of the strongest benefits of PL/SQL is its error-handling capabilities. The error conditions, known as exceptions, in PL/SQL

18.2 CURSORS

When you execute an SQL statement from a PL/SQL block, Oracle assigns a private work area for that statement. The work area, called a cursor, stores the statement and the results returned by execution of that statement. A cursor is created either implicitly or explicitly by you.

TYPES OF CURSORS

The cursor in PL/SQL is of two types:

In a static cursor, the contents are known at compile time. The cursor object for such an SQL statement is always based on one SQL statement.

In a dynamic cursor, a cursor variable that can change its value is used.

The variable can refer to different SQL statements at different times.

You do not declare an implicit cursor. PL/SQL declares, manages, and closes it for every Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE.

You declare an explicit cursor when you have an SQL statement in a PL/SQL block that returns more than one row from an underlying table. The rows retrieved by such a statement into an explicit cursor make up the active set. When opened, the cursor points to the first row in the active set. You can retrieve and work with one row at a time from the active set. With every fetch of

a row, the pointer moves to the next row. The cursor returns the current row to which it is pointing.

18.3 IMPLICIT & EXPLICIT CURSORS AND ATTRIBUTES

IMPLICIT CURSORS

PL/SQL creates an implicit cursor when an SQL statement is executed from within the program block. The implicit cursor is created only if an explicit cursor is not attached to that SQL statement. Oracle opens an implicit cursor, and the pointer is set to the first (and the only) row in the cursor. Then, the SQL statement is fetched and executed by the SQL engine on the Oracle server. The PL/SQL engine closes the implicit cursor automatically. A programmer cannot perform on an implicit cursor all the operations that are possible- on explicit cursor statements. PUSQL creates an implicit cursor for each DML statement in PL/SQL code. You cannot use an explicit cursor for DML statements. You can choose to declare an explicit cursor for a SELECT statement that returns only one row of data, but if you don't declare an explicit cursor for a SELECT statement returning one row of data, an implicit cursor is created for it.

You have no control over an implicit cursor. The implied queries perform operations on implicit cursors. PL/SQL actually tries to fetch twice to make sure that a TOO_MANY _ROWS exception does not exist. The explicit cursor is more efficient, because it does not try that extra fetch. It is possible to use an explicit cursor for a SELECT statement that returns just one row, because you have control over it. For example,

CURSOR deptname_cur IS

SELECT DeptName. Location FROM dept WHERE Deptld = 10;

Here, only one row is retrieved by the cursor with two column values, Finance and Charlotte, which later can be assigned to variables by fetching that row.

EXPLICIT CURSORS

An explicit cursor is declared as a SELECT statement in the PL/SQL block. It is given a name, and you can use explicit statements to work with it. You have total control of when to open the cursor, when to fetch a row from it, and when to close it. There are cursor attributes in PL/SQL to get the status information on explicit cursors. Remember, you can declare an explicit cursor for a SELECT statement that returns one or more rows, but you cannot use an explicit cursor for a DML statement.

Four actions can be performed on an explicit cursor:

- Declare it.
- Open it.
- Fetch row(s) from it.
- Close it.

DECLARING AN EXPLICIT CURSOR

A cursor is declared as a SELECT statement. The SELECT statement must not have an INTO clause in a cursor's declaration. If you want to retrieve rows in a specific order into a cursor, an ORDER BY clause can be used in the SELECT statement.

The general syntax is

DECLARE

CURSOR cursorname IS SELECT statement;

FOR EXAMPLE:

This is a PL/SQL fragment that demonstrates the first step of declaring a cursor. A cursor named C_MyCursor is declared as a select statement of all the rows in the zipcode table that have the tem state equal to "NY."

DECLARE

CURSOR C_MyCursor IS

SELECT *

FROM bookings

WHERE Cust_no = 701;

…

<code would continue here with opening, fetching,

and closing of the cursor>

Cursor names follow the same rules of scope and visibility that apply to the PL/SQL identifiers. Because the name of the cursor is a PL/SQL identifier, it must be declared before it is referenced. Any valid select statement can be used to define a cursor, including joins and statements with

the UNION or MINUS clause.

RECORD TYPES

A record is a composite data structure, which means that it is composed of more than one element. Records are very much like a row of a database table, but each element of the record does not stand on its own.

PL/SQL supports three kinds of records:

(1) table based,

(2) cursor_based,

(3) programmer-defined.

A table-based record is one whose structure is drawn from the list of columns in the table. A cursor-based record is one whose structure matches the elements of a predefined cursor. To create a table-based or cursor_based record use the %ROWTYPE attribute.

<record_name> <table_name or cursor_name>%ROWTYPE

FOR EXAMPLE

```
 DECLARE
vr_customer Customers%ROWTYPE;
BEGIN
SELECT *
INTO vr_customer
FROM Customers
WHERE Cust_no = 701;
DBMS_OUTPUT.PUT_LINE (vr_customer.Cust_name||' '||' has an ID of 701');
EXCEPTION
WHEN no_data_found
THEN
RAISE_APPLICATION_ERROR(-2001,'The Customer '|| 'is not in the database');
END;
```

The variable vr_customer is a record type of the existing database table Customers. That is, it has the same components as a row in the student table. A cursor-based record is much the same, except that it is drawn from the select list of an explicitly declared cursors. When referencing elements of the record, you use the same syntax that you use with tables.

record_name.item_name

In order to define a variable that is based on a cursor record, the cursor must first be declared. In the following lab, you will start by declaring a cursor and then proceed with the process of opening the cursor, fetching from the cursor, and finally closing the cursor.

OPENING A CURSOR

The next step in controlling an explicit cursor is to open it. When the Open cursor statement is processed, the following four actions will take place automatically:

1. The variables (including bind variables) in the WHERE clause are examined.

2. Based on the values of the variables, the active set is determined and the PL/SQL engine executes the query for that cursor. Variables are examined at cursor open time only.

3. The PL/SQL engine identifies the active set of data—the rows from all involved tables that meet the WHERE clause criteria.

4. The active set pointer is set to the first row.

The syntax for opening a cursor is: OPEN cursor_name;

A pointer into the active set is also established at the cursor open time.

The pointer determines which row is the next to be fetched by the cur-sor.

More than one cursor can be open at a time.

FETCHING ROWS IN A CURSOR

After the cursor has been declared and opened, you can then retrieve data from the cursor. The process of getting the data from the cursor is referred to as fetching the cursor. There are two methods of fetching a cursor, done with the following command:

FETCH cursor_name INTO PL/SQL variables;

or

FETCH cursor_name INTO PL/SQL record;

When the cursor is fetched the following occurs:

1. The fetch command is used to retrieve one row at a time from the active set. This is generally done inside a loop. The values of each row in the active set can then be stored into the corresponding variables or PL/SQL record one at a time, performing operations on each one successively.

After each FETCH, the active set pointer is moved forward to the next row. Thus, each fetch will return successive rows of the active set, until the entire set is returned. The last FETCH

will not assign values to the output variables; they will still contain their prior values.

FOR EXAMPLE

```
DECLARE
CURSOR c_customer IS
SELECT *
FROM customers;
vr_customer c_customer%ROWTYPE;
BEGIN
OPEN c_customer;
LOOP
FETCH c_customer INTO vr_customer;
EXIT WHEN c_customer%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(vr_customer.cust_name||
' '||vr_customer.Cust_no);
```

END LOOP;

...

CLOSING A CURSOR

Once all of the rows in the cursor have been processed (retrieved), the cursor should be closed. This tells the PL/SQL engine that the program is finished with the cursor, and the resources associated with it can be freed.

The syntax for closing the cursor is:

CLOSE cursor_name;

The Table lists the attributes of a cursor, which are used to determine the result of a cursor operation when fetched or opened.

a) Now that you know cursor attributes, you can use one of these to exit the loop within the code you developed in the previous example.

Cursor attributes can be used with implicit cursors by using the prefix

SQL, for example: SQL%ROWCOUNT.

Table: Explicit Cursor Attributes

Cursor Attribute Syntax Explanation

%NOTFOUND cursor_name%NOTFOUND A Boolean attribute that re-turns TRUE if the previous FETCH did not return a row, and FALSE if it did.

%FOUND cursor_name%FOUND A Boolean attribute that re-turns TRUE if the previous FETCH returned a row, and FALSE if it did not.

%ROWCOUNT cursor_name%ROWCOUNT # of records fetched from a cursor at that point in time.

%ISOPEN cursor_name%ISOPEN A Boolean attribute that re-turns TRUE if cursor is open, FALSE if it is not.

If you use a SELECT INTO syntax in your PL/SQL block, you will be creating an implicit cursor. You can then use these attributes on the implicit cursor.

FOR EXAMPLE

```
SET SERVEROUTPUT ON

DECLARE

v_Cust_name Customers.Cust_name%type;

BEGIN

SELECT Cust_name

INTO v_Cust_name

FROM customers

WHERE Cust_no = 701;
```

```
IF SQL%ROWCOUNT = 1
THEN
DBMS_OUTPUT.PUT_LINE(v_cust_name ||' has a '||
'id of 701');
ELSIF SQL%ROWCOUNT = 0
THEN
DBMS_OUTPUT.PUT_LINE('The customer 701 is '||
' not in the database');
ELSE
DBMS_OUTPUT.PUT_LINE('Stop harassing me');
END IF;
END;
```

Here is an example of the complete cycle of declaring, opening, fetching and closing a cursor including use of cursor attributes.

```
DECLARE
v_Cust_no Customers.Cust_no%TYPE;
CURSOR c_customer IS
SELECT Cust_no
FROM Customers
WHERE Cust_no < 710;
BEGIN
OPEN c_customer;
LOOP
FETCH c_customer INTO v_Cust_no;
EXIT WHEN c_customer%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Customer ID : '||v_Cust_no);
END LOOP;
CLOSE c_customer;
EXCEPTION
WHEN OTHERS
THEN
IF c_customer%ISOPEN
```

THEN

CLOSE c_customer;

END IF;

END;

Check your progress 1:

What are the steps to work with the Explicit Cursor?

.....
.....
.....
.....

18.4 CURSOR FOR LOOPS

The cursor FOR loop is the easiest way to write a loop for explicit cursors. The cursor is opened implicitly when the loop starts. A row is then fetched into the record from the cursor with every iteration of the loop. The cursor is closed automatically when the loop ends, and the loop ends when there are no more rows. The cursor

FOR loop automates all the cursor actions. The general syntax is

FOR recordname IN cursorname

LOOP Loop statements;

.....

ENDLOOP;

where recordname is the name of the record that is declared implicitly in the loop and is destroyed when the loop ends and cursorname is the name of declared explicit cursor.

Figure 18-1 uses a Cur or FOR loop with a record. When the loop starts, the cursor is opened implicitly. During the loop execution, an implicit fetch retrieves a row into the record for processing with each loop iteration. When an implicit fetch cannot retrieve a row, the cursor is closed, and the loop terminates. The OPEN, FETCH, and CLOSE statements are missing, because these operations are performed implicitly. The record's columns are addressed with recordname.columnname notation. If the record is accessed after the END LOOP statement, it will throw an exception, because the record' scope is only within the loop body.

SQL> SET SERVEROUTPUT ON

SQL> DECLARE

2 CURSOR EMPLOYEE_CUR IS

3 SELECT LNAME, FNAME, SALARY

```
4         FROM EMPLOYEE;
5 BEGIN
6 FOR EMP _REC IN EMPLOYEE_CUR LOOP
7 IF EMP _RECSALARY > 75000 THEN
8 DBMS_OUTPUT.PUT(EMP_RECFNAME ||' ');
9 DBMS_OUTPUT.PUT(EMP_RECLNAME ||' ');
10 DBMS_OUTPUT.PUT_L1NE(EMP _RECSALARY II ' ');
11 END IF;
12 END LOOP;
13 END;
14 /
John Smith 265000
Larry Houston 150000
Derek Dev 80000
```

Figure 18.1 Cursor FOR Loop.

CURSOR FOR LOOP USING A SUBQUERY

Use of a subquery in the cursor FOR loop eliminates declaration of an explicit cursor. The cursor is created by a sub query in the FOR loop statement itself. In Figure 18-2, an explicit cursor is used with implicit actions. One thing that is missing is the cursor name. The cursor declaration is not necessary, because it is created through the subquery. This subquery is similar to the inline view covered in the SQL section of this text.

```
SQL> BEGIN
2 FOR EMP _REC IN
3 (SELECT FNAME, LNAME, SALARY,  COMMISSION
4 FROM EMPLOYEE
5 WHERE DEPTID = 10) LOOP
6  DBMS_OUTPUT.PUT_L1NE
7 (EMP _REC.FNAME || •• II EMP _REC.LNAME ||• $'||
8 TO_CHAR(EMP _REC.SALARY + NVL(EMP REC.COMMISSION. 0)));
9 END lOOP;
10 END;
11 /
```

JOHN SMITH $300000

SANDI ROBERS $75000

SUNNY CHEN $35000

Figure 18.2 Cursor FOR Loop with a subquery

18.5 LET US SUM UP

Check your progress Answers :

1. Declare it.

   Open it.

   Fetch row(s) from it.

   Close it.

LESSON 19

PL/SQL CURSORS WITH PARAMETERS

Contents

19.0 AIMS AND OBJECTIVES

To learn about a private work area for an SQL statement and its active set, called a cursor, the types of cursors and the cursors with parameters.

19.1 SELECT ….FOR UPDATE CURSOR

When you type a SELECT query, the result is returned to you without locking any rows in the table. Row locking is kept to a minimum. You can explicitly lock rows for update before changing them in the program. The FOR UPDATE clause is used with the SELECT query for row locking. The locked rows are not available to other users for DML statements until you release them with COMMIT or ROLLBACK commands. Rows that are locked for update do not have to be updated.

The general syntax is

CURSOR cursorname IS
SELECT columnnames

FROM tablename(s)

[WHERE condition]

FOR UPDATE [OF coloumnnames][NOWAIT];

The optional part of a FOR UPDATE clause is OF columnnames, which enables you to specify columns to be updated. You can actually update any column in a locked row. The optional word NOWAIT tells you right away if another user has already locked the table and lets you continue with other tasks. If you do not use NOWAIT and one or more rows are already locked by another user, you will have to wait until the lock is released.

19.2 WHERE CURRENT OF CLAUSE

In a cursor, data manipulation in the form of UPDATE or DELETE is performed on rows fetched. The WHERE CURRENT OF clause allows you to perform data manipulation only on a recently fetched row. The general syntax is

UPDATE tablename

SET Clause

WHERE CURRENT OF cursorname;

DELETE FROM tablename

WHERE CURRENT OF cursorname;

19.3 CURSOR WITH PARAMETERS

A cursor can be declared with parameters, which allow you to pass values to the cursor. These values are passed to the cursor when it is opened, and they are used in the query when it is executed. With the use of parameters, you can open and close a cursor many times with different values. The cursor with different values will then return different active sets each time it is opened. The general syntax is

CURSOR cursorname

[(parameter1 datatype1,parameter2 datatype2…)]

IS

SELECT query;

where parameter1,parameter2 are formal parameters passed to the cursor datatype is any scalar data type assigned to the parameter. The parameters are assigned only data types; they are not assigned size.

When a cursor is opened, values are passed to the cursor. Each value must match the positional order of the parameters in a cursor's declaration. The values can be passed through literals. PL/SQL variables, or bind variables. The parameters in a cursor are passed into the cursor, but you cannot pass any value out of the cursors through parameters.

A cursor with a parameter can be opened multiple times with a different parameter value to get a different active set.

When you declare a cursor with one or more parameters, you can initialize it to a default value as follows:

CURSOR employee_cur (dept_id employee.Deptld%TYPE :=99)IS

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      V_FIRST EMPLOYEE.FNAME%TYPE;
  3      V_LAST EMPLOYEE.LNAME%TYPE;
  4      D_ID  NUMBER(2) := &DEPARTMENT_ID;
  5      CURSOR EMPLOYEE_CUR (DEPT_NUM EMPLOYEE.DEPTID%TYPE) IS
  6      SELECT LNAME, FNAME
  7      FROM EMPLOYEE
  8      WHERE DEPTID = DEPT_NUM;
  9
 10  BEGIN
 11      OPEN EMPLOYEE_CUR(D_ID);
 12      DBMS_OUTPUT.PUT_LINE
 13      ('EMPLOYEES IN DEPARTMENT ' || TO_CHAR(D_ID));
 14      LOOP
 15          FETCH EMPLOYEE_CUR INTO V_LAST, V_FIRST;
 16            EXIT WHEN EMPLOYEE_CUR%NOTFOUND;
 17          DBMS_OUTPUT.PUT_LINE(V_LAST || ', ' || V_FIRST);
 18      END LOOP;
 19      CLOSE EMPLOYEE_CUR;
 20  END;
 21  /
Enter value for department_id: 10
EMPLOYEES IN DEPARTMENT 10
Smith, John
Roberts, Sandi
Chen, Sunny

PL/SQL procedure successfully completed.

SQL>
```

Check your progress 1:

What is the use of the SELECT…FOR UPDATE and WHERE CURRENT Clause in the Cursors?

……

……

……

……

19.4 CURSOR VARIABLES :AN INTRODUCTION

An explicit cursor is the name of the work area for an active set. A cursor variable is a reference to the work area. A cursor is based on one specific query, whereas a cursor variable can be opened with different queries within a program. A static cursor is like a constant, and a cursor variable is like a pointer to that cursor. You can also use the action statements OPEN, FETCH, and CLOSE with cursor variables. The cursor attributes %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT are available for cursor variables. Cursor variables have many similarities with static cursors.

The cursor variable has other capabilities in addition to the features of a static cursor. It is a variable, so it can be used in an assignment statement. A cursor variable can also be assigned to another cursor variable.

REF CURSOR TYPE

Two steps are involved in creating a cursor variable. First, you have to create a referenced cursor type. Second, you have to declare an actual cursor variable with the referenced cursor type.

The general syntax is

TYPE cursortypename IS REF CURSOR [RETURN return type);

cursorvarname cursortypename;

where cursortypename is the name of the type of cursor. The RETURN clause is optional. The returntype is the RETURN data type and can be any valid data structure, such as a record or structure defined with %ROWTYPE. For example,

TYPE any_cursor_type IS REF CURSOR; any_cursor_var any_cursor_type; TYPE employee_cursor_type IS REF CURSOR

RETURN employee%ROWTYPE;

employee_cursor _var employee_cursor _type;

In this example, the first cursor type, any _cursor _type, is called the weak type, because its RETURN clause is missing. This type of cursor type can be used with any query. The cursor type declared with the RETURN clause is called the strong type, because it links a row type to the cursor type at the declaration time.

OPENING A CURSOR VARIABLE

You assign a cursor to the cursor variable when you OPEN it. The general syntax is OPEN cursorname I cursorvarname FOR SELECT query;

If the cursor type is declared with the RETURN clause, the structure from the SELECT query must match the structure specified in the REF CURSOR declaration. For example

OPEN employee_cursor_var FOR SELECT * FROM employee;

The structure returned by the SELECT query matches the RETURN type employee%ROWTYPE.

The other cursor type, any_cursor_type, is declared without the RETURN clause. It can be opened without any worry about matching the query's result to anything. The weak type is more flexible than the strong type, but there is no error checking. Let us look at some OPEN statements for the weak cursor variable:

OPEN any_cursor_var FOR SELECT * FROM dept;

OPEN any_cursor_var FOR SELECT * FROM employee;

OPEN any_cursor_var FOR SELECT Deptld FROM dept;

It is possible to have all three statements in one program block. The cursor variable assumes different structures with each OPEN.

Fetching from a Cursor Variable

The fetching action is same as that of a cursor. The compiler checks the data structure type after the INTO clause to see if it matches the query linked to the cursor.

The general syntax is

FETCH cursorvarname INTO recordname / variablelist;

19.5 LET US SUM UP

Check your progress Answers:

1. The FOR UPDATE clause is used with the SELECT query for row locking.

The WHERE CURRENT OF clause allows us to perform data manipulation only on a recently fetched row.

LESSON 20

PL/SQL : EXCEPTIONS

Contents

20.0 AIMS AND OBJECTIVES

To learn about the concept of error handling and exceptions.

20.1 EXCEPTIONS

In PL/SQL, a warning or error condition is called an exception. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include division by zero and out of memory. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

In the example below, you calculate and store a price-to-earnings ratio for a company with ticker symbol XYZ. If the company has zero earnings, the predefined exception ZERO_DIVIDE is raised. This stops normal execution of the block and transfers control to the exception handlers. The optional OTHERS handler catches all exceptions that the block does not name specifically.

```
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  SELECT price / earnings INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ';  -- might cause division-by-zero error
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
  COMMIT;
EXCEPTION  -- exception handlers begin
  WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
    COMMIT;
  ...
  WHEN OTHERS THEN  -- handles all other errors
    ROLLBACK;
END;  -- exception handlers and block end here
```

The last example illustrates exception handling, not the effective use of INSERT statements. For example, a better way to do the insert follows:

```
INSERT INTO stats (symbol, ratio)
  SELECT symbol, DECODE(earnings, 0, NULL, price / earnings)
  FROM stocks WHERE symbol = 'XYZ';
```

In this example, a subquery supplies values to the INSERT statement. If earnings are zero, the function DECODE returns a null. Otherwise, DECODE returns the price-to-earnings ratio.

Advantages of PL/SQL Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors:

```
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, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
```

Exceptions improve readability by letting you isolate error-handling routines. The primary algorithm is not obscured by error recovery algorithms. 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.

20.2 TYPES OF EXCEPTIONS

20.2.1 Predefined PL/SQL Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

To handle other Oracle errors, you can use the OTHERS handler. The functions SQLCODE and SQLERRM are especially useful in the OTHERS handler because they return the Oracle error code and message text. Alternatively, you can use the pragma EXCEPTION_INIT to associate exception names with Oracle error codes.

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names in the following list:

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-6530	-6530
CASE_NOT_FOUND	ORA-6592	-6592
COLLECTION_IS_NULL	ORA-6531	-6531
CURSOR_ALREADY_OPEN	ORA-6511	-6511
DUP_VAL_ON_INDEX	ORA-0001	-1
INVALID_CURSOR	ORA-1001	-1001
INVALID_NUMBER	ORA-1722	-1722
LOGIN_DENIED	ORA-1017	-1017
NO_DATA_FOUND	ORA-1403	+100
NOT_LOGGED_ON	ORA-1012	-1012
PROGRAM_ERROR	ORA-6501	-6501
ROWTYPE_MISMATCH	ORA-6504	-6504
SELF_IS_NULL	ORA-0625	-30625
STORAGE_ERROR	ORA-6500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-6533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-6532	-6532
SYS_INVALID_ROWID	ORA-1410	-1410

Exception	Oracle Error	SQLCODE Value
TIMEOUT_ON_RESOURCE	ORA-0051	-51
TOO_MANY_ROWS	ORA-1422	-1422
VALUE_ERROR	ORA-6502	-6502
ZERO_DIVIDE	ORA-1476	-1476

Brief descriptions of the predefined exceptions follow:

Exception	Raised when ...
ACCESS_INTO_NULL	Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.
CASE_NOT_FOUND	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	Your program attempts to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	Your program attempts an illegal cursor

Exception	Raised when ...
	operation such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement does not evaluate to a positive number.
LOGIN_DENIED	Your program attempts to log on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls an aggregate function never raises NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	Your program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.

Exception	Raised when ...
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	Your program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	Your program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	Your program attempts to divide a number by zero.

Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements.

20.2.2 Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION. In the following example, you declare an exception named past_due:

```
DECLARE
  past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

```
block_label.exception_name
```

The following example illustrates the scope rules:

```
DECLARE
  past_due EXCEPTION;
  acct_num NUMBER;
BEGIN
  DECLARE  ---------- sub-block begins
    past_due EXCEPTION;  -- this declaration prevails
    acct_num NUMBER;
  BEGIN
    ...
    IF ... THEN
      RAISE past_due;  -- this is not handled
    END IF;
  END;  ------------- sub-block ends
```

```
EXCEPTION
  WHEN past_due THEN  -- does not handle RAISEd exception
    ...
END;
```

The enclosing block does not handle the raised exception because the declaration of past_due in the sub-block prevails. Though they share the same name, the two past_due exceptions are different, just as the two acct_num variables share the same name but are different variables. Therefore, the RAISE statement and the WHEN clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an OTHERS handler.

20.2.3 Non-Predefined Exceptions

Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT

To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A pragma is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where exception_name is the name of a previously declared exception and the number is a negative value corresponding to an ORA- error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
  ... -- Some operation that causes an ORA-00060 error
EXCEPTION
  WHEN deadlock_detected THEN
    -- handle the error
END;
```

Defining Your Own Error Messages:

Procedure RAISE_APPLICATION_ERROR

The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA- error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call RAISE_APPLICATION_ERROR, use the syntax

raise_application_error(error_number, message[, {TRUE | FALSE}]);

where error_number is a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and as with package STANDARD, you do not need to qualify references to it.

An application can call raise_application_error only from an executing stored subprogram (or method). When called, raise_application_error ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In the following example, you call raise_application_error if an employee's salary is missing:

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS
  curr_sal NUMBER;
BEGIN
  SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
  IF curr_sal IS NULL THEN
    /* Issue user-defined error message. */
    raise_application_error(-20101, 'Salary is missing');
  ELSE
    UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
  END IF;
END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Also, it can use the pragma EXCEPTION_INIT to map specific error numbers returned by raise_application_error to exceptions of its own, as the following Pro*C example shows:

```
EXEC SQL EXECUTE
```

```
  /* Execute embedded PL/SQL block using host
     variables my_emp_id and my_amount, which were
     assigned values in the host environment. */
  DECLARE
     null_salary EXCEPTION;
     /* Map error number returned by raise_application_error
        to user-defined exception. */
     PRAGMA EXCEPTION_INIT(null_salary, -20101);
  BEGIN
     raise_salary(:my_emp_id, :my_amount);
  EXCEPTION
     WHEN null_salary THEN
        INSERT INTO emp_audit VALUES (:my_emp_id, ...);
  END;
END-EXEC;
```

This technique allows the calling application to handle error conditions in specific exception handlers.

Redeclaring Predefined Exceptions

Remember, PL/SQL declares predefined exceptions globally in package STANDARD, so you need not declare them yourself. Redeclaring predefined exceptions is error prone because your local declaration overrides the global declaration. For example, if you declare an exception named invalid_number and then PL/SQL raises the predefined exception INVALID_NUMBER internally, a handler written for INVALID_NUMBER will not catch the internal exception. In such cases, you must use dot notation to specify the predefined exception, as follows:

```
EXCEPTION
  WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
     -- handle the error
END;
```

How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using EXCEPTION_INIT. However, other user-defined exceptions must be raised explicitly by RAISE statements.

Raising Exceptions with the RAISE Statement

PL/SQL blocks and subprograms should raise an exception only when an error makes it undesirable or impossible to finish processing. You can place RAISE statements for a given exception anywhere within the scope of that exception. In the following example, you alert your PL/SQL block to a user-defined exception named out_of_stock:

```
DECLARE
  out_of_stock   EXCEPTION;
  number_on_hand NUMBER(4);
BEGIN
  …
  IF number_on_hand < 1 THEN
    RAISE out_of_stock;
  END IF;
EXCEPTION
  WHEN out_of_stock THEN
    -- handle the error
END;
```

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as the following example shows:

```
DECLARE
  acct_type INTEGER := 7;
BEGIN
  IF acct_type NOT IN (1, 2, 3) THEN
    RAISE INVALID_NUMBER;  -- raise predefined exception
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
    ROLLBACK;
END;
```

How PL/SQL 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 latter case, PL/SQL returns an unhandled exception error to the host environment.

However, exceptions cannot propagate across remote procedure calls (RPCs). Therefore, a PL/SQL block cannot catch an exception raised by a remote subprogram.

Figure 20.1, Figure 20.2, and Figure 20.3 illustrate the basic propagation rules.

Figure 20.1 Propagation Rules: Example 1

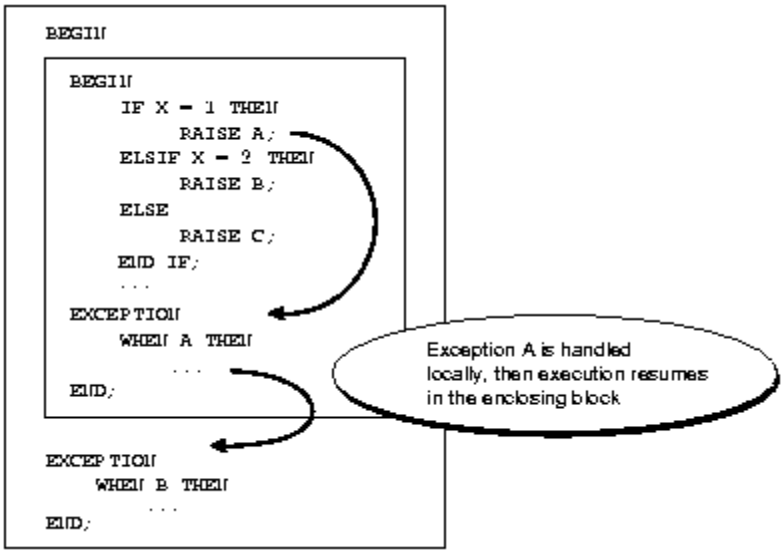


Figure 20.2 Propagation Rules: Example 2

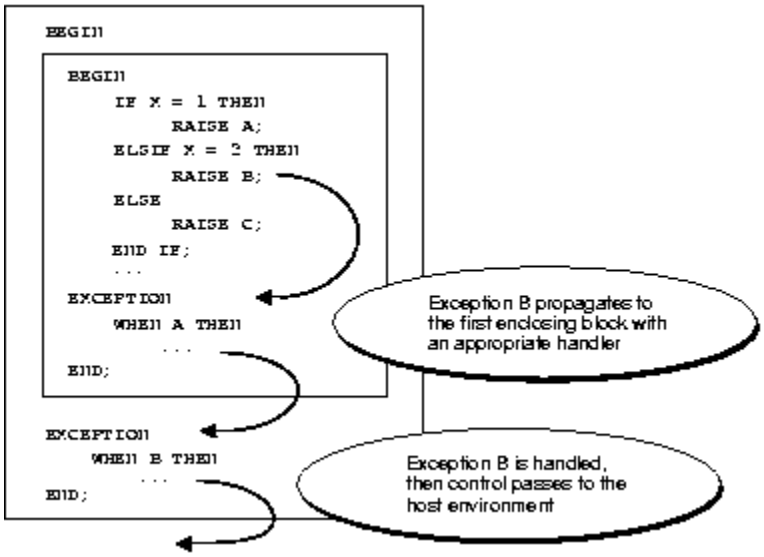
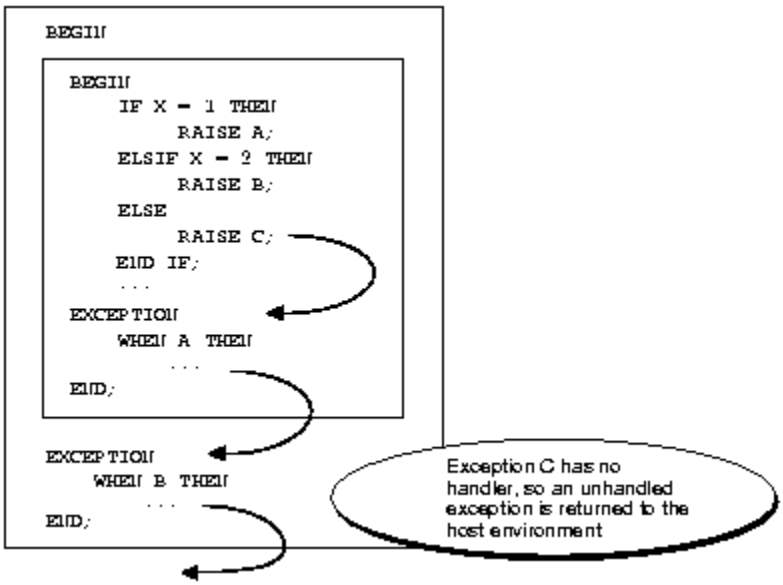


Figure 20.3 Propagation Rules: Example 3



An exception can propagate beyond its scope, that is, beyond the block in which it was declared. Consider the following example:

```
BEGIN
  ...
  DECLARE  ---------- sub-block begins
    past_due EXCEPTION;
  BEGIN
    ...
    IF ... THEN
      RAISE past_due;
    END IF;
  END;  ------------- sub-block ends
EXCEPTION
  ...
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

Because the block in which exception past_due was declared has no handler for it, the exception propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an OTHERS handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets the following error:

ORA-06510: PL/SQL: unhandled user-defined exception

Reraising a PL/SQL Exception

Sometimes, you want to *reraise* an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, simply place a RAISE statement in the local handler, as shown in the following example:

```
DECLARE
  out_of_balance  EXCEPTION;
BEGIN
  ...
  BEGIN  ---------- sub-block begins
    ...
    IF ... THEN
      RAISE out_of_balance;  -- raise the exception
    END IF;
  EXCEPTION
    WHEN out_of_balance THEN
      -- handle the error
      RAISE;  -- reraise the current exception
  END;  ------------ sub-block ends
EXCEPTION
  WHEN out_of_balance THEN
    -- handle the error differently
  ...
END;
```

Omitting the exception name in a RAISE statement--allowed only in an exception handler--reraises the current exception.

Handling Raised PL/SQL Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```
EXCEPTION
  WHEN exception_name1 THEN  -- handler
    sequence_of_statements1
  WHEN exception_name2 THEN  -- another handler
    sequence_of_statements2
  ...
  WHEN OTHERS THEN           -- optional handler
    sequence_of_statements3
END;
```

To catch raised exceptions, you write exception handlers. Each handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Thus, a block or subprogram can have only one OTHERS handler.

As the following example shows, use of the OTHERS handler guarantees that *no* exception will go unhandled:

```
EXCEPTION
  WHEN ... THEN
    -- handle the error
  WHEN ... THEN
    -- handle the error
  WHEN OTHERS THEN
    -- handle all other errors
END;
```

If you want two or more exceptions to execute the same sequence of statements, list the exception names in the WHEN clause, separating them by the keyword OR, as follows:

```
EXCEPTION
  WHEN over_limit OR under_limit OR VALUE_ERROR THEN
    -- handle the error
```

If any of the exceptions in the list is raised, the associated sequence of statements is executed. The keyword OTHERS cannot appear in the list of exception names; it must appear by itself. You can have any number of exception handlers, and each handler can associate a list of exceptions with a sequence of statements. However, an exception name can appear only once in the exception-handling part of a PL/SQL block or subprogram.

The usual scoping rules for PL/SQL variables apply, so you can reference local and global variables in an exception handler. However, when an exception is raised inside a cursor FOR loop, the cursor is closed implicitly before the handler is invoked. Therefore, the values of explicit cursor attributes are *not* available in the handler.

Handling Exceptions Raised in Declarations

Exceptions can be raised in declarations by faulty initialization expressions. For example, the following declaration raises an exception because the constant credit_limit cannot store numbers larger than 999:

```
DECLARE
  credit_limit CONSTANT NUMBER(3) := 5000;  -- raises an exception
BEGIN
  ...
EXCEPTION
  WHEN OTHERS THEN -- cannot catch the exception
    ...
END;
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates *immediately* to the enclosing block.

Handling Exceptions Raised in Handlers

Only one exception at a time can be active in the exception-handling part of a block or subprogram. So, an exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for the newly raised exception. From there on, the exception propagates normally. Consider the following example:

```
EXCEPTION
  WHEN INVALID_NUMBER THEN
    INSERT INTO ...  -- might raise DUP_VAL_ON_INDEX
  WHEN DUP_VAL_ON_INDEX THEN ...  -- cannot catch the exception
END;
```

Branching to or from an Exception Handler

A GOTO statement cannot branch into an exception handler. Also, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is illegal:

```
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ';
  <<my_label>>
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    pe_ratio := 0;
    GOTO my_label;  -- illegal branch into current block
END;
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

Check your progress 1:

What are the types of exceptions?

……
……

20.2.4 Error Message: SQLCODE and SQLERRM

Retrieving the Error Code and Error Message: SQLCODE and SQLERRM

In an exception handler, you can use the built-in functions SQLCODE and SQLERRM to find out which error occurred and to get the associated error message. For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, SQLCODE returns +1 and SQLERRM returns the message: User-Defined Exception.Unless you used the pragma EXCEPTION_INIT to associate the exception name with an Oracle error number, in which case SQLCODE returns that error number and SQLERRM returns the corresponding error message. The maximum length of an Oracle error message

is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message: ORA-0000: normal, successful completion.

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

```
DECLARE
  err_msg VARCHAR2(100);
BEGIN
  /* Get all Oracle error messages. */
  FOR err_num IN 1..9999 LOOP
    err_msg := SQLERRM(err_num); -- wrong; should be -err_num
    INSERT INTO errors VALUES (err_msg);
  END LOOP;
END;
```

Passing a positive number to SQLERRM always returns the message user-defined exception unless you pass +100, in which case SQLERRM returns the message no data found. Passing a zero to SQLERRM always returns the message normal, successful completion.

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
  err_num NUMBER;
  err_msg VARCHAR2(100);
BEGIN
  ...
EXCEPTION
  WHEN OTHERS THEN
    err_num := SQLCODE;
    err_msg := SUBSTR(SQLERRM, 1, 100);
    INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. The functions SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

Note: When using pragma RESTRICT_REFERENCES to assert the purity of a stored function, you cannot specify the constraints WNPS and RNPS if the function calls SQLCODE or SQLERRM.

Catching 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 Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block 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 (unless they are NOCOPY parameters). Also, if a stored subprogram fails with an unhandled exception, PL/SQL does not roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an OTHERS handler at the topmost level of every PL/SQL program.

20.4 LET US SUM UP

Check your progress Answers :

1. The types of exceptions are Predefined Exceptions like NO_DATA_FOUND, LOGIN_DENIED, TOO_MANY_ROWS etc. and the non predefined exceptions such as PRAGMA EXCEPTION_INIT.

UNIT V

LESSON 21

PL/SQL COMPOSITE DATA TYPES : RECORDS

Contents

21.0 AIMS AND OBJECTIVES

To learn about composite data types in PL/SQL, the basics of a PL/SQL record structure and its declaration, assignment of a value and use in a program.

21.1 INTRODUCTION

Composite data types are like scalar data types. Scalar data types are atomic, because they do not consist of a group. Composite data types, on the other hand, are groups, or "collections." Examples of composite data types are RECORD, TABLE, nested TABLE, and VARRAY. In this chapter, we will talk about all four composite data types.

21.2 PL/SQL RECORDS

PL/SQL records are similar in structure to a row in a database table. A record consists of components of any scalar, PL/SQL record, or PL/SQL table type. These components are known as fields, and they have their own values. PL/SQL records are similar in structure to "struct" in the C language. The record does not have a value as a whole; instead, it enables you to access these components as a group. It makes your life easier by transferring the entire row into a record rather than each column into a variable separately.

A PL/SQL record is based on a cursor, a table's row, or a user-defined record type. You learned about a record in a cursor FOR loop in the previous chapter. A record can be explicitly declared based on a cursor or a table:

CURSOR cursorname IS

SELECT query;

Recordname CursorName%ROWTYPE;

A record can also be based on another composite data type called TABLE. We will examine user-defined records in the next section.

21.2.1 Creating a PL/SQL Record

In this section, you will learn to create a user-defined record. You create a RECORD type first, and then you declare a record with that RECORD type. The general syntax is

TYPE recorcftypename IS RECORD

(fleidnamel datatype I variabie% TYPE I table.column% TYPE I

table%ROWTVPE ((NOT NULLI := (DEFAULT Expressioni

1, fieldname2...

FieldNanie3...);

recordname recordtypename;

For example,

TYPE employee_rectype IS RECORD

(elast VARCHAR2(1 5),

e_first VARCHAR2(1 5),

e_sal NUMBER(8,2));

employee_rec employee_rectype;

In this declaration, employee_rectype is the user-defined RECORD type. Three fields are included in its structure; e_last, e_first, and e_sal. The record employee_rec is a record declared with the user-defined record type employee jectype. Each field declaration is similar to a scalar variable declaration.

Now, you will look at another declaration with the %TYPE attribute. For example,

TYPE employeejectype IS RECORD

(ejd NUMBER(3) NOT NULL 111,

elast employee. Lname%TYPE,

e_first employee.Fname%TYPE,

e_sal emptoyee.Salary%TYPE);

employee jec employee_rectype;

The NOT NULL constraint can be used for any field to prevent Null values, but that field must be initialized with a value.

21.2.2 Referencing Fields in a Record

A field in a record has a name that is given in the RECORD-type definition. You cannot reference a field by its name only; you must use the record name as a qualifier:

recordname.fieldname

The record name and field name are joined by a dot (.). For example, you can reference the e_sal field from the previous declaration as

employee_rec.e_sal

You can use a field in an assignment statement to assign a value to it. For example,

employee_rec.esal := 100000;

employee_rec.eJast 'Jordan';

9.2.3 Working with Records

A record is known in the block where it is declared. When the block ends, the record no longer exists. You can assign values to a record from columns in a row by using the SELECT statement or the FETCH statement. The order of fields in a record must match the order of columns in the row. A record can be assigned to another record if both records have the same structure.

A record can be set to NULL, and all fields will be set to NULL. However, do not try to assign a NULL to a record that has fields with the NOT NULL constraint.

For example,

Employee_rec NULL

The record declared with %ROWTYPE has the same structure as the table's row.

For example,

emp_rec employee%ROWTYPE;

Here, emp_rec assumes the structure of the EMPLOYEE table. The fields in emp_rec take their column names and their data types from the table. It is advantageous to use %ROWI'YPE, because it does not require you to know the column names and their data types in the underlying table. If you change the data type and/or size of a column, the record is created at execution time and is defined with the updated table structure. The fields in the record declared with %ROWTYPE are referenced with the qualified name recordname.fieldname.

The program in Figure 21-1 declares a record with a record type. The SELECT query retrieves a row, into the record based on the student ID entered at the prompt. The fields in the record are printed using the recordname.fleldname notation.

```
SQL> DECLARE
2 TYPE STUDENT_RECORD_TYPE IS RECORD
3 (S_LAST VARCHAR2(15),
4 S_FIRST VARCHAR2(15),
5 S_PHONE VARCHAR2(10));
6 STUDENT_REC STUDENT_RECORD_TYPE;
7 BEGIN'
8 SELECT LAST, FIRST, PHONE INTO STUDENT_REC
9 FROM STUDENT WHERE STUDENTID = '&STUDJD';
10 DBMS_OUTPUT.PUT_LINE(STUDENT_REC.S_LAST II',
11 II STUDENT_REC.S_FIRST II --> ' II
12 STUDENT_REC.S_PIIONE);
13 END;
14 /
Enter value for stud_id: 00100
Diaz, Jose --> 9735551111
PLISQL procedure successfully completed.
SQL>
```

Figure 21-1 PL/SQL record.

Check your progress 1:

What is a Record?

……
……
……
……

21.2.4 Nested Records

You can create a nested record by including a record into another record as a field. The record that contains another record as a field is called the enclosing record. For example,

```
DECLARE
TYPE address_rectype IS RECORD
(first VARCHAR2(15),
last VARCHAR2(1 5),
Street  VARCHAR2(25),
City   VARCHAR2(1 5),
State   CHAR (2),
Zip   CHAR (5));

TYPE all_address_rectype IS RECORD
(home_address address_rectype,
bus_address address_rectype,
vacation_address address_rectype);
address_rec aII_address_rectype;
```

In this example, all_address_rectype nests address_rectype as a field type. If you decide to use an unnested simple record, the record becomes cumbersome. There are six fields in address _rectype. You will have to use six fields each for each of the three record fields home_address, bus_address, and vacation_address, which will result in a total of 18 fields.

Nesting records makes code more readable and easier to maintain. You can nest records to multiple levels. Dot notation is also used to reference fields in the nested situation. For example, address_rec.home_address.city references a field called city in the nested record home_address, which is enclosed by the record address_rec.

21.3 LET US SUM UP

Check your progress :Answers:

1. A record consists of components of any scalar, PL/SQL record, or PL/SQL table type. These components are known as fields, and they have their own values. PL/SQL records are similar in structure to “struct” in the C language.

LESSON 22

PL/SQL COMPOSITE DATA TYPES : TABLES & VARRAYS

Contents

22.0 AIMS AND OBJECTIVES

To learn about composite data types in PL/SQL, the basics of a PL/SQL table data type together with its declaration, referencing, and types of assignments.

Built-in methods to obtain table information are outlined. A complex structure, a table of records, is covered. Variable-sized arrays, or varrays, are introduced.

22.1 PL/SQL TABLES

A table, like a record, is a composite data structure in PL/SQL. A PL/SQL-table is a single-dimensional structure with a collection of elements that store the same type of value. In other words, it is like an array in other programming languages. If you know COBOL, arrays are called tables in COBOL terminology, although there are dissimilarities between a traditional array and a PL/SQL table. A table is a dynamic structure that is not constrained, whereas an array is not dynamic in most computer languages.

22.1.1 Declaring a PL/SQL Table

A PL/SQL TABLE declaration is done in two steps, like a record declaration:

1. Declare a PL/SQL table type with a TYPE statement. The structure could use any of the scalar data types.

The general syntax is

2. Declare an actual table based on the type declared in the previous step.

TYPE tabletypename IS TABLE OF

datatype I variablename% TYPE I tablename.columnname% TYPE (NOT NULLI INDEX BY BINARY JNTEGER;

For example,

TYPE deptname_table_type IS TABLE OF deptDeptName%TYPE INDEX BY BINARY INTEGER;

TYPE major_table_type IS TABLE OF VARCHAR2(50)

INDEX BY BINARYINTEGER;

You can declare a table type with a scalar data type (VARCHAR2, DATE, BOOLEAN, or POSITIVE) or with the declaration attribute %TYPE. Optionally, you can use NOT NULL in a declaration, which means that none of the elements in the table may have a Null value. You must, however, add an INDEX BY BINARY_INTEGER clause to the declaration. This is the only available clause for indexing a table at present. Indexing speeds up the search process from the table. The primary key is stored internally in the table along with the data column. The table consists of two columns, the index/primary key column and the data column.

You define the actual table based on the table type declared earlier. The general syntax is

tablename tabletypename;

For example,

deptname_table deptname_table_type;

major_table major_table_type;

Figure 22.1 illustrates a table's structure. It contains a primary key column and a data column. You cannot name these columns. The primary key has the type BINARY_INTEGER, and the data column is of any valid type. There is no limit on the number of elements, but you cannot initialize elements of a table at declaration time.

PRIMARY KEY COLOUMN	DATA COLUMN
……	…..
1	Sales
2	Marketing
3	Information Systems
4	Finance
5	Production
…….	……

Figure 22.1 PL/SQL table structure.

22.1.2 Referencing Table ElementslRows

The rows in a table are referenced in the same way that an element in an array is referenced. You cannot reference a table by its name only. You must use the primary key value in a pair of parentheses as its subscript or index:

tablename (primarykeyvalue)

The following are valid assignments for the table's rows:

deptname_tabte(5) := 'Human Resources':

major_table(100):= vmajor;

You can use an expression or a value other than a BINARY_INTEGER value, and PL/SQL will convert it. For example,

```
/* 25.7 is rounded to 26 */
Deptname_table(25.7) :='Training';
/* '5' || '00' is converted to 500. */
deptname_table ('5' ||'00') := 'Research';
/* v_num + 7 is evaluated. */
deptname_table(v_num + 7) := 'Development';
```

In other programming languages, such as C or Visual Basic, you specify the number of elements in an array when you declare the array. The memory locations re reserved for elements in an array at the declaration time. In a PL/SQL table, the primary key values are not preassigned. A row is created when you assign a value to it. If a row does not exist and you try to access it, the PL/SQL predefined server exception NO_DATA_FOUND is raised. You can keep track of rows' primary key values if you use them in a sequence and keep track of the minimum and the maximum value.

22.1.3 Assigning Values to Rows in a PL/SQL Table

You can assign values to the rows in a table in three ways:

1. Direct assignment.

2. Assignment in a loop.

3. Aggregate assignment.

Direct Assignment. You can assign a value to a row with an assignment statement, as you already learned in the previous topic. This is preferable if only a few assignments are to be made. If an entire database table's values are to be assigned to a table, however, a looping method is preferable.

Assignment in a Loop. You can use any of the three PL/SQL loops to assign values to rows in a table. The program block in Figure 9-3 assigns all Sunday

dates for the year 2004 to a table. The primary key index value will vary from 1 to 52. The table column will contain dates for 52 Sundays. If you are innovative, you can create great applications with loops and tables.

```
SQL> DECLARE
2 TYPE DATE_TABLE_TYPE IS TABLE OF DATE
3 INDEX BY BINARY_INTEGER;
4 SUNDAY_TABLE DATE_TABLE_TYPE;
5 V_DAY BINARYJNTEGER := 1;
6 V_DATE DATE;
7 V_COUNT NUMBER(3) := 1;
8 BEGIN
9 V_DATE '01-JAN-2004';
10 WHILE V_COUNT <= 365 LOOP
11 IF UPPER(TO_CHAR(V_DATE, 'DAY')) LIKE '%SUNDAY%' THEN
12 SUNDAY_TABLE (V_DAY) := V_DATE;
13 DBMS_OUTPUT.PUT_LINE
14 (TO_CHAR (SUNDAY_TABLE (V_DAY). 'MONTH DD, YYYY'));
15 V_DAY V_DAY +1;
16 END IF;
17 V_COUNT v_COUNT +1;
18 V_DATE := V_DATE +1;
19 END LOOP;
20 END;
21 /
JANUARY 04, 2004
JANUARY 11, 2004
JANUARY 18, 2004
JANUARY 25, 2004
FEBRUARY 01, 2004
FEBRUARY 08, 2004
FEBRUARY 15, 2004
FEBRUARY 22, 2004
FEBRUARY 29, 2004
```

DECEMBER 19, 2004

DECEMBER 26, 2004

PL/SQL procedure successfully completed.

SQL>

Figure 9-3 Table row assignment in a loop.

Aggregate Assignment. You can assign a table's values to another table. The data types of both tables must be compatible. When you assign a table's values to another table, the table receiving those values loses all its previous primary key values as well as its data column values. If you assign an empty table with no rows to another table with rows, the recipient table is cleared. In other words, it loses all its rows. Both tables must have the same type for such an assignment.

22.1.4 Built-In Table Methods

The built-in table methods are procedures or functions that provide information about a PL/SQL table. Figure 22.2 lists the built-in table methods and their use. The general syntax is

tabiename.methodname [(indexi [, index2j)j

where methodname is one of the methods described in Figure 13-4. For example,

```
total_rows := deptname_table.COUNT;          /* counts elements */
deptname_table.DELETE(5);                        /*deletes element 5 */
deptname_table.DELETE(7, 10);                /* deletes element 7 to 10 */
deptname_table.DELETE;                       /*deletes all elements*/
next_row := deptname_table.NEXT(25);         /* index after 25 */
previous_row := deptname_table.PRIOR(7);     /* index before 7 */
first_row := deptname_table.FIRST;               /*smallest index */
last_row := deptname_table.LAST;                 /* largest index */
IF deptname_table.EXISTS(11) THEN            /* true, if index 11 exists */
```

Built-in-Method	Use
FIRST	Returns the smallest index number in a PL/SQL table.
LAST	Returns the largest index number in a PL/SQL table.
COUNT	Returns the total number of elements in a PL/SQL table.
PRIOR(n)	Returns the index number that is before index number n.

NEXT(n)	Returns the index number that is after index number n.
EXISTS(n)	Returns TRUE if index n exists in the table.
TRIM	Removes one element from end of the table.
TRIM (n)	Removes n elements from end of the table.
DELETE	Removes all elements from a PL/SQL table.
DELETE (n)	Removes the n-th element from the table.
DELETE (m, n)	Removes all elements in the range m. . . n from a table
EXTEND	Appends a null element to a table.
EXTEND (n)	Appends n null elements to a table
EXTEND (n, x)	Appends n copies of the x-th element to a table.

Figure 22-2 PL/SQL built-in table methods.

Now, look at another example of TABLE, in Figure 22.3, where the program declares two' table types. The two tables based on these TABLE types are parallel tables. The corresponding values in the two tables are related. The program populates these two tables using a cursor FOR loop. Then, another simple FOR loop is used to print information from the two tables.

```
SQL> DECLARE
2 TYPE LNAME_TABLE_TYPE IS TABLE OF EMPLOYEE INAME%TYPE
3 INDEX BY BINARYJNTEGER;
4 NAME_TABLE LNAME_TABLE_TYPE;
5 TYPE SALARY_TABLE_TYPE IS TABLE OF EMPLOYEE.SALARY%TYPE
6 INDEX BY BINARYJNTEGER;
7 SALARY_TABLE SALARY_TABLE_TYPE;
8 C BINARYJNTEGER := 0;
9 CURSOR C_LASTSAL IS
10 SELECT LNAME, SALARY FROM EMPLOYEE;
11 V_TOT NUMBER(2);
12 BEGIN
13 SELECT COUNT(*) INTO V_TOT FROM EMPLOYEE;
14 /* TABLE ASSIGNMENT IN LOOP.*I [
15 FOR LASTSAL_REC IN C_LASTSAL LOOP
16 C:=C+1;
17 NAME_TABLE(C) := LASTSAL_REC.LNAME;
```

```
18 SALARY_TABLE(C) := LASTSAL_REC.SALARY;
19 END LOOP;
20 /* PRINTING TABLE IN LOOP */
21 FOR C IN 1..V_TOT LOOP
22 DBMS_OUTPUT.PUT_LINE(NAME_TABLE(C) II' • II
23 TO_CHAR(SALARY_TABI.E(C), '$999,999.99'));
24 END LOOP;
25 END;
26 /
Smith $291,500.00
Houston $160,500.00
Roberts $80,250.00
McCall $66,500.00
Dev $85,600.00
Shaw $24,500.00
Garner $48,150.00
Chen $35,000.00
PL/SQI. procedure successfully completed.
SQL>
```

Figure 22.3 PL/SQL table.

22.1.5 Table of Records

The PL/SQL table type is declared with a data type. You may use a record type as a table's data type. The %ROWTYPE declaration attribute can be used to define the record type. When a table is based on a record, the record must consist of fields with scalar data types. The record must not contain a nested record. The following example show different ways to declare table types based on records:

- A PL/SQL table type based on a programmer-defined record:

```
TYPE student_record_type IS
        RECORD (stujd NUMBER(3), stu_name VARCHAR2(30));
TYPE student_table_type IS TABLE OF student_record_type
        INDEX BY BINARYJNTEGER;
Student_table student_table_type;
```

- A PL/SQL table type based on a database table:

```
TYPE employee_table_type IS TABLE OF employee%ROWTYPE
    INDEX BY BINARYJNTEGER;
Employee_table employee_table_type;
```

- A PL/SQL table type based on a row returned by a cursor:

```
CURSOR employee_cur IS SELECT * FROM employee;
TYPE employee_cur_table_type IS employee_cur%ROWTYPE
    INDEX BY BINARYJNTEGER;
Employee_cur_table employee_cur_table_type;
```

The %ROWTYPE attribute is not used when the table is based on a user defined record. You use the %ROWTYPE attribute when the table is based on a database table or a cursor.

The fields of a PL/SQL table based on a record are referenced with the following syntax:

```
tablename (index).fieldname
```

For example,

```
Student_table(10).stu_name := 'Ephrem'
Employee_table(13).Salary := 50000;
```

22.2 PL/SQL VARRAYS

A varray is another composite data type or collection type in PL/SQL. Varray stands for variable-size array. They are single-dimensional, bounded collections of elements with the same data type. They retain their ordering and subscripts when stored in and retrieved from a database table. They are similar to a PL/SQL table, and each element is assigned a subscript/index starting with 1.

A PL/SQL VARRAY declaration is done in two steps, like a table declaration:

1. Declare a PL/SQL VARRAY type with a TYPE statement. The TYPE declaration includes a size to set the upper bound of a varray. The lower bound is always one.

2. Declare an actual varray based on the type declared in the previous step.

The general syntax is

```
DECLARE
TYPE varraytypename IS VARRAY (size) OF ElementType (NOT NULL);
varrayname varraytypename;
```

For example,

DECLARE

TYPE Lname_varraytype S VARRAY(5) OF employee.LName%TYPE;

Lname_varray Lname_varray_type := Lname_varray_typeO;

When a varray is declared, it is NULL. It must be initialized before referencing its elements. In the second step of a varray's declaration, the assignment part initializes it. The EXTEND method is used before adding a new element to a varray. In the example above, the upper bound would be five, which limits number of elements to five.

In Figure 9-6, COURSEID_VARRAYJYPE is declared with upper bound of 10. Next, the COURSEID_VARRAY is declared with the varray type and then initialized. A cursor FOR loop then adds elements to the varray. Notice the use of the EXTEND method before assigning a value to the new element. The COUNT method returns the number of elements, the LIMIT method the upper bound, the FIRST method the first subscript, and the LAST method the last subscript.

In Oracle9i, it is possible to create a collection of a collection (multilevel collection) like a varray of varrays. For example,

DECLARE

TYPE varray_typel IS VARRAY(3) OF NUMBER;

TYPE varray_type2 IS VARRAY(2) of varray_type1;

In Figure 22.4, Vi is a varray, and V2 is a varray of varray Vi. Varray Vi contains three elements, and varray V2 contains six elements (2 * 3 = 6). Elements of varray Vi are referenced with one subscript, but elements of varray V2 are referenced with two subscripts.

There is one more type of collection in PL/SQL. Nested tables use a column that has a table type as its data type and are single-dimensional, unbounded collections of elements with the same data type. A nested table can be used in PL/SQL as well as a database table. You can use a column that has a table type as its data type in a database table.

Check your progress 1:

What is a varray?

.....

.....

.....

.....

.....

.....

```
SQL> DECLARE
2 CURSOR COURSE_CUR IS
3 SELECT COURSEID
4 FROM COURSE
5 WHERE ROWNUM <= 5;
6 TYPE COURSEID_VARRAY_TYPE IS VARRAY(10) OF
COURSE.COURSEID%TYPE;
7 COURSEID_VARRAY COURSEID_VARRAY_TYPE :=
COURSEID_VARRAY_TYPEO;
8 V_COUNT NUMBER(1) := 1;
9 BEGIN
10 FOR COURSE_REC IN COURSE_CUR LOOP
11 COURSEID_VARRAY.EXTEND;
12 COURSEID_VARRAY(V_COUNT) := COURSE_REC.COURSEID;
13 DBMS_OUTPUT.PUT_LINE
14 ('Courseid(' || V_COUNT || '):
15 II COURSEID_VARRAY(V_COUNT));
16 V_COUNT := V_COUNT + 1;
17 END LOOP;
18 DBMS_OUTPUT.PUT_LINE
19 ('NUMBER OF ELEMENTS: ' || COURSEID_VARRAY.COUNT);
20 DBMS_OUTPUT.PUT_LINE
21 ('LIMIT ON ELEMENTS: ' || COURSEID_VARRAY.LIMIT);
22 DBMS_OUTPUT.PUT_LINE
23 ('FIRST ELEMENTS: || COURSEID_VARRAY.FIRST);
24 DBMS_OUTPUT.PUT_LINE
25 ('LAST ELEMENTS: ' || COURSEID_VARRAY.LAST);
26 END;
27 /
Courseid(1): EN100
Courseid(2): LA123
Courseid(3): C1S253
Courseid(4): C1S265
```

```
Courseid(5): MA1 50
NUMBER OF ELEMENTS: 5
LIMIT ON ELEMENTS: 10
FIRST ELEMENTS: 1
LAST ELEMENTS: 5
PL/SQL procedure successfully completed.
SQL>
```

Figure 22-4 PL/SQL Varray.

```
SQL> DECLARE
2 TYPE V_TYPE1 IS VARRAY(3) OF NUMBER;
3 TYPE V_TYPE2 IS VARRAY(2) OF V_TYPE1;
4 Vi V_TYPE1 := V_TYPE1(10, 20, 30);
5 V2 V_TYPE2 := V_TYPE2(V1);
6 BEGIN
7 DBMS_OUTPUT_LINE('VARRAY:');
8 FOR I IN 1.3 LOOP
9 DBMS_OUTPUT.PUT_LINE('VI(' 11111 ')=' || Vi(I));
10 END LOOP;
11 V2.EXTEND;
12 V2(2) := V_TYPE 1(100, 200. 300);
13 DBMS_OUTPUT.PUT_LINECVARRAY OF VARRAY');
14 FOR I IN 1..2 LOOP
15 FOR J IN 1..3 LOOP
16 DBMS_OUTPUT.PUT_LINE
17 ('V2(' || I || ')('|| J || ')=' || V2(I)(J));
18 END LOOP;
19 END LOOP;
20 END;
21 /
```

VARRAY:

V1(1) = 10

V1(2) = 20

V1(3) = 30

VARRAY OF VARRAY:

V2(1)U) = 10

V2(1)(2) = 20

V2(1)(3) = 30

V2(2)(i) = 100

V2(2)(2) = 200

V2(2)(3) = 300

PL/SQL procedure successfully completed.

SQL>

Figure 22-5 Varray of varrays.

9.5 LET US SUM UP

Check your progress : Answer

1. A varray is a composite data type or collection type in PL/SQL. Varray stands for variable-size array. They are single-dimensional, bounded collections of elements with the same data type.

LESSON 23

PL/SQL NAMED BLOCKS : PROCEDURES AND FUNCTIONS

Contents

23.0 AIMS AND OBJECTIVES

To learn about PL/SQL modules, also known as named blocks, the basics of a named module called a procedure, Structure of a package and triggers.

23.1 INTRODUCTION

The anonymous block does not have a name, and it cannot be called by another block in the program. It cannot take arguments from another block, either. An anonymous block can call other types of PL/SQL blocks called procedures and functions. The procedures and functions are named blocks, and they can be called with parameters. An anonymous block can be nested within a procedure, function, or another anonymous block. The purpose of a procedure or function call is to modularize a PL/SQL program. A named PL/SQL block is compiled when it is created or when it is altered. The compilation process consists of three steps: syntax error checking, binding, and p-code creation. A syntactically error-free program's variables are assigned storage in the binding stage. Then, the list of instructions, called p-code, is generated for the PL/SQL engine. P-code is stored in the database for all named blocks.

23.2 PROCEDURES

A procedure is a named PL/SQL program block that can perform one or more tasks. A procedure is the building block of modular programming. The general syntax of a procedure is

CREATE (OR REPLACE PROCEDURE procedurename

[(parameterl 1, parameter2 . . .1)]

Is

(constant/variable declarations]

BEGIN

executable statements

[EXCEPTION

exception handling statements I]

END [procedurename];

where procedurename is a user-supplied name that follows the rules used in naming identifiers. The parameter list has the names of parameters passed to the procedure by the calling program as well as the information passed from the procedure to the calling program. The local constants and variables are declared after the reserved word IS. If there are no local identifiers to declare, there is nothing between the reserved words IS and BEGIN. The executable statements are written after BEGIN and before EXCEPTION or END. There must be at least one executable statement in the body. The reserved word EXCEPTION and the exception-handling statements are optional.

23.2.1 Calling a Procedure

A call to the procedure is made through an executable PL/SQL statement. The procedure is called by specifying its name along with the list of parameters (if any) in parentheses. The call statement ends with a semicolon (;). The general syntax is

procedurename [(parameterl,. ..)];

For example,

monthly_salary(v_salary);

calculate_net(v_monthIysaIarc 0.28); display_messages;

In these examples of procedure calls, parameters are enclosed in parentheses. You can use a variable, constant, expression, or literal value as a parameter. If you are not passing any parameters to a procedure, parentheses are not needed.

Procedure Header

The procedure definition that comes before the reserved word IS is called the procedure header. The procedure header contains the name of the procedure and the parameter list with data types (if any). For example,

CREATE OR REPLACE PROCEDURE monthly_salary

(v_salaryjn IN employee.Salary%TYPE)

CREATE OR REPLACE PROCEDURE calculate_net

(v_monthly_salary_in IN employee.Salary%TYPE,

v_taxratejn IN NUMBER)

CREATE OR REPLACE PROCEDURE display_messages

The procedure headers in the examples are based on the procedure calls shown previously. The parameter list in the header contains the name of a parameter along with its type. The parameter names used in the procedure header do not have to be the same as the names used in the call. The number of parameters in the call and in the header must match, and the parameters must be in the same order.

Procedure Body

The procedure body contains declaration, executable, and exception-handling sections. The declaration and exception-handling sections are optional. The executable section contains action statements, and it must contain at least one.

The procedure body starts after the reserved word IS. If there is no local declaration, IS is followed by the reserved word BEGIN. The body ends with the reserved word END. There can be more than one END statement in the program, so it is a good idea to use the procedure name as the optional label after END.

Parameters

Parameters are used to pass values back and forth from the calling environment to the Oracle server. The values passed are processed and/or returned with a procedure execution. There are three types of parameters: IN, OUT, and IN OUT. Figure 23.1 shows the uses of these parameters.

Parameter Type	Use
IN	Passes a value into the program; read-only type of value; it cannot be changed; default parameter type. For example, constants, literal, and expressions can be used as IN parameters.
OUT	Passes a value back from the program;write-only type of value,cannot assign a default value. If a program is successful, value is assigned. For example, a variable can be used as OUT parameter.
IN OUT	Passes a value in and returns a value back; value is read from and then written to. For example, a variable can be used as a IN OUT parameter.

Figure 23.1 Types of parameters.

Actual and Formal Parameters

The parameters passed in a call statement are called the actual parameters. The parameter names in the header of a module are called the formal parameters. The actual parameters and their matching formal parameters must have the same data types. In a procedure call, the parameters are passed without data types. The procedure header contains formal parameters with data types, but the size of the data type is not required. Figure 23.2 shows the relationship between actual and formal parameters.

-- Procedure Call

SEARCH_EMP (543, LAST)

-- Procedure Header

PROCEDURE SEARCH EMP (EMPNO IN NUMBER. LAST OUT VARCHAR2)

Figure 23-2 Actual and formal parameters.

Matching Actual and Formal Parameters

There are two different ways in PL/SQL to link formal and actual parameters:

1. In positional notation, the formal parameter is linked with an actual parameter implicitly by position (Fig. 10-2). Positional notation is more commonly used for parameter matching.

2. In named notation, the formal parameter is linked with an actual parameter explicitly by name. The formal parameter and actual parameters (the values of the parameters) are linked in the call statement with the symbol =>.

The general syntax is

formalparametername => argumentvaIue

For example,

EMPNO => 543

In Figure 23-3, a procedure code is shown. If a procedure with the same name already exists, it is replaced. You can type it in any editor such as Notepad. When you run it, a "Procedure created" message is displayed. The procedure named dependent_info is compiled into p-code and then stored in the database for future execution.

You can execute this procedure from the SQL * Plus environment

(SQL> prompt) with the EXECUTE command. For example,

SQL> EXECUTE dependent_info

SQL> CREATE OR REPLACE PROCEDURE DEPENDENT INFO

2 IS

3 CURSOR DEP CUR IS

4 SELECT LNAME, FNAME, COUNT(DEPENDENTID) CNT

5 FROM EMPLOYEE E, DEPENDENT D

6 WHERE E.EMPLOYEEID = D.EMPLOYEEID

7 GROUP BY LNAME, FNAME;

- 8 BEGIN

9 FOR DEP_REC IN DEP_CUR LOOP

```
10 IF DEP_REC.CNT >= 2 THEN
11 DBMS_OUTPUT.PUT_LINE(DEP_REC.LNAME II ', ' II
12 DEP_REC.FNAMEII' has 'IIDEP_REC.CNTII' dependents');
13 END IF;
14 END LOOP;
15 END;
16 /
Procedure created.

SQL> EXECUTE DEPENDENT_INFO
Dev, Derek has 2 dependents
Chen, Sunny has 3 dependents
PL/SQL procedure successfully completed.
SQL>
```

Figure 23-3 Procedure without parameters .

If you receive an error, use the following command:

SHOW ERROR

The procedure becomes invalid if the table on which it is based is deleted or altered. The compiled version of the procedure is stored, and this version should be re-compiled in case of alterations to the table(s). You can recompile that procedure with the following command:

ALTER PROCEDURE procedurename COMPILE;

Check your progress 1:

What is the difference between a procedure and a function?

.....

.....

.....

.....

.....

In Figure 23-4, the procedure search_emp receives three parameters—i _empid, o_last, and o_first—as IN, OUT, and OUT types, respectively. Parameter i_empid is used for input/reading, and parameters o_last and o_jlrst are used for writing. In Figure 23-5, you will see an anonymous block that calls the procedure in Figure 23-4 with three parameters. The procedure searches for the employee's name based on the V_ID that is passed. If the employee is not found,

the exception is handled in the procedure. If the employee is found, the procedure sends out the last name and first name of the employee to the calling anonymous block. The anonymous block then prints the employee's information.

```
SQL> CREATE OR REPLACE PROCEDURE SEARCH_EMP
2     (LEMPID IN NUMBER,
3       0_LAST OUT VARCHAR2.
4       0_FIRST OUT VARCHAR2)
5   IS
6    BEGIN
7          SELECT INAME, FNAME
8             INTO 0_1.AST, 0_FIRST
9         FROM EMPLOYEE
10  WHERE EMPLOYEEID =I_EMPID;
11         EXCEPTION
12        WHEN OTHERS THEN
13  DBMS_OUTPUT.PUT_LINE('Employeeld 'II
 14  TO_CHAR(LEMPID) II 'does not exist');
15  END SEARCH_EMP;
/
```

Procedure created.

Figure 23.4 Procedure with parameters.

SQL>

```
SQL> DECLARE
2 V_LAST EMPLOYEE.LNAME%TYPE;
3 V_FIRST EMPI.OYEE.FNAME%TYPE;
4 V_ID EMPLOYEE.EMPLOYEEID%TYPE := &EMP_ID;
5 BEGIN
6 SEARCH_EMP( V_ID, V_LAST, V_FIRST);
7 IF V_LAST IS NOT NULL THEN
8 DBMS_OUTPUT.PUT_LINEEmpIoyee: 'II V_ID);
9 DBMS_OUTPUT.PUT_LINE('Name: 'II V_LAST II', 'II V_FIRST);
10 END IF;
```

```
11 END;
12 /
Enter value for empJd: 100
Employeeld 100 does not exist
PL/SQL procedure successfully completed.
SQL> /
Enter value for emp_id: 200
Employee: 200
Name: Shaw, Jinku
P1LSQL procedure successfully completed.
SQL>
```

Figure 23-5 Anonymous block with call to procedure

23.3 FUNCTIONS

A function, like a procedure, is a named PL/SQL block. Like a procedure, it is also a stored block. The main difference between a function and a procedure is that a function always returns a value to the calling block. A function is characterized as follows:

- A function can be passed zero or more parameters.
- A function must have an explicit RETURN statement in the executable section to return a value.
- The data type of the return value must be declared in the function's header.
- A function cannot be executed as a stand-alone program.

A function may have parameters of the IN, OUT, and IN OUT types, but the primary use of a function is to return a value with an explicit RETURN statement. The use of OUT and IN OUT parameter types in functions is rare—and considered to be a bad practice.

The general syntax is

```
CREATE (OR REPLA CE I FUNCTION functionname
((parameterl (,parameter2. . . 1)
RETURN Data Type
Is
[constant I variable declarations ]
BEGIN
executable statements
```

RETURN return value

[EXCEPTION

exception-handling statements

RETURN return value]

END [functionname];

The RETURN statement does not have to be the last statement in the body of a function. The body may contain more than one RETURN statement, but only one is executed with each function call. If you have RETURN statements in the exception section, you need one return for each exception.

Function Header

The function header comes before the reserved word IS. The header contains the name of the function, the list of parameters (if any), and the RETURN data type.

Function Body

The body of a function must contain at least one executable statement. If there is no declaration, the reserved word BEGIN follows IS. If there is no exception handler, you can omit the word EXCEPTION. The function name label next to END is optional. There can be more than one return statement, but only one RETURN is executed in a function call.

RETURN Data Types

A function can return a value with a scalar data type, such as VARCHAR2, NUMBER, BINARY_INTEGER, or BOOLEAN. It can also return a composite or complex data type, such as a PL/SQL table, a PL/SQL record, a nested table, VARRAY, or LOB.

Calling a Function

A function call is similar to a procedure call. You call a function by mentioning its name along with its parameters (if any). The parameter list is enclosed within parentheses. A procedure does not have an explicit RETURN statement, so a procedure call can be an independent statement on a separate line. A function does return a value, so the function call is made via an executable statement, such as an assignment, selection, or output statement. For example,

v_salary := get_salary(&empjd);

IF emp_exists(v_empid)…

In the first exaniple of a function call, the function get_salary is called from an assignment statement with the substitution variable emp_id as its actual parameter. The function returns the employee's salary, which is assigned to the variable v_salary.

In the second example, the function call to the function emp_exists is made from an IF statement. The function searches for the employee and returns a Boolean TRUE or FALSE to the statement.

An anonymous block calls get_deptname function of Figure 23.6 in Figure 23.7, with an employee's department number as a parameter. The function returns the department name back to the calling block. The calling block then prints the employee's information along with the department name.

In this example, the code of function get_deptname is executed in SQL * Plus, which returns a "Function created" message if the function code has no syntactical errors. Then, the calling anonymous block is executed, which calls the compiled function get_deptname with the v_deptid parameter of the NUMBER type. The function searches through the DEPT table, retrieves the corresponding department name, and returns v_deptname, which is assigned to v_dept_name in the anonymous block.

```
SQL> CREATE OR REPLACE FUNCTiON GET_DEPTNAME
2 (LDEPTID IN NUMBER)
9 RETURN VARCHAR2
4 Is
S V_DEPTNAME VARCHAR2(12);
6 BEGIN
7 SELECT DEPTNAME
8 INTO V_DEPTNAME
9 FROM DEPT
10 WHERE DEPTID = 1_DEP11D;
11 RETURN V_DEPTNAME;
12 END GET_DEPTNAME;
13 /
Function created.
SQL>
```

Figure 23.6 Function with parameters.

```
SQL> DECLARE
2 V_DEPTID EMPLOYEE.DEPTID%TYPE;
3 v_DEPT_NAME VARCHAR2(12);
4 V_EMPID EMPLOYEE.EMPLOYEEID%TYPE := &EMPjD;
5 BEGIN
6 SELECT DEPTID
```

```
7 INTO V_DEPTID FROM EMPLOYEE
8 WHERE EMPLOYEEID = V_EMPID;
9 V_DEPT_NAME := GET_DEPTNAME(V_DEPTID);
10 DBMS_OUTPUT.PUT_LINE('Employee: '|| V.. EMPID);
11 DBMS_OUTPUT.PUT_LINE
12 ('Department Name: ' || V_DEPT_NAME);
13 EXCEPTION
14 WHEN OTHERSTHEN
15 DBMS_OUTPUT.PUT_LINE(V_EMPID ||' not found.');
16 END;
17/
Enter value for emp_id: 200
Department Name: Sales
PL/SQL procedure successfully completed.
```

Figure 23.7 Function call.

In the next example, reusability of a function is explained. When a function is compiled and stored, it can be called many times. You write it once, but you can use it many times. Figure 23-8 has a WHILE loop in the anonymous block that calls the function do_total of Figure 23-9 eight times, for values of the counter (or Deptld) equal to 10, 20, 30, and 40. For each value of the counter, the function do_total is called twice, once with two salary parameters and once with two commission parameters. Each time, do_total adds v_sal to v_totsal or v_comm to v_totcomm and returns totals to v_totsal and v_totcomm, respectively. Then, variable v.jotal is used to find the grand total of all salaries and commissions in the anonymous block. The block then prints the total payroll for the company.

```
SQL> DECLARE
2   V_SAL EMPLOYEE.SALARY%TYPE;
3   V_COMM EMPLOYEE.COMMISSION%TYPE;
4   V_TOTSAL NUMBER(8) := 0;
5   V_TOTCOMM NUMBER(S) := 0;
6    v_TOTAL NUMBER(S);
7   COUNTER NUMBER(2) := 10;
8   BEGIN /* MAIN BLOCK */
9   DBMS_OUTPUT.PUT_LINE('DEPTID      SALARY      COMMISSION');
```

```
10  DBMS_OUTPUT.PUT_LINE('—                    -----                    ');
11  WHILE COUNTER <=40 LOOP
12  SELECT SUM(NVL(SALARY, 0)), SUM(NVL(COMMISSION, 0))
13  INTO V_SAL, V_COMM FROM EMPLOYEE
14  WHERE DEPTID = COUNTER;
15  DBMS_OUTPUT.PUT_LINE(COUNTER ||"||
16  TO_CHAR( V_SAl., '$999,999') ||'
17  || TO_CHAR(V_COMM, '$999,999'));
18   V_TOTSAL := DO_TOTAL(V_TOTSAL V_SAL);
19   VTOTCOMM := DO_TOTAL(VTOTCOMM, V_COMM);
20  COUNTER := COUNTER + 10;
21  END LOOP;
22 V_TOTAL := V_TOTSAL + V_TOTCOMM;
23 DBMS_OUTPUT.PUT_LINE('SUBTOTAL' ||
24 TO_CHAR(V_TOTSAL '$999,999') ||"||
25  TO_CHAR(V_TOTCOMM, '$999,999'));
26  DBMS_OUTPUT.PUT_LINE('TOTAL OF SALARY AND COMMISSION: '||
27 TO_CHAR(V_TOTAL '$999,999'));
28 END; /* MAIN BLOCK */
 29 /

10 $375,000 $35,000
20 $146,500 $20,000
30 $69,500 $8,000
40 $150,000 $10,000
SUBTOTAL $741,000 $73,000
TOTAL OF SALARY AND COMMISSION: $814,000
PL/SQL procedure successfully completed.
SQL>
```

Figure 23.8 Function with While Loop

```
SQL> CREATE OR REPLACE FUNCTION DO_TOTAL
2 (I AMOUNT IN NUMBER, LTOTAL IN NUMBER)
3 RETURN NUMBER
4 Is
5 V_RESULT NUMBER(8) 0;
6 BEGIN
7 V_RESULT := LTOTAL + LAMOUNT;
8 RETURN V_RESULT;
9 END DO_TOTAL;
10 /
Function created.
SQL>
```

Figure 23.9 This function (Fig. 23-8) is called multiple times.

Calling a Function from an SQL Statement

A stored function block can be called from an SQL statement, such as SELECT. For example,

```
SELECT get_deptname(10) FROM dual;
```

This function call with parameter 10 will return the department name Finance in the N2 example tables. You can also use a substitution variable as parameter.

23.4 LET US SUM UP

Check your progress Answers :

1. The procedure does not return any value whereas the function returns value.

LESSON 24

PL/SQL NAMED BLOCKS : PACKAGES

Contents

24.0 AIMS AND OBJECTIVES

To learn about PL/SQ Structure of a package, the package body, creation of a package, calling a function from the package and its uses.

24.1 PACKAGES

A package is a collection of PL/SQL objects. The objects in a package are grouped within BEGIN and END blocks. A package may contain objects from the following list:

- Cursors.
- Scalar variables.
- Composite variables.
- Constants.
- Exception names.
- TYPE declarations for records and tables.
- Procedure&
- Functions.

Packages are modular in nature, and Oracle has many built-in packages. If you remember, DBMS_OUTPUT is a built-in package. You also know that a package called STANDARD contains definitions of many operators used in Oracle. There are many benefits to using a package.

The objects in a package can be declared as public objects, which can be referenced from outside, or as private objects, which are known only to the package. You can restrict access to a package to its specification only and hide the actual programming aspect. A package follows some rules of object-oriented programming, and it gives programmers some object-oriented capabilities. A package compiles successfully even without a body if the specification compiles. When an object in the package is referenced for the first time, the entire package is loaded into memory. All package elements are available from that point on, because the entire package stays in memory. This one-time loading

improves performance and is very useful when the functions and procedures in it are accessed frequently. The package also follows top-down design.

24.1.1 Structure of a Package

A package provides an extra layer to a module. A module has a header and a body, whereas a package has a specification and a body. A module's header specifies the name and the parameters, which tell us how to call that module. Similarly, the package specification tells us how to call different modules within a package.

Package Specification

A package specification does not contain any code, but it does contain information about the elements of the package. It contains definitions of functions and procedures, declarations of global or public variables, and anything else that can be declared in a PL/SQL block's declaration section. The objects in the specification section of a package are called public objects.

The general syntax is

```
CREATE [OR REPLACE] PACKAGE packagename
 IS
 [constant, variable and type declarations ]
 [exception declarations ]
 [cursor specifications ]
 [function specifications ]
 [ procedure specifications ]
 END [packagename ];
```

For example,

```
PACKAGE bb_team
IS total_players CONSTANT INTEGER := 12;
player_on_dl EXCEPTION;
FUNCTION team_average(points IN NUMBER, players IN NUMBER)
RETURN NUMBER;
END bb_team;
```

The package specification for the course_info package in Figure 24.1 contains the specification of a procedure called FIND_TITLE and functions HAS_PREREQ and FIND_PREREQ. The COURSE_INFO package contains three modules in

```
SQL> CREATE OR REPLACE PACKAGE COURSEINFO
2 AS
3 PROCEDURE FIND_TITLE
4 (L_ID IN COURSE.COURSEID%TYPE,
5 0_TITLE OUT COURSE.TITLE%TYPE);
6 FUNCTION HAS_PREREQ
7 (L_ID IN COURSE.COURSEID%TYPE)
8 RETURN BOOLEAN;
9 FUNCTION FIND_PREREQ
10 (L_ID IN COURSE.COURSEID%TYPE)
11 RETURN VARCHAR2;
12 END COURSEJNFO;
13 /
Package created.
SQL>
```

Figure 24.1 Package specification.

24.1.2 Package Body

A package body contains actual programming code for the modules described the specification section. It also contains code for the modules not described in specification section. The module code in the body without a description in the package is called a private module, or a hidden module, and it is not visible outside the body of the package.

The general syntax of a package body is

```
PACKAGE BODY packagename
 IS
[ variable and type declarations ]
[cursor specifications and SELECT queries ]
[header and body of functions]
[header and body of procedures ]
 [(BEGiN
executable statements]
[EXCEPTION
```

exception handlers)

END [packagename];

Check your progress 1:

What is a Package?

.....

.....

.....

.....

.....

.....

.....

As a field is to a record, so an object is to a package. When you reference an object in a package, you must qualify it with the name of that package using dot notation. If you do not use dot notation to reference an object, the compilation will fail. Within the body of a package, you do not have to use dot notation for that package's objects, but you definitely have to use dot notation to reference an object from another package.

For example,

IF bb_team.total.players < 10 THEN

For example,

EXCEPTION

WHEN bb_team.player..on_dl THEN

where total.players and player_on_di are modules/objects in the bb_team package.

There is a set of rules that you must follow in writing a package's body:

• The variables, constants, exceptions, and so on declared in the specification must not be declared again in the package body.

• The number of cursor and module definitions in the specification must match the number of cursor and module headers in the body.

• Any element declared in the specification can be referenced in the body.

In Figures 24-1 and 24-2, package specification and body, respectively, are shown for the course_info package. The calls to a procedure and a function in the course_info package are shown in Figures 24-3 and 24-4, respectively.

In Figure 24-3, a call is made to the procedure FIND_TITLE of the COURSE_INFO package in Figure 24-2. The procedure is passed V_COURSEID as an IN parameter. If Courseld is invalid, the procedure throws an exception and then handles that exception with an appropriate message. If Courseld is valid, the OUT parameter V_TITLE is assigned course title.

In Figure 24-4, a call is made to the function HAS_PREREQ of the COURSE_INFO package with one parameter, V_COURSEID. If course does not have a rerequisite, the function displays the appropriate message. If course does not exist, an exception is thrown in the function body, and the message is displayed. The function returns FALSE in both cases. The function returns TRUE if the prerequisite exists. If TRUE is returned, another function, FIND_PREREQ, is called with the V_COURSEID parameter. The function returns concatenated prerequisite ID and title. Figure 24-5 shows output from all three situations mentioned above.

You can also use the EXECUTE command to run a package's procedure:

EXECUTE packagename.procedurename

SQL> CREATE OR REPI.ACE PACKAGE BODY COURSEJNFO AS

2 PROCEDURE FIND_TITLE

3 (L_ID IN COURSE.COURSEID%TYPE, 0_TITLE OUT COURSE.TITLE%TYPE) IS

4 BEGIN

5 SELECT TITLE INTO 0_TITLE FROM COURSE WHERE COURSEID = I_ID'

6 EXCEPTION

7 WHEN OTHERSTHEN

8 DBMS_OUTPUT.PUT_LINE(I_ID ||' not found.');

9 END FIND_TITLE;

10 ----------

11 FUNCTION HAS_PREREQ

12 (L_ID IN COURSE.COURSEID%TYPE) RETURN BOOLEAN IS

13 V_PREREQ VARCHAR2(6);

14 BEGIN

15 SELECT NVL(PREREQ, 'NONE') INTO V_PREREQ

16 FROM COURSE WHERE COURSEID = L_ID;

17 IF V_PREREQ = 'NONE' THEN

18 DBMS_OUTPUT.PUT_LINE('No prerequisite');

19 RETURN FALSE;

20 ELSE RETURN TRUE;

```
21 END IF;
22 EXCEPTION
23 WHEN NO_DATA_FOUND THEN
24 DBMS_OUTPUT.PUT_LINE('Course: '||I_ID || 'does not exist');
25 RETURN FALSE;
26 END HAS_PREREQ;
 27--------------------------------------------------------------------------------
 28 FUNCTION FIND_PREREQ
29 (L_ID IN COURSE.COURSEID%TYPE) RETURN VARCHAR2 IS
30 V_ID VARCHAR2(6);
31 V_TiTLE VARCHAR2(25);
32 V_PRE VARCHAR2(30);
33 BEGIN
34 SELECT NVL(P.COURSEID, 'NONE'), NVL(P.TTTLE, 'NONE')
35 INTO V_ID, V_TITLE FROM COURSE C, COURSE P
36 WHERE C.PREREQ = RCOURSEID AND C.COURSEID = I_ID;
37 V_PRE := VJD || '--' || V_TITLE;
38 RETURN V_PRE;
39 EXCEPTION
40 WHEN OTHERS THEN RETURN 'NONE';
41 END;
42 END COURSEINFO;
43 /
Package body created.
```

Figure 24-2 Package body.

```
SQL> /* Anonymous block calls
DOC> procedure FIND_TITLE in package COURSEINFO */
SQL> DECLARE
2 V_COURSEID COURSE.COURSEID%TYPE '&P_COURSEID';
3 V_TITLE COURSE.TITLE%TYPE;
4 BEGIN
5 COURSE JNFO.FIND.TITLE(VCOURSEID, V_TITLE);
```

```
6 IF V_TITLE IS NOT NULL THEN
7 DBMS_OUTPUT.PUT_LINE(V_COURSEID II': 'II V_TITLE);
8 END IF;
9 END;
10 /
Enter value for p_courseid: C1S265
C1S265: Systems Analysis
PLISQL procedure successfully completed.
SQL> /
Enter value for p_courseid: 05100
CIS 100 not found.
P1./SQL procedure successfully completed.
SQL>
```

Figure 24-3 Call to procedure in the package of Figure 24-2

24.2 LET US SUM UP

Check your progress : Answers :

A package is a collection of PL/SQL objects. The objects in a package are grouped within BEGIN and END blocks.

LESSON 25

PL/SQL NAMED BLOCKS : TRIGGERS

Contents

25.0 AIMS AND OBJECTIVES

To learn the concept of Triggers, Structure of trigger, types of triggers and their uses.

25.1 TRIGGERS

A database trigger, known simply as a trigger, is a PL/SQL block. It is stored in the database and is called automatically when a triggering event occurs. A user cannot call a trigger explicitly. The triggering event is based on a Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE. A trigger can be created to fire before or after the triggering event. For example, if you design a trigger to execute after you INSERT a new employee in the EMPLOYEE table, the trigger executes after the INSERT statement. The execution of a trigger is also known as firing the trigger.

The general syntax is

CREATE [OR REPLACE] TRIGGER triggername

BEFORE \ AFTER \ INSTEAD OF triggeringevent ON tablelview

[FOR EACH ROW]

[WHEN condition]

DECLARE

Declaration statements

BEGIN

Executable statements

EXCEPTION

Exception-handling statements

END;

where CREATE means you are creating a new trigger and REPLACE means you are replacing an existing trigger. The key word REPLACE is optional, and you should only use it to modify a trigger. If you use REPLACE and a procedure, function, or package exists with the same name, the trigger replaces it. If you create a trigger for a table and then decide to modify it and associate it with another table, you will get an error. If a trigger already exists in one table, you cannot replace it and associate it with another table.

A trigger is very useful in generating values for derived columns, keeping track of table access, preventing invalid entries, performing validity checks, or maintaining security. There are some restrictions, however, involving creation of a trigger:

A trigger cannot use a Transaction Control Language statement, such as COMMIT, ROLLBACK, or SAVEPOINT. All operations performed by a trigger become part of the transaction. The trigger operations get committed or rolled back with the transaction.

- A procedure or function called by a trigger cannot perform Transaction Control Language statements,

- A variable in a trigger cannot be declared with LONG or LONG RAW data types.

BEFORE Triggers

The BEFORE trigger is fired before execution of a DML statement. The BEFORE trigger is useful when you want to plug into some values in a new row, insert a calculated column into a new row, or validate a value in the INSERT query with a lookup in another table.

Figure 25-1 is an example of a trigger that fires before a new row is inserted into a table. If a new employee is being added to the EMPLOYEE table, you can use a trigger to get the next employee number from the sequence, use SYSDATE as the employee's hire date, and so on. The trigger in Figure 10-14 fires before the INSERT statement. The naming convention used in the example uses the table name the trigger is for, followed by bi for "before insert," and then the word trigger. A trigger uses a pseudo record called :NEW, which allows you to access the currently processed row. The type of record :NEW is tablename%TYPE. In this example, the type of :NEW is employee%TYPE. The columns in this :NEW record are referenced with dot notation (e.g., :NEW.Employeeld).

SQL>. CREATE OR REPLACE TRIGGER EMPLOYEE_BLTRIGGER

2 BEFORE INSERT ON EMPLOYEE

3 FOR EACH ROW

4 DECLARE

5 V_EMPID EMPLOYEE.EMPLOYEEID%TYPE;

6 BEGIN

```
7 SELECT EMPLOYEE_EMPLOYEEID_SEQ.NEXTVAL
8 INTO V_EMPID FROM DUAL;
9 :NEW.EMPLOYEEID := V_EMPID;
10 :NEW.HIREDATE := SYSDATE;
11 END;
Trigger created.
SQL>
```

Figure 25.1 BEFORE trigger.

The trigger employee_bi_trigger provides values of EmployeeId and HireDate, so you need not include those values in your INSERT statement. If you have many columns that can be assigned values via a trigger, your INSERT statement will be shortened considerably. In Figure 25-2, a row is inserted in the EMPLOYEE table without values for Employeeld and HireDate columns. Those columns are given value with firing of the BEFORE trigger of Figure 25-1.

```
SQL> INSERT INTO EMPLOYEE
2 (LNAME, FNAME, POSOTPMOD, SUPERVISOR, SALARY, DEPTID, QUALID)
3 VALUES
4 ('ZEE', 'SONIA, 3, 543, 100000, 20, 2);
1 row created.
SQL> SET LINESIZE 200
SQL> SELECT * FROM EMPLC"EE WHERE LNAME='ZEE';
SQL> EMPLOYEEID  LNAME  FNAME  POSITIONID  SUPERVISOR
          546          ZEE    SONIA       3                543
```

Figure 25-2 BEFORE trigger—row inserted.

AFTER Triggers

An AFTER trigger fires after a DML statement is executed. It utilizes the built-in Boolean functions INSERTING, UPDATING, and DELETING. If the triggering event is one of the three DML statements, the function related to the DML statement returns TRUE and the other two return FALSE. For example, if the current DML statement is INSERT, then INSERTING returns TRUE, but DELETING and UPDATING return FALSE.

The example in Figure 25-3 uses an existing table named TRANSHISTORY, which keeps track of transactions performed on a table. It keeps track of updates and deletions, the user who performs them, and the dates on which they are performed. The trigger is named employee_adu_trigger, where adu stands for "after delete or update." The trigger uses the transaction type based on the last DML statement. It also plugs in the user name and today's date. The information is then inserted in the TRANSHISTORY table.

Figure 25-4 shows rows inserted in the TRANSHISTORY table on use of DELETE and UPDATE statements by trigger.

For the example in Figure 25-1, we used FOR EACH ROW. Such a trigger is known as a row trigger, if it is based on INSERT, the trigger fires once for every newly inserted row. If it is based on UPDATE statement and the UPDATE affects five rows, the trigger is fired five times, once for each affected row. In Figure 25-3, we did not use a FOR EACH ROW, clause. Such a trigger is known as a statement trigger. A statement trigger is fired only once for the statement, irrespective of the number of rows affected by the DML statement.

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEE_ADU_TRIGGER
2 AFTER DELETE OR UPDATE ON EMPLOYEE
3 DECLARE
4V_TRANSTYPE VARCHAR2(6);
5 BEGIN
6IF DELETING THEN
7V_TRANSTYPE := 'DELETE';
8ELSIF UPDATING THEN
 9V_TRANSTYPE 'UPDATE';
10 END IF;
 11 INSERT INTO TRANSHISTORY
 12 VALUES ('EMPLOYEE', V RANSTYPE, USER, SYSDATE);
 13END;
14/
Trigger created.
```

Figure 25-3 AFTER trigger

```
SQL> DELETE FROM EMPLOYEE
2 WHERE UPPER(LNAME) = 'VIQUEZ';
1 row deleted.
SQL> SELECT * FROM TRANSHISTORY;
TABLENAME              TRANSTYPE          USER..NAME    TRAN.DATE
-----------------      ------------------      ------------------
EMPLOYEE                  DELETE             NSHAH          05-DEC-03
```

```
SQL> UPDATE EMPLOYEE
2 SET COMMISSION = SALARY * 0.10
3 WHERE EMPLOYEEID = 547;
1 rows updated.
SQL.> SELECT * FROM TRANSHISTORY;
TABLENAME          TRANSTYPE        USER_NAME          TRAN_DATE
----------------   ----------------  ----------------   ----------------
--
EMPLOYEE              DELETE            NSHAH            05-DEC-03
EMPLOYEE              UPDATE            NSHAH            05-DEC-03
SQL>
```

Figure 25-4 AFTER trigger—in action.

In Figure 25-4, you see the workings of the AFTER TRIGGER of Figure 25-3. A row is deleted from the EMPLOYEE table, and the trigger inserts a row in the TRANSHISTORY table. Then, a row is updated in the EMPLOYEE table, and the trigger inserts another row in the TRANSHISTORY table.

INSTEAD OF Trigger

The BEFORE and AFTER triggers are based on database tables. From version 8i onward, Oracle provides another type of trigger called an INSTEAD OF trigger, which is not based on a table but is based on a view. The INSTEAD OF trigger is a row trigger. If a view is based on a SELECT querythat contains set operators, group functions, GROUP BY and HAVING clauses, DISTINCT function, join, and/or a ROWNUM pseudocolumn, data manipulation is not possible through it.

An INSTEAD OF trigger is used to modify a table that cannot be modified through a view. This trigger fires “instead of” triggering DML statements, such as DELETE, UPDATE, or INSERT.

In Figure 10-18, a complex view is created with a SELECT query and an outer join. Facultyld 235, 333, and 444 are not used in the STUDENT table; in other words, there is no “child” row in STUDENT table for those faculty members. Faculty ld 235 and 333 are not used in the CRSSECTION table either. The DELETE statement to delete Faculty Id 235 in Figure 25-5 returned an error message. We will accomplish deletion of row by creating an INSTEAD OF trigger.

```
SQL> CREATE OR REPI.ACE VIEW STUDENT_FACULTY
2 AS
3 SELECT S.STUDENTID, S.L.AST, S.FIRST, F.FACULTYID, F.NAME
4 FROM STUDENT S. FACULTY F
5 WHERE S.FACULTYID(÷) F.FACULTYID;
```

```
View created.
SQL> DELETE FROM student_faculty WHERE Facultyld = 235;
DELETE FROM student_faculty WHERE Facultyld = 235
*
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table
SQL>
```

Figure 25-5 No data manipulation through complex view.

In Figure 25-6, an INSTEAD OF DELETE trigger is created on the STUDENT.FACULTY view. Now, when the DELETE statement is issued to delete a faculty member with the complex view, the trigger is fired, and the faculty member is deleted without any error message. Notice the use of pseudorow called :OLD in this trigger, which gets the value of Facultyld 235 from the DELETE statement that the user had issued.

```
SQL> CREATE OR REPLACE TRIGGER faculty_delete jod
2 INSTEAD OF DELETE ON student_faculty
3 FOR EACH ROW
4 BEGIN
5 DELETE FROM faculty
6 WHERE Facultyld = :OLD.Facultyld;
7 END;
8/
Trigger created.
SQL> DELETE FROM student_faculty WHERE Facultyld = 235;
1 row deleted.
SQL>
```

Figure 25-6 Data manipulation and the INSTEAD OF Trigger.

Check your progress 1:

What is the use of a Trigger?

……
……

25.2 DATA DICTIONARY VIEWS

Oracle maintains a very informative Data Dictionary. A few Data Dictionary views are useful for getting information about stored PL/SQL blocks. The following are examples of queries to USER_PROCEDURES (for named

blocks), USER_TRIGGERS (for triggers only), USER_SOURCE (for all source codes), USER_OBJECTS(for any object), and USER_ERRORS (for current errors) views:

SELECT Object_Name, Procedure_Name FROM USER_PROCEDURES;

SELECT Trigger_Name, Trigger_Type, Triggering_Event, Table_Name, Trigger_Body

FROM USER_TRIGGERS;

SELECT Name. Type, Line, Text FROM USER_SOURCE;

SELECT Object_Name, Object_lype FROM USER_OBJECTS;

SELECT Name, Type, Sequence, Line, Position FROM USER_ERRORS;

These views cat provide information ranging from the name of an object to the entire source code. Use the DESCRIBE command to find out the names of columns in each Data Dictionary view, and issue SELECT queries according to the information desired.

25.3 LET US SUM UP

Check your progress Answers:

1. A database trigger, known simply as a trigger, is a PL/SQL block. It is stored in the database and is called automatically when a triggering event occurs. A user cannot call a trigger explicitly. The triggering event is based on a Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE.