

RDBMS Concepts and Oracle

Course Designer and Acquisition Editor

Centre for Information Technology and Engineering

Manonmaniam Sundaranar University

Tirunelveli

RDBMS Concepts and Oracle

CONTENTS

Lecture 1	1
RDBMS Definitions	
Basic Concepts	
Data	
Database	
Database User	
Database System	
Database Modeling	
Lecture 2	15
Entity Relationship Model	
Entity	
Attributes	
Relationships	
Mapping Cardinalities	
Lecture 3	30
RDBMS Concepts	
Object based data models	
OLTP	
DSS	
Transaction	
Large Database Management Control	
Data Concurrency	
Data Consistency	
Data Integrity	
Lecture 4	38
Normalization	
Redundancy and Inconsistent Dependency	
First Normal Form	
Second Normal Form	
Third Normal Form	
Denormalization	
Lecture 5	51
Codd's Rule	
Data Integrity	
Keys	

Lecture 6	61
Client Server Architecture	
Clients	
Servers	
Types of Servers	
Multithreaded Server Architecture	
Advantages of Client/Server Computing	
Lecture 7	77
Front Ends	
Open Clients	
ODBC Support	
Character Based Clients	
GUI Based Clients	
Lecture 8	83
Generation of Languages	
SQL	
SQL command Classification	
Data Types	
Create, View, Manipulate Data	
Lecture 9	115
Data Integrity	
Adding Constraints	
Lecture 10	134
Expressions	
Scalar Functions	
Aggregate Functions	
Lecture 11	179
Joins Tables	
Sub Queries	
Correlated sub Query	
Inline Query	
Pseudo Columns	
Lecture 12	198
Privileges	
Database Objects	
Synonyms	
Sequences	
Understanding Database Structure	
Access Methodologies	

Lecture 13	224
Introduction to PL/SQL	
PL/SQL architecture	
Declares Variables	
Control Structures	
Iteration Control	
Lecture 14	249
Understanding Cursors	
Types of Cursors	
Cursor Variables for using Constrained Cursors	
Lecture 15	276
Exceptions	
Errors	
User - Defined Exceptions	
Unloaded Exceptions	
Lecture 16	295
Introduction to Subprograms	
Procedures to Parameters	
Functions of Notations	
Procedures Vs Functions	
Lecture 17	311
Packages	
Specification	
Package body	
Overloading Package	
Lecture 18	330
Triggers	
Creating Triggers	
Cascading Triggers	
Mutating and Constraining Tables	
Enabling and Disabling Triggers	
Lecture 19	348
Why objects?	
Oops and Object Concepts	
Oracle Supported Data Types	
Collection Types	
Object Views	
Member Functions	
Using Order Method	

Lecture 20	380
Introduction to DBA	
Open Cursor	
Execute	
Oracle DBA	
Lecture 21	386
Java Strategies	
JDBC	
Java with SQL	
Java Virtual Machine	
Java Stored Procedures	
Lecture 22	398
Java Strategies	
JDBC	
Java with SQL	
Java Virtual Machine	
Java Stored Procedures	
Lecture 23	413
Introduction to Forms	
Using forms Builder	
Form Wizards	
Creation of Forms	
Lecture 24	433
Property Class	
Visual Attributes	
Library	
Alerts	
Object Libraries	
Editors	
Lecture 25	454
Master Detail Form	
Creation of Master Detail Form	
Triggers	
Validations	

Lecture 26	475
Working with LOV objects	
Using Multiple Canvases	
Types of Canvas	
Object Groups	
Parameters	
Record Groups	
Lecture 27	500
Stacked Canvas	
Tab Canvas	
Horizontal Toolbar Canvas	
List Item	
Lecture 28	516
Using Menu	
Saving Compiling and Attaching	
Checked, Enabled Menu Items	
Lecture 29	523
Introduction to Reports	
Types of Reports	
Tabular Reports	
Break Report	
Master Detail Report	
Matrix Report	
Report Triggers	
Lecture 30	550
Introduction to Graphics	
Using graphics Builder	
Creating a Graph	
Syllabus	561

Lecture 1

RDBMS Fundamentals

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss the theoretical and physical aspects of a relational database
- ☒ Describe the Oracle implementation of the RDBMS and ORDBMS
- ☒ Describe new features of Oracle8I
- ☒ Describe about Database and Database model
- ☒ Describe the use and user of database

Coverage Plan

Lecture–1

- 1.1 Snap shot
- 1.2 RDBMS definitions
- 1.3 Basic concepts
- 1.4 Data
- 1.5 Database
- 1.6 Database user
- 1.7 Database system
- 1.8 Database modeling
- 1.9 Short summary
- 1.10 Brain Storm

1.1 Snap Shot

In this lesson, you will gain an understanding of the relational database management system(RDBMS) and the object relational database management system(ORDBMS). You will also be introduced to the following: A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a general-purpose software system that facilitates the processes of defining, constructing, and manipulating databases for various applications.

1.2 RDBMS Definitions

1.2.1. Database Management System

In this section, we shall discuss about database and related concepts. A database is a static storage area and the system that works on it is called the Database Management System (DBMS). The user interacts with DBMS and not with the database. The primary purpose of a DBMS is to provide an effective way of storing and retrieving the data faster for both single-user and multi-user systems.

Database Management largely involves:

- Storage of Data
- Manipulation of the data
- Access restriction for unauthorized users.

Functions of a DBMS

Some of the functions of DBMS are:

- ❖ Database Definition – how data is to be stored and organized.
- ❖ Database Creation – Storing data in a defined database
- ❖ Data Retrieval – Querying and reporting
- ❖ Updating – changing the contents of the database.
- ❖ Programming user facilities for system development.
- ❖ Database revision and restructuring
- ❖ Database Integrity control
- ❖ Performance monitoring.

Characteristics Of Dbms

- ❖ Control of data redundancy
- ❖ Sharing data
- ❖ Maintenance of integrity
- ❖ Support for transaction control and recovery
- ❖ Data independence
- ❖ Availability of productivity tools
- ❖ Security
- ❖ Processing speeds
- ❖ Hardware independence

1.2.2 Relational Database Management Systems (RDBMS)

It is the most popular model and solves most of the problems that existed in the previous models. A simple definition of RDBMS is:

"It is a database management system where the data are organized as tables of data values and all the operations on the data work on the these tables"

1.2.3 ORACLE8: OBJECT Relational Database Management system

- User-defined data types and objects
- Fully compatible with relational database
- Support of multimedia and large objects
- High-quality data baser server features

Oracle8 is the first object-capable database developed by Oracle. It extends the data modeling capabilities of Oracle 7 to support a new object relational data base model. Oracle 8 provides a new engine that brings object oriented programming, complex data types, complex business objects, and full compatibility with the relational world.

Oracle 8 extends Oracle 7 in many ways. It includes several features for improved performance and functionality of online transaction processing (OLTP) applications, such as better sharing of runtime data structures, larger buffer caches, and deferrable constraints. Data warehouse application will benefit from enhancements such as parallel execution of insert, update, and delete operations, partitioning, and parallel-aware query optimization. Operating within the Network Computing Architecture(NCA) framework, Oracle 8 supports client-server and Web-based applications that are distributed and multi-tiered.

Oracle 8 can scale tens of thousands of concurrent users, support up to 512 petabytes, and can handle any type of data, including text, spatial, image, sound, video, and time series as well as traditional structured data.

1.2.4 Oracle8i: Internet Platform Database for internet computing features

- Advanced tools to manage all types of data on Web sites.
- More than a simple relational data store: iFS .
- Integrated Java VM in the server: Jserver.
- Better performance, stronger security, language improvement.
- Greater integration with Windows NT environment: AppWizard.

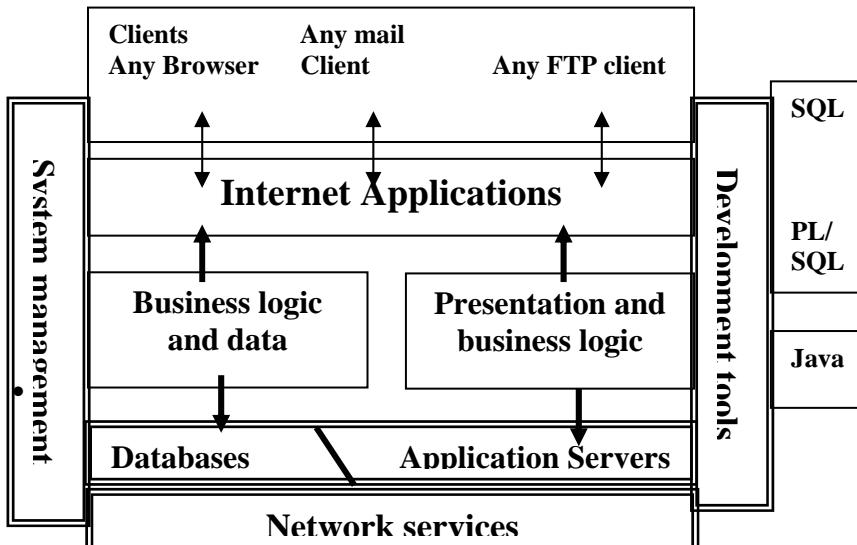
Oracle8i the database for Internet computing, provides advanced tools to mange all types of data in Web sites.

It is much more than a simple relational data store. The Internet File System combines the power of Oracle8i with the ease of use of a file system. It allows users to move all of their data into the Oracle8i database, where it can be stored and managed more efficiently. End users can easily access files and folders in Oracle iFS via a variety of protocols, such as HTML, FTP, and IMAP4, giving them universal access to their data.

Oracle 8i inter Media allows users to web enable their multi media data including image, text, audio, and video data. Oracle 8i includes a robust, integrated, and scalable Java Virtual Machine within the server (Jserver), thus supporting Java in all tiers of applications. This eliminates the necessity of recompiling or modifying Java code when it is to be deployed on a different tier.

With the newly introduced resource management, the DBA can choose the best methods to fit an application's profile and workload. The extended features of parallel servers and networking improves ease of system administration. The extended functionality of advanced replication results in better performance and improved security. Significant new functionalities have been added to languages.

Oracle 8i provides full, native integration with Microsoft Transaction Server (MTS) in the Windows NT environment. Application development is simplified by the Oracle Application Wizard (AppWizard) for Visual Studio, which provides developers with a GUI tool for creating a Visual C ++ Visual Interdev, or Visual Basic applications accessing data in an Oracle database.



Oracle Internet Platform

Oracle offers a comprehensive high performance Internet platform for e-commerce and data warehousing. This integrated platform includes everything needed to develop, deploy, and manage Internet applications. The Oracle Internet Platform is built on three core pieces:

- Browser-based clients to process presentation.
- Application servers to execute business logic and serve presentation logic to browser based clients.
- Data bases to execute data base intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface driven development tools to build business applications, as well as a large suite of software applications, for many areas of business and industry. Stored procedures, function, and packages can be written by using SQL .

1.2.5 Relational Data Structure Definitions

Table:	A table is the organizing principle in a relational database where data are arranged in a rectangular fashion. Each table in a database will have a unique name, which can identify the contents.
Relation:	A relation is the term for defining the association between the tables.
Tuple:	A row or record in a table is called Tuple.
Attribute:	Each column in a table has a column name and every column must have a different name. The column or field is called an Attribute of a table.
Domain:	Domain can be defined as a set of values from where the attributes get their actual values.

1.3 Basic Concepts

Relational Model

The principles of the relational model were first outlined by Dr.E.F.Codd in a June 1970 paper called "A Relational Model of Data for Large Shared Data Banks." In this paper, Dr. Codd proposed the relational model for database systems.

The more popular models used at that time were hierarchical and network, or even simple flat file data structures. Relational database management systems (RDBMS) soon became very popular, especially for their ease of use and flexibility in structure. In addition, a number of innovative vendors, such as SQL, supplemented RDBMS with a suite of powerful application development and user products, providing a total solution.

Components of the Relational Model

- o Collections of objects or relations that store the data
- o A set of operators that can act on the relations to produce other relations
- o Data integrity for accuracy and consistency

Definition of Relational Database

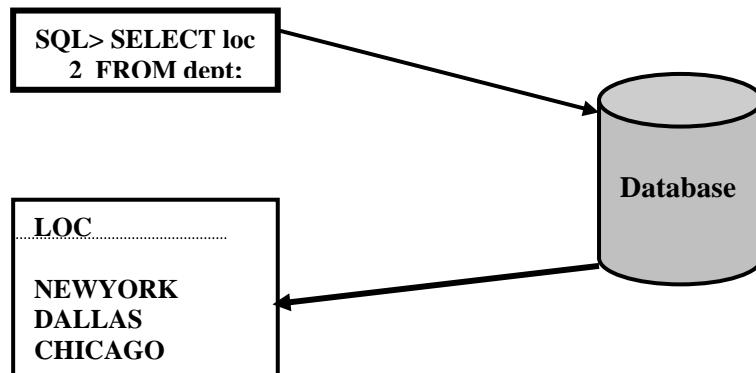
A relational database uses relations or two-dimensional tables to store information. For example, you might want to store information about all the employees in your company. In a relational database, you create several tables to store different pieces of information about your employees, such as an employee table, a department table, and a salary table.

Properties of a Relational Database

In a relational database, you do not specify the access route to the tables, and you do not need to know how the data is arranged physically.

To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating upon relational databases. The language contains a large set of operators for partitioning and combining relations. The database can be modified by using the SQL statements.

Communicating With A RDBMS Using SQL

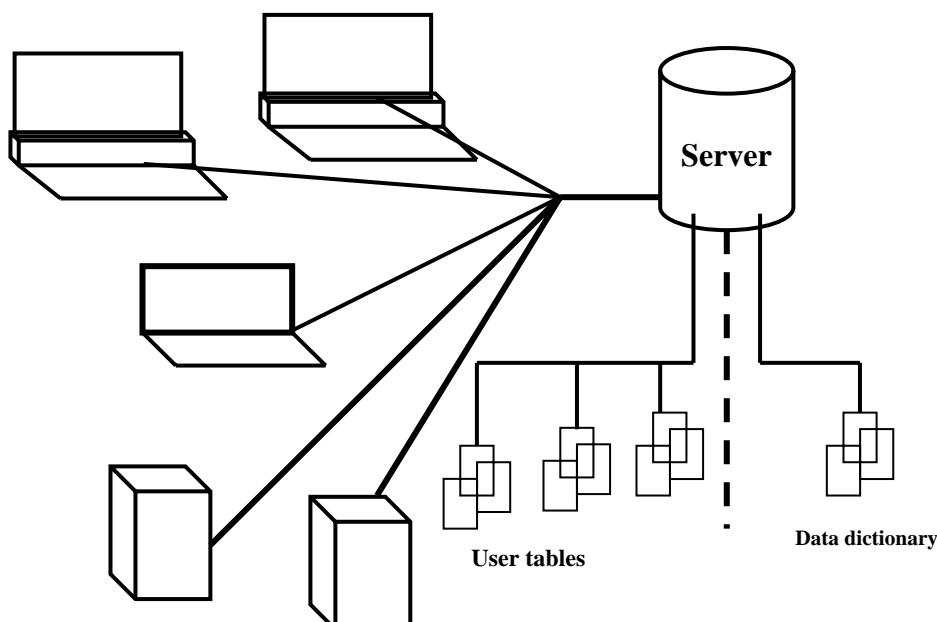


Structured Query Language

SQL allows you to communicate with the server and has the following advantages:

- o Efficient
- o Easy to learn and use
- o Functionally complete (SQL allows you to define, retrieve, and manipulate data in the tables.)

Relational Database Management System



Using its features, you can store and manage data with all the advantages of a relational structure plus PL/SQL, an engine that provides you with the ability to store and execute

program units. The server offers the options of retrieving data based on optimization techniques. It includes consistency and protection of data through locking mechanisms.

In this client-server environment, a wide range of computing resources can be used.

1.4 Data

Data is any information in bits and pieces. Customer names, an employee number, amount of sales, number of centuries scored by Tendulkar are examples of data. To operate with the data, the data needs to be related so that it is easier to search and locate them. Here, data becomes meaningful only if it is related. To illustrate this better let us consider an Office where essential information needs to be stored as shown below:

Office	Files	Manager	Typist	Memo	Salary	Bonus	Sales
--------	-------	---------	--------	------	--------	-------	-------

Data Models

Models are a cornerstone of design. Engineers build a model of a car to work out any details before putting it into production. In the same manner, system designers develop models to explore ideas and improve the understanding of the database design.

Purpose of Models

Models help communicate the concepts in people's minds. They can be used to do the following:

- o Communicate
- o Categorize
- o Describe
- o Specify
- o Investigate
- o Evolve
- o Analyze
- o Imitate

1.5 Database

A database is a collection of interrelated data that are stored in controllable and retrievable form. The collection represents static information of a group of related data that collectively makes sense. In this way, the information can be stored and retrieved quickly with ease using databases.

Databases are created and managed as files. Files are the unit of storage in a computer. When the data is managed, it is important that the relations governing the meaningful data groups are easy to define and manipulate. Each individual information can be shared among several users. Different users can access the same data for different purpose.

The following table illustrates this:

Customer	Product	Orders
Customer No	Product No	Ordercode
Name	Name	Product No
Address	UnitPrice	Customer No
Outstanding_due	Qty_ordered	Order_Value

Table 1.1 Customer, Product and Orders Files

In the above table, a relationship between the Customer No. in the Customer file and the Customer No. in the Orders file has to be established so that the validity of the customer is maintained. Likewise, for the same customer, the product that has been ordered has to be related to the product file. Thus, the database differs depending on how it is organized.

1.6 Database User

One person typically defines, constructs, and manipulates the database. However, many persons are involved in the design, use, and maintenance of a large database with a few hundred users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the “actors on the scene.” And also we consider people who may be called “workers behind the scene” those who work to maintain the database system environment, but who are not actively interested in the database itself.

Database administrators

If any organization where many persons use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the database administrator (DBA).

The DBA is responsible for authorizing access to the database, for coordinating and monitoring its use, and for acquiring software and hardware resources as needed. The DBA is accountable for problems such as breach of security or poor system response time. In large organizations, the DBA is assisted by a staff that helps in carrying out these functions.

Database designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented. It is the responsibility of database designers to communicate with all prospective database users, in order to understand their requirements, and to come up with a design that meets these requirements.

In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop a view of the database that meets the data and processing requirements of this group. These views are then analyzed and integrated with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

End Users

There are the persons whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users.

- ❖ Casual end users occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle or high-level managers or other occasional browsers.
- ❖ Naïve or parametric end users make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates called canned transactions that have been carefully programmed and tested. We are all accustomed to dealing with several types of such users. Bank tellers check balances and post withdrawals and deposits. Reservation clerks for airlines, hotels, and car rental companies check availability for a given request and make reservations. Clerks at receiving stations for courier mail enter package identifications via bar code and descriptive information through buttons to update a central database of received and in-transit packages.
- ❖ Sophisticated end users include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to meet their complex requirements.
- ❖ Stand-alone users maintain personal databases by using ready-made program packages that provide easy-to-use menu or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

A typical DBMS provides multiple facilities to access a database. Naïve end users need to learn very little about the facilities provided by the DBMS; they have only to understand the types of standard transactions designed and implemented for their use. Casual users learn only a few facilities that may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements. Stand-alone users typically become very proficient in using a specific software package.

System Analysts and application programmers

System analysts determine the requirements of end users, especially naïve and parametric end users, and develop specifications for canned transactions that meet these requirements. Application Programmers implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system* environment. These persons are typically not interested in the database itself. We call them the workers behind the scene, and they include the following categories.

DBMS Designers And Implementers

These are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or modules, including modules for implementing the catalog, query language, interface processors, data access, and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.

Tool developers

Tools are software packages that facilitate database system design and use, and help in improving performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. Tool developers include persons who design and implement such tools. In many cases, independent software vendors develop and market these tools.

operators and maintenance personnel

These are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although the above categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database for their own purposes.

1.7 Database system

Database and database technology are having a major impact on the growing use of computers. It is fair to say that database will play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name a few. The word *database* is such common uses that we must begin by defining what a database is. Our initial definition is quite general.

A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, PRADOX, or EXCEL. This is a collection of related data with an implicit meaning and hence is a database.

The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term database is usually more restricted. A database has the following implicit properties:

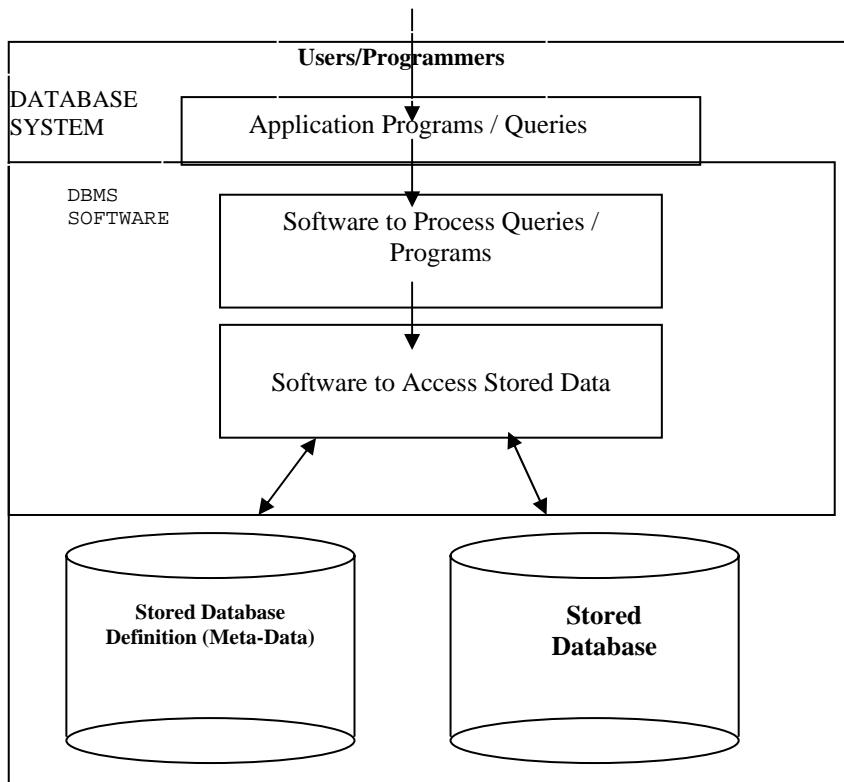
- ❖ A database represents some aspect of the real world, sometimes called the mini world or the Universe of Discourse (UoD), Changes to the miniworld are reflected in the database.
- ❖ A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred-to as a database.
- ❖ A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived application in which these users are interested.

In other words, a database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

A database can be of any size and of varying complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the card catalog of a large library may contain half a million cards stored under different categories-by primary author's last name, by subject, by book title-with each category organized in alphabetic order. A database of even greater size and complexity is maintained by the Internal Revenue Service to keep track of the tax forms filed by taxpayers of the United States. If we assume that there are 100 million taxpayers and if each taxpayer files an average of five forms with approximately 200 characters of information per form, we would get a database of $100*(10^6)*200*5$ characters (bytes) of information. Assuming that the IRS keeps the past three returns for each taxpayer in addition to the current return, we would get a database of $4*(10^{11})$ bytes. This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

A database may be generated and maintained manually or by machine. The library card catalog is an example of a database that may be manually created and maintained. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system.

A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a general-purpose software system that facilitates the processes of defining, constructing, and manipulating databases for various applications. Defining a database involves specifying the data types, structures, and constraints for the data to be stored in the database. Constructing the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. Manipulating a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.



It is not necessary to use general-purpose DBMS software for implementing a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own special-purpose DBMS or not-we usually have to employ a considerable amount of software to manipulate the database in addition to the database itself. We will call the database and software together a database system.

1.8 Database Models

The database models can be broadly classified into two ways:

- Record-based Model
- Object-based Model

The Object-based model is used to define the collection of tools, which describe the data, data relationships etc. The record-based model as the name says describes the access techniques of a DBMS and the data structures.

Record-based Model

The Record-based model consists of three types of models:

- Hierarchical Model
- Network Model
- Relational Model

In the HIERARCHIAL MODEL, data is stored in the form of parent-child relationship. It is more like a tree structure and explodes from a parent to one or more children in the tree. The parent is called root and the children or branches are called nodes. The last node is called the leaf node. Data stored at different levels are accessible from the nodes.

The main disadvantage of this system is that, it is not possible to directly enter a new node.

In the case of the NETWORK MODEL, parent-child relationship is not expected since the data are going to be referred using pointers or locators which locate the data. Also any to any data connections can be made at the time of defining the data. The disadvantage of this type of model is that it leads to complexity in the structure since data are accessed using locators.

Dr. Edgar F. Codd first introduced the RELATIONAL MODEL in 1970s. It was an attempt to simplify the data structure. All the data in the database are represented using a simple row-column type or a tabular structure.

1.9 Short Summary

- Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.
- Oracle8 is based on the object relational database management system.
- Oracle8i server is the database for internet computing
- A relational database can be accessed and modified by executing structured query language statements

1.10 Brain Storm

1. What is RDBMS?
2. Explain the difference between RDBMS and DBMS?
3. Explain about data and database?
4. Explain about different types of users?
5. Explain about classification of database models?

සොස

Lecture 2

Entity Relationship Model

Objectives

After completing this lesson, you should be able to do the following

- ❖ Discuss the theoretical and physical aspects of a relational database
- ❖ Describe the ER model and its notations
- ❖ Describe the entity, attributes and relationship of ER model
- ❖ Discuss about mapping cardinalities

Coverage Plan

Lecture 2

- 2.1 Snap shot
- 2.2 Entity relationship model
- 2.3 Entity
- 2.4 Attributes
- 2.5 Relationships
- 2.6 Mapping cardinalities
- 2.7 Short summary
- 2.8 Brain storm

2.1 Snap Shot

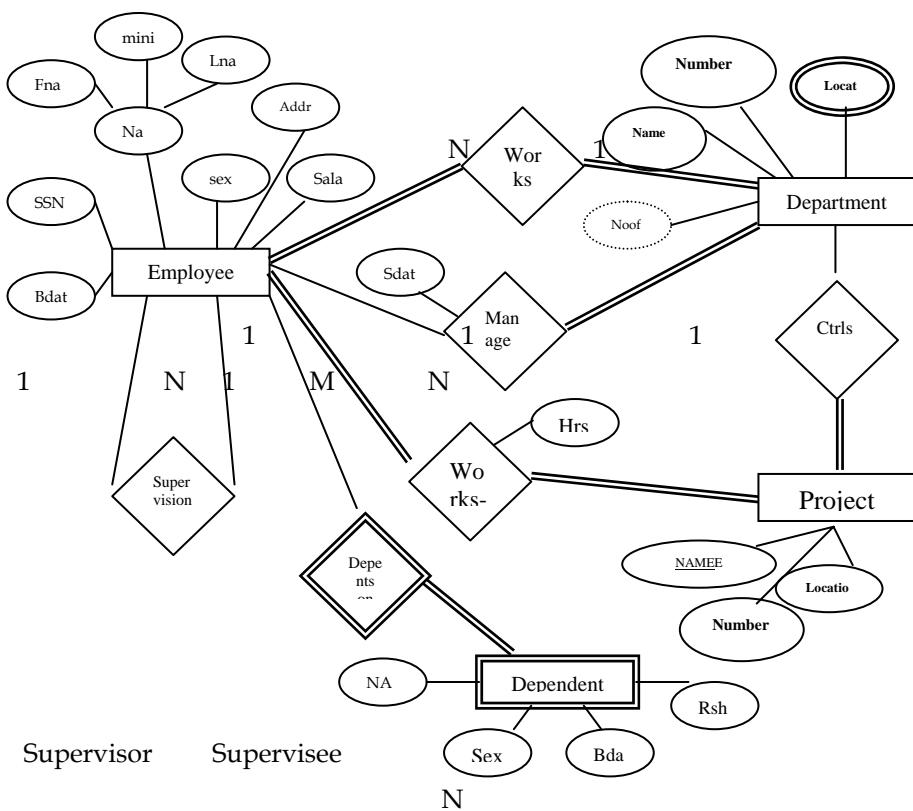
An entity relationship model is derived from business specifications or narratives and built during the analysis phase of the system development life cycle. ER models separate the information required by a business from the activities performed within a business. Although businesses can change their activities, the type of information tends to remain constant.

2.2 Entity-Relationship model

ER model is a popular high level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. In this section we describe an example database, called **company**, that serves to illustrate the ER model concepts and their use in schema design. We list the data requirements for the database here, and then we create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The **company** database keeps track of a **company**'s employees, departments, and projects. Suppose that, after the requirement collection and analysis phase, the database designers started the following description of the 'miniworld' – the part of the company to be represented in the database:

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

The schema for this database application can be displayed by means of the graphical notation known as ER diagrams.



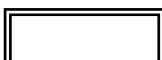
Notation for entity-relationship diagrams

Symbol

Meaning



Entity type



Weak Entity type



Relationship type



Identifying relationship type



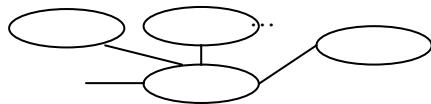
Attribute



Key Attribute



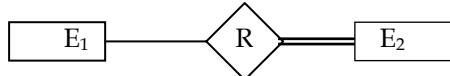
Multivalued attribute



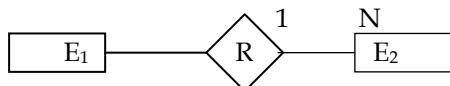
Composite Attribute



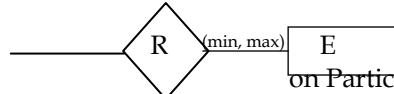
Derived attribute



Total participation of E₂ in R



Cardinality ratio 1:N for E₁:E₂ in R



Structural constraint (min, max)
on Participation of E in R.

ER Model Concepts

Entities, Attributes and Relationships

The relational design process provides a structured approach to modeling an information system's data and the business rules for that data. After you learn the fundamentals of this process, you should be able to take the information learned from the requirements gathered for the system (such as the information presented in the case study) and easily design a database.

The three key components used in relational database design (entities, attributes and relationships) are outlined in following Table.

THE THREE KEY COMPONENTS USED IN RELATIONAL DATABASE DESIGN.

Design Component	Implemented in SQL Server	Examples
-------------------------	----------------------------------	-----------------

Entities	Tables	Student, Course
----------	--------	-----------------

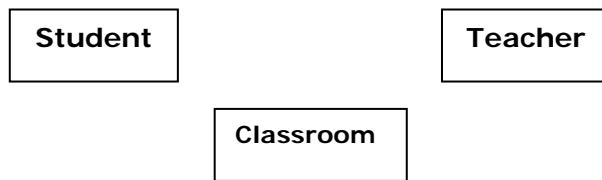
Attributes	Columns in a Table	Student's Name, Course number
------------	--------------------	----------------------------------

Relationships	Primary/Foreign key	Teacher to Class
	Columns or Tables	

The upcoming subsections describe each component in more detail. The chapter then discusses in general terms how these components are implemented in SQL Server.

2.3 Entities

An entity is an object with a distinct set of properties that is easily identified. Entities are the building blocks of a database. Some examples of entities are **Student**, **Teacher** and **Classroom**. You represent an entity using a rectangular box that contains the name of the entity.



An entity instance is a specific value of an entity. Entities are the basic building blocks of relational database design. An entity defines any person, place, thing or concept for which data will be collected. Some examples of entities include the following:

- ❖ **Person:** student, teacher
- ❖ **Place:** classroom, building
- ❖ **Thing:** computer, lab equipment
- ❖ **Concept:** course, student's attendance

When you attempt to discover possible entities in your data model, it is often helpful to look (or listen) for nouns / noun phrases during the requirements analysis. Because nouns describe people, places, things or concepts, this trick usually leads you to an entity. The challenge is to distinguish between the relevant and the irrelevant concepts. Consider the following excerpt from the case study:

"We have ten departments on campus that offer about 400 different courses – everything from Commerce 101 to Zoology 410. Of course, we don't have enough professors to teach every course every semester, but we do manage to offer about 150 different courses each semester."

The highlighted words or phrases are likely candidates for modeling as an entity. The professor, or teacher, concept is an obvious choice, as are the ideas of a course and department. The database will have to keep track of many different teachers and courses. In addition, the database will be used over many semesters, so if detailed information needs to be kept about each semester, the concept of a semester should be modeled as an entity. Notice that the idea of a campus, while important to a college, is not a candidate for an entity in this example. This is because a small college with only one campus will probably build any campus-specific information directly into its data model. If the college were planning to expand to multiple campuses in the future, creating a campus entity then would be important because it would allow you to collect information about each distinct campus.

A good practice is to list all the entities in your database with a one sentence description of what that entity represents. For example, a teacher may be defined as “a person employed by the college who is responsible for instructing students in a class.” Usually, a good entity can be described in one sentence, unless it is a very abstract concept.

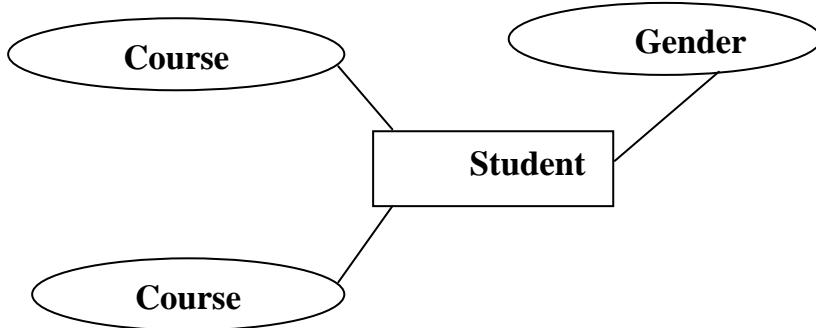
Review the case study to see if you can identify some entities that will be used in the data model. Then, review the following list of entities:

- ❖ **Teacher:** A person employed by the college who is responsible for instructing a class sessions.
- ❖ **Student:** A person who is enrolled in classes at the college and attends class sessions.
- ❖ **Course:** The subject material taught in a class
- ❖ **Class:** A scheduled instance of a course that is taught by one teacher and that meets in a particular room during specific times of the week.
- ❖ **Class Session:** An instance of a class that occurs at a particular date and time.

When speaking of an entity in the data model, one usually is referring to an instance of that entity. An instance is a particular occurrence of an entity that is distinguishable from all other occurrences of that entity. For example, the college has multiple professors, such as Professors Noel, Professor Press and Professor Smith. Each is an instance of a teacher. The concept of an instance is important to understanding relationships.

2.4 Attributes

An attribute is a property of an entity that differentiates it from other entities and provides information about the entity. An attribute type is a property of an entity type. For example, the attributes of the entity Student are **Course Name**, **Course Number** and **Gender**. In an ER diagram, you represent attributes as ellipses and label them with the name of the attribute.



The second major data-modeling concept that you must understand is that of attributes. Attributes are additional characteristics or information defined for an entity.

An entity's attributes don't define an entity, but they provide additional information about an entity that may be useful elsewhere. The concept of a teacher can be defined (definition) without knowing the teacher's name, salary, or educational level (attributes). However, knowing nothing about its teachers does the college little good. And that's where the attributes come in handy.

Review the case study and the entities described in the previous section. Try to list some attributes for each entity, then review the following list of possible attributes for each entity:

- **Teacher:** Name, gender, social security number, address, salary, years tenured
- **Student:** Name, gender, social security number, billing address, college address, class level, grade point average
- **Course:** Course number, course name, prerequisites
- **Class:** Course number, scheduled meeting times, scheduled room, assigned teacher, maximum enrollment, students enrolled.
- **Class Session:** Course number, date and time of session, students attending.

There is no set method or trick for discovering attributes of a particular entity. Usually, brainstorming on the different characteristics of an entity is sufficient. If you find that you are having difficulty, you may want to check your definition of the entity to see if it is specific enough.

2.5 Relationships

Entities and attributes unable you to explicitly define what information, or data, is being stored in the database. Relationships are the other powerful feature of relational modeling and give the modeling technique its name. A relationship is a logical linkage between two entities that describes how those entities are associated with each other. Think of relationships as the logical links in a database that turn simple data into useful information.

For instance, our definition of a teacher reads: "A person employed by the college who is responsible for instructing students in a class."

You will now see to identify not only the need for a relationship, but also the type of relationship.

Identifying relationships in a data Model

If entities can be thought of as the nouns in data modeling, then relationships are best described as the verbs. In fact, relationships often have a verb phrase associated with them in the data model, much like entities have an associated definition. One trick to discovering relationships between entities is to look closely at the entity definitions. Consider the following entity definitions from earlier in the chapter:

- **Teacher:** A person employed by the college who is responsible for instructing a CLASS of STUDENTS.
- **Student:** A person who is enrolled in CLASSES at the college and attends CLASS SESSIONS.
- **Course:** Defines the subject material taught in a CLASS.

- **Class:** A scheduled instance of a COURSE that is taught by one TEACHER and that meets in particular room during specific times of the week.
- **Class Session:** An instance of a CLASS that occurs at a particular date and time.

Notice that the definitions now are written to show other entities in all caps, and significant verbs or verb phrases appear in italics. Formatting your entity definitions in this way makes it easy to identify possible relationships between entities. These definitions can be distilled into several simple statements that highlight the relationships between the entities in this case study:

A TEACHER instructs CLASSES
A COURSE defined subject material for CLASSES.
STUDENTS are enrolled in CLASSES
STUDENTS attend CLASS SESSIONS
A CLASS occurs as CLASS SESSIONS

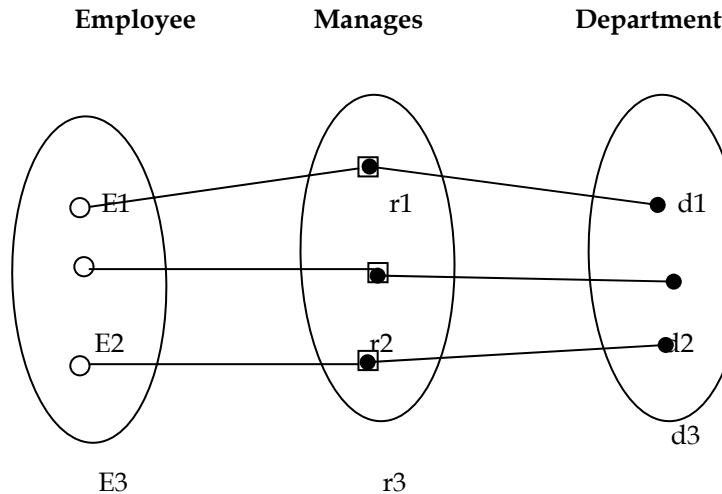
In an ER diagram, relationships are represented as lines drawn between entities. Often, the verb phrase appears near the line to describe the relationship further.

2.6 Mapping cardinality

The identifying property of a relationship is called its cardinality. The cardinality of a relationship allows the database modeler to specify how instances of each entity relate to each other. There are three major types of cardinality:

- **One-to-One.** A single instance of one entity is associated with a single instance of another entity. This type of relationship is relatively uncommon; it is typically used when an entity can be classified into several subtypes.

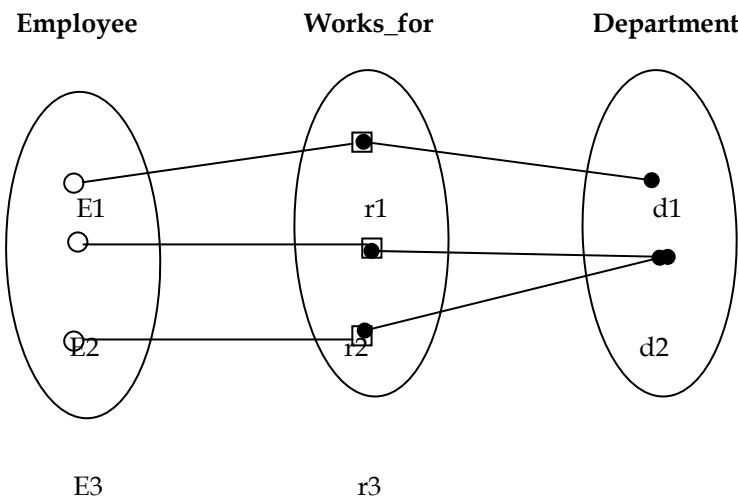
If One-to-One relationship is more complicated than a simple One-to-One cardinality, the line is usually denoted in some way (such as with a dot) to indicate the cardinality.



Above example of a 1 : 1 binary relationship type is manages, which relates a department entity to the employee who manages that department. This represents the constraints that an employee can manage only one department and that a department has only one manager.

- **One-to-Many.** An instance of an entity (called the parent) is associated with zero or several instances of another entity (called the child). An example of this type of relationship can be found by examining the teacher and class entities. A teacher may instruct zero or more classes during the course of a semester.

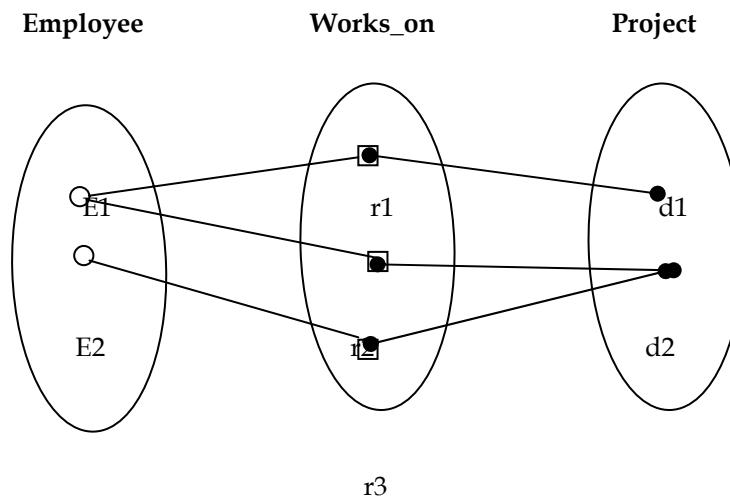
A One-to-Many relationship is drawn as a line between the entities involved. The child end of the relationship typically has a dot.



Above example, consider a relationship type `works_for` between the two entity types `Employee` and `Department`, which associates each employee with the department the employee works for. Each relationship instance in `Works_for` associates one employee entity and one department entity.

- **Many-to-Many.** Many instances of an entity are associated with many instances of another entity. Consider the enrollment relationship between a student and a class. A single student may be enrolled in many classes, and a single class may enroll many students. SQL Server's implementation of the relational model does not directly support a Many-to-Many relationships to a third entity.(table)..

A many-to-many relationship is drawn as a line between the entities involved. Both ends of the line have a dot.



Data Modeling Concepts

You have just learned about entities, attributes, and relationships in an abstract sense. The most important things to take away from this section are the following definitions:

- An entity defines any person, place, thing, or concept for which data will be collected.
- Attributes are additional characteristics or information defined for an entity.
- A relationship is a logical linkage between two entities that describes how the entities are associated with each other.

You should now know what each of these terms means in the context of data modeling, and so are ready to see how these ideas are used in SQL Server. The following sections introduce more concrete terminology that will rigidly define entities, attributes, and relationships in the terms that SQL Server understands.

Implementing Entities and Attributes

Now that you have become familiar with the three main concepts used in relational modeling, it's time to see how these concepts are used in SQL Server. Entities and attributes are implemented in SQL Server as tables and columns, respectively. The following subsections describe table and column characteristics in SQL Server's terminology. You will see how to choose a primary key and apply appropriate column constraints. The following main section deals with how relationships are defined between tables in a database in SQL Server.

Characteristics of Tables

Recall that the earlier discussion of entities introduced the concept of an instance of an entity. In SQL Server, tables are used to store information about each instance of an entity.

First Name	Last Name	Sex	Social Security Number	Salary	Years Tenured
Mathu	Sankar	M	001-03-1869	\$45,000	12
Saravana	Kumar	M	079-91-2060	\$52,500	15
Kalai	Selvi	F	114-78-1342	\$59,000	16
Meena	Devi	F	001-23-1903	\$60,100	18
Diana	Joseph	F	001-14-1803	\$55,000	19

Note that a table stores each instance of an entity as a row in the table, with each attribute stored in a column. Rows and columns are sometimes referred to as records and fields, respectively. The order of the rows and columns in a table is not important, although columns that store related information should be grouped together. Note that in the example table, the FirstName and LastName columns are grouped in this way.

The following list summarizes the terminology just mentioned:

- Entities are modeled as tables.
- In a table, each instance of an entity is called a row
- Attributes are modeled as columns in a table

- Programmers often refer to rows and columns as records and fields, respectively

Although you can choose any names you want for tables and columns, following a few guidelines can make your data model consistent and easier to read. These guidelines include:

- Table and column names are usually singular; this is a relational modeling convention.
- Tables are almost always named after the entity they represent
- Mixed case is preferable to using underscores to separate words.
- Ensure that columns that store the same type of information in different tables have the same name. For instance, if the teacher table and the student table both store address information, make sure that the column storing the ZIP code has the same name in both tables.
- Ensure that similarly named columns all have the same data types.

SQL Server does place some restrictions on table and column names. The restrictions include the following:

- Table and column names cannot be longer than 128 characters.
- Table names must be unique within the database
- Column names must be unique within a table

SQL Server enforces the uniqueness of table and column names for you, but for a relational model to work properly, each row must also be unique. This concept is known as entity integrity or row integrity. In other words, each instance of an entity must be distinguishable from all other instances. SQL Server does not automatically enforce it, the person designing the database must build this row uniqueness into the data model. Row uniqueness is enforced by using a special column in the table called the primary key.

2.7 Short Summary

- ▶ An entity relationship model is an illustration of various entities in a business and the relationships between them.

- **Entity:** An entity is an object with a distinct set of properties that is easily identified.
- **Attribute:** Each column in a table has a column name and every column must have a different name. The column or field is called an Attribute of a table.
- **Relation :** A relation is the term for defining the association between the tables.
- **Relationship:** A named association between entities showing optionality and degree.

2.8 Brain Storm

1. What is entity?
2. Explain about the types of relationship?
3. Give one simple example for ER model?
4. Which type of Relationship is in between the purchase and customer?

ANSWER

Lecture 3

Database Concepts

Objectives

After Completing This Lesson, You Should Be Able To Do The Following

- ☒ Discuss the theoretical and physical aspects of a database
- ☒ Describe the database design
- ☒ Discuss about Object database design model
- ☒ Discuss about on-line transaction processing and DSS
- ☒ Discuss about Data concurrency and data integrity

Coverage Plan

Lecture 3

- 3.1 Snap shot
- 3.2 RDBMS concepts
- 3.3 Object based data models
- 3.4 OLTP
- 3.5 DSS
- 3.6 Transaction
- 3.7 large Database management control
- 3.8 Data concurrency
- 3.9 Data consistency
- 3.10 Data Integrity
- 3.11 Short summary
- 3.12 Brain Storm

3.1 Snap Shot

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases will play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name a few. Having identified all the data in the system, it is necessary to arrive at the logical database design. Database design involves designing the conceptual model of the database.

3.2 RDBMS Concepts

RDBMS is a distinct type of DBMS and offers implicit relational connections through values and references. These are only possible because the data in RDBMS is stored as tabular structure, which grows as a spreadsheet of information in a two-dimensional direction.

RDBMS enables a better understanding of the E-R model and the relational algebra. RDBMS has specific features and characteristics such as non-procedural constructs through which the developer can program the applications more effectively incorporating validations and security of data access.

We shall now see the concepts and features supported by RDBMS.

3.3 Object-based Data Model

Object-based models can be classified as:

- Binary model
- Entity-Relationship model
- Infological Model

The widely accepted model is the Entity-Relationship model. It has gained the acceptance as the ideal model for any database design. Peter Chen first introduced the Entity-Relationship model in 1976. It is based on a perception of the real world things, which represent information. E-R model helps in database design through a better conception and arrangement. Through the E-R model, the application developer can look forward to a

foolproof data definition and table design incorporating the real world application functionality completely. The definitions are dealt below.

3.4 OLTP

On-Line Transaction Processing or OLTP applications are multi-user on-line systems where multiple users access the same data simultaneously and modify the information, which are sensitive in nature. OLTP systems should ensure consistent view of data for modification in that multi-user environment and make sure that the transactions are serialized. In OLTP applications, decisions can wait but data accuracy can't be compromised. Examples of OLTP environment are reservation systems, bank teller counters, stock exchange counters, betting system in racecourses and so on.

In OLTP systems, data is selected for updating critical information. So, though you get a feeling that you get simultaneous data access, actually the user operations are serialized. i.e., they are carried out by the system one after another.

While there is indeed a great deal of complexity in building online transaction processing systems, they can usually be broken down into fairly standard finite steps. For example, to update a customer's address record, a developer would have to code the following steps:

- ☞ Verify user authorization to update database
- ☞ Ask for the customer number
- ☞ Verify it is valid
- ☞ Retrieve the record and populate the screen
- ☞ Position the cursor on the address
- ☞ Accept user input
- ☞ Verify input against any rules
- ☞ Ask for user feedback on correctness
- ☞ Rewrite the record

To the user and developer, it should be as easy as "update a customer's address record." The system should be concerned with the numerous steps it takes to accomplish the task. Smart OLTP interfaces will take care of the detailed transaction management steps. In client/server architectures, smart OLTP interfaces handle distributed transaction management session management, and service-request management.

Currently, the only client/server development tool available with a smart OLTP interface is Ellipse from cooperative solutions.

3.5 DSS

Decision Support Systems or DSS systems are multi-user online applications where decision making is the key factor for processing. In DSS applications, data sensitivity is not very critical but data availability without delay is very important. Examples of DSS applications are manufacturing applications, ERP applications, factory automation environments, and so on.

In DSS, the decision making is crucial and the RDBMS supporting a DSS application should ensure that the user does not wait for a long time to see a permanent version of the data for decision making.

3.6 Transaction

Transaction in business data processing is a logical unit of work that would make sense on completion as a whole. This means, either all the operations in the unit are done fully or none of them are done. For example, when you go to bank to draw money from your account, the teller does a set of operations like checking your account, making entries in a set of registers, handing over the amount to you and you check the amount before you leave the counter. If any of these operations fail, then the rest of the operations together do not make sense and they have to be undone. If there is no sufficient balance in your account, for example, then the teller has to undo all the entries that are made about your cheque number and other information about your account from the registers.

Transaction in RDBMS terms is similar to what has been explained above. It involves one or more of the database operations like insert, delete or update, which are either done together, are done permanently in the database or none of the operations are done.

RDBMS gives powerful commands to simulate a real life transaction with the database. So, you need not strike out or undo manually if one of the operations go wrong. You can rather use simple commands like 'ROLLBACK' to automatically undo those operations, which are part of the incomplete transaction. Alternatively to make permanent the operations of a

transaction you can simply use the ‘COMMIT’ command. A transaction is initiated as well as ended by either committing or rolling back.

3.7 Large Database Management Control

RDBMS offers to control database with large sizes of information running to Giga bytes of storage. For examples, Oracle8 is capable of managing large databases of probably an unlimited size or close to hundreds of Giga bytes of storage. The database space management in Oracle 7 is extensive and space control is done interactively through simple, English like commands and a full and complete space control is possible for the database administrator.

3.8 Data Concurrency

Data concurrency is the phenomenon by which multiple users operate on the same data simultaneously and get response concurrently without the loss of consistent view of data and without waiting for response from the database.

RDBMS allows a large number of users to execute concurrently a variety of applications and operate on the same data simultaneously. With this multi-user database access, RDBMS still minimizes data contention and assures data concurrency and consistency. For instance, Oracle 8’s data concurrency feature enables it to suit well for both OLTP and DSS applications. This feature could even be viewed as a unique offering from Oracle8.

3.9 Data Consistency

Data consistency is a feature by which the RDBMS ensures what you see or understand about the state of the data is what is actually the state in the database. This means you get a consistent view of the database from the RDBMS every time you query the database so that you can go ahead with processing the data.

Oracle 8 provides foolproof data consistency by offering two types of data consistency, one at the *statement level* and other, at the *transaction level*. This is due to the fact that since the consistent view of data is required for two distinct kinds of operations, one to make a decision and other to effect a change based on the view. So, the first operation is just a query

and is more of a DSS nature and the second are a transaction where a query and a subsequent modification together make sense and this is of OLTP nature.

Statement level read consistent view of data is consistent during the execution time and the next execution of the same query might give a different view of data.

Transaction level read consistent view of data is consistent for the whole transaction. This means, the view of data is the same for one or more execution of the query inside the transaction. Once the transaction ends then the view of the data may change.

3.10 Data Integrity

Data integrity ensures that the data entered into the database by the user is checked for its correctness and is the data that is supposed to go into the database. Thus, the data gets automatically validated when it is entered as per the instructions or commands of the designer and by this way RDBMS ensures that the application is of a high degree of data security and integrity.

RDBMS ensures data integrity through automatic validation of data using integrity constraints. The integrity constraints are non-procedural constructs and by just specifying those, the designer can automate the validation process at the time of data entry. Popular integrity constraints are NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK constraints.

3.11 Short summary

- ❖ The object based logical model can be defined as a collection of ideal tools for describing data, data relationships, and data constraints.
- ❖ Transaction in business data processing is a logical unit of work that would make sense on completion as a whole.
- ❖ The object relational model supports both object oriented and relational concepts. It eliminates certain discrepancies in the relational model. In this model it is possible to provide well defined interfaces for the application. A structure once created can be reused- this is the fundamental property of the oops concepts.
- ❖ Data integrity this ensures the accuracy of the data.

3.12 Brain Storm

1. what is OLTP and DSS?
2. What is transaction?
3. What is the difference between Data concurrency and data consistency?
4. What is Large Database management control?

SOOR

Lecture 4

Normalization

Objectives

After completing this lesson, you should be able to do the following

- Discuss the Redundancy and Inconsistent dependency
- Know the purpose for normalization
- Understand the steps involved for normalization.
- Describe the use of normalization
- Discuss about Denormalization
- Discuss about functional dependency

Coverage Plan

Lecture 4

- 4.1 Snap shot
- 4.2 Normalization
- 4.3 Redundancy and inconsistent dependency
- 4.4 First normal form
- 4.5 Second normal form
- 4.6 Third normal form
- 4.7 Denormalization
- 4.8 Short summary
- 4.9 Brain Storm

4.1 Snap Shot

In this session we will concentrate on normalization which is an important step in database design, particularly for relational DBMSs. The relational data model is based on a relation. When structuring data that is to be stored, the analyst must anticipate the need to access the data to meet unexpected requirements, and to reduce redundancy. These can be achieved through the techniques of normalization that provides a systematic way of boiling data structures down to their simplest possible forms.

4.2 Normalization

Normalization is the process of organizing data in a database. This includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating two factors: redundancy and inconsistent dependency. Normalizing a logical database design involves organizing the data into more than one table. Normalization involves performance by reducing redundancy. Redundancy can lead to:

Inconsistencies – errors are more likely to occur when facts are repeated.

Update anomalies – Inserting, modifying and deleting data may cause inconsistencies.

Normalization is a technique that can be applied to data to ensure that a set of tables is derived that contains no redundant data. Normalization is a process of simplifying the relationship between data elements in a record. It is the transformation of complex data stores to a set of smaller, stable data structures. Normalized data structures are simpler, more stable and are easier to maintain. Normalization can therefore be defined as a process of simplifying the relationship between data elements in a record.

Purpose for Normalization

Normalization is carried out for the following four reasons:

- ❖ To structure the data so that there is no repetition of data, this helps in saving space.
- ❖ To permit simple retrieval of data in response to query and report requests.

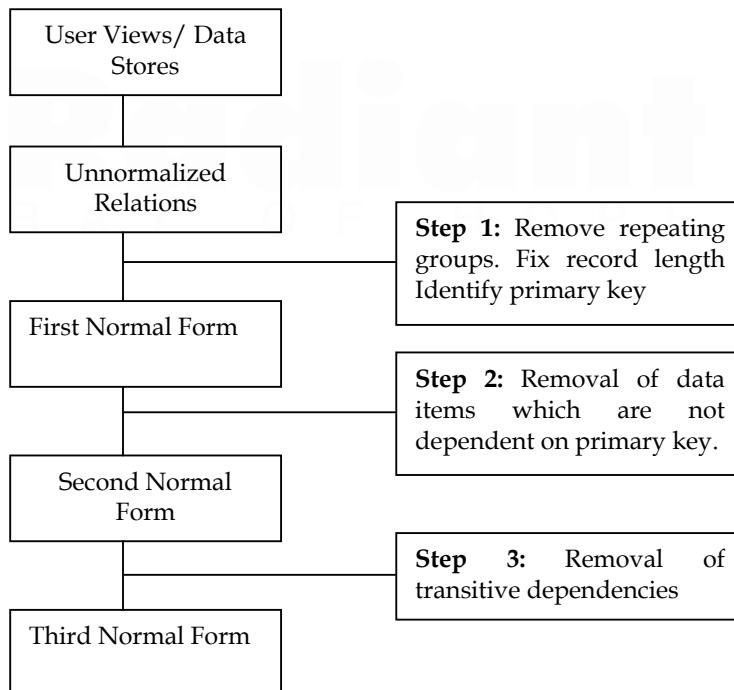
- ❖ To simplify the maintenance of the data through updates, insertions and deletions.
- ❖ To reduce the need to restructure or reorganize data when new application requirements arise.

Steps Of Normalization

Systems analysts should be familiar with the steps in normalization, since this process can improve the quality of design for an application. Starting with a data store developed for a data dictionary the analyst normalizes a data structure in three steps. Each step involves an important procedure to simplify the data structure.

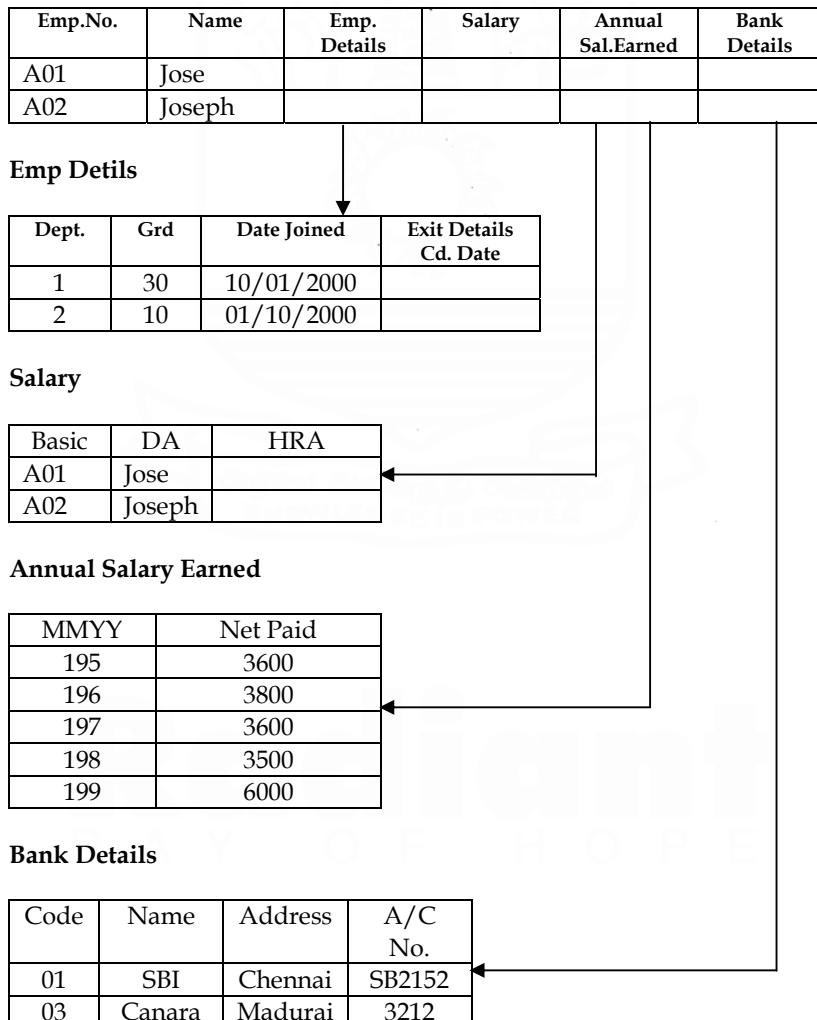
It consists of basic three steps:

1. First Normal form, which decomposes all data, groups into two-dimensional records.
2. Second Normal Form, which eliminates do not fully depend on the primary key of the record.
3. Third Normal Form, which eliminates any relationships that, contain transitive dependencies.



The relation obtained from the data store such as Employee register will most likely be unnormalized. It will consist of repeating groups and record will not be of fixed length.

Unnormalized Employee Record



Employee no., employee name, employee details (department code, grade, date of joining, exit code and exit date), annual salary earned (MMYY, net paid), bank details (bank code, bank name, address, employees A/C no).

Here it is clearly seen that the employee's annual salary earned details which are: Month and Year paid, net paid, are being repeated. Therefore this relation is not a first normal form.

4.3 Redundancy and Inconsistent dependency

Redundant data wastes disk space and creates maintenance problems. If data that exists in more than one place must be changed, then the data must be changed in exactly the same way in all the locations. A customer address change is much easier to implement if that data is stored only in the Customers table and nowhere else in the database.

What is an "inconsistent dependency"? While it is intuitive for a user to look into the Customers table for the address of a particular customer, it may not make sense to look there for the salary of the employee who calls on that customer. The employee's salary is related to, or dependent on, the employee and thus should be moved to the Employees table. Inconsistent dependencies can make data difficult to access; the path to find the data may be missing or broken.

There are a few rules for database normalization. Each rule is called a "normal form." If the first rule is observed, the database is said to be in "first normal form." If the first three rules are observed, the database is considered to be in "third normal form." Although other levels of normalization are possible, third normal form is considered the highest level necessary for most applications.

The three rules of Normalization are:

1. No **repeating groups** of attributes (fields).

Note: Remove all the duplicating info(attributes). Create a new entity for those.

2. Every non-key attribute (field) is fully functionally dependent upon the key field.

Note: Remove those attributes which are dependent on other attributes.

3. No non-key fields are dependent upon the other non-key field(s).

Note: Remove those attributes that come out by itself (ie.,) automatically.

Let us illustrate this with an example.

Consider an object called Invoice, which contains attributes that are required to prepare an Invoice.

Invoice Object Attributes

Invoice_code, Inv_Date, Order_code, Order_Date, order_qty, Cust_code, cust_name, Address, item_code, description, unitcost, ordervalue, invoice_value

4.4 First Normal Form

The basic improvement the analyst should make to such a record structure is to design the record structure so that all records in the file are of fixed length.

A repeating group, that is, the reoccurrence of a data item or group of data items within a record, is actually another relation. This is removed from the record and treated as an additional record structure, or relation.

First Normal Form – Employee Record

Emp.No.	Name	Emp. Details	Salary	Bank Details	I.Tax Detials
A01	Jose				
A02	Joseph				

Emp.No.	MMYY	Net Paid
A01	195	3600
A01	196	3800
A01	197	3600
A01	198	3500
A02	199	6000

As mentioned above the first normal form is carried out by removing the repeating group. In this case we remove the Annual salary earned items and include them in a new file or relation called Annual Salary earned record. Employee number is still the primary key in the employee record. A combination of employee number and MMYY is the primary key in the annual salary earned record.

We thus form two record structures of fixed length: **Employee record** consisting of Employee no., employee name, employee details (department code, grade, date of joining, exit code and exit date), bank details (bank code, bank name, address, employees A/C no)

Annual salary earned record consisting of – employee no., month & year (MMYY) and net paid.

In the attributes mentioned above, the first Normal form must not contain repeated groups. Here the repeated groups are products. When the Invoice Attribute is in 1st Normal Form, the tables would be:

Invoice_Master	Invoice_Item
Invoice_code	Invoice_No
Inv_date	item_code
Order_code	ord_qty
Order_qty	ucst
Cust_code	Ord_value
Cust_name	description
Address	
Invoice_value	

The attributes that are indicated as bold denote primary key.

4.5 Second Normal Form

The second normal form is achieved when every data item in a record that is not dependent on the primary key of the record should be removed and used to form a separate relation.

The PF department ensures that only one employee in the state is assigned a specific PF number. This is called a **one-to-one relation**. The PF number uniquely identifies a specific employee; an employee is associated with one and only one PF number. Thus, if you know the Employee no., you can determine the PF number. This is **functional dependency**. Therefore a data item is functionally dependant if its value is uniquely associated with a specific data item.

Second Normal Form – Employee Record

Emp.No.	Name	Emp. Details	Salary	A/C No.	Bank Code	I.Tax Detials
A01	Jose			SB2152	01	
A02	Joseph			3212	03	

Annual Salary Earned Record

Emp.No.	MMYY	Net Paid
A01	195	3600
A01	196	3800
A01	197	3600
A01	198	3500
A02	199	6000

Bank Record

Code	Name	Address
01	SBI	Chennai
03	Canara	Madurai

The three record structures that are created are:

1. **Employee record** consisting of: _Employee no., employee name employee details (department code, grade, date of joining, exit code and exit date), bank details (bank code, bank name, address, employees A/C no.)
2. **Annual salary earned record** consisting of - employee no., month & year(MMYY) and net paid
3. **Bank record** consisting of: bank code, bank name and bank address. All the attributes of this relation are **fully dependent** on Bank code.

A table in Second Normal Form must indicate that all the attributes, which are not dependent of primary key, must be removed. The table in Second Normal Form will be

Invoice_Master	Invoice_Item	Item_Master
Invoice_code	Invoice_No	Item_code
Inv_date	item_code	description
Order_code	rate	
Cust_code	ord_qty	
Cust_name	Ord_value	
Address		
Invoice_value		

There are no changes in the Invoice details.

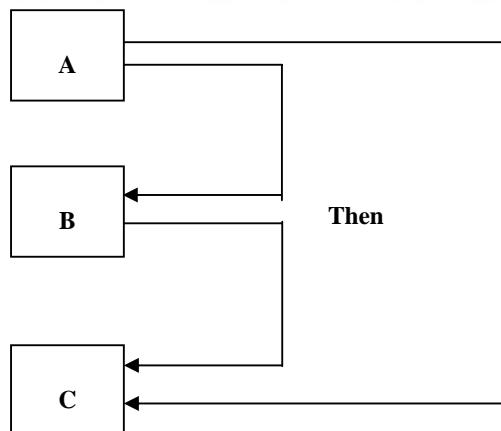
4.6 Third Normal Form

The third normal form indicates that there must not be any dependency between non-key attributes. Resolving the above we get,

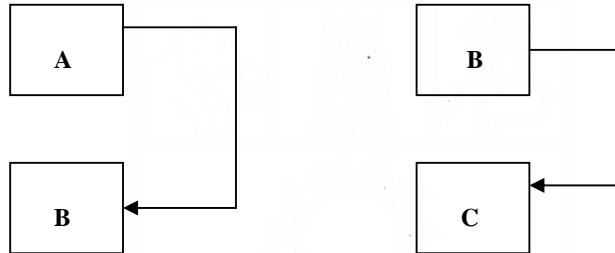
Invoice_Master	Invoice_Item	Item_Master
Invoice_code	Invoice_No	Item_code
Inv_date	item_code	description
Order_code	description	rate
Invoice_Value	ord_qty	Ord_value

Order_Cust	Customer_Master
Order_code	cust_code
Cust_code	cust_name
Order_date	address

Third normal form is achieved when transitive dependencies are removed from a record design. Some of the non-key attributes are dependent not only on the primary key but also on a non-key attribute. This is referred to as a transitive dependency.



Conversion to third normal form removes transitive dependence by splitting the relation into two relations.



Reason for concern. When there is a transitive dependence, deleting A will cause deletion of B and C as well.

- A, B and C are three data items in a record
- If C is functionally dependent on B and
- B is functionally dependent on A
- Then C is functionally dependent on A
- Therefore, a transitive dependency exists.

There are **no transitive dependencies**, so it is also in third normal form.

4.7 Demoralization

The end product of normalization is a set of related tables, which comprise the database. The virtues of normalization have already been enumerated. However, sometimes, to get a simple output, you have to join multiple tables. This affects the performance of a query. In such cases, it is wiser to introduce a degree of redundancy in tables either by introducing extra columns or extra tables.

The intentional introduction of redundancy in a table in order to improve performance is called demoralization.

For example, consider the following tables, which maintain the details of tables

Employee

NAME	SSN	BDATE	SEX	SALARY	SUPERSSN	DNO
Bala	123456	06-10-72	M	5500	789012	5
Diana	789012	06-07-74	F	4850	654321	5
Madhu	654321	05-07-73	M	5080	147258	4
Kalai	147258	22-05-77	F	4500	258369	4
Marthu	258369	01-06-69	M	4800	369147	1
Meena	369147	01-08-77	F	4500	Null	2

Department

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	5	789012	09-10-1999
Administration	4	147258	05-07-2000
HeadQuarters	1	369147	01-06-1999

Select * from department join employee on department.MGRSSN = employee.SUPERSSN;

Resultant table

Dname	Dnumber	MGRSSN	Name	SSN
Research	5	789012	...	Bala	123456
Administration	4	147258	Madhu	654321
HeadQuarters	1	369147	Maruthu	258369

Select Dname, Dnumber, Name from resultant

Name	Dnumber	Name
Research	5	Bala
Administration	4	Madhu
HeadQuarters	1	Maruthu

This table structure has simplified the query and speeded up the processing of the query. By storing extra columns, you are introducing redundancy in the table but improving the performance of queries.

The decision to denormalize will obviously result in a trade-off between performance and data integrity. Denormalization also increases disk space utilization.

4.8 Short summary

- ➲ Normalization is a technique that is more applicable to record-based data models.
- ➲ Normalization is the process of simplifying the relationship between data elements in a record.
- ➲ Normalization is carried out for four reasons which are:
 - ➲ To structure the data
 - ➲ To permit simple retrieval of data in response to query
 - ➲ To simplify the maintenance of data.
 - ➲ To reduce the need to restructure or reorganize data for new requirements.
- ➲ The relation obtained from a data store developed for a data dictionary will most likely be ‘unnormalized’. The normalization process consists of three basic steps which are:
 1. Removal of repeating groups, fixing record length, identification of primary key – first normal form.
 2. Removal of data items which are not dependent on primary key – second normal form.
 3. Removal of transitive dependencies – third normal form.

4.9 Brain Storm

1. What are the steps involved for normalization?
2. What is normalization?
3. What is redundancy?
4. What is the purpose of normalization?
5. What is Denormalization?

Rules of RDBMS

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about the rules of relational database management system
- ☒ Describe about the Data integrity
- ☒ Describe about types of integrity
- ☒ Discuss about types of keys
- ☒ Describe the uses of keys and its examples.

Coverage Plan

Lecture 5

- 5.1 Snap shot
- 5.2 Codd's rule
- 5.3 Data integrity
- 5.4 Keys
- 5.5 Short summary
- 5.6 Brain Storm

5.1 Snap Shot

In this session we discuss about the rules of Codd's. For any RDBMS to be accepted as a full-fledged RDBMS, it has to follow the twelve Codd's rules. There are a number of products called RDBMSs attempting to meet all twelve Codd's rules, but all the twelve rules are, until now, not been satisfied by any RDBMS. Oracle is also, not 100% relational, since it does not obey all Codd's rules.

5.2 Codd's Rules

For a database to be ready to use, Dr. E.F. Codd designed twelve rules. The rules are listed below:

Information Rule

All the information in a relational database must be represented explicitly by values in tables. Knowledge of only one language is necessary to access all data such as description of the table and attribute definitions, integrity constraints, action to be taken when constraints are violated, and security information.

Guaranteed Access Rule

Every piece of data in the relational database can be accessed by using a combination of a table name. A primary key value that identifies the row and a column name that identifies the cell. The benefit of this is that user productivity is improved since there is no need resort to using physical pointers addresses. Provides data independence.

Systematic Treatment of Null Values

RDBMS handles records that have unknown values in a predefined way. It distinguishes between zeroes, blanks and nulls in records and handles such values in a consistent manner that produces correct answers, comparisons and calculations.

Active On-Line Catalog Based on the Relational Model

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

Comprehensive Data Sub-language Rule

A RDBMS may support several languages, but atleast one of them allows the user to do all of the following: define table view, query and update data, set integrity constraints, set authorization and define transactions.

View Updating Rule

Any view is theoretically updateable, if changes can be made to the tables that effect the desired changes in the view. Data consistency is ensured since changes in the underlying tables are transmitted to the view they support. Logical data independence reduces maintenance cost.

High Level Inserts, Update and Delete

RDBMS supports insertion, updatation and deletion at a table level. With this the RDBMS can improve performance by optimizing the path to be undertaken to execute the action. Ease of use is improved since commands act on set of records.

Physical Data Independence

The execution of the requests and application programs is not affected by the changes in the physical data access and storage methods. Database administrators(DBA's) can make the changes to physical access and storage methods, which improve performance but do not change in the application programs or requests. This reduces the maintenance costs.

Logical Data Independence

Logical changes in tables and views such as adding/deleting columns or changing field lengths do not necessitate modifications in application programs or in the format of adhoc requests.

Integrity Independence

Like table/view definitions, integrity constraints are stored in the on-line catalog and therefore can be changed without necessitating changes in application programs or in the format of the requests.

Distribution Independence

Application programs are not affected by change in the distribution of the physical data.

Non-subversion Rule

Different levels of the language cannot subvert or bypass the integrity rules and constraints. In order to adhere to this rule, RDBMS must have an active on-line catalog that contains the constraints.

5.3 Data Integrity

Data integrity ensures that the data entered into the database by the user is checked for its correctness and is the data that is supposed to go into the database. Thus, the data gets automatically validated when it is entered as per the instructions or commands of the designer and by this way RDBMS ensures that the application is of a high degree of data security and integrity.

RDBMS ensures data integrity through automatic validation of data using integrity constraints. The integrity constraints are non-procedural constructs and by just specifying those, the designer can automate the validation process at the time of data entry. Popular integrity constraints are NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK constraints.

Data integrity falls into the following categories

- ❖ Entity integrity
- ❖ Domain integrity
- ❖ Referential integrity

Entity integrity

Entity integrity ensures that each row can be uniquely identified by an attribute called the primary key. The primary key cannot have a NULL value.

Domain Integrity

Domain integrity refers to the range of valid entries for a given column. It ensures that there are only valid entries in the column.

Referential integrity

Referential integrity ensures that for every value of a foreign key, there is a matching value of the primary key.

5.4 Keys

Enforcing data integrity ensures that the data in the database is valid and correct. Keys play an important role in maintaining data integrity.

The various types of keys that have been identified are the

- ❖ Candidate key
- ❖ Primary key
- ❖ Alternate key
- ❖ Composite key
- ❖ Foreign key

Candidate key

It is important to have an attribute in a table that uniquely identifies a row is called a candidate key. This attribute has values that are unique. Consider the table airport.

Airport

Airlineno	SourceName	DestinationName
023451	London	India
023452	Newyork	Tokiya
023456	Madurai	Bangalore

The value of the attribute Airlineno is unique in every row. So this is called as candidate key. This is also called surrogate key.

Primary Key

The candidate key that you choose to identify each row uniquely is called the primary key. In the table Employee, if you choose Empid# to identify rows uniquely, Empid# is the primary key.

Employee

Empid#	Empname	Salary
1437	Kalaiselvi	5000
1337	Meena	5000
1137	Diana	5000

Alternate key

A candidate key that is not chosen as a primary key is an alternate key. In the table Purchase

If you choose Serial# as the primary key, Itemcode is the alternate key. It is important that you understand that a primary key is the only sure way to identify the rows of a table. Hence, an alternate key may have the value NULL. A NULL value is not to be permitted in a primary key since it would be difficult to uniquely identify rows containing NULL values.

Purchase

Serial#	Itemcode	Price
01437	I454	500
01438	I667	550
01439	I777	200

Composite Key

In certain tables, a single attribute cannot be used to identify rows uniquely and a combination of two or more attributes is used as a primary key. Such keys are called composite keys.

Consider the following table, customer, which is used to maintain the purchases made by various customers.

Customers

Custcode	Productcode	Qty
c0147	P454	50
c0148	P777	55
c0147	P777	20

You can see all values are not unique for any of the attributes. However, a combination of Custcode and Productcode results in all unique values. Hence, the combination can be used as a composite primary key.

Foreign key

When a primary key of one table appears as an attribute in another table, it is called the foreign key in the second table. A foreign key is used to relate two tables.

Consider the tables, department and staff.

Department

Deptcode	Name	HOD
CSC	COMPUTER	SARAVANAN
EEE	ELECTRONIC	MATHU
EEC	ELECTRICAL	BALAMURUGA

Staff

Staffcode	Deptcode	Name
S345	CSC	DIANA
S346	EEC	KALAI
S347	EEE	MEENA

Since deptcode is unique in the department table, you can choose it as a primary key. Since it appears in the staff table as an attribute, it will be the foreign key in the staff table. You must ensure that the values of the foreign key match with any one value of the primary key.

You can use the following conventions to represent the keys in the table structure.

Fk – Foreign key

Pk – Primary key

Department
Deptcode (Pk)
Name
HOD

Staff
Staffcode (Pk)
Deptcode (Fk)
Name

5.5 Short Summary

- ❖ Enforcing data integrity ensures that the data in the database is valid and correct. Keys play an important role in maintaining data integrity.
- ❖ Rules of RDBMS are Information Rule, Guaranteed Access Rule, Systematic Treatment of Null Values, Active On-Line Catalog Based on the Relational Model, Comprehensive Data Sub-language Rule, View Updating Rule, High Level Inserts, Update and Delete, Physical Data, Independence, Logical Data Independence, Integrity Independence, Distribution Independence and Non-subversion Rule.
- ❖ The various types of keys that have been identified are the
 - Candidate key
 - Primary key
 - Alternate key
 - Composite key
 - Foreign key

5.6 Brain Storm

1. What are the various types of keys we used in RDBMS?
2. Why we need keys?
3. What is Data integrity?
4. Explain about the different categories of integrity?
5. Explain the rules of RDBMS?

END

Lecture 6

Client-Server Architecture

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about client server architecture
- ☒ Describe the Client and server parts.
- ☒ Describe about server and types of server
- ☒ Describe about advantages of client/server concepts
- ☒ Describe the multithreaded server architecture

Coverage Plan

Lecture 6

- 6.1 Snap shot
- 6.2 Client server architecture
- 6.3 Clients
- 6.4 Servers
- 6.5 Types of servers
- 6.6 Multithreaded server architecture
- 6.7 Advantages of client/server computing
- 6.8 Short summary
- 6.9 Brain Storm

6.1 Snap Shot

In this session we discuss about the client/server architecture and also we discuss about the types of client/server architecture concepts, and its usage. In this session we also discuss about server and types of server. We describe about multithreaded server architecture. Discuss about the pros and cons of client/server architecture.

6.2 Client-Server Architecture

Client/server computing uses local processing power-the power of the desktop platform. A simple definition of client/server computing is that *server* software requests for data from *client* manipulates the data and presents the results to the user or, acting as a server, sends the results to the client that requested it. To make it sound more technical: most of the application processing is done on a programmable desktop computer, which obtains application services from another computer in a master/slave configuration.

The emphasis of client/server computing is not hardware. The hardware components have been around for quite a while. While the hardware does indeed deserve some attention, the major focus of this book will be the technology that makes client/server computing possible-the software.

Client/server computing provides Information Systems (IS) professionals with options by allowing applications to be segmented into tasks. Each task can be run on a different platform, under a different operating system and with a different network protocol. Each task can be developed and maintained separately, accelerating application development. Data can be placed closer to the user. Users can access their data with a comfortable interface and tools for manipulating that data into meaningful information. Suddenly users are able to do more for themselves, rely less on IS for assistance, and receive quick turnaround time for applications requested of IS. The end result is more efficient use of existing equipment, increased effectiveness of future spending, and more productive workers.

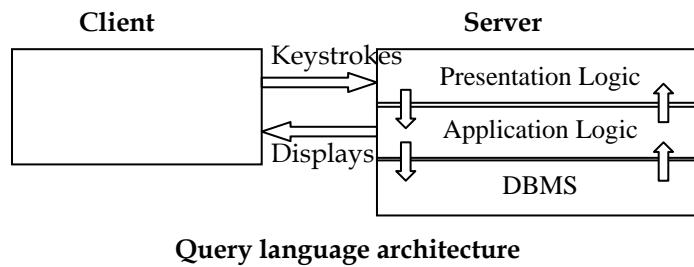
Application processing performed on more than one machine in a network may be either distributed computing or cooperative processing. Distributed computing partitions the data between two or more computers, which may be geographically dispersed. The user has transparent access to the data. Cooperative processing splits an application's functions

between two or more computers in a peer-to-peer relationship. Most client/server network structures are based on distributed access, not distributed computing. Client/server architecture uses a master/slave configuration where processing may be performed by both the master and the slave.

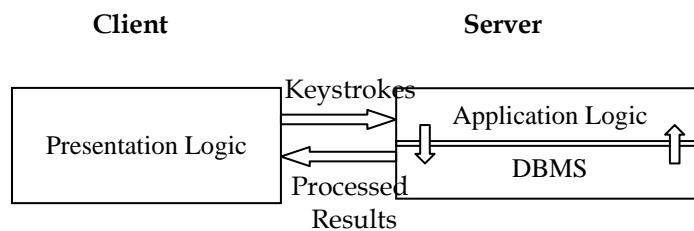
The use of open systems (hardware, software, operating systems, databases, and networks) enhances client / server computing. By adhering to standards, open hardware and software can provide interoperability and portability. Open systems offer IS greater flexibility in linking divergent technologies. However, in some cases there are multiple standards and in others, none at all.

While this new technology seems very straightforward, it is actually more complex than the technology it is replacing. Even if all the pieces adhere to standards, multiple components must be integrated and managed. The network and all its nodes are treated as “the computer.: Department LAN-based systems become part of the enterprise-wide network. Before changes are made to departmental systems, impact on the enterprise-wide system must be identified and resolved.

Before an organization ventures into the world of client/server computing, an application's functions should be reviewed. This breakdown parallels the division of duties in client/server computing.

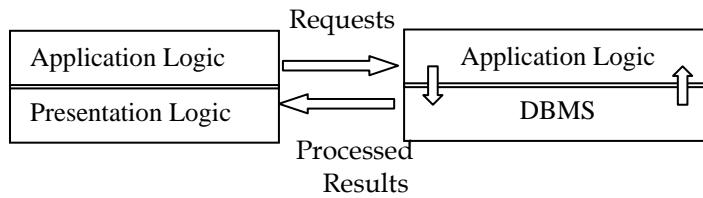


Query language architecture



Original Client/server applications

Client Processed SQL Server



Distribution of processing in client/server model

The last sections discussed about the database and the system that is required to operate on the database and the most popular system of today i.e., the RDBMS.

The RDBMS acts as a very good interface between you and the database and processes data as per your instructions. Instructions come to the RDBMS by way of commands and other interface specific operations. The RDBMS performs the following two distinct operations or tasks when you operate with it:

- Gets the commands from you using a friendly user interface and displays the results through friendly reporting
- Processes the commands given by you and processes data subsequently based on the commands

The Fig illustrates these operations of the database system.

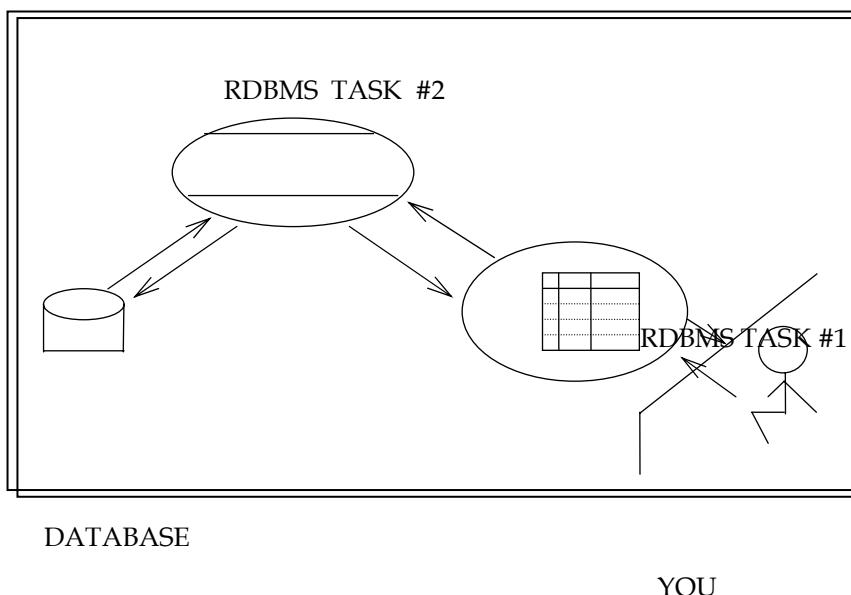


Figure 1 Different operations for a database

Task #1 interacts with the user and receives commands through a highly user-friendly interface, sends the commands to task #2 and displays the results got from task #2 with

highly user-friendly reporting facilities. Task #2 processes the commands and as per the instructions, processes the data and sends the results to the task #1.

This way of processing where the overall job processing is done through two processes, one working as a front-end and other working as a back-end is called *Client-Server Model* and the database system that works on this architecture is called *Client-Server System*. Task #1 is called *Client* because it always requests for the command to be processed and it works as a front-end in the whole system. Task #2 is called *Server* because it serves the requests of task #1 by way of processing the commands given.

Clients always interact with the end-user and servers always interact with the database files and the vice-versa is not possible. The client and the server interact with each other to effectively do the database processing as requested by the end-user.

6.3 Clients

The client or the front-end is usually the *application* that interacts with the end-user through the keyboard, display and pointing device such as mouse. The client portion has no data access responsibilities. It concentrates on *requesting, processing and presenting* the data managed by the server portion. The client workstation can be optimized for its job. For example, it might not need large disk capacity or it might benefit from graphic capabilities. Client deals with user interface and data presentation.

The client hardware is the desktop machine that runs client software. It could be a micro or a workstation. The client hardware has to be robust enough to support the presentation requirements and the client based processing of the application.

The client software formulates data requests and passes the requests to the network software. This software sends the requests to the server, accepts the results from the server and passes the results back to the client software. The client software may perform some application logic on the results of the request before passing it on to the presentation component of the software.

The presentation component produces the interface that the user views and interacts with. It is often, but not always, a graphical user interface. GUIs provide a graphic-oriented presentation front-end to applications and provide (or simulate) multitasking processing (the

ability to run two or more applications at the same time). The major windowing environments are Windows from Microsoft, Presentation Manager from IBM, Motif from Open System Foundation (OSF), and OpenLook from USL.

For those who have not seen or used a windowed environment: Using a mouse as a pointing device, users select options by positioning the cursor on the item of choice and clicking (pressing a button on the mouse). Options are displayed as icons or lists. If a user must choose from a valid list of values, a scrollable list box appears, the user scrolls through the list until the choice is highlighted and then clicks. The user can toggle between multiple tasks and / or programs. Each task/program is in its own window and the user can position the windows so that they are side-by-side or overlaid. Every effort is made to reduce keyboard use. The interface hides most, if not all, of the requirements for data retrieval and system administration from the user.

There is also an operating system running on the client hardware, which may or may not be the same as the server's operating system. The client operating system must be robust enough to support the presentation processing and application logic processing required for the applications. Each of the major windowing environments was designed for a particular operating system. Windows runs under DOS, Presentation Manager under OS/2, and Motif and OpenLook under UNIX.

There is also communications software running on the client hardware. The client-based network software handles the transmissions of requests and receives the results of the requests. However, individual network operating systems do not support all available client and server operating systems.

The client may also be executing runtime support for applications generated with client/server development tools. Using the development tool, the application logic is specified and partially executable code is generated. The generated code is executed by the client's runtime version of the software.

In some cases, a client is actually a server acting as a client (then the server is called an agent) by requesting data from another server.

6.4 Servers

The server portion runs the RDBMS software and handles the functions required for concurrent, shared database access. The server portion receives and processes the commands originating from the client applications. The computer that manages the server portion can be optimised for its duties. It can have large disk space and fast CPUs.

A server is the machine that runs data management software that has been designed for server functionality. Compared to a desktop micro, server hardware has increased memory capabilities; increased storage capabilities; increased processing power, including, in some cases, parallel processors, improved cycle times; and improved reliability with built-in reliability features, such as uninterruptible power supply, fault tolerance, and disk mirroring.

When evaluating server hardware, the following measures should be kept in mind:

- **Reliability.** How often does it fail? What is the mean time between failures?
- **Availability.** How quickly does the system come back into service after a failure? To assure high availability, some systems have self-healing routines, continuous processors, fault-tolerant hardware, alarms to highlight problems before they become serious and continuous functioning in a reduced configuration. Some can be rebooted from remote sites.
- **Flexibility and scalability.** How easily can the server be expanded as processing needs grow?

A server has operating system software, data management software, and a portion of the network software. The operating system has to interact reliably with network software and be robust enough to handle server technology.

The data management software responds to requests for data retrieves, updates, and stores data. Relational databases have become the *de facto* standard structure and SQL the *de facto* standard data access language. Gateways to non-relational data sources are offered by most major vendors of server DBMS software, as well as third-party vendors. Server DBMS software incorporates many of the services taken for granted in mainframe-based applications, such as backup and recovery routines and testing and diagnostic tools.

Two of the keys to successful client/server applications are the separation of presentation management from other application services and the distribution of application logic between the client and the server. In most cases, the distribution of application logic is determined by the developer when the application is designed or installed. In some cases, this split can be determined by the application software, based on current resources, at runtime.

<u>Client Functions</u>	<u>Server Services</u>
GUI	File, print, database server
Distributed application processing	Distributed application processing
Local application	E-mail
E-mail	Communications
Terminal emulation	Network management
	Resource management
	Configuration management

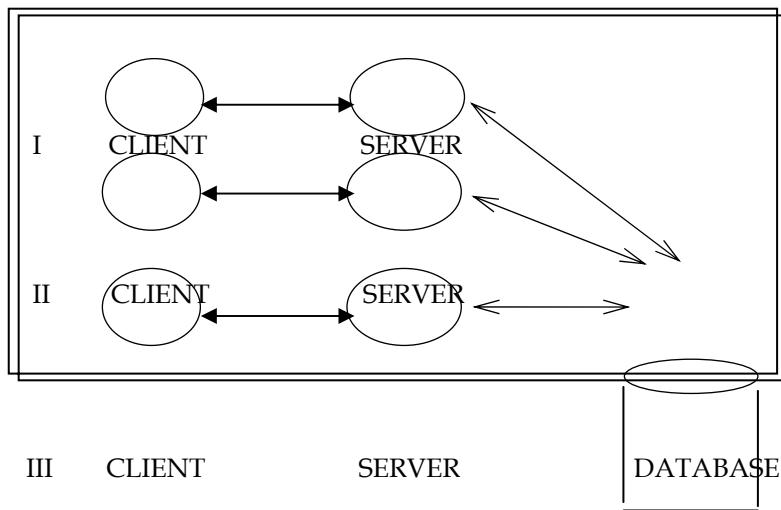


Figure 2 Multi-User Database environment with Client-Server Architecture

6.5 Types of Servers

Every client working in the two-task architecture requires a server to process the commands and the data on its behalf. While working on this kind of a setup, it is possible to optimize the performance by making a server to process the requests of more than one client. This optimization can be realized when multiple clients try to process the data in a centralized database using their own servers.

In Figure 2 the server in each case is the RDBMS server that services the requests coming from each client and each server acts as a dedicated one for each client. This client-server architecture is also called *dedicated client-server architecture*.

When multiple clients get their requests serviced by a single server it becomes *multi-threaded client-server architecture* and the server servicing in that environment is a called *multi-threaded server*. (Figure 3)

The six types of servers are:

- File server
- Application server
- Data server
- Compute server
- Database server
- Communication server

Their difference are based on where data is handled and how it is transferred.

File server

File servers manage a work group's applications and data files, so that they may be shared by the group. File servers are very I/O oriented. They pull large amounts of data off their storage subsystems and pass the data over the network. When data from the file is requested, a file server transmits all records of a file and the entire index to the client. The client either selects records(based on query criteria) as they are received or loads the whole file and its index into memory and then reviews it. File servers require many slots for network connections and a large capacity, fast hard disk subsystem.

File locking is handled by locking the entire file or by locking byte ranges. There is no differentiation between read locks and write locks at this level. When multiple users access

Shared files, the file server engine checks for contention. If it detects contention at the file lock level, it waits until the resource is free.

There can be no scheduling multiple users, no cache management, no lock manager, and minimal concurrency control in the DBMS sense because there is no single engine to which all the required information is available. These DBMS like features are usually handled by the client software which anticipates the best way to process the data. Unless each data file is

locked for exclusive use and some client side indexing technique is used, all data must be moved across the network before filtering, sort, or merge operating can be applied. This situation forces heavy network traffic.

Two techniques used to minimize the amount of data that passes over the network are:

Organizing data so that the data needed by a particular application request is stored in a single contiguous block.

Storing copies of data accessed by more than one user to help with concurrency problems.

Of course, these techniques require developers to build integrity and synchronization handling into the processing of the application.

Application Server

An application server is a machine that serves as a host replacement (and in some cases actually is a host.). when applications are downsized from a host, one option is to install the applications on a smaller machine that runs the same software and to hook all the users to the new box. This process requires no modifications to the host based application software. For client server applications that are classified as host based the host is the serve to the GUI based clients, as illustrated in Figure 8.3

Data Server

A data server is data oriented and used only for data storage and management, as illustrated in Figure 8.4. a data server is used in conjunction with a compute server and may be used by more than one compute server. A data server does not perform any application logic processing). The processing done on a data server is rule based procedures, such as data validation, required as part of the data management function.

Data servers performs multiple searches through large amounts of data and frequently update massive tables. These tasks require fast processors, large amounts of memory, and substantial hard disk capacity. However, for the most part, these computers send relatively small amounts of data across the network.

Compute Server

A compute server passes client requests for data to a data server and forwards the results of those requests to clients. Compute servers may perform application logic on the results of the data requests before forwarding data to the client.

Compute servers require processors with high performance capabilities and large amounts of memory but relatively low disk subsystem capacity and throughput.

By separating the data from the computation processing, and organization can optimize its processing capabilities. Since a data server can serve more than one compute server, compute intensive applications can be spread among multiple servers.

Database Server

This is the most typical user of server technology in client server applications. Most if not all, of the applications is run on the client. The database server accepts request for data, retrieves the data from its database (or makes a request for the data from another node), and passes the results of the request back to the client. Compute servers working with data servers provide the same functionality.

Using a database server or a combination of data and compute servers, the data management functions on the server and the client program consists of application specific code as well as presentation logic. Because the database engine is separate from the client, the disadvantages of file servers disappear. Database servers can have a lock manager, multi-user cache management, and scheduling and thus have no need for redundant data.

Data base and data compute servers improve request handling by processing a SQL client request and sending back to the client only the data that satisfies the request. This is much more efficient in terms of network load than a file server architecture, where the complete file is often sent from the server to the client.

Because SQL allows records to be processed in sets, an application can, with a single SQL statement, retrieve or modify a set of server database. Older database system have to issue separate sequential requests for each desired record of each of the base tables. Because SQL

can create a results table that combines, filters, and transforms data from base tables, considerable savings in data communications are realized even for data retrieval.

The requirements for these servers are a function of the size of the database, the speed with which the database must be updated, the number of users, and the type of network used.

Communication Server

Communication servers provide gateways to other LANs networks, midrange computers, and mainframes. They have relatively modest system requirements, with perhaps the greatest demands being those for multiple slots and fast processors to translate networking protocols.

6.6 Multi-Threaded Server Architecture

Multi-threaded servers improve performance in database processing because they reduce the number of processes and the corresponding load to the CPU. In a multi-user database environment, the clients and servers are all individual processes to the computer system and it is always recommended that the RDBMS is configured to work with Multi-threaded servers. The following figure explains the multi-threaded server architecture.

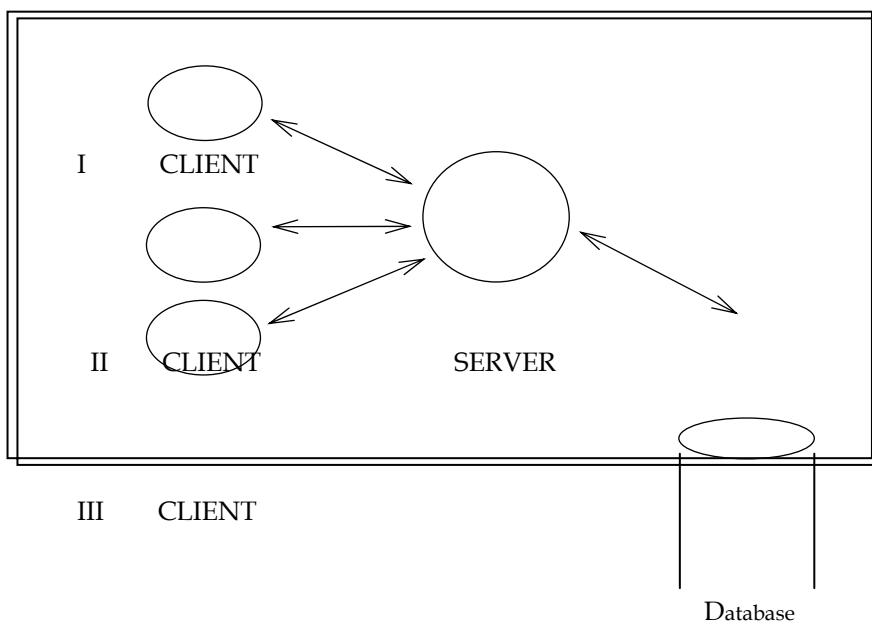


Figure 3 Multi-threaded Client Server Architecture

Concept of Multi-threading

Multi-threaded server keeps track of the jobs submitted by one or more clients and switches context between the jobs through an internal scheduling. The job scheduling is usually a pre-emptive one which means the server executes a client's job request for a definite capsule of time and the job gets completed in many instances. In the case of the expiry of the time capsule allotted, the server keeps the job's status as of that instance of time in the process stack and switches to the next client's job waiting in the queue. When the job thus pre-empted gets its next chance, the server resumes the execution from the point where it left in the previous instance. The job switch is so quick that the client does not realise it is waiting and also the client gets a feeling that it has a dedicated server for itself. The word *multi-threading* is used because individual job status is maintained like a thread inside the server process. The RDBMS of modern age works on Client-Server architecture and works on multi-threaded server technique explained above.

A thread is the smallest unit of execution that the system can schedule to run; a path of execution through a process.

6.7 Advantages of Client-Server Computing

Client-Server setup facilitates a better planning and allocation of resources when you want to implement a computerized business solution. With Client-Server computing, it is a lot easier to access information. The computing environment can be heterogeneous with the best of the operating systems, languages and the computing platforms and the data access to the end-user can still be transparent and distributed. Operations of a corporate business setup can be widespread and the location can no longer be a constraint.

In a nutshell, the advantages of Client-Server computing are as follows:

- Access to more information is a lot easier
- Investments can be protected against obsolescence
- Planned and marginalised investments
- Free of processor, OS and software tool knowledge and operations dependencies
- Partitioning of applications
- Enterprise-wide transparent resource sharing
- Client/Server Computing is Easily Implemented

- Current Desktop Machines Are Sufficient
- Minimal Training Is Required
- All Data Are Relational
- Development Time Is Shorter

Implementing distributed remote data sharing in a transparent way can be done using a client-server setup. The following figure illustrates this further.

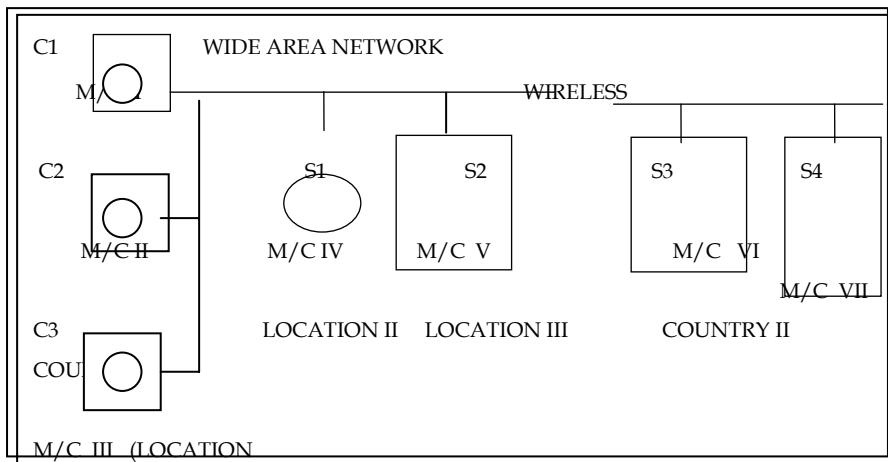


Figure 4 Wide Area Network

In the above figure, the client-server setup is explained in a massive *wide-area network*. Wide-area network spreads across regions and countries and taps the resources through communication links including telephone lines, satellite communication network. Through proper protocols or arrangements, the machines and the clients running in those machines can interact with servers of their choice anytime and anywhere. The servers could be any RDBMS and the machines running those servers could be from multiple vendors like SUN, HP, IBM and so on.

Servers can also share multiple clients and clients can run on PCs in bitmapped GUI environments. Servers can run on Super-mini machines or even mainframes. Together what you achieve is the highest utilization of the resources transparently in an enterprise-wide network. RDBMS of modern age are designed to take advantage of client-server computing. Examples of client-server RDBMS that run on a variety of platforms include SQL SERVER, Oracle, Sybase, INFORMIX, Ingres and so on.

6.8 Short Summary

- Ý Client/server computing is that *server* software requests for data from *client* manipulates the data and presents the results to the user or, acting as a server, sends the results to the client that requested it.
- Ý Client software sends the requests to the server, accepts the results from the server and passes the results back to the client software.
- Ý A server has operating system software, data management software, and a portion of the network software. The operating system has to interact reliably with network software and be robust enough to handle server technology.
- Ý *Dedicated client-server architecture* is the server in each case is the RDBMS server that services the requests coming from each client and each server acts as a dedicated one for each client.

6.9 Brain Storm

1. Explain about client/server computing?
2. What is client?
3. What is server?
4. Differentiate the client/server architectures?
5. Mention types of Servers.
6. Explain about multithreaded server architecture?

808

Lecture 7

Clients

Objectives

After completing this lesson, you should be able to do the following :

- ☞ Discuss about the Client concepts.
- ☞ Describe the details of open client and front-end client.
- ☞ Describe the features of ODBC.
- ☞ Describe about CUI and GUI concepts.
- ☞ Differentiate the CUI clients and GUI client concepts.

Coverage Plan

Lecture 7

- 7.1 Snap shot
- 7.2 Front ends
- 7.3 Open clients
- 7.4 ODBC support
- 7.5 Character based clients
- 7.6 GUI based clients
- 7.7 Short summary
- 7.8 Brain Storm

7.1 Snap Shot

In this session we discuss about the client concepts for this we deal the front-end clients, open clients, character-based clients and graphical based clients. In this lecture we deal the Open database connectivity concepts. We differentiate the character-based and graphical based clients.

7. 2 Front-ends (Clients)

Clients act as front-ends of the data processing system setup and they deal with the user-interface and applications which are developed to present data better and to do data entry, validation and manipulation with a friendly interface. Front-ends, as their names indicate act closely with the end-users and hence they offer excellent user-friendly interfaces to accept input commands and data and to display results in the required formats. A typical client application interface is presented in the following figure:

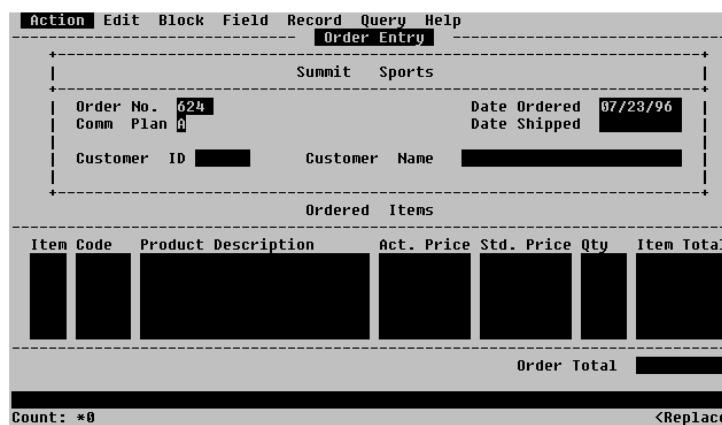


Figure 1.6 : Client Application Interface

The above figure illustrates an order entry application and has required fields which are entered with proper values by the user. The application is also built with validation constructs to check for the validity of the data entered by the user. The interface is intuitive and helps the user in understanding the application better by presenting a friendly entry format.

Front-ends are offered by all vendors who offer RDBMS products. Front-ends are also offered by independent third-party vendors. While front-ends supplied by the RDBMS vendors

usually talk to their own RDBMS servers, front-end tools offered by third-party vendors talk to a variety of RDBMS servers.

7.3 Open Clients

Open front-ends or client tools are those who can connect to any RDBMS and exchange information. The implementations of those clients are such that separate modules (Also referred to as *Drivers*) are built in to extend support for multiple RDBMS connectivity. Front-ends of today's technology are all *Open Clients*. Popular open clients are PowerBuilder, Oracle Forms / Reports, VisualBasic and so on.

7.4 ODBC Support

Microsoft Corp., USA has devised interface standards for database connectivity from open front-ends, which is called Open Database Connectivity. With ODBC in picture, a front-end application, which wants to talk to a specified RDBMS server, need not have the connectivity pre-defined before writing the application. The Open Database Connectivity (ODBC) interface allows applications to access data in database management systems (DBMS) using Structured Query Language (SQL) as a standard for accessing data.

The interface permits maximum *interoperability* – a single application can access different database management systems. This allows an application developer to develop, compile, and ship an application without targeting a specific DBMS. Users can then add modules called database *drivers* that link the application to their choice of database management systems.

7.5 Character-based Clients

Character-based Clients work on dumb terminals and are usually tied tightly with an RDBMS server and are not of Open Client kind. In Character-based setup, the interface screen resolution is one character and all the presentations are drawn using one or other characters. Character-based Clients are slowly loosing demand with the advent of Windows based environment. Character-based Clients are descriptive in nature and do not offer extensive visual interface and do not deal with graphical representation of information.

7.6 GUI based Clients

Graphical User Interface (GUI) based Clients are Windows based and work on bit-mapped interface rather than character-based one. In a bit-mapped screen, the interface resolution is a bit or a pixel. This facilitates a greater resolution on the Computer monitor and as a result, pictures and other graphical visual presentations are possible. GUI based clients, as their name suggest, take advantage of the graphical native look-and-feel environment and give a lot of intuitive features to the users including multi-media and video motion picture interface. Images with 3D effect, command buttons, pick-and-choose objects, selection by clicking over the region by mousses are some of the features of a GUI based interface.

GUI based Clients address closer to real-world application situations and help user to work in a more intuitive and interactive manner. Following figure presents one such GUI based Client interface.

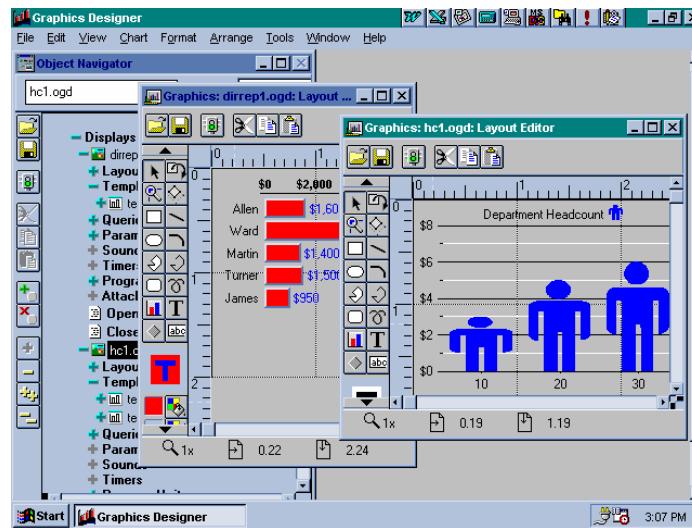


Figure 1.7 GUI Based Clients

GUI based clients are of non-procedural type which means programming efforts are greatly reduced. Visual possibilities improve intuitive learning and the GUI based front-ends could very well sit over a desktop and access information through a Workgroup environment. GUI based clients are fast catching up and becoming the de facto standards of operations.

7.7 Short summary

- ➲ Front-ends indicate act closely with the end-users and hence they offer excellent user-friendly interfaces to accept input commands and data and to display results in the required formats.
- ➲ The implementations of those clients are such that separate modules are built in to extend support for multiple RDBMS connectivity. Front-ends of today's technology are all Open Clients.
- ➲ ODBC is a library that acts as a middle ware between any client and any server and talks in their native language.
- ➲ Character-based Clients work on dumb terminals and are usually tied tightly with an RDBMS server and are not of Open Client kind.
- ➲ GUI based clients, as their name suggest, take advantage of the graphical native look-and-feel environment and give a lot of intuitive features to the users including multimedia and video motion picture interface.

7.8 Brain Storm

1. How graphical based clients are differing from character based clients?
2. Explain about front-end clients?
3. Explain the concepts of ODBC?
4. Define the open clients?

Lecture 8

Introduction to SQL

Objectives

After completing this lesson, you should be able to do the following

- ⌚ Discuss the Different generations of languages
- ⌚ Describe about Structured query language
- ⌚ How to logging to SQL and what is SQL * plus
- ⌚ Understand how to create, alter a table and understand to manipulate with data
- ⌚ Discuss about Selecting data from a table

Coverage Plan

Lecture 8

- 8.1 Snap shot
- 8.2 Generation of languages
- 8.3 SQL
- 8.4 SQL command classification
- 8.5 Data types
- 8.6 Create, view, manipulate data
- 8.7 Short summary
- 8.8 Brain Storm

8.1 Snap Shot

In this session we learn about the generation of language and give example for each generation. Discuss about the structured query language and SQL * plus. In this lecture we describe about how to logging in to SQL * plus and how to operate the commands.

In SQL we have several command categories each categories have different type of commands. We learn all these commands and we also learn about the data types of oracle.

8.2 Generation Of Languages

First Generation of Languages [Machine Language]

This is considered to be the First generation of Language. Every computer has its own machine language, which is the only language understood by the computer. Originally, programs were written in machine language. Machine language instructions are represented by binary numbers i.e., sequences consisting of 0's and 1's.

For eg:- 001010001110

could represent a 12-bit machine language instruction. This instruction is divided into two parts, an operation code(or op code) and an operand, e.g:-

p code	Operand
001	010001110

The Op code specifies the operation (add, multiply, move....) and the operand is the address of the data item that is to be operated on. Besides remembering the dozens of code numbers for the operations, the programmer also has to keep track of the addresses of all the data items. Thus programming in machine language is highly complicated and subject to error. Also, the program is machine dependent, i.e good only for a particular machine since different computers uses different machine languages.

Second Generation of Languages [Assembly Language]

The Second Generation of Computer Languages saw a difference in the development of languages. Here, the sequences of 0's and 1's that serve as operation codes in machine language are replaced by *mnemonics* (memory-aiding, alphabetic codes). Each assembly-language instruction may have three parts, not all of which is required to be specified. The first part is the label or tag. These are the programmer-defined symbols that give the address of the instruction. Then follow the op code and the operand, as with the machine-language instruction. Here the operands are symbolized ad hoc, in letters chosen by the programmers.

Third Generation of Languages [High-Level Languages]

The Third Generation of Language saw a very big change with the advent of high-level languages where programming was made comparatively easier. These are called high-level languages or compiler languages, since they require a special program called a *compiler*, which translates programs written into machine language. The original program written in high-level language is called as the *source program*, and its translation into machine language is called the *object program*. Common high-level languages are FORTRAN, COBOL, BASIC, etc.

Fourth Generation of Languages

Even-though third generation of languages such as C, BASIC brought in a new change in programming, in order to use these languages one must understand the syntax and the logic for writing codes and compiling them. The data that must be located and read must also be specified since there is no way of automatically searching and retrieving the information. The Fourth Generation language made this very simpler and easier by bringing out a very English-like Language. Structured Query Language falls into this generation of computer languages.

8.3 Structured Query Language (SQL)

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

Structured Query Language is a simple English-like language. SQL is also pronounced as “sequel” and consists of layers of increasing complexity and capability. End-users with little or no experience in data processing can learn SQL features very quickly. It is a Fourth Generation Language.

SQL was first introduced by IBM Research and was introduced into the commercial market first by Oracle Corporation in 1979. A committee at the American National Standards Institute has endorsed SQL as the standard language for RDBMS.

SQL provides the following functionalities:

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

Non-Procedural Language

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

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

Database Access through SQL

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

SQL * Plus

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

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

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

Logging to Oracle

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

Plus80w (From Oracle 8) in the Start-Run option.

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

The Database Administrator configures this database alias name.

The figure explains this:

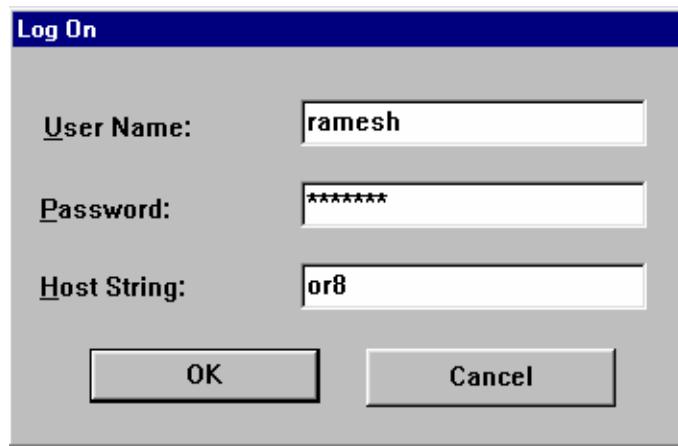


Figure 8.1 Logging to Oracle

Shortcuts to starting SQL * Plus

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

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

PLUS80W SCOTT/TIGER (or) SQLPLUS SCOTT/TIGER

If the user is connected to a network, SQL * Plus can be started by giving

PLUS80W SCOTT/TIGER@ORACLE

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

SQL Buffer

The area where SQL * Plus stores the most recently typed SQL commands or PL/SQL commands is called the SQL Buffer. The command remains in the buffer until another command is entered. Thus, if the same command or block has to be re-executed or edited, it

can be done without retying the same. A detailed list of SQL * Plus commands are dealt later.

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

8.4 SQL Command Classifications

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

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

Before discussing about creating tables, a detail description about datatypes is dealt with.

8. 5 Datatypes

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

- BUILT-IN
- USER-DEFINED (dealt in a later chapter)

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

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

Character Datatypes

Char(n)

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

Syntax

Char(n)

Example:

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

Varchar2(size)

Varchar2 datatypes are variable length character strings. They can store alpha-numeric values and the size must be specified. The maximum length of varchar2 datatype is 4000 bytes. Unlike char datatype, blank spaces are not padded to the length of the string. So, this is more preferred than character datatypes since it does not store the maximum length.

Syntax

Varchar2(Size)

Example:

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

Numeric datatypes

Number

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

Syntax

Number (p, s)

Example:

Sal number – Here the scale is 0 and the precision is 38

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

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

DATE datatype

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

RAW Datatype

RAW(n)

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

Syntax

Raw(n)

LONG Datatype

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

LONG RAW Datatype

Stores upto 2 Gigabytes(GB) of raw binary data. The use of LONG Values are restricted. The restrictions are :

- A Table cannot contain more than one LONG column
- LONG columns cannot appear in Integrity constraints (dealt later)
- They cannot appear in WHERE, ORDER BY clauses of SELECT statements
- Cannot be a part of expressions or conditions
- Cannot appear in the SELECT list of CREATE TABLE as SELECT statement.

LOB Datatypes

In addition to the above datatypes, Oracle 8 supports LOB datatypes. LOB is the acronym for LARGE OBJECTS. The LOB datatypes stores upto 4 GB of data. This datatype is used for storing video clippings, large images, history documents etc. LOB datatypes can be

CLOB	Character Large Objects (Internal LOB)
BLOB	Binary Large Objects (Internal LOB)
BFILE	Binary File (External LOB)

The LOB datatypes store values, which are called locators. These locators indicate the place where the objects are stored. The location can be out-of-line or in an external file like CD-ROM. In order to manipulate the LOB type of data, DBMS_LOB package is used.

8.6 Create, View, Manipulate Data

8.6.1 Data Creation through SQL

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

Using the CREATE TABLE Command

Oracle Database is made up of tables that contain rows (horizontal) and columns (vertical). Each column contains a data value at the intersection of a row and a column. The table definition contains the name of the attribute (property or field) and the type of the data that

the column contains. To create a table, use CREATE TABLE command. CREATE command is used to define the structure of a table or any object.

Syntax:

```
CREATE TABLE <tablename> (column1 datatype,  
                           column2 datatype,...);
```

Here, tablename refers to the name of the table or entity, column1 is the name of the first column, column2 the name of the second column and so on. For each column there must be an appropriate datatype which describes the type of data it can hold. The statement is terminated by a semi-colon.

The following example illustrates the creation of a table.

Example 2.1

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

This would display:

Table created.

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

The structure of the data would look like:

Name	Type
EMPNO	NUMBER
EMPNAME	CHAR(10)
DOJ	DATE

While creating tables, consider the following points:

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

8.6.2 Viewing the Table structure

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

Syntax:

DESC[ribe] <tablename>

Example:

DESC EMPLOYEE

The output would look like:

Name	Type
EMPNO	NUMBER
EMPNAME	CHAR[10]
DOJ	DATE

Using the ALTER TABLE Command

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

Syntax:

```
Alter table <tablename> add (column1 datatype, column2 datatype ...);  
Alter table <tablename> modify (column1 datatype, column2 datatype ...);
```

Example:

Consider the previous example where a new column called Salary is to be added.

```
ALTER TABLE employee ADD (salary NUMBER);
```

After this command is issued, it displays,

Table altered.

Alter command can be used to modify the existing datatype of a column to another datatype or to change the column width. The next example illustrates this:

Example:

```
ALTER TABLE employee MODIFY (empname VARCHAR2 (20));
```

Here the number of characters that the column *empname* can contain is increased from 10 to 20. The maximum length of the value in *empname* column is 20.

Note: In the case of decreasing the data width, the column on which the width is going to decrease must not contain any value. Whereas, in the case of increasing, there is no such restriction.

After altering the structure by adding a new column and modifying the data width, the structure of the table would look like this:

Name	Type
EMPNO	NUMBER
EMPNAME	VARCHAR2(20)
DOJ	DATE
SALARY	NUMBER

8.6.3 Removing Tables – DROP Command

Tables can be dropped the same way they are created. This command drops the data and the structure is permanently removed from the database.

Syntax:

```
Drop table <tablename>;
```

Example:

```
DROP TABLE Employee;
```

This command removes the structure along with the rows from the database and displays the message “Table Dropped”.

Using the TRUNCATE command

Unlike the DROP command, TRUNCATE command deletes all the records and does not remove the structure.

Syntax:

```
Truncate table <tablename>;
```

Example:

```
TRUNCATE TABLE employee;
```

All the records in the employee table are deleted with its structure available in the database. It displays “Table Truncated”.

8.6.4 Manipulating with the Data

After creating the table, data is entered using the Data manipulation statements. Insert command is used to insert new records, the update command replaces the old value with the new value and the delete command is used to delete records based on certain conditions.

Using INSERT Command

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

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

Case 1:

Consider inserting records onto all the fields in the table.

Syntax:

Insert into <tablename> Values (value1, value2, value3...);

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

Example:

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

The message displayed will be:

1 row created.

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

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

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

Case 2:

Consider inserting Values into Selective fields

Syntax:

Insert into <tablename>(selective column1, selective column2) Values (value1, value2);

Example:

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

Displays the feedback as

1 row created.

Case 3:

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

Syntax:

Insert into <tablename> Values (&col1, &col2, &col3...);

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

Example

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

Output will be:

```
Enter value for eno: 1247
```

```
Enter value for name: Meena
Enter value for doj: 10-jan-2000
Enter value for sal: 5000
```

```
old 1: insert into employee values(&eno,'&name','&doj,&sal)
new 1: insert into employee values(1247,'meena','10-jan-2000',5000)
```

1 row created.

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

Now, consider inserting records continuously for selective fields. This is similar to case 2.

Insert into <tablename>(selective column1, selective column2) Values (&col1, &col2);
The following example inserts records into the *empno* and *empname* columns.

Example:

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

Output

```
Enter value for eno: 1440
```

```
Enter value for name: Diana
```

```
old 1: insert into employee values(&eno,'&name')
new 1: insert into employee values(1440,'Diana')
```

1 row created.

Case 4:

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

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

Retrieving Records - Using the SELECT Statement

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

Select some data (column name(s)) FROM a table or more tables(table name(s))

Retrieval of records can be done in various ways:

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

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

Example

SELECT empno,empname,doj,salary FROM employee;

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

Example

SELECT * FROM employee;

This would display:

EMPNO	EMPNAME	DOJ	SALARY
1237	Kalai	10 - MAR - 2000	5000
1330	Saravanan		

1247	Meena	10 - JAN - 2000	5000	
1440	Diana			

'*' indicates all the column names.

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

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

Syntax

```
SELECT col1,col2,.. FROM <tablename>;
```

Example

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

The records would be displayed as follows:

EMPNAME	SALARY
Kalai	5000
Saravanan	
Meena	5000
Diana	

This statement retrieves the column values of empname and salary.

8.6.5 Conditional Retrieval

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

Syntax:

Select (column name(s)) FROM (table name(s)) WHERE condition(s) Consider selecting employee records whose salary is equal to or greater than 3000. The query can be written as,

Example:

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

The records selected will be,

EMPNO	EMPNAME	SALARY
1237	Kalai	5000
1247	Meena	5000

Example:

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

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

Retrieving Records in a sorted order

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

Syntax:

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

Example:

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

This would display

EMPNO	EMPNAME	DOJ	SALARY
1440	Diana		
1237	Kalai	10 - MAR - 2000	5000

1247	Meena	10 - JAN - 2000	5000
1330	Saravanan		

There is a difference in the display. This query displays all the records in the employee table sorted in ascending order of the employee name. By default ascending is the order in which the records are displayed. If the records need to be displayed in descending order, use DESC along with the column name. For example,

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

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

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

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

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

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

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

EMPNO	EMPNAME	SALARY
1247	Meena	5000
1237	Kalai	5000
1440	Saravanan	
1330	Diana	

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

Copying the Structure with Records

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

Syntax:

```
CREATE TABLE <tablename2> AS SELECT <columnlist> FROM <tablename1> [WHERE  
<conditions>]
```

Example:

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

```
CREATE TABLE employee1 AS SELECT * FROM EMPLOYEE;
```

Displays the feedback,

Table created.

The structure of the employee1 would be the same as the structure of employee table. The records will also be the same. To view the records of employer table, use

```
SELECT * FROM employee1;
```

This would display

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

If only specific columns need to be copied use,

```
CREATE TABLE employee2 AS SELECT empno,empname FROM EMPLOYEE;
```

The statements given above create new tables called employee1 and employee2 respectively. In the case of employee1 table, the structure, which exists in the employee table, is copied and the records are inserted. In the case of employee2 table , two columns are copied from the employee table with the records and the structure. The above statements can alternately be written as

```
CREATE TABLE <tablename>
```

and

```
INSERT INTO <tablename> SELECT <columnlist> FROM <tablename> AS (explained in the  
section on Inserting records using SELECT Statement).
```

Copying the Structure

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

```
CREATE TABLE employee3 AS SELECT * FROM employee WHERE 1=2;
```

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

Inserting records using SELECT statement

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

Syntax:

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

Example

```
INSERT INTO employee3 SELECT * FROM employee WHERE salary >=2000;
```

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

Modifying data – Update Command

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

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

Syntax:

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

Example:

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

```
ALTER TABLE employee2 ADD deptno NUMBER;
```

Before the update command is issued on issuing.

```
SELECT * FROM employee2;
```

This display will be:

EMPNO	EMPNAME	DEPTNO
1237	Kalai	
1330	Saravanan	
1247	Meena	
1440	Diana	

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

```
UPDATE employee2  
SET deptno=10;
```

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

4 rows updated.

Example:

```
UPDATE employee2  
SET deptno=20 WHERE empno >=1003;
```

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

```
SELECT * FROM employee2;
```

EMPNO	EMPNAME	DEPTNO
1237	Kalai	10
1330	Saravanan	10
1247	Meena	10
1440	Diana	20

To update more than one column, separate the columns with a ‘,’ as illustrated below

```
UPDATE employee1  
SET salary=6000,doj='09-SEP-2000' WHERE doj is NULL;
```

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

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

8.6.6 Removing Data – DELETE Statement

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

Syntax:

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

Example

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

This statement removes all the records from the employee2 table for which the deptno 20. The following example removes all the records are from the table.

Example:

```
DELETE FROM employee2;
```

8.6.7 Working with Transactions

Transaction means a logical unit of work that would make sense only on the completion of all the operations as a whole. This means that, either all the operations in the unit are completed fully or none of them are completed. In Oracle, operations can be any Data Manipulation Language statements. SQL offers two commands to simulate real-life transactions in the database. The commands are:

- COMMIT
- ROLLBACK

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

```
INSERT INTO <tablename> Values(.....);
```

```
INSERT INTO <tablename> Values(.....);
```

```
UPDATE <tablename> SET .....
```

```
COMMIT;
```

Displays

Commit complete.

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

Now , consider the following:

```
DELETE FROM <tablename>;
```

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

```
ROLLBACK;
```

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

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

Temporary Markers

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

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

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

Syntax:

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

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

```
DELETE FROM <tablename> WHERE <condition>;  
SAVEPOINT s1;  
INSERT INTO <tablename> VALUES (....);  
ROLLBACK TO s1;  
.....
```

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

Note: SAVEPOINTS can be used within transactions. After a transaction is completed, the same savepoints cannot be used in the next transaction.
--

DCL Statements are dealt in the chapter on database objects.

Differences between Delete and Truncate

The following table illustrates the difference between Truncate and Delete

Truncate	Delete
Deletes all the records	Can delete all the records or selective rows alone.
This is a DDL statement	This is a DML statement
Commits Implicitly Hence ROLLBACK	Explicit commit is required and can be used to rollback.
Hence Not possible	

SQL * PLUS Commands

DESC	Used to describe the structure of an object
SET PAGESIZE n	Sets the Pagesize to n length
SET FEEDBACK [on/off]	SQL displays the feedback for every statement that is successfully executed. By turning it off, the feedback is not displayed on the screen.
SET HEADING [on/off]	In query displays, the heading of the column name is displayed. By turning it off, the heading is not shown on the screen. It can be turned on again.
SET LINESIZE x	Sets 'x' number of lines to be displayed on the screen.

SET PAUSE on/off Pauses the display till the user presses
<Enter> key.

/ Executes the latest command in the buffer.

8.7 Short Summary

- SQL is a Structured Query Language used for all RDBMS.
 - SQL * Plus is a tool that is used to work with and the commands are executed.
 - SQL* Plus statements are not stored in the buffer.
 - DDL – Data Definition Language that defines the structure of an object
 - They are CREATE, ALTER, DROP, TRUNCATE, RENAME
 - DML – Data Manipulation Language that is used to Insert , Update and Delete records.
 - INSERT, UPDATE and DELETE commands are called DML commands
 - TCL – Controls the Transactions. They stand for Transaction Control Language
 - They are COMMIT, ROLLBACK and SAVEPOINT.

8.8 Brain Storm

1. Which Generation SQL belong to?
2. What is SQL * Plus?
3. Columns cannot be added to an existing table
 - a. True
 - b. False
4. While creating table we can insert records into it?
5. Can we drop a Column in a table?
6. How many records can be inserted using a single Insert command?
7. Using update command, can we update more than one column?
8. Using Alter table command, can we decrease the column width?
9. What is the difference between Delete and Truncate?
10. The statement that marks the end of a transaction is _____

80CR

Lecture 9

Data Integrity

Objectives

After completing this lesson, you should be able to do the following

- ❖ Discuss about data Integrity
- ❖ Describe about the types of constraints
- ❖ Create and maintain constraints
- ❖ Describe Adding Constraints
- ❖ Drop the constraints

Coverage Plan

Lecture 9

- 9.1 Snap shot
- 9.2 Data integrity
- 9.3 Adding constraints
- 9.4 Short summary
- 9.5 Brain Storm

9.1 Snap Shot

In this Lecture is to introduce about the importance of data integrity, enforcing business rules using constraints. This chapter deals with creating different types of constraints in different ways. This chapter briefly deals with enabling, disabling and dropping constraints.

9.2 Data Integrity

Data integrity ensures that the data entered into the database by the user is checked for its correctness and is the data that is supposed to go into the database. Thus, the data gets automatically validated when it is entered as per the instructions or commands of the designer and by this way RDBMS ensures that the application has a high degree of data security and integrity.

RDBMS ensures data integrity through automatic validation of data using integrity constraints. The integrity constraints are non-procedural constructs and by just specifying those, the designer can automate the validation process at the time of data entry. Popular integrity constraints are NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK constraints.

9.2.1 Constraints

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

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

Entity Integrity Constraint

Entity Integrity constraints can be classified as

- PRIMARY KEY
- UNIQUE KEY

Choosing a table's Primary Key

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

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

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

Using UNIQUE Key Constraints

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

9.2.2 Domain Integrity Constraints

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

- Not Null Constraints
- Check Constraints

Choosing NULL Constraint

By default all columns can contain null values. NOT NULL constraints are used for columns that absolutely require values at all times. NOT NULL constraints are often combined with

other types of constraints to further restrict the values that can exist in specific columns of a table.

Choosing Check Constraint

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

- Conditions must be a boolean expression that can be evaluated using the values in the record being inserted or updated.
- Conditions must not contain sub-queries
- Conditions cannot contain any SQL functions
- Conditions cannot contain pseudocolumns.

9.2.3 Referential Integrity Constraint

This constraint establishes the relationship between tables. A single or combination of columns, which can be related to the other tables, is used to perform this operation. Foreign key is used to establish the relationship. This kind of relationship can be referred to as a Parent-Child relationship. The table containing the referenced key from where other tables refer for values is called the Parent table and the table containing the foreign key is called the child table.

9.3 Adding Constraints

Constraints can be added in two different ways. They are

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

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

9.3.1 Adding Entity Integrity Constraints

Both primary key and unique key constraints can be added at the time of creation of tables. Let us consider creating a table called item_master, which contains item_code, item_name and unit_price.

Example

Create table Item_master(item_code number primary key,

```
        Item_name varchar2(20) unique,  
        Unit_price number(9,2)  
    );
```

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

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

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

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

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

*

ERROR at line 1:

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

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

Create table Item_master(item_code number CONSTRAINT pkit_code primary key,

```
        Item_name varchar2(20) CONSTRAINT unqit_name unique,  
        Unit_price number(9,2)  
    );
```

Naming the constraints always provides better readability.

9.3.2 Adding Domain Integrity Constraints

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

What is NULL?

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

Example

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

While inserting records to this table,

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

ERROR at line 1:

```
ORA-01400: cannot insert NULL into ("HEMA"."EMPLOYEE_MASTER"."EMPNAME")
```

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

Check Constraints are used to perform the checking of conditions based on the values entered. The CHECK constraint is used to enforce business rules as shown in the following example:

Example

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

);

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

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

```
INSERT INTO employee_master VALUES (2,'Aditya','b')
```

```
*
```

ERROR at line 1:

```
ORA-02290: check constraint (HEMA.CHK_SEX) violated
```

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

Example

```
CREATE TABLE employee_master (Empcode NUMBER, empname VARCHAR2(20),
```

```
Salary NUMBER CONSTRAINT CHK_SAL CHECK(salary BETWEEN 1000 and 7000));
```

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

Relationships between Parent and Child tables

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

- Parent table must be created before creating the child table.
- The Parent table column, which is referenced in the child table, must contain either Primary Key or Unique Key constraints.

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

The structures of the tables look like:

CUST_MASTER

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

ORDER_MASTER

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

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

```
CREATE TABLE cust_master (ccode NUMBER(5) CONSTRAINT pk_ccode PRIMARY KEY,  
Name VARCHAR2(40), Address      VARCHAR2(100));
```

```
CREATE TABLE order_master (ocode NUMBER(4), ccode NUMBER(5) CONSTRAINT  
fk_ccode REFERENCES cust_master(ccode),  
odate DATE, oval NUMBER);
```

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

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

Example

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

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

Inserting records into child table.

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

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

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

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

*

ERROR at line 1:

ORA-02291: integrity constraint (HEMA.FK_CCODE) violated - parent key not found

This raises to an error since there is no reference record in the master table for Customer code 4.

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

```
CREATE TABLE employee_master (Empcode NUMBER, empname VARCHAR2 (20),
```

Salary NUMBER, sex NUMBER,

```
CONSTRAINT pk_empno PRIMARY KEY (empcode), CONSTRAINT chk_sal CHECK  
(salary BETWEEN 1000 and 7000), CONSTRAINT chk_sex CHECK (sex IN ('M','F','m','f'));
```

The table-level constraint for adding a referential integrity constraint is shown below:

```
CREATE TABLE order_master (ocode NUMBER(4), ccode NUMBER(5), odate DATE, oval  
NUMBER,  
  
CONSTRAINT fk_ccode FOREIGN KEY(ccode) REFERENCES cust_master(ccode));
```

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

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

Using ALTER Statement

Constraints can be created for existing tables using ALTER statement. The following example illustrates this. Assume that the tables order_master, employee_master, item_master are created without constraints.

To add a primary key constraint,

Syntax:

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

Example

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

To add a unique constraint,

Syntax:

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

Example

```
ALTER TABLE item_master ADD CONSTRAINT unq_name UNIQUE(item_name);
```

Check constraints are added using,

Syntax:

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

Example

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

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

Syntax:

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

Example

```
ALTER TABLE order_master ADD CONSTRAINT fk_custcode FOREIGN KEY(ccode)  
REFERENCES cust_master(ccode);
```

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

Example

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

9.3.3 Enabling Constraints

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

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

```
ALTER TABLE emp1
```

```
ADD PRIMARY KEY (empno);
```

The above two statements add the primary key constraint. The first adds the primary key constraint at the time of creation of the table. The next statement tries to add the constraint after the table is created. Here if the records are not available, the constraint is added. If records are already present, the ALTER TABLE statement will fail.

9.3.4 Disabling Constraints

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

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

```
ALTER TABLE emp
    ADD PRIMARY KEY (empno) DISABLE;
```

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

Enabling and Disabling Defined Integrity Constraints

Use the ALTER TABLE command to

- enable a disabled constraint, using the ENABLE clause
- disable an enabled constraint, using the DISABLE clause

Enabling Disabled Constraints

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

```
ALTER TABLE dept
    ENABLE CONSTRAINT dname_ukey;
```

```
ALTER TABLE dept  
    ENABLE PRIMARY KEY,  
    ENABLE UNIQUE (dname, loc);
```

The above 2 statements tries to enable the constraints.

An ALTER TABLE statement that attempts to enable an integrity constraint fails when the rows of the table violate the integrity constraint.

9.3.4 Disabling Enabled Constraints

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

```
ALTER TABLE dept  
    DISABLE CONSTRAINT dname_ukey;
```

```
ALTER TABLE dept  
    DISABLE PRIMARY KEY,  
    DISABLE UNIQUE (dname, loc);
```

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

9.3.5 Dropping Integrity Constraints

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

```
ALTER TABLE dept  
    DROP UNIQUE (dname, loc);
```

```
ALTER TABLE emp  
    DROP PRIMARY KEY,  
    DROP CONSTRAINT dept_fkey;
```

```
DROP TABLE emp CASCADE CONSTRAINTS;
```

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

Specifying Referential Actions for Foreign Keys

Oracle allows referential integrity actions to be enforced, as specified the definition of a FOREIGN KEY constraint:

- **The ON DELETE CASCADE Action** This action allows referenced data in the parent key to be deleted (but not updated). If referenced data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint. For example:

Example

```
CREATE TABLE emp ( . . .
FOREIGN KEY (deptno) REFERENCES dept
ON DELETE CASCADE);
```

In this example, whenever a foreign key is added with an ON DELETE CASCADE constraint, the corresponding record in the master table(dept) is checked. This means that, whenever a record in dept table is deleted, corresponding record in the emp table is also deleted.

Example

Consider the following CREATE TABLE statements that define a number of integrity constraints, and the following example:

```
CREATE TABLE dept (
deptno      NUMBER(3) PRIMARY KEY,
dname       VARCHAR2(15),
loc         VARCHAR2(15),
CONSTRAINT dname_ukey UNIQUE (dname, loc),
```

```
CONSTRAINT loc_check1
CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO'));

CREATE TABLE emp (
    empno      NUMBER(5) PRIMARY KEY,
    ename      VARCHAR2(15) NOT NULL,
    job        VARCHAR2(10),
    mgr        NUMBER(5) CONSTRAINT mgr_fkey
                REFERENCES emp ON DELETE CASCADE,
    hiredate    DATE,
    sal         NUMBER(7,2),
    comm        NUMBER(5,2),
    deptno     NUMBER(3) NOT NULL
CONSTRAINT dept_fkey REFERENCES dept);
```

This creates the 2 table namely the primary key along with other constraints are specified. In emp table, along with foreign key on delete cascade is also specified which automatically ensures that whenever a record in Dept table is deleted corresponding rows in Emp table also deleted.

Deferred Constraint Checking

Constraints can be deferred for validity until the end of the transaction.

- A constraint is deferred if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back.
- If a constraint is immediate (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.

If a constraint causes an action (for example, delete cascade), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate.

Constraint Attributes

Constraints can be defined as either deferrable or nondeferrable. These attributes can be different for each constraint. Keywords that must appear in the Constraint clause is:

- DEFERRABLE or NOT DEFERRABLE

SET CONSTRAINTS Mode

The SET CONSTRAINTS statement makes constraints either DEFERRED or IMMEDIATE for a particular transaction. This statement is used to set the mode for a list of constraint names or for ALL constraints.

The SET CONSTRAINTS mode lasts for the duration of the transaction or until another SET CONSTRAINTS statement resets the mode.

SET CONSTRAINTS ... IMMEDIATE causes the specified constraints to be checked immediately on execution of each constrained statement. Oracle first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction (as long as all the checked constraints are consistent and no other SET CONSTRAINTS statement is issued). If any constraint fails the check, an error is signalled; at that point, a COMMIT would cause the whole transaction to roll back.

Example

Create table testconst(x number, y number constraint ck1 check (y > 20) deferrable);

Set constraint ck1 deferred;

Will display like:

Constraint set.

When records are inserted with violation checking is not performed immediately. The following is illustrative of this:

insert into testconst values(12,11);

1 row created.

commit;

commit

*

ERROR at line 1:

ORA-02091: transaction rolled back

ORA-02290: check constraint (HEMA.CK1) violated

Only when a COMMIT takes place the constraint is validated and the whole transaction is rolled back. Also, the set statement is valid only for one single transaction. Before the second transaction starts, the set statement must be given again.

9.4 Short Summary

Constraints are mechanisms that are used by Oracle to restrict invalid data from being entered into the table.

Constraints can be of 3 types:

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

Entity Integrity Constraints

Primary Key - No duplicates and no null values allowed. A table can have only one primary key.

Unique Key - No duplicates but any number of null values.

Domain Integrity Constraints

NOT NULL - No null values allowed

Check – Checks for conditions

Referential Integrity Constraints

Foreign Key – Establish Master-child Relationship.

Constraints can be enabled or disabled.

They can be dropped using alter command.

9.5 Brain Storm

1. How do you Enforce Entity and Referential Integrity?
2. How do you make a constraint inactive?
3. What constraints can be enforced [Using Enforce Option]
4. What is the difference between Primary Key and Unique Key?
5. Can a table have more than one Primary Key?
6. Explain the usage of ON DELETE CASCADE
7. How a constraint can be dropped?
8. Can a check condition check for more than one column?
9. How a constraint can be made to check later?
10. How a disabled constraint can be enabled?

Lecture 10

Operators and Functions

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss the operators and functions
- ☒ Describe and use Expressions
- ☒ Describe and create Functions
- ☒ Describe about Scalar functions
- ☒ Describe about Aggregate functions

Coverage Plan

Lecture 10

- 10.1 Snap shot
- 10.2 Operators
- 10.3 Functions
- 10.4 Scalar functions
- 10.5 Aggregate functions
- 10.6 Short summary
- 10.7 Brain Storm

10.1 Snap Shot

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

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

10.2 Operators

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

Operators

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

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

Arithmetic Operators

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

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

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

Character Operators

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

Example

```
SELECT 'This is an' || 'Example for' || 'Concatenation operator' FROM DUAL;  
Displays,
```

THIS IS AN EXAMPLE FOR CONCATENATION OPERATOR

This is an Example for Concatenation operator

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

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

Relational Operators

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

Operator Purpose	Example
= Equality test	SELECT empno, salary FROM employee WHERE deptno=10
!= ,<> Un-equality test	SELECT * FROM employee WHERE salary != 4000
> Greater	SELECT * FROM employee than specified value WHERE salary > 2000
< Lesser than specified value	SELECT * FROM students WHERE marks < 40
>= Greater than or equal to	SELECT * FROM product WHERE unit_price >=4
<= Lesser than or equal to	SELECT * FROM product WHERE unit_price <=4

Between The value lies within the range	UPDATE student SET grade='A' WHERE marks BETWEEN 80 AND 90
IN	Displays records that satisfy the list of values WHERE deptno IN (10,20,30) (Records where the department number is either 10 or 20 or 30 are displayed)
IS[NOT] Values	Is used to check for employee WHERE empname The only operator that checks NULL Values
LIKE	Matches a specified pattern. To match a single character, use '_' and for multiple characters use '%' SELECT empname FROM employee WHERE ename LIKE '_A%' (Displays employee names where the second letter of the name is 'A')

Logical Operators

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

- AND
- OR
- NOT

AND Operator

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

Example

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

This would display,

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1383	Saran	PRESIDENT		17-NOV-81	5000	10	

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

OR Operator

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

Example

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

This display would be as follows:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000	10	
7782	CLARK	MANAGER	7839	09-JUN-81	2450	10	
7566	JONES	MANAGER	7839	02-APR-81	2975	20	
7902	FORD	ANALYST	7566	03-DEC-81	3000	20	
7369	SMITH	CLERK	7902	17-DEC-80	800	20	
7788	SCOTT	ANALYST	7566	09-DEC-82	3000	20	
7876	ADAMS	CLERK	7788	12-JAN-83	1100	20	
7934	MILLER	CLERK	7782	23-JAN-82	1300	10	

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

NOT Operator

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

Example

SELECT * FROM students WHERE NOT (sname IS NULL);

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

SET operators

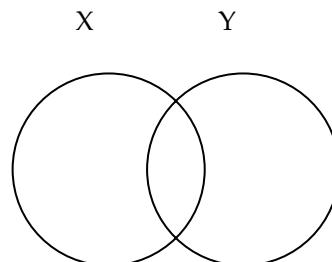
Set operators combine the results of two component queries into a single result. Queries containing set operators are called Compound Queries.

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

The SET operators are discussed below.

Union

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



Data in the tables

Table X		Table Y	
Itemcode	Name	Itemcode	Name
1	Powders	2	Soaps
2	Soaps	3	Creams
4	Pens	4	Pencils

Example

```
SELECT * FROM x
UNION
SELECT * FROM y
```

The output of this query looks like:

ITEMCODE	NAME
----------	------

1	Powders
2	Soaps
3	Creams
4	Pencils
4	Pens

UNION ALL Operator

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

Example

```
SELECT * FROM x
UNION ALL
SELECT * FROM y
```

The output of this query looks like:

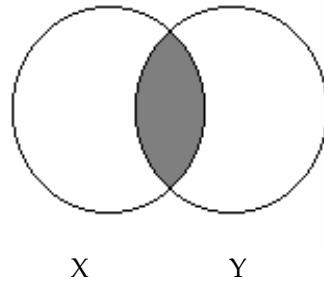
ITEMCODE	NAME
1	Powders
2	Soaps
2	Soaps
3	Creams
4	Pencils

4

Pens

INTERSECT operator

All the common distinct values are alone selected by the INTERSECT Operator.



Example

```
SELECT * FROM x
```

```
INTERSECT
```

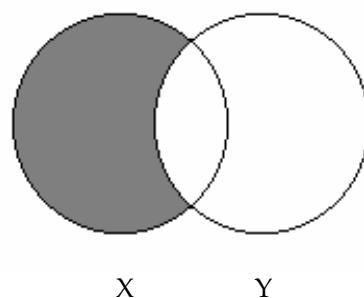
```
SELECT * FROM Y
```

This would return

ITEMCODE	NAME
2	Soaps

MINUS Operator

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



Example

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

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

The output will look like:

ITEMCODE	NAME
1	Powders
4	Pens

All set operators have equal precedence. If a SQL statement contains multiple sets of operators, ORACLE evaluates them from left to right. If the order is to be changed, parentheses has to be used to explicitly specify another order. These querying tables can contain different structures. But the type of the data in the tables must match.

10.3 Functions

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

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

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

10.4 Scalar Functions

Scalar Functions can be classified as follows:

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

Number Functions

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

ABS

Syntax

ABS(n)

Returns the absolute value of n

Example

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

Absolute

15

FLOOR

Syntax

FLOOR(n)

Returns the largest integer equal to or less than n.

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL
```

Floor

15

CEIL

Syntax

CEIL(n)

Returns the smallest integer greater than or equal to n.

Example

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL
```

Ceiling

16

EXP

Syntax

EXP(n)

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

Example

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

e to the 4th power

54.59815

LN

Syntax

LN(n)

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

Example

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

Natural log of 95

4.5538769

LOG

Syntax

LOG(m,n)

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

Example

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

Log base 10 of 100

2

MOD

Syntax

MOD(m,n)

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

Example

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

Modulus

3

POWER

Syntax

POWER(m,n)

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

Example

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

Raised

9

ROUND

Syntax

ROUND(n[,m])

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

Example

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

Round

15.2

SIGN

Syntax

SIGN(n)

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

Example

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

Sign

-1

SQRT

Syntax

SQRT(n)

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

Example

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

Square root

5.0990195

TRUNC

Syntax

TRUNC(n[,m])

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

Example

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

Truncate

15.7

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

Truncate

10

Character Functions

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

ASCII

Syntax

ASCII(char)

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

Example

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

```
ASCII('Q')
```

```
-----  
81
```

INSTR

Syntax

INSTR(char1,char2[,n[,m]])

Searches char1 beginning with its nth character for the mth occurrence of char2 and returns the position of the character in char1 that is the first character of this occurrence. If n is negative, ORACLE counts and searches backward from the end of char1. The value of m must be positive. The default values of both n and m are 1, meaning ORACLE begins searching at the first character of char1 for the first occurrence of char2. The return value is relative to the beginning of char1, regardless of the value of n, and is expressed in characters. If the search is unsuccessful (if char2 does not appear m times after the nth character of char1) the return value is 0.

Examples

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

Instring

14

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

Reversed Instring

2

INSTRB

Syntax

INSTRB(char1,char2[,n[,m]])

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

Example

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

Instring in bytes

27

LENGTH

Syntax

LENGTH(char)

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

Example

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

Length in characters

7

LENGTHB

Syntax

LENGTHB(char)

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

Example

Assume a double-byte database character set:

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

Length in bytes

14

Character Functions Returning character Values

CHR

Syntax

CHR(n)

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

Example

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

Character

K

CONCAT

Syntax

CONCAT(char1, char2)

Returns char1 concatenated with char2. This function is equivalent to the concatenation operator (||).

Example

This example uses nesting to concatenate three character strings:

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

Job

JAMES is a CLERK

INITCAP

Syntax

INITCAP(char)

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

Example

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

The Soap

LPAD

Syntax

LPAD(char1,n [,char2])

Returns char1, left-padded to length n with the sequence of characters in char2; char2 defaults to ' ', a single blank. If char1 is longer than n, this function returns the portion of char1 that fits in n.

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

Example

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

LPAD example

```
-----
```

```
*.*.*Page 1
```

LTRIM

Syntax

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

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

Example

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

Left trim example

```
-----
```

```
XxyLAST WORD
```

REPLACE

Syntax

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

Returns char with every occurrence of search_string replaced with replacement_string. If replacement_string is omitted or null, all occurrences of search_string are removed. If search_string is null, char is returned. This function provides a superset of the functionality

provided by the TRANSLATE function. TRANSLATE provides single character, one to one, substitution. REPLACE allows you to substitute one string for another as well as to remove character strings.

Example

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

```
-----  
BLACK and BLUE
```

RPAD

Syntax

```
RPAD(char1, n [,char2])
```

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

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

Example

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

```
RPAD example
```

```
-----  
TURNERababa
```

RTRIM

Syntax

```
RTRIM(char [,set])
```

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

Example

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

Right trim example

```
-----  
TURNERYxX
```

SOUNDEX

Syntax

```
SOUNDEX(char )
```

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

Example

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

```
ENAME
```

```
-----  
SMITH
```

SUBSTR

Syntax

```
SUBSTR(char,m [,n])
```

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

Example

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

Substring

CD

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

Substring

EF

SUBSTRB

Syntax

SUBSTRB(char,m [,n])

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

Example

Assume a double-byte database character set:

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

Substring with bytes

CD

TRANSLATE

Syntax

TRANSLATE(char,from,to)

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

Example

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

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

Tiles

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

```
SELECT TRANSLATE('2KRW229', '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',  
'0123456789') "Translate example" FROM DUAL
```

Translate example

2229

UPPER

Syntax

UPPER(char)

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

Example

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

Uppercase

LARGE

LOWER

Syntax

LOWER(char)

Returns char, with all letters in lowercase. The return value has the same datatype as the argument char (CHAR or VARCHAR2).

Example

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

Lowercase

```
-----  
mr. samuel hillhouse
```

Date Functions

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

ADD_MONTHS

Syntax

ADD_MONTHS(d,n)

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

Example

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

Next mont

```
-----  
04-JAN-01
```

LAST_DAY

Syntax

LAST_DAY(d)

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

Example

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

SYSDATE	Last	Days Left
---------	------	-----------

04-DEC-00	31-DEC-00	27
-----------	-----------	----

```
SELECT ADD_MONTHS(LAST_DAY(sysdate),5) "Five months" FROM dual;
```

Five mont

31-MAY-01

MONTHS_BETWEEN

Syntax

MONTHS_BETWEEN(d1,d2)

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

Example

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

Months

10.829904

NEXT_DAY

Syntax

NEXT_DAY(d,char)

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

Example

```
SELECT NEXT_DAY('06-DEC-00','TUESDAY') "NEXT DAY" FROM DUAL;
```

NEXT DAY

12-DEC-00

ROUND

Syntax

ROUND(d,[fmt])

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

Example

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

FIRST OF

01-JAN-01

TRUNC

Syntax

TRUNC(d,[fmt])

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

Example

```
SELECT TRUNC(SYSDATE, 'MM') "First Of The Month" FROM DUAL
```

First Of

01-DEC-00

Conversion Functions

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

TO_CHAR, date conversion

Syntax

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

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

Example

```
SELECT TO_CHAR(SYSDATE,'Month DD, YYYY') "New date format"
```

```
FROM dual;
```

New date format

December 04, 2000

TO_CHAR, number conversion

Syntax

```
TO_CHAR(n [, fmt])
```

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

Example

```
SELECT TO_CHAR(17145,'099G999') "Char" FROM DUAL
```

Char

```
-----  
017,145
```

TO_DATE

Syntax

```
TO_DATE(char [, fmt])
```

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

Do not use the TO_DATE function with a DATE value for the char argument. The returned DATE value can have a different century value than the original char, depending on fmt or the default date format.

Example

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

Output

```
TO_DATE('
```

```
-----  
25-SEP-00
```

Date Format Models

The date format models can be used in :

- TO_CHAR function to translate a DATE value that is in a format other than the default date format
- TO_DATE function to translate a character value that is in a format other than the default date format

Date Format Elements

A date format model is composed of one or more date format elements. NO TAG lists the date format model elements.

Element	Meaning
SCC or CC	Century; "S" prefixes BC dates with "-".
YYYY or SYYYY	4-digit year; "S" prefixes BC dates with "-".
IYYY	4-digit year based on the ISO standard.
YY or YY or Y	Last 3, 2, or 1 digit(s) of year.
IYY or IY or I	Last 3, 2, or 1 digit(s) of ISO year.
Y, YYY	Year with comma in this position.
SYEAR or YEAR	Year, spelled out; "S" prefixes BC dates with "-".
RR	Last 2 digits of year; for years in other countries.
BC or AD	BC/AD indicator.
B.C. or A.D.	BC/AD indicator with periods.

Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1)
MM	Month (01-12; JAN = 01)
RM	Roman numeral month (I-XII; JAN = I).
MONTH	Name of month, padded with blanks to length of 9 characters.
MON	Abbreviated name of month.
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
DDD	Day of year (1-366).
DD	Day of month (1-31).
D	Day of week (1-7).
DAY	Name of day, padded with blanks to length of 9 characters.
DY	Abbreviated name of day.
AM or PM	Meridian indicator.
A.M. or P.M.	Meridian indicator with periods.
HH or HH12	Hour of day (1-12).

HH24 Hour of day (0-23).

MI Minute (0-59).

SS Second (0-59).

SSSS Seconds past midnight (0-86399).

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values of other centuries. The RR date format element allows you to store twenty-first century dates in the twentieth century by specifying only the last two digits of the year. It also stores the twentieth century dates in the twenty-first century in the same way if necessary.

If TO_DATE function is used with the YY date format element, the date value returned is always in the current century. Instead if RR date format element is used, the century of the return value varies according to the specified two-digit year and the last two digits of the current year.

The following example demonstrates the behavior of the RR date format element.

Example

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR'), 'YYYY')  
      "4-digit year"    FROM DUAL
```

4-digit year

1995

```
SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY')  
      "4-digit year"  
      FROM DUAL
```

4-digit year

2017

Note: If the year is between 0 to 49 its places them in the current century.

Date Format Element Prefixes & Suffixes

Following table lists prefixes/suffixes that can be added to date format elements:

Prefix/Suffix	Meaning	Element	Value
FM	Spelled character	fmMONTH	'June'
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

TO_NUMBER

Syntax

TO_NUMBER(char [,fmt [, 'nlsparams']])

Converts char, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model fmt, to a value of NUMBER datatype.

Example

```
UPDATE emp SET sal = sal + TO_NUMBER ('89')  
WHERE ename = 'BLAKE'
```

The update statement updates the record where the employee name is BLAKE.

Other Functions

DUMP

Syntax

```
DUMP(expr [,return_format [, start_position [, length]] ])
```

Returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of expr. The argument return_format specifies the format of the return value and can have any of these values:

- 8 - returns result in octal notation.
- 10 - returns result in decimal notation.
- 16 - returns result in hexadecimal notation.
- 17 - returns result as single characters.

The argument start_position and length combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If expr is null, this function returns 'NULL'.

Examples

```
SELECT DUMP(ename, 8, 3, 2) "OCTAL" FROM emp WHERE ename = 'SCOTT'  
OCTAL
```

```
Type=1 Len=5: 117,124
```

```
SELECT DUMP(ename, 10, 3, 2) "ASCII" FROM emp WHERE ename = 'SCOTT'
```

```
ASCII
```

```
Type=1 Len=5: 79,84
```

GREATEST

Syntax

```
GREATEST(expr [,expr] ...)
```

Returns the greatest of the list of exprs. All exprs after the first are implicitly converted to the datatype of the first prior to the comparison. ORACLE compares the exprs using non-padded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher value. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT GREATEST('HARRY','HARRIOT','HAROLD') "GREATEST" FROM DUAL
```

```
GREATEST
```

```
-----
```

```
HARRY
```

LEAST

Syntax

```
LEAST(expr [,expr] ...)
```

Returns the least of the list of exprs. All exprs after the first are implicitly converted to the datatype of the first prior to the comparison. ORACLE compares the exprs using non-padded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST" FROM DUAL
```

```
LEAST
```

```
-----
```

```
HAROLD
```

NVL

Syntax

```
NVL(expr1, expr2)
```

If expr1 is null, it returns expr2; if expr1 is not null, it returns expr1. The arguments expr1 and expr2 can have any datatype. If their datatypes are different, ORACLE converts expr2 to the datatype of expr1 before comparing them. The datatype of the return value is always the

same as the datatype of expr1, unless expr1 is character data in which case the return value's datatype is VARCHAR2.

Example

```
SELECT ename, NVL(TO_CHAR(COMM),'NOT APPLICABLE') "COMMISSION"  
FROM emp  
WHERE deptno = 30  
ENAME  
COMMISSION  
-----  
ALLEN    300  
WARD     500  
MARTIN   1400  
BLAKE    NOT APPLICABLE  
TURNER   0  
JAMES    NOT APPLICABLE
```

UID

Syntax

UID

Returns an integer that uniquely identifies the current user.

USER

Syntax

USER

Returns the current ORACLE user with the datatype VARCHAR2. ORACLE compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the UID and USER functions identify the user on the local database. These functions cannot be used in a CHECK constraint.

Example

```
SELECT USER, UID FROM DUAL
```

USER UID

OPS\$KING 9

SYSDATE

Syntax

SYSDATE

Returns the current date and time. Requires no arguments. In distributed SQL statements, this function return the date and time on the local database. This function cannot be used in the condition of a CHECK constraint.

Example

SELECT SYSDATE "TODAY" FROM DUAL

TODAY

04-DEC-00

DECODE

Syntax:

DECODE(variable,<condition1>,<value1>,<condition2>,<value2>...)

This function is used to check for the if..then condition and displays the corresponding values.

Example:

SELECT ENAME, DEPTNO,

DECODE(deptno,10,'Accounting',20,'Sales',30,'Purchase','others') FROM EMP;

The output will be:

ENAME	DEPTNO	DECODE(DEP)
KING	10	Accounting
BLAKE	30	Purchase
CLARK	10	Accounting
JONES	20	Sales
MARTIN	30	Purchase
ALLEN	30	Purchase
TURNER	30	Purchase
JAMES	30	Purchase
WARD	30	Purchase
FORD	20	Sales
SMITH	20	Sales
SCOTT	20	Sales
ADAMS	20	Sales
MILLER	10	Accounting

The records whose deptno is 10, their corresponding name in the DECODE function is displayed, where the deptno is 20, the corresponding values are displayed. But this function must not be used inside PL/SQL.

The following Table is lists of all the functions:

Functions	Name
Numeric functions	ABS(),EXP(),FLOOR(),CEIL(),LN(),LOG(),MOD(),POWER(),ROUND(),SIGN(),SQRT(),TRUNC()
Character functions returning numeric values	ASCII(),INSTR(),INSTRB(),LENGTH()
Character functions returning character values	CHR(),CONCAT(),INITCAP(),LPAD(),LTRIM(),REPLACE(),RPAD(),RTRIM(),SOUNDEX(),SUBSTR(),SUBSTRB(),TRANSLATE(),UPPER(),LOWER()
Date functions	ADD_MONTHS(),LAST_DAY(),MONTHS_BETWEEN(),NEXT_DAY()

	,ROUND(),TRUNC()
Conversion functions	TO_CHAR(),TO_DATE(),TO_NUMBER()
Other functions	DUMP(),UID(),GREATEST(),LEAST(),NVL(),UID,USER,SYSDATE

10.5 Aggregate Functions

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

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

Following are the examples of the use these functions.

AVG(colname)

Calculates the average of the column name specified.

Example

```
SELECT AVG(sal) FROM EMP;
```

Displays the average of the sal column.

MAX(colname)

Example

```
SELECT MAX(qty) FROM ORDPRD;
```

This query displays the maximum quantity in the ORDPRD table.

The output will be:

MAX(QTY)

293

MIN(colname)

Example

SELECT MIN(qty) FROM ORDPRD;

This query displays the minimum quantity in the table.

The output will be:

MIN(QTY)

2

COUNT(colname)

Displays the total number of column values for the column specified in the colname format.
Discards any NULL values for that column.

SELECT COUNT(empno) FROM emp;

COUNT(*)

This function returns the total number of records in a table. Includes NULL values also.

Example

SELECT COUNT(*) FROM emp;

GROUP BY Clause

Group By clauses are used to group selected rows and return a single row of summary information. Expressions in the GROUP BY clause can contain any column name that appears in the table. Consider a situation such that the total number of employees in each department is to be listed out along with the department number. The following example illustrates this.

Example

```
SELECT deptno,COUNT(*) FROM emp GROUP BY deptno;
```

The above statement would display

DEPTNO	COUNT(*)
10	3
20	5
30	6

Restrictions in using Group By clause

1. If a SELECT list contains any columns other than these in the aggregate functions, the columns that appear in the Select list must be grouped and must appear in the GROUP By clause.
2. If a SELECT list contains any nested aggregate functions, no other column names can appear in the SELECT list and GROUP By must contain the column name on which the nested aggregate functions are performed.

Having Clause

HAVING clause is used to restrict the groups of rows defined by the GROUP BY clause. This is similar to the WHERE clause which acts as a filter to the SELECT clause. HAVING is used along with GROUP BY clause. The order of precedence in a SELECT list are listed below:

1. If a query contains a WHERE clause, the rows which do not satisfy the condition are removed.

2. Oracle calculates and forms the groups as specified in the GROUP BY clause.
3. Oracle removes the records that do not satisfy the condition in the HAVING clause.

Consider a situation where all the clerks in every department who lie within the specified maximum and minimum salary must be displayed. The minimum salary must not be less than 1000 and the maximum salary must not exceed 5000. The following example performs this checking.

Example

```
SELECT deptno,MIN(sal),MAX(sal) FROM emp WHERE job='CLERK'  
GROUP BY deptno  
HAVING MIN(sal) >=1000 AND MAX(sal) <=5000;
```

The output looks like:

DEPTNO	MIN(SAL)	MAX(SAL)
10	1300	1300

In this case, there is only one department that satisfies both the conditions.

The records are processed in this way:

First, the records where the job is not CLERK are eliminated. Next, the minimum and maximum salary of every department is determined and the HAVING condition is performed on the result of the rows selected by the GROUP BY clause.

Next, let us display the total number of employees in each department is displayed designation wise. Also, the number of employees in each department and for each job must exceed 2.

Example

```
SELECT job,deptno,COUNT(*) FROM emp GROUP BY job,deptno HAVING COUNT(*) >=2  
ORDER BY deptno;
```

This query selects the deptno, job and the total count from the employee table and grouping is done first on the job and then based on the deptno and the records are displayed where the count is greater than or equal to 2..

The output would look like:

JOB	DEPTNO	COUNT(*)
ANALYST	20	2
CLERK	20	2
SALESMAN	30	4

10.6 Short Summary

Operators: They are used to manipulate individual data items and then return a result.

The different operators are classified based on the data on which they perform operations.

They are classified as:

- Arithmetic
- Character
- Relational
- Logical
- Set

Built-in Functions are used to perform specific task. The functions are classified based on the arguments that they take:

- Numeric Functions
- Character Functions
- Date Functions
- Conversion Functions
- Other Functions

Aggregate Functions perform a group of values. They use the Group by clause to group the values and display as an aggregate value. Having clause acts as a filter for the Grouped Values.

10.7 Brain Storm

1. Prove In, Any, = Some are same.
2. Write a query to select managers from department 20
3. Select Lpad ('Radian',5 '**') from dual;
 - a. ***** Radian
 - b. RTRIM('Radian', '*')
 - c. Radia
 - d. None of the above
4. Prove Oracle is Y2K complain? (Hint: RR Format)
5. Write a query to find out whether given year is a leap year?
6. Select greatest (sal) from emp; observe the result and find out difference between Max and Greatest.
7. Mention conversion functions with brief explanation?
8. If null values are present in Table for which we apply count function. What would be the result?

ANSWER

Lecture 11

Joins and Subqueries

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss Joins and Subqueries
- ☒ Describe the joins tables
- ☒ Describe Subqueries
- ☒ Describe about Correlated subquery
- ☒ Describe about Inline query
- ☒ Discuss about pseudo columns

Coverage Plan

Lecture 11

- 11.1 Snap shot
- 11.2 Joins tables
- 11.3 Sub-queries
- 11.4 Correlated sub-query
- 11.5 Inline query
- 11.6 Pseudo columns
- 11.7 Short summary
- 11.8 Brain Storm

11.1 Snap Shot

For this chapter is to introduce the concepts of Joins and Subqueries. This chapter deals with the various types of joins, Subqueries and usage of Pseudocolumns like ROWID, ROWNUM, and LEVEL.

- Cartesian Product
- Different Joins
- Equi-Join
- Self Join
- Outer Join
- Non-Equi Join
- Subqueries
- Correlated Subquery
- Pseudocolumns

11.2 Joining Tables

Till now, we have seen querying being performed on only one table. If any information needs to be queried from more than one table, a concept called JOINS is used. This allows data to be selected from one or more tables and to combine the selected data into a single result table.

Cartesian Products

If a join condition is not specified, Oracle performs a Cartesian Product. Oracle combines each row of one table with each row of the other. For example, if emp table contains 10 records and dept table contains 4 records, the number of rows selected by the query without a join condition yields 40 records. Always when data is selected from different tables, a join condition is required.

Join Conditions

Most Join queries contain WHERE clause conditions that compare two columns, one from each table. Such a condition is called a JOIN Condition. To execute a Join, Oracle combines

pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. Joins can be of different types:

- Equi-Join
- Self Join
- Outer Join
- Non-Equi Join

Equi-Joins

This is the most common type of joins. This join contains a condition containing an equality operator. An equi-join combines rows that have equivalent values for the columns specified in the Join. A simple example for using Equi-Joins is shown below.

Example 11.1

```
SELECT ename,dname FROM emp,dept WHERE emp.deptno=dept.deptno;
```

This would display all the employees and their department names based on the outcome of the join condition.

Here, emp.deptno=dept.deptno is the Join condition. The records are selected and displayed when the condition is evaluated to TRUE.

The display will be:

ENAME	DNAME
KING	ACCOUNTING
BLAKE	SALES
CLARK	ACCOUNTING
JONES	RESEARCH
MARTIN	SALES
ALLEN	SALES
TURNER	SALES

JAMES	SALES
WARD	SALES
FORD	RESEARCH
SMITH	RESEARCH
SCOTT	RESEARCH
ADAMS	RESEARCH
MILLER	ACCOUNTING

Table Aliasing

In the previous query, the table names were specified in the Join condition since both the tables had the same column name. In order to avoid any ambiguity in selecting the records, table aliasing must be used. The table alias in the previous query is the table name itself. Any alias name can be specified for the table which will be valid for that query alone. Refining the above example,

```
SELECT ename, dname FROM emp e, dept d WHERE e.deptno=d.deptno;
```

The example given below displays the products ordered along with the customer name and the product codes. This example is a join of 4 tables : product table, orders table, customer table, ordprd table.

The following example is a join from more than 2 tables.

```
SELECT op.pcode, pname, cname FROM product p, orders o, ordprd op, customer c  
WHERE c.ccode=o.ccode and  
o.ocode=op.ocode and  
op.pcode=p.pcode
```

Pcode	Pname	Cname
102	storewell	Mohan
102	storewell	Mohan
102	storewell	Mohan
103	cinthol old	Ramanath

103	cinthol old	Mohan
103	cinthol old	Sridhar
103	cinthol old	Ramanath
103	cinthol old	Mohan
104	1.5 ton a/c	Ramanath
105	1.0 ton a/c	Ramanath
106	0.75 ton a/c	Ramanath
106	0.75 ton a/c	Mohan
106	0.75 ton a/c	Sridhar
106	0.75 ton a/c	Ramanath
107	puf refrigerator	Ramanath
107	puf refrigerator	Ramanath
107	puf refrigerator	Ramesh
107	puf refrigerator	Mohan
108	hair dye	Ramesh
108	hair dye	Selvam

Self Joins

Self joins are a type of joins where the join operation takes place within the same table. Consider a situation where an organization needs a report of all the employee names along with their manager's name. For performing this query, the manager number is available in the same table under mgr column. By equating the manager number with the employee number can be selected the records. In order to query from the same table, a copy of the same table is required in the SELECT list to display records. To differentiate the two tables, table aliasing is used. The following query illustrates this concept.

Example 11.2

```
SELECT e1.ename || ' works for ' || e2.ename FROM emp e1,emp e2 WHERE e1.mgr=e2.empno;
```

Each record in table e1 is matched for the corresponding record in table e2. The records that match the condition are selected and displayed.

Outer Joins

The outer join extends the result of a simple join or equi join. Outer join returns all the rows that satisfy the join condition and those rows from one table for which no rows from the other table satisfy the join condition.

In order to write a query that performs an outer join of table A and table B and records from A, the outer join operator (+) is applied to all columns of B in the join condition. The example given below performs this operation.

Example 11.3

Consider a situation product details where are the to be displayed for all the products ordered as well as not ordered in a single query.⁹

This can be done using

```
SELECT a.pcode,b.ocode FROM product a, ordprd b WHERE a.pcode=b.pcode(+);
```

The above query displays all the product codes from the product table and all the order codes from the ordprd table. In addition to this, the product codes available in the Product table that do not exist in the ordprd table are also displayed. To achieve this, the (+) operator is used.

Outer Joins are subject to the following restrictions:

- The (+) operator can appear only in the WHERE clause and can be applied only to a column of a table or view.
- If two tables are joined by multiple conditions, the (+) operator must appear in all these conditions.

- A condition cannot use IN comparison operator to compare a column marked within the (+) operator to another expression.

Non-Equi Joins

Non-Equi Joins specify the relationship between the tables not in terms of columns but in terms of the relational operators or any comparison operators used. The following example deals with this type of join:

Example 11.4

```
SELECT ename,grade FROM emp, salgrade WHERE sal BETWEEN losal AND hisal;
```

The relationship between the tables emp and salgrade are established using the values and the BETWEEN operator.

11.3 Subqueries

A Sub-query is a query that can contain multiple query statements each nested within another. The Sub-query is used to retrieve selected data from tables that depend on the values in the table. It is also called as a nested query. Statement containing a subquery is called the parent statement and the query inside is called the child query. The parent statement uses the rows returned by the child query. Subqueries can be used in the following conditions:

- To define the set of rows to be inserted into the target table of a INSERT or CREATE TABLE statement.
- To define one or more values to be assigned to existing rows of an UPDATE statement.

Consider a case where the details of an employee who earns the maximum salary must be displayed. In such situations, first the maximum salary must be calculated. Then the maximum salary must be compared with values in the emp table and their details must be displayed. These 2 queries can be combined into a single query called sub-query.

Example 11.5

```
SELECT ename,deptno,sal FROM emp WHERE sal=(SELECT MAX(sal) FROM emp);
```

In the above query, first the maximum salary is computed and the result is compared with the parent query for the output.

The output will look like:

<u>ENAME</u>	<u>DEPTNO</u>	<u>SAL</u>
KING	10	5000

In the employee table the record with the employee name KING has the maximum salary of 5000.

Example 11.6

```
SELECT PCODE , PNAME FROM PRODUCT WHERE PCODE=
(SELECT PCODE FROM ORDPRD WHERE QTY=
(SELECT MAX(QTY) FROM ORDPRD))
```

The output will be:

PCODE	PNAME
106	0 . 75 ton a/c

This example displays the product code and the product name from the product table for which the ordered quantity is the maximum.

Using ANY,ALL,SOME operators

If the inner-query returns only one value, the outer query can process and display values that match the condition. At all times, the inner-query may not return only one value. In these

cases, the equality operator is used to check for the condition, which can check for utmost only one value. If more than one value is selected, Oracle requires an operator that can check and display the values. To do this operation ANY,ALL or SOME operators are used.

Any Operator

Consider the following query

```
SELECT ename,sal,deptno FROM emp WHERE sal >=ANY(SELECT AVG(sal) FROM emp  
group by deptno);
```

This query displays the employee name,salary and the department number of employees whose salary is greater than or equal to any of the values generated by the inner query. If any of these operators are not used, Oracle generates an error message as shown below.

ERROR at line 1:

ORA-01427: single-row subquery returns more than one row

Some Operator

Consider the following query

```
SELECT ename,sal,deptno FROM emp WHERE sal >=SOME(SELECT Sum(sal) FROM emp  
group by deptno);
```

This query works the same way as the ANY operator. There is no difference in the execution of the query.

Note: ANY and SOME operators are equivalent to the OR operator and the IN operator.

All Operator

ALL operator is used to check for all the values returned by the list. The condition is checked for all the values and the records that match all the values is displayed. Consider the following example:

```
SELECT pcode  FROM ordprd WHERE qty >=ALL(SELECT AVG(qty) FROM ORDPRD GROUP BY pcode)
```

Output will be:

PCODE

106

This example first calculates the average qty of all the products grouped by product code and then checks for the product for which the qty is greater than or equal to the average qty. The records for which the condition is satisfied are displayed.

EXISTS OPERATOR

The Outer query is executed if inner query evaluates to TRUE.

Example 11.7

```
SELECT DEPTNO, DNAME, LOC FROM DEPT WHERE EXISTS  
(SELECT * FROM EMP)
```

The output looks like:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

11.4 Correlated Subquery

A Correlated subquery is a subquery that is evaluated once for every row processed by the parent statement. For example, let us display the information of employees whose salary is greater than the average salary of their own department.

The first query displays the deptno along with the average salary.

```
SELECT AVG(SAL),DEPTNO FROM EMP GROUP BY DEPTNO;
```

<u>AVG(SAL)</u>	<u>DEPTNO</u>
-----------------	---------------

2916.6667	10
2175	20
1566.6667	30

The following query displays the records where the matching condition is satisfied.

Example 11.8

```
SELECT EMPNO,ENAME,DEPTNO,SAL FROM EMP B WHERE SAL >=
(SELECT AVG(SAL) FROM EMP A WHERE A.DEPTNO=B.DEPTNO GROUP BY DEPTNO);
```

The execution works in the following manner:

1. The department number is obtained.
2. The department number is then equated with the Parent Statement
3. If that row's salary is greater than the average salary that particular row is returned.

EMPNO	ENAME	DEPTNO	SAL
7839	KING	10	5000
7698	BLAKE	30	2850
7566	JONES	20	2975
7499	ALLEN	30	1600
7902	FORD	20	3000
7788	SCOTT	20	3000

For every parent record selected the inner-query is executed.

The following example displays the nth maximum salary:

Example 11.9

```
SELECT SAL FROM EMP A WHERE &N =  
(SELECT COUNT(DISTINCT(SAL)) FROM EMP WHERE A.SAL <=SAL);
```

The output will be:

Enter value for n: 1

```
old    1: SELECT SAL FROM EMP A WHERE &N =  
new    1: SELECT SAL FROM EMP A WHERE 1 =
```

SAL

5000

11.5 In-Line Queries

Inline Queries are a type of queries which can be selected from the 'FROM' clause of the select statement.

Example 11.10

```
SELECT ENAME,DEPTNO FROM (SELECT * FROM EMP) WHERE DEPTNO=20;
```

The output will be:

<u>ENAME</u>	<u>DEPTNO</u>
JONES	20
FORD	20
SMITH	20
SCOTT	20

ADAMS 20

The following query displays the average salary of the employees along with their salary.

```
SELECT ENAME,SAL, X "AVERAGE SALARY" ,DEPTNO FROM (SELECT AVG(SAL) X,DEPTNO
B FROM EMP
GROUP BY DEPTNO),EMP WHERE DEPTNO=B;
```

The output will be:

ENAME	SAL	Average Salary	DEPTNO
KING	5000	2916.6667	10
CLARK	2450	2916.6667	10
MILLER	1300	2916.6667	10
JONES	2975	2175	20
SCOTT	3000	2175	20
ADAMS	1100	2175	20
SMITH	800	2175	20
FORD	3000	2175	20
BLAKE	2850	1566.6667	30
MARTIN	1250	1566.6667	30
ALLEN	1600	1566.6667	30
TURNER	1500	1566.6667	30
JAMES	950	1566.6667	30
WARD	1250	1566.6667	30

11.6 Pseudo-columns

A pseudocolumn behaves like a column of a table, but it is not actually stored in the table. Selection can be done from Pseudocolumns but DML operations are not possible. The Pseudocolumns are listed below:

- ROWNUM
- ROWID
- LEVEL
- CURRVAL
- NEXTVAL

(CURRVAL and NEXTVAL are dealt in the chapter on Sequences.)

Rownum

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle retrieves the record. The ROWNUM holds a numeric value. For example,

Example 11.11

```
SELECT deptno, dname, ROWNUM FROM dept;
```

This query returns the department number along with the department name and the ROWNUMber.

The output will be:

DEPTNO	DNAME	ROWNUM
10	ACCOUNTING	1
20	RESEARCH	2
30	SALES	3
40	OPERATIONS	4

Consider the next example which sorts the display based on the department name.

```
SELECT deptno, dname, ROWNUM FROM dept ORDER BY dname;
```

The output for this query will be:

DEPTNO	DNAME	ROWNUM
10	ACCOUNTING	1
40	OPERATIONS	4
20	RESEARCH	2
30	SALES	3

Here the display is changed. Sorting the department name does not result in changing the row number since the row number is selected as soon as the record is retrieved from the table.

The next example shows condition based retrieval using ROWNUM.

```
SELECT ename,deptno FROM emp WHERE ROWNUM > 1;
```

The output of this query will be:

no rows selected

The row number is assigned only after the selection is made. Here, even before the selection is done, the WHERE condition is executed which results in no records being selected. Hence, the row numbers are not assigned. This leads to the above output.

Rowid

A ROWID is created by Oracle for each new row in every table, and it is a pseudo column and it has a value for every row in a table. The ROWID gives us the physical address of a row and is the fastest way to access any row. The ROWID contains 4 bits of information, they are :-

1. Object Id
2. The File Id within the Object
3. Block Id within the File
4. Row # within the block.

The ROWID has three important uses, it is :-

1. The fastest path to any row.
2. Unique identifier of any given row.

A ROWID does not change during the lifetime of a row. However, it must not be used as a primary key. The ROWID of a deleted record may be assigned to a new record that is inserted later.

Example 11.12

```
SELECT ROWID, ENAME FROM EMP WHERE DEPTNO=20;
```

The output of this query is ,

ROWID	ENAME
-------	-------

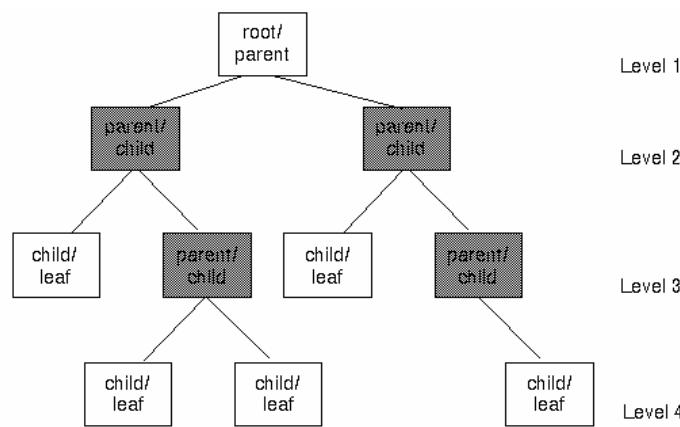
AAAApVAAFAAAACZAAD	JONES
AAAApVAAFAAAACZAAJ	FORD
AAAApVAAFAAAACZAAK	SMITH
AAAApVAAFAAAACZAAL	SCOTT
AAAApVAAFAAAACZAAM	ADAMS

In the above output ,

AAAApV represents Object Id (6 bytes)
AAF represents the File Id (3 bytes)
AAAACZ represents Block Id (6 bytes)
AAD represents Row Id (3 bytes)

Level

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for the root node, 2 for a child node and so on. A root node is the highest node within an inverted tree. A child node is any non-root node. A parent node is any row that has children. A leaf node is any row without children. The following figure shows the nodes of an inverted tree with their LEVEL values.



To define the hierarchical relationship in a query, use START WITH and CONNECT BY clause. The following example illustrates this:

```
SELECT LPAD(ENAME,10*LEVEL) FROM EMP  
START WITH MGR IS NULL  
CONNECT BY MGR=PRIOR EMPNO;
```

The output of this query will be:

```
LPAD(ENAME,10*LEVEL)
```

```
KING  
-----  
BLAKE  
    MARTIN  
    ALLEN  
    TURNER  
    JAMES  
    WARD  
CLARK  
    MILLER  
    JONES  
    FORD  
    SMITH  
    SCOTT  
    ADAMS
```

Since King is the top most employee, the execution will start with the start with clause and the next connection will be based on CONNECT BY clause.

In this example CONNECT BY MGR=PRIOR EMPNO compares the last selected empno (first King's employee no) with the matching manager No's in Manager Column and displays those records under KING and so on.

11.7 Short Summary

Joins – Joins are used to join one or more tables. The condition that appears in the WHERE clause is called Join Condition.

Joins can be of different types;

- Equi Join

- Self Join
- Outer Join
- Non-Equi Join

Subqueries – Subqueries are a type of queries where multiple statements can be embedded.

Correlated Subquery – A subquery which gets evaluated for each and every parent query execution.

PseudoColumns appear as a column in a table but are not part of the table.

Rowid – Identifies the physical address of a row

Rownum – Number that is assigned in the SELECT statement for each and every record selected.

11.8 Brain Storm

1. If a problem can be solved using joins and subqueries, Which works faster?
2. For how many tables can we set join condition in a single query?
3. Do you need a common column to set a join condition?
4. What is Cartesian Product?
5. Differentiate between Normal Subquery and Correlated Subquery?
6. Explain Self Join?
7. What is Pseudo Column?
8. Which Pseudo Column is used in Autogeneration of Numbers?
9. Observe Rownum in query which has order by clause?
10. Get details of employee in an ascending order of ename [Hint: Without using order by].



Lecture 12

Database Objects

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss the Database Objects
- ☒ Describe privileges
- ☒ Describe Synonyms and Sequences
- ☒ Understanding Database Structures
- ☒ Discuss Access Methodologies

Coverage Plan

Lecture 12

- 12.1 Snap shot
- 12.2 Privileges
- 12.3 Database objects
- 12.4 Synonyms
- 12.5 Sequences
- 12.6 Understanding database structure
- 12.7 Access methodologies
- 12.8 Short summary
- 12.9 Brain Storm

12.1 Snap Shot

In this Lecture we learn the following concepts they are privileges, different Database Objects. This chapter introduces the usage of Views, Synonyms and Sequences. Access Methodologies such as Indexes and Clusters are also dealt with.

- Granting Privileges
- Revoking Privileges
- Creating Views
- Creating Synonyms
- Creating Sequences
- Working with Indexes and Clusters

12.2 Privileges

A privilege is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to

- create a table
- select rows from another user's table
- Execute other procedures, functions etc.

Granting and Revoking System Privileges

GRANT or REVOKE is used to give system privileges to users. System privileges are granted to or revoked from users using the following:

The SQL commands GRANT and REVOKE

The DELETE, INSERT, SELECT, and UPDATE privileges allow the DELETE, INSERT, SELECT, and UPDATE DML operations, respectively, on a table or view.

The ALTER, INDEX, and REFERENCES privileges allow DDL operations to be performed on a table.

The one object privilege for procedures (including standalone procedures and functions, and packages) is EXECUTE. You should grant this privilege only to users who need to execute a procedure.

Syntax:

To grant a privilege:

```
Grant <privilege name> on <tablename> to <username>;
```

To revoke a privilege

```
Revoke <privilege name> on <tablename> from <username>;
```

Example:

To grant a select privilege to a user B from A, use

```
GRANT SELECT ON EMP TO B;
```

DISPLAYS,

Grant succeeded.

While revoking ,

```
REVOKE select on emp from b;
```

Revoke succeeded.

12.3 Database Objects

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

- Tables
- Views
- Synonyms
- Sequences
- Indexes
- Clusters

Out of these, tables are already dealt.

12.3.1 Views

Views are a tailored presentation of the data contained in one or more tables or views. A view takes the output of a query and displays it. Views contain queries which are selected from one or more tables or in turn views itself. These are called Base tables. Views provide the following advantages:

Restricted Access to specific columns of a table thereby providing additional security. They hide data complexity. They are used to simplify commands since the queries that are issued very frequently can be placed inside a view and the view can just be selected.

Direct DML operations are possible on Views with some restrictions.

Based on the query contained in the definition of a view, Views can be classified into three types

- Simple
- Join
- Force

12.3.2 Creating a Simple view

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

Syntax:

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

Example:

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

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

This displays the feedback

View created.

From now on, querying need not be done from the EMP table. The records can be queried using the View as shown below:

```
SELECT * FROM empview;
```

Displays,

ENAME	SAL	DEPTNO
KING	5000	10
BLAKE	2850	30
CLARK	2450	10
JONES	2975	20
MARTIN	1250	30
ALLEN	1600	30
TURNER	1500	30
JAMES	950	30
WARD	1250	30
FORD	3000	20
SMITH	800	20
SCOTT	3000	20
ADAMS	1100	20
MILLER	1300	10

Any DML operations can be made to this view. Records inserted through the view will be effected in the base table and vice versa. But not all columns can be inserted through the view. Only the column names that appear in the select list of the query can be inserted or updated.

There are some Restrictions on performing DML operations. They are:

If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, rows cannot be inserted into, updated in, or deleted from the base tables using the view.

If a view is defined with the WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view) if the view cannot select the row from the base table.

If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, a row cannot be inserted into the base table using the view.

If the view was created by using an expression, such as DECODE(deptno, 10, 'SALES', ...), rows cannot be inserted into or updated in the base table using the view.

A View creation cannot contain an ORDER BY clause.

12.3.3 Creating Force Views

Even if the Sometimes a table may be created at a later time for which a view must be created first. In those cases, a view can be created for the tables that are not available but querying cannot be performed. The keyword FORCE is used to do this:

Syntax:

```
CREATE OR REPLACE FORCE VIEW FV AS <QUERY>;
```

Example:

```
CREATE OR REPLACE FORCE VIEW fv AS SELECT * FROM students;
```

This view generates an output like:

Warning: View created with compilation errors.

Selection from the view is not possible till the table is created. Only then the view becomes valid. Before the table is created, the querying from the View will result in:

```
SELECT * from fv;
SELECT * from fv
*
ERROR at line 1:
ORA-04063: view "HEMA.FV" has errors
```

12.3.4 Views with Constraints

Consider the following example:

Example

```
CREATE OR REPLACE VIEW empview1 AS SELECT EMPNO,ENAME,DEPTNO,SAL FROM emp  
WHERE SAL >=3000;
```

Insertion onto this view is possible. A record can be inserted which violates the WHERE clause. But while selecting only the WHERE clause is checked and the records that match the condition are only displayed. If a WHERE condition must be checked while inserting records through a view, CHECK OPTION is used. The example illustrates this:

```
CREATE OR REPLACE VIEW empview1 AS SELECT EMPNO,ENAME,DEPTNO,SAL FROM emp  
WHERE SAL >=3000 WITH CHECK OPTION;
```

If any DML operation which violates the WHERE clause is issued, Oracle raises the following error:

```
INSERT INTO empview1 VALUES(14,'Suresh',20,1000);  
INSERT INTO empview1 VALUES(14,'Suresh',20,1000)  
*
```

ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause violation

12.3.5 Read-Only Views

Certain Views can be created for selection purpose alone. Those views will not permit a user or the owner of the view to perform any DML operation. The example shows this view:

```
CREATE OR REPLACE VIEW read_view As  
SELECT * FROM dept WITH READ ONLY;
```

View created.

```
INSERT INTO read_view VALUES(50,'Purchase','Texas');  
INSERT INTO read_view VALUES(50,'Purchase','Texas')  
*  
ERROR at line 1:  
ORA-01733: virtual column not allowed here
```

12.3.6 Join Views

Joining of tables results in creation of Join Views. Join View is a view that is created out of a Join condition. There are certain restrictions in Join Views. Assume a situation where the employee name along with the department name must be displayed. The join condition can be used here. The following example shows this:

```
CREATE OR REPLACE VIEW join_view AS
SELECT      empno,ename,dept.deptno,dname      FROM      emp,dept      WHERE
emp.deptno=dept.deptno;
```

When this view is selected, this results in:

```
sql> SELECT * FROM join_view;
```

EMPNO	ENAME	DEPTNO	DNAME
7839	KING	10	ACCOUNTING
7698	BLAKE	30	SALES
7782	CLARK	10	ACCOUNTING
7566	JONES	20	RESEARCH
7654	MARTIN	30	SALES
7499	ALLEN	30	SALES
7844	TURNER	30	SALES
7900	JAMES	30	SALES
7521	WARD	30	SALES
7902	FORD	20	RESEARCH
7369	SMITH	20	RESEARCH
7788	SCOTT	20	RESEARCH
7876	ADAMS	20	RESEARCH
7934	MILLER	10	ACCOUNTING

Inserting records onto the view will insert records in emp table and dept table.

```
INSERT INTO join_view VALUES(15,'radhika',50,'purchase');
INSERT INTO join_view VALUES(15,'radhika',50,'purchase')
*
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join view
```

View cannot directly insert into more than one base table. In order to perform this INSTEAD OF Triggers are used. This view can insert only onto one table.

Key-Preserved Tables

The concept of a key-preserved table is fundamental to understanding the restrictions on modifying join views. A table is key preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.

Note: It is not necessary that the key or keys of a table be selected for it to be key preserved. It is sufficient that if the key or keys were selected, then they would also be key(s) of the result of the join.

Attention: The key-preserving property of a table does not depend on the actual data in the table. It is, rather, a property of its schema and not of the data in the table. For example, if in the EMP table there was at most one employee in each department, then DEPT.DEPTNO would be unique in the result of a join of EMP and DEPT, but DEPT would still not be a key-preserved table.

12.3.7. Dropping Views

Dropping Views can be done using the statement:

```
DROP VIEW <viewname>;
```

12.4. Synonyms

A synonym is a database object, which is used as an alias name for any object. The main advantages of using Synonyms are:

Simplify SQL statements

Hide the real identity of an object

Use in database links

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

Syntax for creating a synonym name:

Syntax:

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

Example:

```
CREATE SYNONYM synemp FOR EMP;  
SELECT * FROM synemp will display all the employee records in the emp table.
```

From now on, querying can be done using the synonym synemp. Also using this synonym, DML operations can be performed. Synonyms can be created for non-existing objects also. The problem comes only when querying is done on the synonym. This example creates a synonym for a non-existing object.

```
CREATE SYNONYM s1 FOR TESTSS;  
Synonym created.  
SELECT * FROM s1;  
SELECT * FROM s1  
*  
ERROR at line 1:  
ORA-00980: synonym translation is no longer valid
```

Dropping Synonyms

Dropping of Synonyms can be done using the Drop command:

```
DROP SYNONYM synemp;
```

12.5 Sequences

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

12.5.1 Creating Sequences

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

Syntax:

```
CREATE SEQUENCE <SEQUENCENAME>
[ <INCREMENT BY> <VALUE>
<START WITH> <VALUE>
<MINVALUE> <VALUE>
<MAXVALUE> <VALUE>
<CACHE> <VALUE>
<CYCLE> <VALUE>
```

where,

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

START WITH - Specifies the first sequence number to be generated.

MINVALUE - Indicates the minimum value in the sequence It must be less than or equal to START WITH and less than MAXVALUE.

MAXVALUE - Specifies the maximum value the sequence can generate.

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

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

Example

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

Displays,

Sequence Created.

12.5.2. Referencing Sequences

A sequence is referenced in SQL statements using NEXTVAL and CURRVAL pseudocolumns. NEXTVAL indicates the sequence's new number and CURRVAL indicates the current sequence number.

Example:

```
SELECT s1.NEXTVAL FROM dual;
```

Displays

NEXTVAL

1

They can be used in Insert operations also.

```
INSERT INTO emp(empno,ename,job)
VALUES(s1.NEXTVAL,'sita','manager');
```

The record will be inserted where the employee number will be the value stored in the NEXTVAL pseudocolumn.

12.5.3. Altering a sequence

Sequences can be altered to perform: Set or eliminate minimum or maximum value or altering the incremental values or changing the number of cached sequence numbers.

```
ALTER SEQUENCE s1 MAXVALUE 25;
```

12.6 Understanding Database Structure

An Oracle database is a collection of data that is treated as a unit. The general purpose of a database is to store and retrieve related information. The database has logical structures and physical structures.

Logical Database Structure

Logical Database Structures contains various divisions such as tablespace, schema objects, data blocks, extents and segments. The following sections explain this.

Tablespaces

A database is divided into logical storage units called tablespaces. A tablespace is used to group related logical structures together. For example, tablespaces commonly group all of an application's objects to simplify certain administrative operations.

Schemas and Schema Objects

A schema is a collection of objects. Schema objects are the logical structures that directly refer to the database's data. Schema objects include such structures as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links. (There is no relationship between a tablespace and a schema; objects in the same schema can be in different tablespaces, and a tablespace can hold objects from different schemas.)

Oracle Data Blocks

At the finest level of granularity, an Oracle database's data is stored in data blocks. One data block corresponds to a specific number of bytes of physical database space on disk. A data block size is specified for each Oracle database when the database is created. A database uses and allocates free database space in Oracle data blocks.

Extents

The next level of logical database space is called an extent. An extent is a specific number of contiguous data blocks, obtained in a single allocation, used to store a specific type of information.

Segments

The level of logical database storage above an extent is called a segment. A segment is a set of extents allocated for a certain logical structure. For example, the different types of segments include the following:

Data Segment

Each non-clustered table has a data segment. All of the table's data is stored in the extents of its data segment. Each cluster has a data segment. The data of every table in the cluster is stored in the cluster's data segment.

Index Segment

Each index has an index segment that stores all of its data.

Rollback Segment

One or more rollback segments are created by the database administrator for a database to temporarily store "undo" information.

Temporary Segment

Temporary segments are created by Oracle when a SQL statement needs a temporary work area to complete execution. When the statement finishes execution, the temporary segment's extents are returned to the system for future use.

Physical Database Structures

The following sections explain the physical database structures of an Oracle database, including datafiles, redo log files, and control files.

Datafiles

Every Oracle database has one or more physical datafiles. A database's datafiles contain all the database data. The data of logical database structures such as tables and indexes is physically stored in the datafiles allocated for a database. The following are characteristics of datafiles:

A datafile can be associated with only one database.

Database files can have certain characteristics set to allow them to automatically extend when the database runs out of space.

12.7 Access Methodologies

The performance of an application is always critical. That's because the productivity of an application user directly relates to the amount of time that the user must sit idle while the application tries to complete work. With database applications, performance depends greatly on how fast an application can access table data. Typically, disk I/O is the primary performance determining factor for table access—the less disk I/O that's necessary to access table data, the better the dependent applications will perform. In general, it's best to try and minimize the amount of disk access that applications must perform when working with database tables.

The judicious use of table indexes is the principal method to reduce disk I/O and improve the performance of table access. Just like an index in a book, an index of a table column (a set of columns) allows Oracle to quickly find specific table records. When an application queries a table and uses an indexed column in its selection criteria, Oracle automatically uses the index to quickly find the target rows with minimal disk I/O. Without an index, Oracle has to read the entire table from disk to locate rows that match a selection criteria.

The presence of an index for a table is entirely optional and transparent to users and developers of database applications. For example, Applications can access table data with or without associated indexes.

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

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

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

Oracle8 supports several different types of indexes to satisfy many types of application requirements. The following sections explain more about the various types of indexes that you can create for a table's columns.

When an index is created, Oracle fetches and sorts the columns to be indexed and stores the ROWID along with index value for each row. Oracle loads the index from the bottom up. They are logically and physically independent of the data associated with the tables.

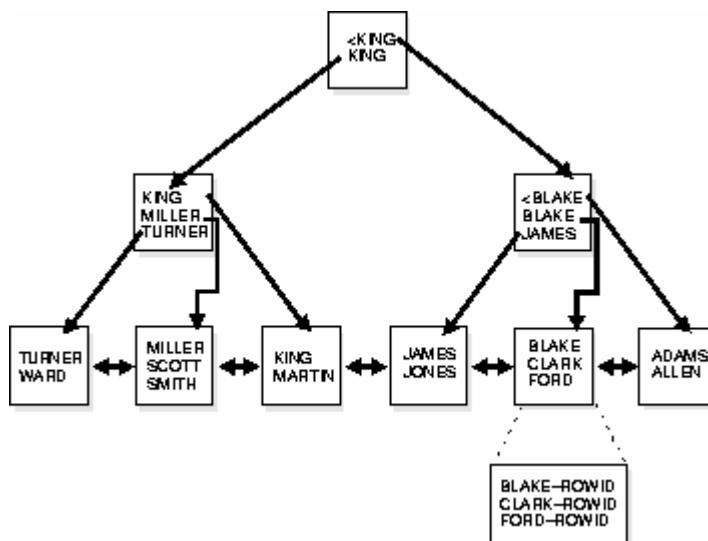
How Indexes Are Stored

When you create an index, Oracle automatically allocates an index segment to hold the index's data in a tablespace.

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

The Internal Structure of Indexes

Oracle uses B*-tree indexes that are balanced to equalize access times to any row. The following figure illustrates this:



The upper blocks (branch blocks) of a B*-tree index contain index data that points to lower level index blocks. The lowest level index blocks (leaf blocks) contain every indexed data value and a corresponding ROWID used to locate the actual row; the leaf blocks are doubly linked. Indexes in columns containing character data are based on the binary values of the characters in the database character set.

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

Advantages of B*-tree Structure

The B*-tree structure has the following advantages:

All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.

- B*-tree indexes automatically stay balanced.
- All blocks of the B*-tree are three-quarters full on the average.
- B*-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.
- B*-tree performance is good for both small and large tables, and does not degrade as the size of a table grows.

What Is a Bitmap Index?

Oracle provides five indexing schemes: B*-tree indexes (currently the most common), B*-tree cluster indexes, hash cluster indexes, reverse key indexes, and bitmap indexes. These indexing schemes provide complementary performance functionality.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value.

In a regular index, this is achieved by storing a list of rowids for each key corresponding to the rows with that key value. (The Oracle Server stores each key value repeatedly with each stored rowid.)

In a bitmap index, a bitmap for each key value is used instead of a list of rowids. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value.

A mapping function converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a regular index even though it uses a different representation internally. If the number of different key values is small, bitmap indexes are very space efficient.

Bitmap indexing efficiently merges indexes that correspond to several conditions in a WHERE clause. Rows that satisfy some, but not all conditions are filtered out before the table itself is accessed. As a result, response time is improved, often dramatically.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. If the values in a column are repeated more than a hundred times, the column is a candidate for a bitmap index. Even columns with a lower number of repetitions (and thus higher cardinality), can be candidates if they tend to be involved in complex conditions in the WHERE clauses of queries.

For example, in a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B*-tree index, particularly when this column is often queried in conjunction with other columns.

B*-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as CUSTOMER_NAME or PHONE_NUMBER. A regular B*-tree index can be several times larger than the indexed data. Used appropriately, bitmap indexes can be significantly smaller than a corresponding B*-tree index.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be quickly resolved by performing the corresponding boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example

Table 12-1 shows a portion of a company's employee data.

EMPLOYEE_NO	NAME	JOB	DEPTNO	SALARY
101	CLARK	MANAGER	10	1000
102	SCOTT	SALESMAN	10	3000

103	WARD	SALESMAN	20	2000
104	ALLAN	MANAGER	20	1000
105	MILLER	CLERK	10	2444
106	FORD	ANALYST	20	5643

Table 12-1: Bitmap Index Example

Since NAME, JOB, DEPTNO, SALARY are all low-cardinality columns. It is appropriate to create bitmap indexes on these columns. A bitmap index should not be created on EMPLOYEE # because this is a high-cardinality column. Instead, a unique B*-tree index on this column in order would provide the most efficient representation and retrieval.

DEPTNO=10	DEPTNO=20
1	0
1	0
0	1
0	1
1	0
0	1

Table 12-2: Sample Bitmap

Each entry (or "bit") in the bitmap corresponds to a single row of the EMPLOYEE table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap DEPTNO=10 contains a one as its first bit: this is because the deptno is 10 in the first row of the EMPLOYEE table.

A sample query for a bitmap example:

```
SELECT COUNT(*) FROM EMPLOYEE  
WHERE JOB='manager' and deptno IN (10,20);
```

Bitmap indexes can process this query with great efficiency by merely counting the number of ones in the resulting bitmap.

Unique indexes

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

For example, Primary Key and unique constraints are by default creates unique index.

Composite Indexes

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

Creating Indexes

Indexes can be created in the following ways:

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

Example:

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

Dropping Indexes

Index can be dropped by using Drop command.

Syntax:

```
Drop index <indexname>;
```

Example:

```
DROP INDEX I1;
```

Clusters

Clusters are an optional method of storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together.

For example, the EMP and DEPT table share the DEPTNO column. When you cluster the EMP and DEPT tables, Oracle physically stores all rows for each department from both the EMP and DEPT tables in the same data blocks.

Because clusters store related rows of different tables together in the same data blocks, properly used clusters offer two primary benefits:

Disk I/O is reduced and access time improves for joins of clustered tables.

In a cluster, a cluster key value is the value of the cluster key columns for a particular row. Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value.

Therefore, less storage might be required to store related table and index data in a cluster than is necessary in non-clustered table format.

For example, notice how each cluster key (each DEPTNO) is stored just once for many rows that contain the same value in both the EMP and DEPT tables.

The Cluster Key

The cluster key is the column, or group of columns, that the clustered tables have in common. You specify the columns of the cluster key when creating the cluster. You subsequently specify the same columns when creating every table added to the cluster.

For each column specified as part of the cluster key (when creating the cluster), every table created in the cluster must have a column that matches the size and type of the column in the cluster key. No more than 16 columns can form the cluster key, and a cluster key value cannot exceed roughly one-half (minus some overhead) the available data space in a data block. The cluster key cannot include a LONG or LONG RAW column.

You can update the data values in clustered columns of a table. However, because the placement of data depends on the cluster key, changing the cluster key for a row might cause Oracle to physically relocate the row. Therefore, columns that are updated often are not good candidates for the cluster key.

The Cluster Index

You must create an index on the cluster key columns after you have created a cluster. A cluster index is an index defined specifically for a cluster. Such an index contains an entry for each cluster key value. To locate a row in a cluster, the cluster index is used to find the cluster key value, which points to the data block associated with that cluster key value. Therefore, Oracle accesses a given row with a minimum of two I/Os (possibly more, depending on the number of levels that must be traversed in the index).

The following Example illustrates the creation of Clusters

- Create a cluster for the common column.
- Create an index for the cluster
- Create the table along with clusters

Without creating indexes, records cannot be inserted. The following example creates the cluster along with tables.

```
CREATE CLUSTER CLEMP(deptno NUMBER(3));
Create index clind on cluster clemp;
Create table dept (deptno number(3),dname varchar2(20),loc varchar2(20))
cluster clemp(deptno);
Create table emp(empno number,ename varchar2(20),deptno number(3)) cluster
clemp(deptno);
```

When records are selected, the rowid of the first deptno will be the same.

Output:

```
SELECT ROWID FROM CLUSTER clemp;
```

```
ROWID
-----
AAAAzmAAFAAAAJ+AAA
AAAAzmAAFAAAAJ / AAA
AAAAzmAAFAAAAKAAAA
AAAAzmAAFAAAAKBAAA
AAAAzmAAGAAAAIAAAA
```

Dropping Clusters

Clusters are dropped using DROP CLUSTER command.

Example:

```
DROP CLUSTER CLEMP INCLUDING TABLES;
```

12.8 Short Summary

Privileges are the right to execute a particular SQL statement. They can be granted and revoked using GRANT and REVOKE statements respectively.

Database objects can be classified as

Views - Used as a tailor made presentation of a query

Views can be of Simple View, Join View or Force view. Views created out of a Join condition can be manipulated with some restrictions.

Synonyms - They are alias name for an object

Sequences - Created for automatic generation of numbers

Indexes - Used to improve performance on columns that appear in WHERE clause often.

Clusters - They are another performance techniques that improve queries on joins.

12.9 Brain Storm

1. What is a View?
2. Can create command of view have order by clause.

3. What is Force View?
4. What is Key preserved Table & Non preserved table? Is Non Key preserved table updateable? Explain.
5. What are the different types of Views?
6. Which views are updateable? Mention non-updateable views.
7. How to document Sequence Value?
8. Which parameters in Sequence can be altered?
9. What is Synonym? How does it differ from View?
10. What are the different types of Synonyms?
11. What is Cluster?
12. Mention disadvantages of Clusters?
13. Can Long & Long Raw columns be indexed?

808

Lecture 13

Introduction to PL/SQL

Objectives

After completing this lesson, you should be able to do the following

- ❖ Discuss the Introduction to PL/SQL
- ❖ Describe PL/SQL Architecture
- ❖ Discuss the Declares Variables
- ❖ Describe about Control Structures
- ❖ Describe Iteration Control

Coverage Plan

Lecture 13

- 13.1 Snap shot
- 13.2 Introduction to pl/sql
- 13.3 Pl/sql architecture
- 13.4 Declares variables
- 13.5 Control structures
- 13.6 Iteration control
- 13.7 Short summary
- 13.8 Brain Storm

13.1 Snap Shot

This chapter deals with the Procedural Language extension to SQL, its advantages in the programming aspect and the complete architecture of PL/SQL. This chapter also deals with various controls structures, Iterative control statements.

- Introduction to PL/SQL
- Architecture of PL/SQL
- Different types of Blocks
- PL/SQL Delimiters
- Writing simple programs
- Working with Control Structures
- Using Loop statements

13.2 Introduction to PL/SQL

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

What is PL/SQL?

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

Features of PL/SQL

- Allows users to embed one or more SQL statements together and execute as a single SQL unit.
- Allows declaration of variables.

- Allows usage of conditional constructs
- Allows programming of error-handling using exceptions
- Allows row-by-Row processing of data using cursors
- Allows triggers to be created and fired.

Advantages of PL/SQL

SQL Support

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

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

Better performance

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

Support for Object Oriented Programming

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

Portability

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

Higher Productivity

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

Integration with Oracle

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

13.3 PL/SQL Architecture

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

- Declarative part
- Executable part
- Error handling part

Declarative Part

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

Syntax:

DECLARE

 Set of variables

Executable Part

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

Error handling part

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

Architecture of PL/SQL

PL/SQL runtime system is a technology, which has an engine that executes PL/SQL blocks and subprograms. The engine can be installed in the Oracle server or in an application development tool such as Forms and Reports. The engine can reside in either of the two environments:

- Oracle server
- Oracle Tools

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

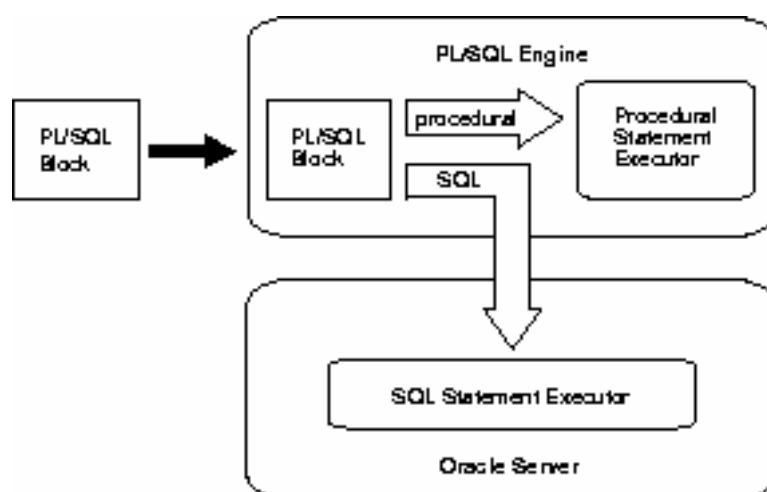


Figure 7.1 PL/SQL Engine

PL/SQL Engine In the Oracle Server

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

- Anonymous Blocks
- Named Blocks
- Stored Subprograms
- Triggers

Anonymous Blocks

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

Named Blocks

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

Stored Subprograms

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

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

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

Each stored subprogram has a name and is called or manipulated using the name in Oracle. PL/SQL structure and the associated rules are the same when a subprogram is constructed using them. Procedures and functions are known as subprograms. They can be called inside triggers also.

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

Database Triggers

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

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

PL/SQL engine In Oracle Tools

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

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

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

PL/SQL Delimiters

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

Delimiter	Name	Description
;	Statement Terminator	Each statement terminates with a Semicolon.
:=	Assignment Operator	Used to assign values to (A:=10) variables.
	Label Indicator	Used to define the labels inside a block.
--	Single Line	Commenting a single line Comment
/* .. */	Multiline Comment	Many statements can be placed inside this and they will not be executed.
..	Range indicator	This is used in For Loops which indicate the minimum range and maximum range.
&	Substitution Variable	Used to substitute values to variables.
	Bind or Session Variable accessed.	Global Variable can be
%	Attribute Indicator	They are used to assign the datatype of a column to a variable.
	DBMS_OUTPUT.T.PUT_LINE	used to print messages on the Screen

Datatypes

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

NUMBER Datatypes

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

Binary_Integer

Binary_Integer datatype is used to store signed integers. Its magnitude ranges from -2147483647...2147483647. They are used to store array type of data. This type requires less storage compared to Number values.

Subtypes

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

- NATURAL
- NATURALN
- POSITIVE
- POSITIVEN
- SIGNTYPE

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

Collection Datatypes

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

Boolean Datatypes

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

Exception Datatype

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

Writing a Simple Program

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

```
BEGIN
```

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

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

Example 7.1

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

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

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

Example 7.2

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

The assignment operator (&) is used to accept the value in the numb variable and this value assigned to x. It is similar to the continuous insert statements dealt in chapter2.

Using the SELECT Statement inside PL/SQL

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

Example 7.3

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

The variable s is assigned the salary of the employee number 7369 and the result would be displayed.

13.4 Declaring Variables

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

Using DEFAULT

```
DECLARE
    S NUMBER DEFAULT 10;
BEGIN
    DBMS_OUTPUT.PUT_LINE(s);
END
```

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

Using CONSTANT

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

Example 7.4

```
DECLARE
    Pi CONSTANT REAL:=3.14;
    Area NUMBER;
    R NUMBER:=&R;
BEGIN
    Area:=pi*r**2;
    DBMS_OUTPUT.PUT_LINE('THE AREA OF CIRCLE WITH RADIUS '||r||' IS '||area);
END;
```

In the above example, the variable pi is declared as a constant variable whose value cannot change anywhere inside the program. Re-assigning the value for the same variable leads to an error.

Using NOT NULL

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

Example 7.5

```
DECLARE
    s VARCHAR2(10) NOT NULL := 'RADIANT';
BEGIN
    DBMS_OUTPUT.PUT_LINE(s);
END;
```

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

13.5 Control Structures

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

```
IF...THEN
IF...THEN..ELSE
IF..THEN..ELSEIF
IF-THEN
```

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

Syntax

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

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

Example 7.6

```
DECLARE  
    s number;  
BEGIN  
    S:=&a;  
    If s>=10 THEN  
        DBMS_OUTPUT.PUT_LINE('S greater than or equal to 10');  
    END IF;  
END;
```

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

IF-THEN-ELSE

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements. The general syntax is 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 yields FALSE or NULL. Thus, the ELSE clause ensures that a sequence of statements is executed. Modifying the above example.

Example 7.7

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

IF-THEN-ELSIF

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

Syntax:

```
IF CONDITION1 THEN
    SEQUENCE_OF_STATEMENTS1;
ELSIF CONDITION2 THEN
    SEQUENCE_OF_STATEMENTS2;
ELSE
    SEQUENCE_OF_STATEMENTS3;
END IF;
```

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

Example 7.8

```
DECLARE
```

```
A NUMBER :=&A;
B NUMBER :=&B;
BEGIN
IF A>B THEN
DBMS_OUTPUT.PUT_LINE('THE GREATEST NUMBER IS '||A);
ELSIF B> A THEN
DBMS_OUTPUT.PUT_LINE('THE GREATEST NUMBER IS '||B);
ELSE
DBMS_OUTPUT.PUT_LINE('BOTH ARE EQUAL');
END IF;
END;
```

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

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

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

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

13. 6 Iterative Control

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

- LOOP
- WHILE LOOP
- FOR LOOP

LOOP

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

Syntax

```
LOOP  
Sequence_of_statements;  
End LOOP.
```

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

Example 7.9

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

The output leads to an error. Because the number of times the loop must be executed is not specified and it goes for an infinite loop. In order to terminate the loop some form of termination statement is required. The EXIT statement is used to perform this operation.

Example 7.10

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

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

EXIT-WHEN

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

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

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

Example 7.11

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

```
a:=a+1;  
EXIT WHEN a>20;  
END LOOP;  
END;
```

While loop

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

Syntax

```
While condition loop  
Sequence_of_statements;  
END LOOP;
```

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

Example 7.12

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

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

FOR-LOOP

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

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

The range is evaluated when the FOR loop is first entered and is never re-evaluated. 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. The following example displays the reversal of a string:

Example 7.13

```
DECLARE
    S VARCHAR2(20) := '&s';
    S1 VARCHAR2(20);
BEGIN
    FOR I IN 1..LENGTH(S)
    LOOP
        S1 := S1 || SUBSTR(S, -I, 1);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(s1);
END;
```

In the above example, a string is accepted. The number of iterations is fixed by the length of the string. Each character in the string is extracted and assigned to the variable S1. The variable is called a counter variable and it cannot be assigned or declared.

Example 7.14

```
DECLARE
```

```
s VARCHAR2(20):='&s';
s1 VARCHAR2(20);

BEGIN
    FOR i IN 1..LENGTH(s)
    LOOP
        s1:=s1||SUBSTR(s,-i,1);
        DBMS_OUTPUT.PUT_LINE('The counter value is'||i);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(s1);
END;
```

This displays the value of the counter variable.

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

```
FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    sequence_of_statements; -- executes three times
END LOOP;
```

Nevertheless, the range bounds are to be written in ascending (not descending) order. Inside a FOR loop, the loop counter can be referenced like a constant. So, the loop counter can appear in expressions but it cannot be assigned values, 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; -- illegal
    END IF;
END LOOP;
```

Iteration schemes

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

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
code IN ASCII('A')..ASCII('J')
```

LOOP LABELS

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

```
<<outer>>
LOOP
...
LOOP
..
Exit outer WHEN... exit both loops
End Loop;
...
End Loop outer;
```

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

13. 7 Short Summary

PL/SQL stands for Procedural Language Extension for SQL. They implement the procedural constructs that are available in other programming languages.

Basic unit of PL/SQL is a block.

Blocks are of different types:

- Anonymous Blocks
- Named Blocks
- Subprograms
- Triggers

PL/SQL contains three iterative control statements

- Loop
- While loop
- For loop

13.8 Brain Storm

1. PL/SQL is a _____
2. PL/SQL increases performance
 - a. True
 - b. False
3. Boolean data type stores _____, _____ and _____
4. Which type attribute provides a record type that represents a row I na table
 - a . Varray type
 - b. %rowtype
 - c. %type
5. The advantages of subtypes are _____, _____ and _____.
6. In which loop construct Reverse keyword is used
 - a. While loop
 - b. if...endif
 - c. loop..until
 - d. for loop

7. Consider a PL/SQL block

```
Begin  
.....  
Goto ins_1  
.....  
<<ins_7>>  
End;
```

Will this block be executed successfully?

8. Null includes

- a. Missing Values
- b. 0
- c. Both missing values and 0

¤...¤

Lecture 14

Cursors

Objectives

After completing this lesson, you should be able to do the following

- ☒ Describe about Understanding Errors
- ☒ Describe types of Cursors
- ☒ Discuss about Cursor Variables
- ☒ Discuss about eg. For using constrained cursors

Coverage Plan

Lecture 14

- 14.1 Snap Shot
- 14.2 Understanding cursors
- 14.3 Types of cursors
- 14.4 Cursor variables
- 14.5 Using constrained & Unconstrained cursor variables
- 14.6 Short summary
- 14.7 Brain Storm

14.1 Snap Shot

This chapter deals with working of Cursors and different types of Cursors. This chapter also gives an insight into the working of composite datatypes and their methods. This chapter also introduces Cursor Variables and their usage.

- Cursors
- Implicit Cursors
- Explicit Cursors
- Cursor Attributes
- Constrained Cursors and Unconstrained Cursors
- Using Cursor For update
- Using Composite Datatypes

Oracle uses workareas to execute SQL statements and store processing results. A PL/SQL construct called Cursor allows a workarea to be named, to store and access its information. They are typically used in areas where the query inside the PL/SQL block retrieves more than one record. A cursor is a pointer to handle the context area or the memory area or workarea. The result set of information is called Result Set.

14.2 Understanding Cursors

Why Cursors?

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

14.3 Types of Cursors

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

- Implicit Cursors

- Explicit Cursors

Implicit Cursors

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

Explicit cursors

SELECT statements that occur within PL/SQL blocks are known as embedded. They must return one row and may only return one row. To get around this a SELECT statement is defined as a cursor (an area of memory), the query is executed and the returned rows are manipulated within the cursor. Explicit Cursors can be of two types.

- Static Cursors
- Dynamic Cursors

Static Cursors

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

Dynamic Cursors

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

Each cursor has four attributes

%ROWCOUNT Returns the number of rows processed
by a SQL statement.

%FOUND Holds TRUE if at least one row is
processed

%NOTFOUND Holds TRUE if no rows are processed.

%ISOPEN Holds TRUE if a cursor is open or
FALSE if cursor has not been opened or
been closed. They are used only
in connection with explicit cursors.

%ROWCOUNT

The %ROWCOUNT attribute is used to return the number of records fetched. This is in accordance with the number of FETCHes done.

%FOUND

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

%NOTFOUND

It is the logical opposite of %FOUND. If records are fetched, the %NOTFOUND is FALSE and TRUE if there are no more records to be processed. Using this we can terminate from the loop.

%ISOPEN

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

The tabular structure illustrates this better.

Attribute	Is TRUE	Is FALSE
%ISOPEN	If the cursor is opened.	If the cursor is not open.
%FOUND	When records are fetched using FETCH statement	If there are no more records to be fetched and processed
%NOTFOUND	When there are no records available to be fetched	When records are fetched by the FETCH statement
%ROWCOUNT	Holds the number of records	-

Explicit Cursors

Explicit cursor manipulation is performed using four commands

- DECLARE
- OPEN
- FETCH
- CLOSE

Declaring a Cursor

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

Syntax:

CURSOR <cursorname> IS <query>

Example:

CURSOR C1 is SELECT ename,job FROM emp;

Opening a Cursor

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

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

Syntax:

OPEN <cursor name>

Example

OPEN c1;

Fetching Records from the cursor

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

Syntax

FETCH <cursor name> INTO <variable list>

Example

FETCH C1 INTO x,y;

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

Closing a Cursor

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

The following examples illustrate the concept of using Static cursors.

Example

This example is used to select all the employee names and their jobs from the table.

```
DECLARE
    CURSOR c1 IS SELECT ename,job FROM emp;
    Mname VARCHAR2(20);
    Mjob VARCHAR2(20);
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO mname,mjob;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(mname||' is a ' ||mjob);
    END LOOP;
    CLOSE c1;
END;
```

The output of the program is :

```
KING is a PRESIDENT
BLAKE is a MANAGER
CLARK is a MANAGER
JONES is a MANAGER
MARTIN is a SALESMAN
ALLEN is a SALESMAN
TURNER is a SALESMAN
JAMES is a CLERK
WARD is a SALESMAN
FORD is a ANALYST
SMITH is a CLERK
SCOTT is a ANALYST
ADAMS is a CLERK
MILLER is a CLERK
```

All the employee names and their respective jobs are displayed from the employee table. Here, additionally there is a EXIT WHEN statement which was dealt in chapter 7. The output differs depending on where the FETCH statement is placed. If the EXIT statement is placed after the DBMS_OUTPUT.PUT_LINE statement, the last record is displayed twice since the variables contain the last fetched record. Consider the following example is:

```
DECLARE
  CURSOR c1 IS SELECT ename,job FROM emp;
  Mname VARCHAR2(20);
  Mjob VARCHAR2(20);
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO mname,mjob;
    DBMS_OUTPUT.PUT_LINE(mname||' is a ' ||mjob);
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
END;
The output will look like:
KING is a PRESIDENT
BLAKE is a MANAGER
CLARK is a MANAGER
JONES is a MANAGER
MARTIN is a SALESMAN
ALLEN is a SALESMAN
TURNER is a SALESMAN
JAMES is a CLERK
WARD is a SALESMAN
FORD is a ANALYST
SMITH is a CLERK
SCOTT is a ANALYST
ADAMS is a CLERK
MILLER is a CLERK
MILLER is a CLERK
```

Example

This example SELECTS records from the table based on the department number.

```
DECLARE
  CURSOR c1 IS SELECT ename,job,deptno FROM emp where deptno=10;
  Mname VARCHAR2(20);
  Mjob VARCHAR2(20);
  mdno number;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO mname,mjob,mdno;
    EXIT WHEN c1%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE(mname||' is a ' ||mjob||' working in '||mdno);
      END LOOP;
      CLOSE c1;
    END;
```

The output of the program will be:

```
KING is a PRESIDENT working in 10
CLARK is a MANAGER working in 10
MILLER is a CLERK working in 10
```

Using CURSOR FOR Loops

A FOR CURSOR opens the cursor, fetches the records and closes the cursors. In the case of FOR CURSOR loops, declaration alone must be given. Changing the above example to a FOR CURSOR loop, the programming code is reduced.

Example

```
DECLARE
  CURSOR c1 IS SELECT ename,job,deptno FROM emp where deptno=10;
  Mname VARCHAR2(20);
  Mjob VARCHAR2(20);
  mdno number;
BEGIN
  FOR EREC IN C1
  LOOP
    DBMS_OUTPUT.PUT_LINE(EREC.Ename||' is a '||EREC.job||' working in
' ||EREC.DEPTNo);
  END LOOP;
END;
```

The output will be the same. In the above example, there is no explicit OPEN, FETCH and CLOSE statements since FOR automatically takes care of.

Using %TYPE

The %TYPE attribute provides the datatype of a variable or database column. Supposing a variable has to be defined to be of the same datatype as a column in a table, then this attribute is used. This defines the exact datatype. The syntax is :

Variablename <tablename>.<columnname>%TYPE

Example

Refining example,

```
DECLARE
    CURSOR c1 IS SELECT ename,job,deptno FROM emp where deptno=10;
    Mname emp.ename%type;
    Mjob emp.job%type;
    mdno emp.deptno%type;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO mname,mjob,mdno;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(mname||' is a ' ||mjob||' working in '||mdno);
    END LOOP;
    CLOSE c1;
END;
```

There is no change in the output.

Using %ROWTYPE

The %ROWTYPE attribute provides a record type that represents a row in a table. The variable declared as this type can store an entire row of data selected from the table or fetched from a cursor. This type is used in cases where all the column names need to be retrieved from the table.

Example

This query displays all the records from the table.

```
DECLARE
```

```
CURSOR c1 IS SELECT * from emp;
e_rec emp%rowtype;
BEGIN
OPEN c1;
LOOP
FETCH c1 INTO e_rec;
EXIT WHEN c1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(e_rec.ename||' is a '||e_rec.job||' and earns
'||e_rec.sal);
END LOOP;
CLOSE c1;
END;
```

The output of the program will be:

```
KING is a PRESIDENT and earns 5000
BLAKE is a MANAGER and earns 2850
CLARK is a MANAGER and earns 2450
JONES is a MANAGER and earns 2975
MARTIN is a SALESMAN and earns 1250
ALLEN is a SALESMAN and earns 1600
TURNER is a SALESMAN and earns 1500
JAMES is a CLERK and earns 950
WARD is a SALESMAN and earns 1250
FORD is a ANALYST and earns 3000
SMITH is a CLERK and earns 800
SCOTT is a ANALYST and earns 3000
ADAMS is a CLERK and earns 1100
MILLER is a CLERK and earns 1300
```

The queries that appear in the Cursor declaration can be a JOIN or a SUB-QUERY.

Understanding COMPOSITE DATATYPES

Composite Datatypes are a set of datatypes that can hold multiple columns and multiple records. Records and Tables together are called as Composite Datatypes.

A PL/SQL record is a variable that contains a collection of separate fields. Each field is individually addressable. Referencing the field names can be done in both assignments and

expressions. The fields within a record may have different datatypes and sizes, like the columns of a database table. Records are a convenient way of storing a complete fetched row from a database table. Using %ROWTYPE attribute to declare a record based upon a collection of database columns from a table or view, the fields within the record take their names and datatypes from the columns of the table or view.

Using Record Type

Record type variables are declared using the keyword RECORD. The following example shows this:

Example

```
DECLARE
  TYPE TREC IS RECORD(A VARCHAR2(20),B NUMBER);
  TVAR TREC;
  CURSOR C1 IS SELECT PNAME,PCODE FROM PRODUCT;
  BEGIN
    OPEN C1;
    LOOP
      FETCH C1 INTO TVAR;
      EXIT WHEN C1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(TVAR.A||' '||TVAR.B);
    END LOOP;
    CLOSE C1;
  END;
```

The Output will look like:

```
navtal lock 101
storewell 102
cinthol old 103
1.5 ton a/c 104
1.0 ton a/c 105
0.75 ton a/c 106
puf refrigerator 107
hair dye 108
shaving cream 109
jumpin juke 110
thomas cook 111
```

Using Table Type

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

Syntax:

Type <typename> is Table of <datatype> index by binary_integer;

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

Example

```
DECLARE
  TYPE VTAB IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
  TAB VTAB;
  J BINARY_INTEGER:=0;
  CURSOR C1 IS SELECT ENAME FROM EMP;
BEGIN
  FOR I IN C1 LOOP
    J:=J+1;
    TAB(J):=I.ENAME;
    DBMS_OUTPUT.PUT_LINE(TAB(J));
  END LOOP;
END;
```

In this example, vtab is declared as a table type. Since it can hold any number of data the index must be specified. This displays all the records. In order to display specific values, collection methods are used.

Using Collection Methods

There are certain methods that can be used along with the table type. Following are the list of collection methods:

- COUNT

- EXISTS
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

Using COUNT

Count returns the number of elements that the collection contains. For example, if there are 10 elements the count will display 10.

Using FIRST and LAST

As the names indicate, FIRST and LAST are used to return the first and the last index number in the collection.

PRIOR and NEXT

PRIOR(n) indicates the number that precedes the index n in the collection. NEXT is used to traverse to the next element .

Using DELETE

DELETE is used to delete the specified element from the collection. Note that this does not delete data from the table. Delete without parameter deletes all the elements and delete with a number deletes the specified element alone.

Using Exists

In a collection, if a user would like to know the existence of an element, this can be done using the collection method Exists which takes in a value and checks for its existence.

An example using the collection types follows:

```
DECLARE
  TYPE VTAB IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
  TAB VTAB;
  J BINARY_INTEGER:=0;
  CURSOR C1 IS SELECT ENAME FROM EMP;
BEGIN
  FOR I IN C1 LOOP
    J:=J+1;
    TAB(J):=I.ENAME;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(TAB.FIRST);
END;
```

Note, in the above example, it displays the index number 1 and not the value. In order to display the value, the statement must be re-written as

```
Dbms_output.put_line(tab(tab.first));
```

The following example displays other values,

```
DECLARE
  TYPE VTAB IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
  TAB VTAB;
  J BINARY_INTEGER:=0;
  CURSOR C1 IS SELECT ENAME FROM EMP;
BEGIN
  FOR I IN C1 LOOP
    J:=J+1;
    TAB(J):=I.ENAME;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('THE TOTAL NUMBER OF RECORDS ARE '||TAB.COUNT);
  DBMS_OUTPUT.PUT_LINE('THE FIRST RECORD IS '||TAB(TAB.FIRST));
  DBMS_OUTPUT.PUT_LINE('THE LAST RECORD IS '||TAB(TAB.LAST));
  DBMS_OUTPUT.PUT_LINE('THE SECOND RECORD IS '||TAB(TAB.NEXT(TAB.FIRST)));
  DBMS_OUTPUT.PUT_LINE('THE LAST BUT PREVIOUS RECORD IS
'||TAB(TAB.PRIOR(TAB.LAST)));
END;
```

The displays would be as follows

The total number of records are 14

```
The first record is KING
The last record is MILLER
The second record is BLAKE
The last but previous record is ADAMS
```

Here KING is the first record. If the record is deleted, then the second record will be displayed as the first record.

```
DECLARE
TYPE VTAB IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
TAB VTAB;
J BINARY_INTEGER:=0;
CURSOR C1 IS SELECT ENAME FROM EMP;
BEGIN
FOR I IN C1 LOOP
J:=J+1;
TAB(J):=I.ENAME;
END LOOP;
DBMS_OUTPUT.PUT_LINE('THE TOTAL NUMBER OF RECORDS ARE '||TAB.COUNT);
DBMS_OUTPUT.PUT_LINE('THE FIRST RECORD IS '||TAB(TAB.FIRST));
DBMS_OUTPUT.PUT_LINE('THE LAST RECORD IS '||TAB(TAB.LAST));
DBMS_OUTPUT.PUT_LINE('THE SECOND RECORD IS '||TAB(TAB.NEXT(TAB.FIRST)));
DBMS_OUTPUT.PUT_LINE('THE LAST BUT PREVIOUS RECORD IS
'||TAB(TAB.PRIOR(TAB.LAST)));
TAB.DELETE(1);
DBMS_OUTPUT.PUT_LINE('THE TOTAL NUMBER OF RECORDS ARE '||TAB.COUNT);
DBMS_OUTPUT.PUT_LINE('THE FIRST RECORD IS '||TAB(TAB.FIRST));
DBMS_OUTPUT.PUT_LINE('THE LAST RECORD IS '||TAB(TAB.LAST));
DBMS_OUTPUT.PUT_LINE('THE SECOND RECORD IS '||TAB(TAB.NEXT(TAB.FIRST)));
DBMS_OUTPUT.PUT_LINE('THE LAST BUT PREVIOUS RECORD IS
'||TAB(TAB.PRIOR(TAB.LAST)));
END;
```

The display would now be,

```
The total number of record are 14
The first record is KING
The last record is MILLER
The second record is BLAKE
The last but previous record is ADAMS
The total number of records are 13
The first record is BLAKE
The last record is MILLER
```

```
The second record is CLARK  
The last but previous record is ADAMS
```

CURSOR FOR UPDATE

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

- The FOR UPDATE clause in the cursor declaration
- The WHERE CURRENT OF clause in an UPDATE or DELETE statement.

FOR UPDATE

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

The syntax is

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

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

Normally SELECT operation will not take any locks on the rows being accessed. This allows other sessions connected to the database to change the data being selected. The result set is still consistent. When the cursor is opened, the active set is determined. Any changes that have been committed prior to this point are reflected in the active set. Any changes made after this point, even if they are committed, are not reflected unless the cursor is reopened, which will evaluate the active set again. This is called as read-consistency. However, in the case of FOR UPDATE clause, exclusive row locks are taken on the rows in the active set before the OPEN returns. These locks prevent other sessions from changing the rows in the active set until the transaction is committed.

If another session already has locks on the rows in the active set, then the SELECT FOR UPDATE operation will wait for this waiting period- the SELECT FOR UPDATE will hang

until the other session releases the lock. To handle this situation, the NOWAIT clause can be used. If the rows are locked by another session, then OPEN will return immediately with the Oracle error:

ORA-54:resource busy and acquire with NOWAIT specified

Where Current of

If the cursor is declared with the FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an UPDATE or DELETE statement. This statement refers to the latest row fetched from the cursor. The syntax for this clause is

WHERE CURRENT OF cursorname

where *cursorname* is the name of a cursor that has been declared with a FOR UPDATE clause. The WHERE CURRENT OF clause evaluates to the row that is just retrieved by the cursor.

Note that the UPDATE statement updates only the column listed in the FOR UPDATE clause of the cursor declaration. If no columns are listed, then any column can be updated.

It is legal to execute a query with a FOR UPDATE clause, but not reference the rows fetched via WHERE CURRENT OF. In this case, the rows are still locked and thus can only be modified by the current session (which holds the lock). UPDATE and DELETE statements that modify these rows will not block if they are executed by the session holding the lock. The following example is illustrative of this:

```
DECLARE
  CURSOR c1 IS SELECT * FROM emp FOR UPDATE;
  CURSOR c2 IS SELECT * FROM emp;
BEGIN
  FOR emp_rec IN c1
  LOOP
    IF emp_rec.job='MANAGER' THEN
      UPDATE emp SET sal=sal+500 WHERE CURRENT OF c1;
    ELSIF emp_rec.job='CLERK' THEN
      UPDATE emp SET sal=sal+200 WHERE CURRENT OF c1;
    END IF;
  END LOOP;
  FOR erec IN c2
```

```
LOOP
    DBMS_OUTPUT.PUT_LINE(erec.ename || ' now earns '||erec.sal);
END LOOP;
END;
```

In the above-mentioned example the program first checks whether the job of the employee is MANAGER or CLERK.

If the condition evaluates to TRUE the cursor updates that particular record's salary as given in the program. This is achieved by using WHERE CURRENT OF clause. Otherwise the salary will be updated as mentioned for the whole table if the fetch results in TRUE condition. The output of the above program will be:

```
KING now earns 5000
BLAKE now earns 3350
CLARK now earns 2950
JONES now earns 3475
MARTIN now earns 1250
ALLEN now earns 1600
TURNER now earns 1500
JAMES now earns 1150
WARD now earns 1250
FORD now earns 3000
SMITH now earns 1000
SCOTT now earns 3000
ADAMS now earns 1300
MILLER now earns 1500
```

14.4 Cursor Variables

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

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

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

Declaring a Cursor Variable

Reference types in PL/SQL are declared using the syntax:

```
REF type  
where type is a previously defined type.
```

The REF keyword indicates that the new type will be a pointer to the defined type. The type of a cursor variable is therefore REF CURSOR. . The complete syntax for defining a cursor variable type is

```
TYPE type_name IS REF CURSOR RETURN return_type;
```

where type_name is the name of the new reference type, and return_type is a record type indicating the types of the select list that will eventually be returned by the cursor variable. The return type for a cursor variable must be a record type. It can be declared explicitly as a user-defined record or implicitly using %ROWTYPE.

Once the reference type is defined, the variable can be declared. The following declarative section shows different declarations for cursor variables.

```
DECLARE  
    TYPE emp_ref IS REF CURSOR RETURN emp%ROWTYPE;  
    TYPE t_emprecord IS RECORD (name emp.ename%type, desg emp.job%TYPE);  
    v_emprecord t_emprecord;  
    TYPE t_emprecordRef IS REF CURSOR RETURN t_empRecord;  
    TYPE t_empRef2 IS REF CURSOR RETURN v_emprecord%TYPE;  
    v_emprecordCV      t_emprecord;  
    v_emprecord      t_emprecordRef;
```

14.5 Using Constrained and Unconstrained Cursor Variables

The cursor variables in the previous section are constrained-they are declared for a specific return type only. When the variable is later opened, it must be opened for a query whose

select list matches the return type of the cursor. If not, the predefined exception ROWTYPE_MISMATCH is raised.

PL/SQL, however, allows the declaration of unconstrained cursor variables. An unconstrained cursor variable does not have a RETURN clause. When an unconstrained cursor variable is later opened, it can be opened for any query. The following declarative section declares an unconstrained cursor variable.

```
DECLARE  
  --TYPE t_FlexibleRef IS REF CURSOR;  
  v_CursorVar t_FlexibleRef;
```

Allocating Storage for Cursor Variables

Since a cursor variable is a reference type, no storage is allocated for it when it is declared. Before it can be used, it needs to point to a valid area of memory. This memory can be created automatically by the PL/SQL engine.

Controlling Cursor Variables

There are three statements that control a cursor variable: OPEN-FOR, FETCH and CLOSE. OPEN-FOR is used for a multi-row query. FETCH is used to retrieve rows from the result set one at a time. After all the rows are processed ,CLOSE closes the cursor variable and releases the memory.

Opening a Cursor Variable for a Query

In order to associate a cursor variable with a particular SELECT statement, the OPEN syntax is extended to allow the query to be specified. This is done with the OPEN FOR syntax.

```
OPEN cursor_variable FOR select_statement;
```

where cursor_variable is a previously declared cursor variable, and select_statement is the desired query.

Note that if the cursor variable is constrained, then the select list must match the return type of the cursor. If it is not constrained, the error

ORA-6504: PL/SQL: return types of result set variables or query do not match

For example,given a cursor variable declaration like this,

```
DECLARE
    TYPE T_PRODREF IS REF CURSOR RETURN PRODUCT%ROWTYPE;
    V_PRODUCT      T_PRODREF;
    V_PRODUCTTAB   PRODUCT%ROWTYPE;

    --WE CAN OPEN V_PRODUCT WITH:
BEGIN
    OPEN V_PRODUCT FOR SELECT * FROM PRODUCT;
    LOOP
        FETCH V_PRODUCT INTO V_PRODUCTTAB;
        EXIT WHEN V_PRODUCT%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(V_PRODUCTTAB.PNAME || ' ' || V_PRODUCTTAB.PCODE || '
' || V_PRODUCTTAB.DEPTNO);
    END LOOP;
    CLOSE V_PRODUCT;
END;
```

The output of the program will be:

```
navtal lock 101 40
storewell 102 40
cinthol old 103 10
1.5 ton a/c 104 20
1.0 ton a/c 105 20
0.75 ton a/c 106 20
puf refrigerator 107 20
hair dye 108 10
shaving cream 109 10
jumpin juke 110 10
thomas cook 111 30
```

If,on the other hand,we attempt to open v_productCV the following way,

```
OPEN v_productCV FOR SELECT deptno,pname,pcode FROM product;
```

Raises ORA-6504, since the select list of the query does not match the return type of the cursor variable.

Closing Cursor Variables

Cursor variables are closed just like static cursors using the CLOSE statement. This frees the resources used for the query. It does not necessarily free the storage for the cursor variable itself. The storage for the variable is freed when the variable goes out of scope. It is illegal to close a cursor or cursor variable that is already closed. Cursor variables can be closed on either the client or the server. In the preceding program, the Cursor is opened on the server (via the embedded anonymous block), and fetched from and closed back on the client. Since the cursor variable is declared as a host variable, it is unconstrained.

Example for using Constrained Cursors

Consider a query that displays records from multiple tables.

If Emp table is selected, it displays empno,ename and if Dept table is selected, it must display deptno,dname and if product is selected, it displays pcode and pname. In the above, records come from different tables but the type of data that is retrieved is of the same structure.

```
DECLARE
    TYPE rectab IS RECORD ( code NUMBER, name VARCHAR2(40));
    TYPE refcur IS REF CURSOR RETURN rectab;
    Recvar RECTAB;
    Refcur_var refcur;
    Opt NUMBER:=&N;
BEGIN
    IF opt=1 THEN
        OPEN refcur_var FOR SELECT empno,ename FROM emp;
    ELSIF opt=2 THEN
        OPEN refcur_var FOR SELECT deptno,dname FROM dept;
    ELSIF opt=3 THEN
        OPEN refcur_var FOR SELECT pcode,pname FROM product;
    ELSE
        DBMS_OUTPUT.PUT_LINE ('INVALID OPTION SELECTED');
    END IF;
    LOOP
        FETCH refcur_var INTO recvar;
```

```
        EXIT WHEN refcur_var%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Code is '||recvar.code ||' Name is
'||recvar.name);
      END LOOP;
      CLOSE refcur_var;
END;
```

When this block gets executed and if the option entered is 1, it displays,

```
Enter value for n: 1
old   6: Opt NUMBER:=&N;
new   6: Opt NUMBER:=1;
Code is 7839 Name is KING
Code is 7698 Name is BLAKE
Code is 7782 Name is CLARK
Code is 7566 Name is JONES
Code is 7654 Name is MARTIN
Code is 7499 Name is ALLEN
Code is 7844 Name is TURNER
Code is 7900 Name is JAMES
Code is 7521 Name is WARD
Code is 7902 Name is FORD
Code is 7369 Name is SMITH
Code is 7788 Name is SCOTT
Code is 7876 Name is ADAMS
Code is 7934 Name is MILLER
```

Likewise for 2 and 3 options, the corresponding values are selected.

But if the option is not a valid one, it displays

```
INVALID OPTION SELECTED
DECLARE
*
ERROR at line 1:
ORA-01001: invalid cursor
ORA-06512: at line 18
```

In order to capture this Exceptions are used.

Example for Un-Constrained Variables

```
DECLARE
  TYPE TCUR IS REF CURSOR;
  TVAR TCUR;
  EREC EMP%ROWTYPE;
  DREC DEPT%ROWTYPE;
  CHOICE NUMBER:=&C;
BEGIN
  IF CHOICE=1 THEN
    OPEN TVAR FOR SELECT * FROM EMP;
  LOOP
    FETCH TVAR INTO EREC;
    EXIT WHEN TVAR%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(erec.ename);
  END LOOP;
  CLOSE TVAR;
  ELSIF CHOICE=2 THEN
    OPEN TVAR FOR SELECT * FROM DEPT;
  LOOP
    FETCH TVAR INTO DREC;
    EXIT WHEN TVAR%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(drec.deptno||'          '||drec.dname);
  END LOOP;
  CLOSE TVAR;
  ELSE
    DBMS_OUTPUT.PUT_LINE('INVALID CHOICE');
  END IF;
```

The display of the program based on the condition will be:

Enter value for c: 2

```
old  6: CHOICE NUMBER:=&C;
new  6: CHOICE NUMBER:=2;
10   ACCOUNTING
20   RESEARCH
30   SALES
      OPERATIONS
```

14.6 Short Summary

- ❖ Cursors are named workareas that are used for multiple row processing in PL/SQL blocks.

- ❖ They can be of two types:
 - Implicit Cursors and Explicit Cursors
 - The four Cursor attributes which check the status of a cursor are %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.
- ❖ Composite Datatypes are the datatypes that can hold multiple columns and multiple records.
- ❖ Dynamic cursors can accept the table name at run-time. They can be further classified as Constrained and Un-constrained Cursors. Cursor variables can be associated with different statements at run time.

14.7 Brain Storm

1. Cursor is _____
2. The two types of cursors are _____ & _____
3. What are the statements used to control the cursor
 - a. open
 - b. fetch
 - c. close
 - d. declare
4. What is a Cursor variable?
5. _____ and _____ are the two types of REF cursor type
6. If the last fetch returned a row, % not found returns _____
7. The value of % row count before the first fetch is _____
8. SQL %isopen always yields _____
9. Locks can be released by
 - a. Commit
 - b. Savepoint
 - c. Rollback

Lecture 15

Exceptions

Objectives

After completing this lesson, you should be able to do the following

- ☒ Define PL/SQL exceptions
- ☒ Recognize unhandled exceptions
- ☒ List and use different types of PL/SQL exception handlers
- ☒ Trap unanticipated errors
- ☒ Describe the effect of exception propagation in nested blocks
- ☒ Customize PL/SQL exception messages

Coverage Plan

Lecture 15

- 15.1 Snap shot
- 15.2 Exceptions
- 15.3 Errors
- 15.4 User - defined exceptions
- 15.5 Unhandled exceptions
- 15.6 Short summary
- 15.7 Brain Storm

15.1 Snap Shot

This chapter deals with handling run-time errors that are raised due to various reasons and to give an understandable user-friendly message. Creating and displaying user-defined error messages are also dealt with.

- Introduction to Exception
- Predefined exception
- List of Predefined exceptions
- Handling user-defined exceptions
- Using Raise_application_error
- Usage of SQLCODE and SQLERRM

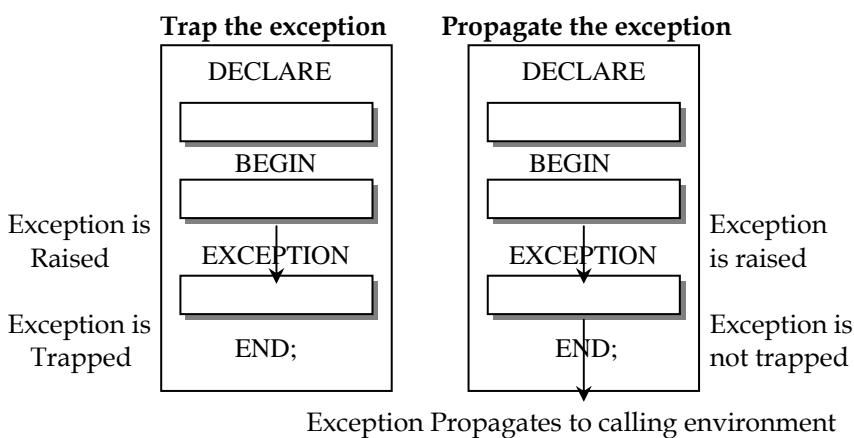
15.2 Exceptions

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

Two methods for raising an exception

1. An oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-04103 occurs when no rows are retrieved from the database in a select statement, then PL/SQL raises the exception NO_DATA_FOUND.
2. You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user defined or predefined.

Handling exceptions



15.3 Errors

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

- Runtime Errors
- Compile-time Errors

Exceptions can be broadly classified into:

- Predefined Exceptions
- User-defined Exceptions
- Un-defined Exceptions

Pre-defined Exceptions

Predefined exceptions are exceptions that are already defined by Oracle. The following list gives the set of Predefined Exceptions.

Advantages of Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time a command is issued, execution errors must be checked, as follows:

```
BEGIN
    SELECT ...
        -- check for 'no data found' error
    SELECT ...
        -- check for 'no data found' error
    SELECT ...
        -- check for 'no data found' error
```

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

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

NO_DATA_FOUND	A SELECT INTO statement returns no rows, or a deleted element is referenced in a nested table, or an uninitialized element is referenced in an index-by table. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised. SQL group functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls a group function will never raise NO_DATA_FOUND.
NOT_LOGGED_ON	PL/SQL 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 you pass an open host cursor variable to a stored subprogram, the return types of the actual and formal parameters must be compatible.

STORAGE_ERROR	PL/SQL runs out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	A nested table or varray element is referenced using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	You reference a nested table or varray element using an index number that is outside the legal range (-1 for example).
TIMEOUT_ON_RESOURCE	A timeout 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 a column value is selected 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 to a number fails. In SQL statements, INVALID_NUMBER is raised.
ZERO_DIVIDE	User tries to divide a number by zero.

The following examples illustrates the usage of Pre-defined exceptions

Zero-Divide

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

Example

```
DECLARE
  X NUMBER := &X;
  Y NUMBER := &Y;
BEGIN
  DBMS_OUTPUT.PUT_LINE('RESULT IS ' || X/Y);
END;
```

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

ORA-01476: divisor is equal to zero

In order to handle the exception, exception clause must contain the Exception Zero_Divide.
Refining the above example,

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

The second example illustrates the usage of NO_DATA_FOUND

Example

```
DECLARE
  MYENAME VARCHAR2(20);
BEGIN
  SELECT ENAME INTO MYENAME FROM EMP WHERE EMPNO=&X;
  DBMS_OUTPUT.PUT_LINE('THE NAME OF THE EMPLOYEE IS ' || MYENAME);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('NO RECORD FOUND');
END;
/
```

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

Example

```
DECLARE
  MYSAL NUMBER;
BEGIN
  SELECT SAL INTO MYSAL FROM EMP WHERE DEPTNO=10;
  DBMS_OUTPUT.PUT_LINE(MYSAL);
END;
```

This would display

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

Raises the pre-defined exception `too_many_rows`. To avoid this, handle the exception in the `Exception` clause.

```
DECLARE
  MYSAL NUMBER;
BEGIN
  SELECT SAL INTO MYSAL FROM EMP WHERE DEPTNO=10;
  DBMS_OUTPUT.PUT_LINE(MYSAL);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Use Cursors');
END;
```

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

Using Others

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

Example

```
DECLARE
  MYSAL NUMBER;
  MYENAME VARCHAR2(20);
BEGIN
  SELECT SAL INTO MYSAL FROM EMP WHERE DEPTNO=10;
  DBMS_OUTPUT.PUT_LINE(MYSAL);
  SELECT ENAME INTO MYENAME FROM EMP WHERE EMPNO=&X;
  DBMS_OUTPUT.PUT_LINE('THE NAME OF THE EMPLOYEE IS ' || MYENAME);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
```

Exception	Raised when ...
ACCESS_INTO_NULL	User tries to assign values to the attributes of an uninitialized (atomically null) object.
COLLECTION_IS_NULL	User tries to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or assigns values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	User tries 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, that cursor cannot be opened inside the loop.
DUP_VAL_ON_INDEX	User tries to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	User tries an illegal cursor operation such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of character string to a number fails because the character string does not represent a valid number. In procedural statements, VALUE_ERROR is raised.
LOGIN_DENIED	User tries logging on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or a deleted element is referenced in a nested table, or an uninitialized element is referenced in an index-by table. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised. SQL group functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls a group function will never raise NO_DATA_FOUND.

```

DBMS_OUTPUT.PUT_LINE('Use Cursors');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('OTHERS EXCEPTION HANDLED');
END;

```

Note: Always OTHERS must be the last handler in the Exception Section.

15.4 User-Defined Exceptions

PL/SQL allows users to define their own exceptions. Unlike Predefined exceptions, user-defined exceptions must be declared exceptions can be raised using one of the following:

- RAISE
- RAISE_APPLICATION_ERROR

In the case of user-defined exceptions, the following three steps must be performed:

- Declare the exception
- Raise the Exception
- Handle the Exception

Declaring Exceptions

Exceptions are declared in the declarative part of a PL/SQL block, subprogram or package. It contains a name followed by the datatype called Exception. The following syntax shows this:

```
DECLARE  
  Ex1 EXCEPTION;
```

Exception declaration is very much similar to variable declaration. But they cannot appear as a part of an assignment statement.

Raising Exceptions

Raising Exceptions means the exception declared by the user is raised explicitly unlike Predefined exceptions. The keyword RAISE is used to perform this operation. Syntax for using this is :

```
RAISE ex1;
```

Handling Exceptions

After the exception is raised, it must be handled. Handling of exceptions is performed using the exception handling section similar to handling of pre-defined exceptions.

WHEN ex1 THEN

.....

The following example illustrates this:

Example

```
DECLARE
  EX1 EXCEPTION;
  N NUMBER(5) := &x;
BEGIN
  IF N > 20000 THEN
    RAISE ex1;
  END IF;
EXCEPTION
  WHEN ex1 THEN
    DBMS_OUTPUT.PUT_LINE('Value must not exceed 20000');
END;
```

Consider another example. The following example adds the accepted value onto the table. If the salary exceeds 30000 the exception SALARY_RAISE is raised and handled.

```
DECLARE
  ENO NUMBER := &NO;
  NAME VARCHAR2(20) := '&B';
  SALARY NUMBER := &P;
  SALARY_RAISE EXCEPTION;
BEGIN
  IF SALARY > 30000 THEN
    RAISE SALARY_RAISE;
  ELSE
    INSERT INTO EMP(EMPNO,ENAME,SAL) VALUES(ENO,NAME,SALARY);
  END IF;
EXCEPTION
  WHEN SALARY_RAISE THEN
    DBMS_OUTPUT.PUT_LINE('SALARY EXCEEDS 30000');
END;

RAISE_APPLICATION_ERROR
```

Raise_Application_Error stops the program and terminates the block abruptly. Raise_Application_error is primarily used in triggers to rollback the transaction and give suitable error messages. The syntax is

```
RAISE_APPLICATION_ERROR ( erronumber , MESSAGE);
```

where the error messages span between -20000 and -20999. Message is a character data that can hold upto 2000 bytes. The following example illustrates this:

In the following example, raise_application_error is called if an employee's salary is missing:

```
DECLARE
    Emp_id number := &no;
    current_salary NUMBER;
    increase number:=&x;
BEGIN
    SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
    IF current_salary IS NULL THEN
        /* Issue user-defined error message. */
        raise_application_error(-20101, 'Salary is missing');
    ELSE
        UPDATE emp SET sal = current_salary + increase
        WHERE empno = emp_id;
    END IF;
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20000,'NO MATCHING RECORD');
END raise_salary;
```

This program accepts the employee number and the salary increase. If the corresponding employee is found, it updates the value else it raises an error. Assume 765 is a number which is not available in the table. The error message in this case will be:

```
ERROR at line 1:
ORA-20000: NO MATCHING RECORD
ORA-06512: at line 17
```

Using EXCEPTION_INIT - Use of PRAGMA

To handle unnamed internal exceptions, you must use the OTHERS handler or the pragma EXCEPTION_INIT must be used. A **pragma** is a compiler directive, which can be thought of as a parenthetical remark to the compiler. Pragmas (also called pseudo instructions) are 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 allows any internal exception to be referenced by name and to write a specific handler for it.

PRAGMA EXCEPTION_INIT is given 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. The pragma must appear somewhere after the exception declaration in the same declarative part, as shown in the following example:

```
DECLARE
  X EXCEPTION;
  PRAGMA EXCEPTION_INIT(X,-1400);
BEGIN
  INSERT INTO EMP(EMPNO) VALUES(NULL);
EXCEPTION
  WHEN X THEN
    DBMS_OUTPUT.PUT_LINE('VALUE CANNOT HOLD NULL VALUES');
END;
```

In the above example, if a null value is being inserted onto a column which cannot contain null values, an exception is raised.

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 limit cannot store numbers larger than 999:

```
DECLARE
    limit CONSTANT NUMBER(3) := 5000; -- raises an exception
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN ... -- cannot catch the exception
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates immediately to the enclosing block.

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:

Example

```
EXCEPTION
    WHEN INVALID_NUMBER THEN
        INSERT INTO ... -- might raise DUP_VAL_ON_INDEX
    WHEN DUP_VAL_ON_INDEX THEN . -cannot catch the exception
```

Branching to or from an Exception Handler

A GOTO statement cannot branch into an exception handler; nor can it 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
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

Using SQLCODE and SQLERRM

In an exception handler, the functions SQLCODE and SQLERRM can be used to find out which error has 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 tables and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message.

ORA-0000: normal, successful completion

An error number can be passed to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass It must be ensured that negative error numbers are not passed 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

When +100 is passed, in which case SQLERRM returns the message:

ORA-01403: no data found

Passing a zero to SQLERRM always returns the following message:

ORA-0000: normal, successful completion

SQLCODE or SQLERRM cannot be used directly in a SQL statement. Instead, their values must be assigned to local variables, which can then be used in the SQL statement, as the following example shows:

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);
```

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. SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell which internal exception was raised.

15.5 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. Also, if a stored subprogram fails with an unhandled exception, PL/SQL does not roll back database work done by the subprogram.

Unhandled exceptions can be avoided by coding an OTHERS handler at the topmost level of every PL/SQL block and subprogram.

How Exceptions Propagate

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

An exception can propagate beyond its scope, that is, beyond the block in which it is 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 it was declared has no handler for the exception named `past_due`, it 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.

15.6 Short Summary

Exceptions are error conditions or warnings that are raised whenever the programs raise any errors. Exceptions are of three types viz:

- Predefined Exceptions
- User Defined exceptions
- Un defined Exceptions

Some of the predefined exceptions are
`ZERO_DIVIDE`, `NO_DATA_FOUND`, `INVALID_CURSOR` and so on.

`Raise` and `Raise_Application_Error` are two statements that are used to handle user-defined exceptions.

`Pragma` are precompiler directive that attaches the error number to the user-defined exception.

15.7 Brain Storm

1. Runtime errors are handled by the _____ of the PL/SQL block

2. Which of the following is not a pre-defined exception?
No_dat_found
Invalid_null
Login_denied
Timeout_on_resource

3. Exception_init is used to handle _____

4. Exceptions raised in the declarative part can be handled in the same block.
a. True b. False

5. SQLCODE returns _____ for user-defined exceptions.

ANSWER

Lecture 16

Subprograms

Objectives

After completing this lesson, you should be able to do the following

- ❖ Discuss Introduction to Subprograms
- ❖ Describe about Procedures to Parameters
- ❖ Describe Functions to Notations
- ❖ Discuss about Procedures Vs Functions

Coverage Plan

Lecture 16

- 16.1 Snap shot
- 16.2 Introduction to subprograms
- 16.3 Procedures to parameters
- 16.4 Functions of notations
- 16.5 Procedures Vs functions
- 16.6 Short summary
- 16.7 Brain Storm

16.1 Snap Shot

This chapter shows how to use subprograms, which let a user to name and encapsulate a sequence of statements. Subprograms aid application development by isolating operations. They are like building blocks.

- Introduction to Subprograms
- Procedures
- Parameter Modes
- Declaring and using procedures and Functions
- Notations
- Procedures Vs Functions

16.2 Introduction TO Subprograms

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

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

Advantages of Subprograms

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

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

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

16.3 Procedures to Parameters

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

```
CREATE OR REPLACE PROCEDURE <procedurename>[parameter list] AS  
....  
          PL/SQL statements
```

A Procedure contains 2 parts

- Specification
- Body

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

Parameters

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

- ACTUAL
- FORMAL

Actual Parameters

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

Formal Parameters

They are the variables given while declaring subprograms.

Parameter Modes

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

- IN
- OUT
- IN OUT

IN Mode

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

Example

```
CREATE OR REPLACE PROCEDURE DISP_ENAME(ENO IN NUMBER) IS
MNAME VARCHAR2(20);
BEGIN
SELECT ENAME INTO MNAME FROM EMP WHERE EMPNO=ENO;
DBMS_OUTPUT.PUT_LINE('THE NAME IS ' || MNAME);
END;
```

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

```
EXEC disp_ename(7902);

Displays,

EXEC DISP_ENAME(7902);
the name is FORD

PL/SQL procedure successfully completed.
```

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

OUT Mode

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

```
CREATE OR REPLACE PROCEDURE DISP_SAL (ENO IN NUMBER, SALARY OUT NUMBER) AS
  MNAME VARCHAR2(20);
BEGIN
  SELECT SAL, ENAME INTO SALARY,MNAME FROM EMP WHERE EMPNO=ENO;
  DBMS_OUTPUT.PUT_LINE('THE EMPLOYEE NAME IS ' || MNAME);
END;
```

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

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

```
VAR <variablename> <datatype>
```

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

```
EXEC <procedurename>(value,:<variablename>);
```

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

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

```
PRINT <variablename>
```

The example and the output is shown below:

```
VAR n NUMBER
EXEC disp_sal(7902,:n);
the employee name is FORD
```

PL/SQL procedure successfully completed.

```
PRINT n
```

```
      N
-----
3000
```

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

```
DECLARE
MSAL NUMBER;
MENO NUMBER:=&N;
BEGIN
DISP_SAL(MENO,MSAL);
DBMS_OUTPUT.PUT_LINE('THE SALARY IS '||MSAL);
END;
```

When this block gets executed, it displays

```
Enter value for n: 7902
old   3: Meno number:=&n;
new   3: Meno number:=7902;

the employee name is FORD

the salary is 3000
PL/SQL procedure successfully completed.
```

IN OUT Mode

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

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

The output of the above program is ,
Procedure created.

The procedure, which contains an IN OUT parameter, cannot be executed at SQL prompt. A subprogram that contains an IN OUT parameter must be executed only inside a PL/SQL block. The below program shows how to execute an IN OUT parameter.

Example

```
DECLARE
Name VARCHAR2(20);
Var_x NUMBER:=&X;
BEGIN
DISP_PROD(VAR_X,NAME);
DBMS_OUTPUT.PUT_LINE('name is '||name ||' rate is '||var_x);
END;
```

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

The output would look like:

```
Enter value for x: 102
old   3: Var_x NUMBER:=&X;
new   3: Var_x NUMBER:=102;
```

```
name is storewell rate is 10000
```

PL/SQL procedure successfully completed.

The table below illustrates the differences between IN, OUT and IN OUT Parameter modes.

IN	OUT	IN OUT
Default	Must be specified	Must be specified
Passes value to a subprogram	Returns value to the caller	Passes initial value to a sub program and returns updated value to the caller
Formal Parameter acts like a constant	Acts like an uninitialized variable	Acts like an uninitialized variable
Formal parameter cannot be assigned with a value	Formal parameter cannot be used in an expression and must be assigned with a value	Formal parameter must be assigned with a value
Actual Parameter can be a constant, literal or an expression	Actual Parameter is passed by value(copy of the value is passed out)	Similar to OUT but the copy of the value can be read and written.

Passing Default Values

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

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

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

```
Exec create_dept;
```

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

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

Forward Declarations

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

```
DECLARE
  ...
  PROCEDURE award_bonus ( ... ) IS
    BEGIN
      calc_rating( ... ); -- undeclared identifier
      ...
    END;
  PROCEDURE calc_rating ( ... ) IS
    BEGIN
      ...
    END;
```

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

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

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

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

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

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

16.4 Functions of Notations

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

```
CREATE OR REPLACE FUNCTION <FUNCTIONNAME>[parameter list] RETURN datatype IS
    PL/SQL Statements.
    Return <value/variable>;
END;
```

An example of selecting the department information is shown below:

Example

```
CREATE OR REPLACE FUNCTION seldept(dno IN NUMBER) RETURN VARCHAR2 IS
    M_dNAME VARCHAR2(20);
```

```
BEGIN  
SELECT DNAME INTO M_DNAME FROM DEPT WHERE DEPTNO=DNO;  
RETURN M_DNAME;  
END;
```

Displays the corresponding department name.

Function created.

Unlike Procedures, Functions cannot be called using EXEC statement. They must be called using Select Statement. The following shows this:

```
SELECT seledept(10) FROM DUAL;
```

Normally a function is used to compute specific values. They are not meant to perform any DML operations on the table. For example, if a function tries to insert a record onto a table, Oracle throws an error. The example illustrates this:

```
CREATE OR REPLACE FUNCTION INSDEPT(DNO NUMBER,NAME VARCHAR2) RETURN NUMBER  
AS  
BEGIN  
INSERT INTO DEPT(DEPTNO,DNAME) VALUES(DNO,NAME);  
RETURN 1;  
END;
```

Function created.

```
select insdept(23,'purchase') from dual;  
select insdept(23,'purchase') from dual  
*  
ERROR at line 1:  
ORA-06571: Function INSDEPT does not guarantee not to update database
```

The function has to satisfy some purity levels for performing any DML operations. Purity Level determines to what extent the function does not do any alterations to the database object. The four levels of purity levels are shown in the following table.

Purity Level	Expansion	Description
WNDS	Write No Database State	Function does not perform any DML

		operations on tables.
RNDS	Read No Database State	Reading is restricted from any table
RNPS	Read No Package State	Function must not alter any packaged variables
WNPS	Write No Package State	Function cannot write onto any package variable.

Depending on the purity level, a function is restricted to the following restrictions:

1. A function callable from a SELECT statement must not perform any operation to any tables.
2. Functions can take only IN parameter and not OUT and IN OUT.
3. Return type of a function must be a database type.

Notations

When calling a subprogram, the actual parameters are written using either positional or named notation. That is, the association between an actual and formal parameter can be indicated by position or name. For example, given the declarations

Example

```
DECLARE
    acct INTEGER;
    amt REAL;
    PROCEDURE credit (acctno INTEGER, amount REAL) IS ...
```

you can call the procedure credit in four logically equivalent ways:

```
BEGIN
    credit(acct, amt);      -- positional notation
    credit(amount => amt, acctno => acct); -- named notation
    credit(acctno => acct, amount => amt); -- named notation
    credit(acct, amount => amt);           -- mixed notation
```

Positional Notation

The first procedure call uses positional notation. The PL/SQL compiler associates the first actual parameter, acct, with the first formal parameter, acctno. And, the compiler associates the second actual parameter, amt, with the second formal parameter, amount.

Named Notation

The second procedure call uses named notation. The arrow (called an association operator) associates the formal parameter to the left of the arrow with the actual parameter to the right of the arrow.

The third procedure call also uses named notation and shows that you can list the parameter pairs in any order. Therefore, you need not know the order in which the formal parameters are listed.

Mixed Notation

The fourth procedure call shows that you can mix positional and named notation. In this case, the first parameter uses positional notation, and the second parameter uses named notation. Positional notation must precede named notation. The reverse is not allowed. For example, the following procedure call is illegal:

```
credit(acctno => acct, amt); -- illegal
```

16.5 Procedures Vs Functions

The following table lists out the differences between procedures and functions

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

Viewing Procedures and Functions

Procedures and Functions are viewed using the Data_dictionary Views:

```
SELECT * FROM USER_SOURCE;
```

Dropping Subprograms

Subprograms when not required can be dropped using Drop command.

Syntax:

```
DROP FUNCTION <function name>;  
DROP PROCEDURE <procedure name>;
```

16.6 Short Summary

- ❖ Subprograms are set of PL/SQL blocks that are stored and executed. They can be of two types:
 - Procedures
 - Functions
- ❖ Procedures are used to perform specific tasks whereas Functions are used to compute calculations.
- ❖ Functions and Procedures are created using CREATE PROCEDURE / FUNCTION statement.
- ❖ The parameter modes are IN for READ, OUT for WRITE, IN OUT for READ WRITE.
- ❖ A function to be callable from a SELECT statement must be pure from any side-effects.
- ❖ Functions and procedures are dropped using DROP statement.

16.7 Brain Storm

1. Subprograms are_____
2. Give any two advantages of subprograms
3. What is the function of a return statement?
4. The variables declared in a subprogram specifications are called_____parameters

5. The three types of notations followed in calling a subprograms are _____, _____, _____.
6. The default mode for a parameter is _____
7. What do you mean by overloaded subprograms?
8. Default values can be assigned to parameters
 - a) True
 - b) False
9. A function can have only one return statement
 - a) True
 - b) False
10. What is a recursive subprogram?

ANSWER

Lecture 17

Packages

Objectives

After completing this lesson, you should be able to do the following

- ☒ Describe Packages and list their possible components
- ☒ Create a package to group together related variables, cursors, constants, exceptions , procedures, and functions
- ☒ Make a Package construct either public or private
- ☒ Describe about Overloading of Package

Coverage Plan

Lecture 17

- 17.1 Snap shot
- 17.2 Packages
- 17.3 Specification
- 17.4 Package body
- 17.5 Overloading package
- 17.6 Short summary
- 17.7 Brain Storm

17.1 Snap Shot

This chapter shows how to bundle related PL/SQL programming constructs into a package. The packaged constructs might include a collection of procedures or a pool of type definitions and variable declarations. For example, a Human Resources package might contain hiring and firing procedures. Once written, the general-purpose package is compiled, then stored in an Oracle database, where, like a library unit, its contents can be shared by many applications.

- Understanding Packages
- Advantages of Packages
- Package Specification and Body
- Overloading
- Understanding Some of the Packages

17.2 What is a Package?

A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the specification.

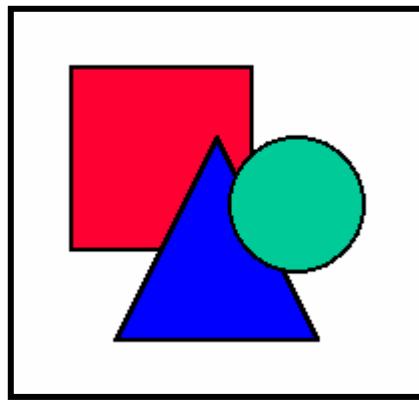
A package once compiled and stored, gets life in a session when any of the components of the package is referred in the session. Then onwards, the variables, arrays, cursors or subprograms defined in the package are allocated and loaded in the memory for that session and any change that is done to the components are live for the whole session across operations and across all the subprograms and triggers. This is unlike a subprogram where the variables have effect only inside the subprogram and not across programs in the session.

Unlike subprograms, packages cannot be called, parameterized, or nested. Still, the format of a package is similar to that of a subprogram as shown below.

```
CREATE PACKAGE name AS  -- specification (visible part)
    -- public type and item declarations
    -- subprogram specifications
```

```
END [name];  
  
CREATE PACKAGE BODY name AS -- body (hidden part)  
  -- private type and item declarations  
  -- subprogram bodies  
[BEGIN  
  -- initialization statements]  
END [name];
```

The specification holds public declarations, which are visible to the application. The body holds implementation details and private declarations, which are hidden from the application. This concept is very useful when modular applications are build which would enable a server centric application with all the application related programs and variables stored as part of the package.



You can debug, enhance, or replace a package body without changing the interface (package specification) to the package body.

To create packages and store them permanently in an Oracle database, CREATE PACKAGE and CREATE PACKAGE BODY statements are used. These statements can be executed interactively from SQL*Plus or Enterprise Manager.

In the example given below, you package a record type, a cursor, and two employment procedures are packaged. Notice that the procedure hire_employee uses the database sequence empno_seq and the function SYSDATE to insert a new employee number and hire date, respectively.

```
CREATE PACKAGE emp_actions AS -- specification
```

```
PROCEDURE hire_employee (
    ename  VARCHAR2,
    job    VARCHAR2,
    mgr    NUMBER,
    sal    NUMBER,
    comm   NUMBER,
    deptno NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS -- body
PROCEDURE hire_employee (
    ename  VARCHAR2,
    job    VARCHAR2,
    mgr    NUMBER,
    sal    NUMBER,
    comm   NUMBER,
    deptno NUMBER) IS
BEGIN
    INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
                           mgr, SYSDATE, sal, comm, deptno);
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

This package creates two procedures called hire_employee and fire_employee. The two procedures can be executed in the following way:

```
EXEC EMP_ACTIONS.HIRE_EMPLOYEE('SUDHA','MANAGER',7902,10000,200,10);
EXEC EMP_ACTIONS.FIRE_EMPLOYEE(2);
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. So, the body (implementation) can be changed without having to recompile the calling programs.

Advantages of Packages

Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance.

Modularity

Packages logically related types, items, and subprograms in a named PL/SQL module to be encapsulated. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

Easier Application Design

When designing an application, all that is needed initially is the interface information in the package specifications. A specification can be coded and compiled without its body. Then, stored subprograms that reference the package can be compiled as well. The package bodies are fully compiled only until the user is ready to complete the application.

Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the definition of the private subprogram so that only the package (not your application) is affected if the definition changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

Added Functionality

Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow data to be maintained across transactions without having to store it in the database.

Better Performance

When a packaged subprogram is called for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and so avoid unnecessary recompiling. For example, if

the definition of a packaged function is changed. Oracle need not recompile the calling subprograms because they do not depend on the package body.

17.3 Package Specification

The package specification contains public declarations. The scope of these declarations is local to the database schema and global to the package. So, the declared items are accessible from the application and from anywhere in the package.

The specification lists the package resources available to applications. All the information that the application needs to use the resources is in the specification. For example, the following declaration shows that the function named fac takes one argument of type INTEGER and returns a value of type INTEGER:

```
FUNCTION fac (n INTEGER) RETURN INTEGER; -- returns n!
```

That is the information that need the user to call the function. The underlying implementation of fac (whether it is iterative or recursive, for example). Be need not considered

Only subprograms and cursors have an underlying implementation or definition. So, if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary. Consider the following bodiless package:

```
-- a bodiless package
```

```
CREATE or replace PACKAGE trans_data AS
    TYPE TimeRec IS RECORD (
        minutes SMALLINT,
        hours   SMALLINT);
    TYPE TransRec IS RECORD (category VARCHAR2(1), account  INTEGER, amount
REAL,
        time      TimeRec);
    minimum_balance CONSTANT REAL := 10.00;
    number_processed  INTEGER;
    insufficient_funds EXCEPTION;
END trans_data;
```

The package trans_data needs no body because types, constants, variables, and exceptions do not have an underlying implementation. Such packages global variables-usable by subprograms and database triggers-that persist throughout a session to be defined

Referencing Package Contents

To reference the types, items, and subprograms declared within a package specification, the dot notation is to be used as follows:

```
package_name.type_name  
package_name.item_name  
package_name.subprogram_name
```

Package contents can be referenced from a database trigger, a stored subprogram, an Oracle Precompiler application, an OCI application, or an Oracle tool such as SQL*Plus.

For example, the packaged procedure hire_employee might be called from SQL*Plus, as follows:

```
EXECUTE emp.actions.hire_employee('TATE', 'CLERK', ...);
```

17.4 Package Body

The package body implements the package specification. That is, the package body contains the definition of every cursor and subprogram declared in the package specification. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specifications also appear in the package specification.

To match subprogram specifications and bodies, PL/SQL does a token-by-token comparison of their headers. So, except for white spaces. the headers must match word for word. Otherwise, PL/SQL raises an exception, as the following structure shows:

```
CREATE PACKAGE emp_actions1 AS  
  ...  
  PROCEDURE calc_bonus (date_hired emp.hiredate%TYPE, ...);  
END emp_actions;  
  
CREATE PACKAGE BODY emp_actions1 AS
```

```
...
PROCEDURE calc_bonus (date_hired DATE, ...) IS
-- parameter declaration raises an exception because 'DATE'
-- does not match 'emp.hiredate%TYPE' word for word
BEGIN
...
END calc_bonus;
END emp_actions;
```

The package body can also contain private declarations, which define types and items necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and items are inaccessible except from within the package body. Unlike a package specification, the declarative part of a package body can contain subprogram bodies.

Following the declarative part of a package body is the optional initialization part, which typically holds statements that initialize some of the variables previously declared in the package.

The initialization part of a package plays a minor role because, unlike subprograms, a package cannot be called or passed parameters. As a result, the initialization part of a package is run only once. That is the first time you reference the package is referenced.

Recall that if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary. However, the body can still be used to initialize items declared in the specification.

Examples

Consider the package named **sord** given below. The package specification declares the following subprograms:

- procedures ordproc, custchk and prodchk

After writing the package, applications that reference its types, call its subprograms, use its cursor, and raise its exception can be developed. When the package is created, it is stored in an Oracle database for general use.

```
CREATE OR REPLACE PACKAGE SORD AS
  PROCEDURE CUSTCHK(CNO NUMBER);
  PROCEDURE PRODCHK(PNO NUMBER);
  PROCEDURE ORDPROC(CNO NUMBER,PNO NUMBER,QTY NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY SORD AS
  FUNCTION CHKQTY(PNO NUMBER,QTY NUMBER) RETURN NUMBER IS
    STOCK NUMBER:=0;
  BEGIN
    SELECT STOC INTO STOCK FROM PRODUCT WHERE PCODE=PNO;
    IF STOCK > QTY THEN
      RETURN 1;
    ELSE
      RETURN -1;
    END IF;
  END;
  PROCEDURE PRODUPD(PNO NUMBER,QT NUMBER) IS
  BEGIN
    UPDATE PRODUCT SET STOC=STOC-QT WHERE PCODE=PNO;
  END;
  PROCEDURE CUSTCHK(CNO NUMBER) IS
    C_ID NUMBER;
  BEGIN
    SELECT CCODE INTO C_ID FROM CUSTOMER WHERE CCODE=CNO;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20000,'CCODE NOT FOUND');
  END;
  PROCEDURE PRODCHK(PNO IN NUMBER) AS
    P_ID NUMBER;
  BEGIN
    SELECT PCODE INTO P_ID FROM PRODUCT WHERE PCODE=PNO;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20000,'PCODE NOT FOUND');
  END;
  PROCEDURE ORDPROC(CNO IN NUMBER,PNO IN NUMBER,QTY IN NUMBER) AS
    RES NUMBER:=0;
    OC NUMBER:=0;
  BEGIN
    CUSTCHK(CNO);
    PRODCHK(PNO);
```

```
RES:=CHKQTY(PNO,QTY);
IF RES=-1 THEN
  RAISE_APPLICATION_ERROR(-20000,'SORRY NO STOCK');
ELSE

  SELECT MAX(OCODE)+1 INTO OC FROM ORDERS;
  INSERT INTO ORDERS(OCODE,CCODE,ODAT)
  VALUES(OC,CNO,SYSDATE);
  INSERT INTO ORDPRD VALUES(OC,PNO,QTY);
  PRODUPD(PNO,QTY);
END IF;

END;
END;
END;
```

The output of the program would be:

```
SQL> EXEC SORD.ORDPROC(1002,101,20);
begin SORD.ORDPROC(1002,101,20); end;

*
ERROR at line 1:
ORA-20000: SORRY NO STOCK
ORA-06512: at "HEMA.SORD", line 37
ORA-06512: at line 1
```

When a quantity that is more than the existing is entered, this raises an error. Look at the following:

```
SELECT STOC FROM PRODUCT WHERE PCODE=104
```

```
STOC
-----
716.1
```

```
EXEC SORD.ORDPROC(1002,104,20);
```

```
PL/SQL procedure successfully completed.
SELECT STOC FROM PRODUCT WHERE PCODE=104;
```

```
STOC
-----
696.1
```

the product stoc is updated.

Remember, the initialization part of a package is run just once, the first time you reference the package.

Every time the procedure ordproc is called, a record in the table orders is inserted if the product code and the customer code exist and there is sufficient qty on hand.

Private versus Public Items

Consider the following example:

```
CREATE OR REPLACE PACKAGE TESTVAR AS  
  X NUMBER := 0;  
END;
```

Look again at the package sord. The package body declares a variable named X, which is initialized to zero. Unlike items declared in the specification of SORD, items declared in the body are restricted to use within the package. Therefore, PL/SQL code outside the package cannot reference the function chkqty. Such items are termed private.

However, items declared in the specification of testvar such as the variable x are visible outside the package. Therefore, any PL/SQL code can reference the variable x. Such items are termed public.

When items are to be maintained they must be throughout a session or across transactions, placed in the declarative part of the package body. If the items are to be made public, in the package specification.

<p>Note: When a packaged subprogram remotely, the whole package is reinstated and its previous state is lost.</p>

17.5 Overloading Package

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when a subprogram has to accept parameters that have different datatypes.

In the next example, packaging of a mini calculator is shown. Assume that the operations to be performed are addition, subtraction , multiplication and division. This procedure is an example of OVERLOADING.

```
CREATE OR REPLACE PACKAGE MYCALCI AS
PROCEDURE ADDNUM(X IN NUMBER,Y IN NUMBER);
PROCEDURE SUBNUM(X IN NUMBER,Y IN NUMBER);
PROCEDURE MULTNUM(X IN NUMBER,Y IN NUMBER);
PROCEDURE DIVNUM(X IN NUMBER,Y IN NUMBER);
PROCEDURE ADDNUM(X IN NUMBER,Y IN NUMBER,X1 IN NUMBER,Y1 IN NUMBER);
PROCEDURE SUBNUM(X IN NUMBER,Y IN NUMBER,X1 IN NUMBER,Y1 IN NUMBER) ;
PROCEDURE MULTNUM(X IN NUMBER,Y IN NUMBER,X1 IN NUMBER,Y1 IN NUMBER);
PROCEDURE AVGNUM(X IN NUMBER,Y IN NUMBER,X1 IN NUMBER,Y1 IN NUMBER) ;
END;

CREATE OR REPLACE PACKAGE BODY MYCALCI IS
PROCEDURE ADDNUM(X IN NUMBER,Y IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X+Y;
DBMS_OUTPUT.PUT_LINE('THE TOTAL IS '||RES);
END;

PROCEDURE SUBNUM(X IN NUMBER,Y IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X-Y;
DBMS_OUTPUT.PUT_LINE('THE DIFFERENCE IS '||RES);
END;

PROCEDURE MULTNUM(X IN NUMBER,Y IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X*Y;
DBMS_OUTPUT.PUT_LINE('THE RESULT IS '||RES);
END;

PROCEDURE DIVNUM(X IN NUMBER,Y IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X/Y;
DBMS_OUTPUT.PUT_LINE('THE RESULT IS '||RES);
END;

PROCEDURE ADDNUM(X IN NUMBER,Y IN NUMBER, X1 IN NUMBER,Y1 IN NUMBER) IS
RES NUMBER:=0;
```

```
BEGIN
RES:=X+Y+X1+Y1;

DBMS_OUTPUT.PUT_LINE('THE TOTAL IS '||RES);
END;

PROCEDURE SUBNUM(X IN NUMBER,Y IN NUMBER, X1 IN NUMBER,Y1 IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X-Y-X1-Y1;
DBMS_OUTPUT.PUT_LINE('THE DIFFERENCE IS '||RES);
END;

PROCEDURE MULTNUM(X IN NUMBER,Y IN NUMBER, X1 IN NUMBER,Y1 IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=X*Y*X1*Y1;
DBMS_OUTPUT.PUT_LINE('THE RESULT OF FOUR NUMBERS ARE '||RES);
END;

PROCEDURE AVGNUM(X IN NUMBER,Y IN NUMBER, X1 IN NUMBER,Y1 IN NUMBER) IS
RES NUMBER:=0;
BEGIN
RES:=(X+Y+X1+Y1)/4;
DBMS_OUTPUT.PUT_LINE('THE AVERAGE OF THE NUMBERS ARE '||RES);
END;
END;
/
The output would look like:
```

```
SQL> EXEC MYCALCI.SUBNUM(34,2);
THE DIFFERENCE IS 32
```

```
PL/SQL procedure successfully completed.
```

```
SQL> EXEC MYCALCI.SUBNUM(100,1,2,3);
THE DIFFERENCE IS 94
```

```
PL/SQL procedure successfully completed.
```

```
SQL> EXEC MYCALCI.AVGNUM(12,20,30,40);
THE AVERAGE OF THE NUMBERS ARE 25.5
```

```
PL/SQL procedure successfully completed.
```

The first procedure accepts addnum accepts two numbers, while the second procedure accepts four numbers.. Yet, each procedure handles the data appropriately.

Package STANDARD

A package named STANDARD defines the PL/SQL environment. The package specification globally declares types, exceptions, and subprograms, which are available automatically to every PL/SQL program. For example, package STANDARD declares the following built-in function named ABS, which returns the absolute value of its argument:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package STANDARD are directly visible to applications. So, can be called ABS from a database trigger, a stored subprogram, an Oracle Precompiler application, an OCI application, and various Oracle tools including Oracle Forms, Oracle Reports, and SQL*Plus.

If ABS is redeclared in a PL/SQL program, the local declaration overrides the global declaration. However, the built-in function can still be called by using dot notation, as follows:

```
... STANDARD.ABS(x) ...
```

Most built-in functions are overloaded. For example, package STANDARD contains the following declarations:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves a call to TO_CHAR by matching the number and datatypes of the formal and actual parameters.

Product-specific Packages

Oracle and various Oracle tools are supplied with product-specific packages that help PL/SQL-based applications to be built. For example, Oracle is supplied with the packages

DBMS_STANDARD, DBMS_OUTPUT, DBMS_PIPE, UTL_FILE, UTL_HTTP, DBMS_SQL, DBMS_ALERT and others.

DBMS_STANDARD

Package DBMS_STANDARD provides language facilities that help the applications to interact with Oracle. For instance, a procedure named raise_application_error allows

user-defined error messages to be issued. This way errors can be reported to an application and the returning of unhandled exceptions can be avoided.

DBMS_OUTPUT

Package DBMS_OUTPUT enables the output to be displayed from PL/SQL blocks and subprograms, which makes it easier to test and debug them. The put_line procedure outputs information to a buffer in the SGA. The information is displayed by calling the procedure get_line or by setting SERVEROUTPUT ON in SQL*Plus or Enterprise Manager.

For example, suppose the following stored procedure is created.

```
CREATE PROCEDURE calc_payroll (payroll IN OUT REAL) AS
  CURSOR c1 IS SELECT sal,comm FROM emp;

BEGIN
  payroll := 0;

  FOR clrec IN c1 LOOP
    clrec.comm := NVL(clrec.comm, 0);
    payroll := payroll + clrec.sal + clrec.comm;
  END LOOP;
  /* Display debug info. */
  dbms_output.put_line('payroll: ' || TO_CHAR(payroll));
END calc_payroll;
```

When the following commands are issued. SQL*Plus displays the value of payroll calculated by the procedure:

```
SET SERVEROUTPUT ON
VARIABLE num NUMBER
EXECUTE calc_payroll(:num)
```

This would display

```
payroll: 33525
```

DBMS_PIPE

Package DBMS_PIPE allows different sessions to communicate over named pipes. (A pipe is an area of memory used by one process to pass information to another.) The procedures pack_message and send_message can be used to pack a message into a pipe, then send it to another session in the same instance.

At the other end of the pipe, the procedures receive_message and unpack_message can be used to receive and unpack (read) the message. Named pipes are useful in many ways.

For example, you can write routines in C that allow external servers to collect information, then send it through pipes to procedures stored in an Oracle database.

UTL_FILE

Package UTL_FILE allows PL/SQL programs to read and write operating system (OS) text files. It provides a restricted version of standard OS stream file I/O, including open, put, get, and close operations.

When a text file is to be read or written is called, you call the function fopen, which returns a file handle for use in subsequent procedure calls. For example, the procedure put_line writes a text string and line terminator to an open file. The procedure get_line reads a line of text from an open file into an output buffer.

PL/SQL file I/O is available on both the client and server sides. However, on the server side, file access is restricted to those directories explicitly listed in the accessible directories list, which is stored in the Oracle initialization file.

UTL_HTTP

Package UTL_HTTP allows PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can be used to retrieve data from the internet, or to call Oracle Web Server cartridges. The package has two entry points, each of which accepts a URL (universal resource

locator) string, contacts the specified site, and returns the requested data, which is usually in hypertext markup language (HTML) format.

DBMS_SQL

Package DBMS_SQL allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time.

DBMS_ALERT

Package DBMS_ALERT database triggers to be used to alert an application when specific database values change. The alerts are transaction based and asynchronous (that is, they operate independently of any timing mechanism). For example, a company might use this package to update the value of its investment portfolio as new stock and bond quotes arrive.

Guidelines

When writing packages, they should be as general as possible so that they can be reused in future applications. Avoid writing packages that duplicate some feature already provided by Oracle.

Package specifications reflect the design of the application. So, define them before the package bodies. Place in a specification only the types, items, and subprograms that must be visible to users of the package. That way, other developers cannot misuse the package by basing their code on irrelevant implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package specification. Changes to a package body do not require Oracle to recompile dependent procedures. However, changes to a package specification require Oracle to recompile every stored subprogram that references the package.

17.6 Short Summary

- ➲ Packages are a collection of procedures, functions, variables and exceptions. They define the application. Packages are mainly used to increase the performance since any call to a packaged object immediately loads the entire package to the memory.
- ➲ Packages provide encapsulation by way of writing local procedures and functions inside the body of the package which is not visible to application.
- ➲ They also support Polymorphism by the way of Overloading subprograms which differ in arguments or datatypes.
- ➲ Different built-in packages are DBMS_OUTPUT.PUT_LINE , DBMS_SQL, DBMS_PIPE and so on.

17.7 Brain Storm

1. Packages is a collection of _____
2. The two parts of packages are _____ & _____
3. Advantages of packages are _____, _____ & _____
4. What is the use of restrict_references?
5. Can we access a variable declared within a package body outside a package?
6. Name two available packages
7. What is DBMS_SQL package used for?
8. How will you refer a subprogram within a package?

©...QR

Lecture 18

Triggers

Objectives

After completing this lesson, you should be able to do the following

- ☒ Describe database triggers and their use
- ☒ Create database triggers
- ☒ Describe database trigger firing rules
- ☒ Remove database triggers

Coverage Plan

Lecture 18

- 18.1 Snap shot
- 18.2 Triggers
- 18.3 Creating triggers
- 18.4 Cascading triggers
- 18.5 Mutating and constraining tables
- 18.6 Enabling and disabling triggers
- 18.7 Short summary
- 18.8 Brain Storm

18.1 Snap Shot

This chapter shows how to enforce complex constraints using the concept called Triggers. The trigger includes PL/SQL statements that get automatically invoked whenever any DML operations are performed on a table.

- Understanding Triggers
- Types of Triggers
- Designing Triggers
- Writing Trigger body
- Using INSTEAD OF triggers
- Disabling and Enabling triggers

18.2 Triggers

A database trigger is a stored PL/SQL block that is associated with a table. Triggers are automatically executed when a specified SQL statement is issued against the table. Triggers are mainly used for the following purposes:

- To automatically generate values
- To provide auditing.
- To prevent invalid transactions.
- To check for complex integrity constraints that cannot be given in Constraints.
- To maintain replicate tables.

Triggers contain PL/SQL constructs and they can be fired only for DML Statements like INSERT, UPDATE and DELETE. Triggers are event based.

Parts of a Trigger

A Trigger contains 3 parts. They are

- Triggering event or statement
- Trigger Restriction
- Trigger Action

Triggering event or Statement

It is a SQL statement that causes the trigger to be fired. The valid statements are the DML statements – Insert, Update or Delete. Additionally, trigger can be made to be fired when a particular column is updated. Also it can contain multiple DML statements.

Triggering Restriction

This is a Boolean expression which must be TRUE for the trigger to be fired.

Trigger Action

This is a procedure containing SQL and PL/SQL statements to be executed when a triggering statement is issued and also the triggering restriction is true.

Types of Triggers

Triggers are classified into different types depending on when the trigger is to be fired.

They are:

- BEFORE
- AFTER
- INSTEAD OF (dealt later in this chapter)

BEFORE / AFTER

The BEFORE option of a trigger is used to specify when the trigger must be fired. If the option BEFORE is chosen, the trigger is fired before the triggering statement is executed. In the case of AFTER, the trigger is fired after executing the statement.

When to Use Before and After

BEFORE

Before triggers are used when the triggering action determines whether the triggering statement must be allowed to be completed or not. This eliminates un-necessary processing of the triggering statement. Before triggers can also be used to derive specific column values before completing an Insert or Update statement.

AFTER

This trigger is executed after the trigger statement is issued. It is used when the trigger statement has to be completed before the trigger action. If a Before Trigger is already present, an After trigger can be used to perform different actions on the same statement.

Based on how many records are to be affected by the trigger, triggers can be classified as;

- Row Level
- Statement Level

Row Level Triggers

These types of triggers are fired for each row that is affected by the triggering statement. For example, UPDATE statement for multiple rows of a table. In this case, the ROW level triggers are executed for every row that is affected by the UPDATE statement.

Statement Level Triggers

This trigger is fired once on behalf of the triggering statement regardless of the number of rows in the table that the triggering statement affects.

Based on the combinations of the above, there are twelve types of triggers that can be written on a table.

Statement Level	Row Level
Before Insert	Before Insert
Before Update	Before Update
Before Delete	Before Delete

After Insert	After Insert
After Update	After Update
After Delete	After Delete

18.3 Creating Triggers

Triggers are created using CREATE TRIGGER statement. The name of the trigger must be unique with respect to other triggers in the same schema. The syntax for creating the trigger is:

Syntax:

```
CREATE OR REPLACE TRIGGER <triggername> [ BEFORE/AFTER ]
[ INSERT/UPDATE/DELETE ] ON <tablename> [ FOR EACH ROW ] [ WHEN <condition> ]
```

The following example creates a simple trigger.

Example

```
CREATE OR REPLACE TRIGGER instremp BEFORE INSERT ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE('Inserting Record');
END;
/
```

The trigger gets created.

When an Insert statement is issued, the output will look like:

```
INSERT INTO emp (empno,ename,deptno) VALUES (1,'Giri',10);
```

This would display

Inserting Record

1 row created.

In the above example, the triggering statement is

BEFORE INSERT ON emp

Hence the message 'Inseting record is displayed before the row is created

There is no trigger restriction the body or action of the trigger starts from the Begin statement.

Example

This trigger displays the total number of records available in the table if insertion is done.

```
CREATE OR REPLACE TRIGGER INSTR1 BEFORE INSERT ON EMP
DECLARE
X NUMBER;
BEGIN
SELECT COUNT(*) INTO X FROM EMP;
DBMS_OUTPUT.PUT_LINE(X+1 || ' NUMBER OF RECORDS ARE AVAILABLE');
END;
```

While insertion is done, the output will look like:

```
insert into emp(empno,ename,deptno) values(12,'hari',30);
15 number of records are available
```

Before insertion the total records were 14. After insertion the total records increases to 15. Note here, there is an explicit DECLARE statement and no IS/AS is specified. If a trigger contains any variable, DECLARE must be specified.

The trigger execution is performed in the following ways:

1. All the Before statement triggers are executed.
2. The Before Row Level triggers are executed.
3. After Row Level triggers are executed.
4. After Statement triggers are executed that apply to the statement.

Accessing Column Values

When a trigger action or the trigger body contains statements that require access to the table values or for checking the new value with the old value, two correlation names are used: -

:new and :old. :new refers to the new values entered in the trigger statement and :old refers to the old existing value in the table. These are also called as pseudorecords since they do not contain any permanent record. Also these can be given only for **row-level** triggers. The following table illustrates the values held by these with respect to the DML operations.

Statement	:new	:old
Insert	New values entered in the Insert command	<u>Null</u>
Update	New values entered in the Update statement	Old values available in the table
Delete	null	Old values in the table

They are available for both Before and After triggers.

Example

This example uses the :new and displays the newly inserted record.

```
CREATE OR REPLACE TRIGGER INSTR2 AFTER INSERT ON DEPT FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE(:NEW.DEPTNO || ' ' || :NEW.DNAME || ' ' || :NEW.LOC);
END;
```

The output will look like:

```
INSERT INTO DEPT VALUES(12,'SALES','CHENNAI');
12  SALES  CHENNAI
```

1 row created.

The following example checks for the deptno availability in the dept table. If the corresponding deptno is not available in the table it displays a message as shown in the following example.

```
CREATE OR REPLACE TRIGGER CHKDEPTNO BEFORE UPDATE OF DEPTNO ON EMP FOR EACH
ROW
DECLARE
  OLD_DNO NUMBER;
```

```
BEGIN  
  SELECT DEPTNO INTO OLD_DNO FROM DEPT WHERE DEPTNO=:NEW.DEPTNO;  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    DBMS_OUTPUT.PUT_LINE('DEPTNO NOT AVAILABLE');  
END;
```

The output of the program

```
UPDATE EMP SET DEPTNO=34 WHERE EMPNO=7900;
```

```
DEPTNO NOT AVAILABLE
```

1 row updated.

Using Referencing Option

The Referencing option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named.

Using Conditional Predicates

If more than one type of DML operation can fire a trigger, all the operations can be placed inside a single trigger each differentiated by what are called conditional predicates.

For referring to Insert operation, INSERTING is used, for Delete operation, DELETING and for update operation, UPDATING predicates are used. These are Boolean values which return TRUE only when the statement that fired a trigger is any one of DML statements. The syntax is given below:

```
IF INSERTING THEN.... .  
  
ELSIF UPDATING THEN .... .  
  
ELSIF DELETING THEN .... .  
  
END IF;
```

Example

```
CREATE OR REPLACE TRIGGER prdtrig AFTER INSERT OR UPDATE OR DELETE ON
product
```

```
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting operation');
    ELSIF UPDATING THEN
        DBMS_OUTPUT.PUT_LINE('Updating operation');
    ELSIF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting operation');
    END IF;
END;
```

Using Raise_Application_Error

As we have already seen in Exceptions, Raise_Application_Error terminates the operation abruptly and comes to the SQL prompt. Now, consider a situation such that, a record gets inserted and the trigger based on that causes another insert on another table.

For example consider the following trigger. This trigger checks for the job and the corresponding salary in the table. If there is any violation, it reports to the user.

```
CREATE OR REPLACE TRIGGER CHKSALJOB AFTER INSERT ON EMP FOR EACH ROW
BEGIN
    IF NOT(:NEW.JOB='MANAGER' AND (:NEW.SAL > 4000 AND :NEW.SAL < 6000))
    THEN
        DBMS_OUTPUT.PUT_LINE('Limit for Manager is between 4000 and
6000');
    END IF;
END;
```

The trigger gets created. When a record is inserted onto the table EMP which violates the condition the message is displayed as

```
INSERT INTO emp (empno,ename,deptno,mgr,job,sal) VALUES
(1234,'ARUN',20,7902,'MANAGER',8000);
Limit for Manager is between 4000 and 6000
```

1 row created.

Even though the message is displayed, still this does not stop the record from being inserted. Now let us try to raise this as an exception with RAISE statement and the code is re-written as follows:

```
CREATE OR REPLACE TRIGGER CHKSALJOB AFTER INSERT ON EMP FOR EACH ROW
DECLARE
    raise_ex EXCEPTION;
BEGIN
    IF NOT(:NEW.JOB='MANAGER' AND (:NEW.SAL > 4000 AND :NEW.SAL <6000)) THEN
        RAISE raise_ex;
    END IF;
EXCEPTION
    WHEN RAISE_EX THEN
        DBMS_OUTPUT.PUT_LINE('Limit for Manager is between 4000 and
6000');
END;
```

The trigger is created.

After this, when the insert statement is issued,

```
INSERT INTO emp(empno,ename,deptno,mgr,job,sal) VALUES
(1234,'ARUN',20,7902,'MANAGER',8000);
Limit for Manager is between 4000 and 6000
```

1 row created.

Even this does not stop from inserting the record onto the table. RAISE_APPLICATION_ERROR is used to perform this operation. The following example does not insert record into the table if the condition is not satisfied.

Example

```
CREATE OR REPLACE TRIGGER CHKSALJOB AFTER INSERT ON EMP FOR EACH ROW
BEGIN
```

```
IF NOT(:NEW.JOB='MANAGER' AND (:NEW.SAL > 4000 AND :NEW.SAL <6000))
THEN
    RAISE_APPLICATION_ERROR(-20000,'MANAGER SALARY CANNOT EXCEED 8000 AND
CANNOT BE LESS THAN 4000');
END IF;
END;
```

The output when record is inserted is :

```
INSERT INTO emp(empno,ename,deptno,mgr,job,sal) VALUES
(1234,'ARUN',20,7902,'MANAGER',8000);
INSERT INTO emp(empno,ename,deptno,mgr,job,sal) VALUES
(1234,'ARUN',20,7902,'MANAGER',8000)
*
ERROR at line 1:
ORA-20000: Manager Salary cannot exceed 8000 and cannot be less than 4000
ORA-06512: at "HEMA.CHKSALJOB", line 4
ORA-04088: error during execution of trigger 'HEMA.CHKSALJOB'
```

18.4 Cascading Triggers

Triggers are said to be cascading whenever the body of the trigger causes another trigger to be fired. Consider an example such that whenever a product is deleted in the product table the corresponding record in the ordprd table must be deleted. The following trigger is created for the ordprd table:

```
CREATE OR REPLACE TRIGGER ORDPRDDEL AFTER DELETE ON ORDPRD
DECLARE
    X NUMBER:=0;
BEGIN
    SELECT COUNT(*) INTO X FROM ORDPRD;
    DBMS_OUTPUT.PUT_LINE('RECORD DELETED');
    DBMS_OUTPUT.PUT_LINE('RECORDS AVAILABLE ARE '||X);
END;

/ CREATE OR REPLACE TRIGGER PRODDEL AFTER DELETE ON PRODUCT
FOR EACH ROW
BEGIN
    DELETE FROM ORDPRD WHERE PCODE=:OLD.PCODE;
END;
```

When a record is deleted,

```
DELETE FROM product WHERE pcode=102;  
record deleted  
records available are 18
```

18. 5 Mutating and Constraining Tables

A mutating table is a table that is currently being modified by an UPDATE, DELETE Or INSERT statement, or a table that might need to be updated. In other words, the trigger body or the triggering action must not contain any operations on the same table on which the trigger is written. They are not considered to be mutating in the case of Statement Triggers.

A constraining table is a table that a triggering statement might need to read either directly for a SQL statement or indirectly for a declarative statement.

Example

```
CREATE OR REPLACE TRIGGER INSPROD AFTER INSERT ON PRODUCT FOR EACH ROW  
DECLARE  
    TOTREC NUMBER;  
BEGIN  
    SELECT COUNT(*) INTO TOTREC FROM PRODUCT;  
    DBMS_OUTPUT.PUT_LINE('THE TOTAL RECORDS ARE ' || TOTREC);  
END;
```

Trigger gets created.

When a record is inserted,

```
INSERT INTO PRODUCT (pcode, pname) VALUES(189, 'books');  
INSERT INTO PRODUCT(pcode, pname) VALUES(189, 'books')  
*  
ERROR at line 1:  
ORA-04091: table HEMA.PRODUCT is mutating, trigger/function may not see it  
ORA-06512: at "HEMA.INSPROD", line 4  
ORA-04088: error during execution of trigger 'HEMA.INSPROD'
```

In order to avoid this problem, if the total number of records are to be displayed, use the trigger along with a package variable. The following example is illustrative of this:

Before writing a trigger, write a package with specification alone:

```
CREATE OR REPLACE PACKAGE TEST1 AS  
N NUMBER:=0;  
END;
```

Thereafter write a Before Trigger as shown below:

```
CREATE OR REPLACE TRIGGER INSPROD1 BEFORE INSERT ON PRODUCT FOR EACH ROW  
BEGIN  
SELECT COUNT(*) INTO TEST1.N FROM PRODUCT;  
END;
```

Also write a After Trigger which has the following statements:

```
CREATE OR REPLACE TRIGGER INSPROD AFTER INSERT ON PRODUCT FOR EACH ROW  
DECLARE  
  
TOTREC NUMBER;  
BEGIN  
TOTREC:=TEST1.N+1;  
DBMS_OUTPUT.PUT_LINE('THE TOTAL RECORDS ARE '||TOTREC);  
END;
```

Now, if the insert statement is issued, it displays

```
INSERT INTO PRODUCT(PCODE,PNAME) VALUES(189,'BOOKS');  
THE TOTAL RECORDS ARE 12
```

1 row created.

Using Instead of Triggers

Triggers can be written only for tables. But the Instead of Trigger is intended to be written for views. The Instead of Option in the Create Trigger statement is an alternative way of modifying views that cannot be modified by the Insert, Update or Delete statements. Recall the Join View. In the case of Join View, updating or inserting can be done only on the key preserved table. This trigger is called Instead of Trigger because unlike other triggers, the

trigger body is executed instead of executing the triggering statement. The trigger performs the operations directly on the underlying tables. They are activated for each row.

The INSTEAD OF Option

INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement. The trigger performs update, insert, or delete operations directly on the underlying tables.

Users write normal UPDATE, INSERT, and DELETE statements against the view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place.

By default, INSTEAD OF triggers are activated for each row.

Views That Are Not Modifiable

A view cannot be modified by UPDATE, INSERT, or DELETE statements if the view query contains any of the following constructs:

- Set operators
- Group functions
- GROUP BY, CONNECT BY, or START WITH clauses
- DISTINCT operator
- Joins (a subset of join views are updatable)

If a view contains pseudocolumns or expressions, you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

Example of an INSTEAD OF Trigger

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER_INFO view.

```
CREATE OR REPLACE VIEW manager_info AS
  SELECT e.ename, e.empno, d.dname, d.deptno
    FROM   emp e, dept d
   WHERE  e.deptno = d.deptno

CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info FOR EACH ROW
BEGIN
  INSERT INTO emp(EMPNO,ENAME) VALUES(:new.empno, :new.Ename);
  INSERT INTO dept(DEPTNO,DNAME) VALUES(:new.deptno, :new.dname);
END;
```

Here, instead of executing the statement for the view, the body of the view is executed thereby inserting records onto the respective tables.

18.6 Enabling Triggers and Disabling triggers

By default, all triggers are automatically enabled when They are created. However, They can be disabled. Once a task is completed, the trigger can be re-enabled. Enabling and Disabling triggers are performed using enable and disable keywords.

Syntax:

```
ALTER TRIGGER <triggername> ENABLE|DISABLE;
```

Example:

```
ALTER TRIGGER empdel DISABLE;
```

For a particular table, all triggers can be enabled using a single command.

```
ALTER TABLE <tablename> DISABLE ALL TRIGGERS;
```

Dropping triggers

Triggers are dropped using the Drop Trigger Command. The syntax follows:

Syntax:

```
DROP TRIGGER <triggername>;
```

Example:

```
DROP TRIGGER chkprod;
```

Restrictions on Triggers

Even-though there are various advantages that a trigger can provide there are still some restrictions in using Triggers.

Trigger body cannot contain ROLLBACK or COMMIT statements. Trigger body must contain keyword DECLARE for declarative section unlike Subprograms. Trigger specification must not contain any IS/ AS keywords.

18.7 Short Summary

Triggers are set of subprograms that are automatically executed whenever a DML statement is issued on a particular table. They are used to automatically generate values, prevent invalid transactions and so on.

Triggers contain 3 parts:

- Triggering Event
- Restriction
- Action

There are 12 types of triggers.

Triggers are created using CREATE TRIGGER statement.

Instead of Triggers are used for Views which contain Joins.

Triggers can be enabled, disabled or dropped.

18.8 Brain Storm

1. The advantages of triggers are _____, _____ & _____
2. How triggers are executed?
3. How will you disable a trigger?

4. What are the three triggering statements?
5. Views that cannot be modified by insert, update, delete statements can be modified by _____
6. Mutating table is_____
7. How will you enable the entire trigger for a given table?
8. The two correlation names for row level trigger is_____ and _____
9. How Row lever trigger can be created?
10. Alter trigger <trigger name> command is used to_____

ANSWER

Lecture 19

OOPS and Object Concepts

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about Oops and Object Concepts
- ☒ Describe Why Objects?
- ☒ Discuss about Oracle supported Data Types
- ☒ Describe about Collection Types
- ☒ Describe Object views
- ☒ Discuss about member functions
- ☒ Discuss about Using order method

Coverage Plan

Lecture 19

- 19.1 Snap shot
- 19.2 Why objects?
- 19.3 Object Oriented
- 19.4 Oracle supported data types
- 19.5 Collection types
- 19.6 Object views
- 19.7 Member functions
- 19.8 Using order method
- 19.9 Short summary
- 19.10 Brain Storm

19.1 Snap Shot

This chapter deals with Object Oriented Programming concepts and creating abstract datatypes and using the same. This chapter talks about the collection objects, creating Object tables , using Object views and creating Member Functions.

- OOP concepts
- Creating Abstract Datatypes
- Embedding the object types to tables
- Using Constructor methods
- Understanding Varrays
- Understanding Nested Tables
- Using REF and DEREF constructs
- Using Member Functions
- Understanding object views

19.2 Why Objects

In general, datatypes like number, varchar2 and date are used. But representation of objects as real world objects is possible in Object Relational Database Management Systems (ORDBMS) and Object Related Database Management Systems (OODBMS). In Oracle 8 the objects are referred to as real world objects by incorporating the features of ORDBMS and OODBMS.

19.3 Object – Oriented Concepts

The important concepts of the object-oriented methodology are discussed below:

Classes

CLASS refers to the definition of an object. In simpler terms, it is the blue print for an object.

It is a standard representation consisting of data members and their functions. Functions and variables that are part of the class declaration of a class definition are considered to be public members of that class.

Abstract datatypes model classes of data within the database. These datatypes can share the same structure. For example, if you want to move a car, you can either move the car (an object) or you can break it into its components (tires, steering column etc.), move the components individually, and then perform a join in the new location. Treating the car as an object is a more natural way of relating to it and simplifies your interaction with it.

Methods & Messages

A method is the incorporation of a specific behavior assigned to an object. A method is a function of a particular class. They implement the operations to be performed on a real world object and the output of the method is message. Thus a message is essentially an executed function belonging to a class member. The method is the means by which objects communicate with one another.

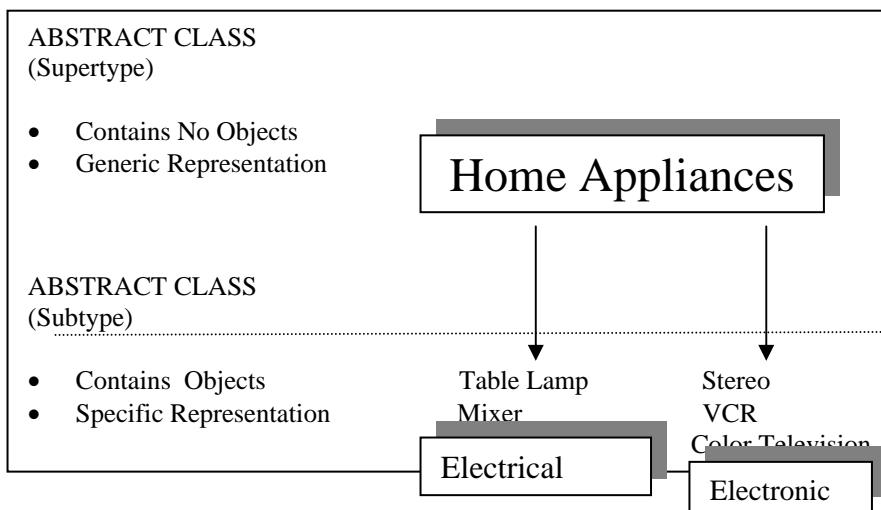
Abstraction

Abstract datatypes are a part of the Object Oriented concept called abstraction. It is the conceptual existence of classes within a database. The structural existence of the abstract types is sufficient. The methods attached to each level of abstraction may be called from the higher levels of abstraction.

Consider an example where HOME APPLIANCES refers to a class of data. "Electronic" is a subclass of the abstract class Home Appliances. The relationship between the new abstract class "Home Appliances" and the subclass "Electronic" would be known as a supertype to subtype relationship, in the logical model of a relational database. The class "Home Appliances" is an abstract class, it has no actual instances of objects in it. It exists to describe general features that pertain to all Home Appliances, not only Electronic.

Inheritance

Inheritance is the ability of one class to inherit the properties of its ancestor. Inheritance allows an object to inherit a certain set of attributes from another object while allowing the addition of specific features.



In addition to inheriting data structures, classes can inherit the behavior of their “parent” classes – a concept called implementation inheritance. That is, you can execute a method against an object even if the object does not use that method – provided the method is defined for one of the classes that are the “parents” of the current object.

During the creation of objects, they inherit the structures of the data elements they are descended from. For example, if “CAR” were a class of data, then “Ford”, a type of car, would inherit the structural definitions of “CAR”. From a data perspective, nested abstract datatypes inherit the representations of their “parents”.

Encapsulation

ORACLE’s Object Oriented implementation is based on a relational system. In a relational system the means of accessing data is never fully limited. Thus, data within the ORACLE cannot usually be completely encapsulated. Encapsulation can be achieved by limiting access to tables and forcing all access to be accomplished via procedures and functions, but this prevents the realization of the true power of a relational database.

Polymorphism

Polymorphism enables different objects to have methods by the same name that accomplish similar tasks, but in different ways. Oracle incorporates this feature in Packages by incorporating the concept of Method Overloading.

19.4 Oracle Supported Data Types

The Oracle8 database provides support for two distinct types of data: -

- BUILT-IN DATA TYPES
- USER DEFINED DATA TYPES

Built-In Datatypes supported in Oracle have already been discussed.

Object Option

The objects option makes Oracle an object-relational database management system (ORDBMS), which means that users can define additional kinds of data—specifying both the structure of the data and the ways of operating on it—and use these types within the relational model. This approach adds value to the data stored in a database.

Oracle with the objects option stores structured business data in its natural form and allows applications to retrieve it that way. For that reason it works efficiently with applications developed using object-oriented programming techniques.

Oracle's support for user-defined datatypes makes it easier for application developers to work with complex data like images, audio, and video.

Abstract Data Types (User-Defined Types)

User-defined datatypes use Oracle built-in datatypes and other user-defined datatypes as the building blocks to model the structure and behavior of data in applications. User-Defined Datatypes can be classified into two types based on the number of data it can hold. The two types are

- Simple Object Types
- Collection Object Types

Simple Object Types

They are implemented using Abstract Datatypes. These datatypes define the structure of a datatype, which can be embedded as another datatype for a column.

19.5 Collection Object types

Collection Datatypes are similar to arrays and they can hold multiple values inside a single column.

Object Types

Object types are abstractions of the real-world entities. It consists of three components.

- Name which is used to identify the object.
- Attributes which are the characteristics of an object.
- Methods or behaviors which are functions or procedures that are written and stored in the database.

Object types are just templates of an object. A data unit that matches the template is called an object. Object types can be:

- Single Row Object type
- Multiple – Row Object type

Single Row Object Type

Single Row Object types are object types that are defined for specific columns and can hold singular value only like other datatypes. They are created as objects, which are further embedded as a datatype to a column.

Example

```
CREATE OR REPLACE TYPE user_date AS OBJECT
(
    day    number(2),
    month  number(2),
    year   number(2)
```

```
) ;

CREATE OR REPLACE TYPE user_name AS OBJECT
(
    fname varchar2(20),
    mname varchar2(20),
    lname varchar2(20)
);
CREATE TABLE employee_data
(
    empno      number(3),
    ename      user_name,
    dob        user_date,
    salary     number(9,2)
);
```

The employee_data table is a relational table with an object type defining one of its columns. Objects that occupy columns of the relational table are called COLUMN OBJECTS (EMBEDDED OBJECTS).

Inserting Values

Every object type has a system-defined constructor method, that is, a method that makes a new object according to the object type's specification. The name of the constructor method is the name of the object type. The constructor method is a function. It returns the new object as its value. It is used to initialize the attributes with values.

Example

Data inserted into the table EMPLOYEE_data using CONSTRUCTOR METHOD

```
INSERT INTO employee _data VALUES ( 101, user_name ('Sachin', 'Ramesh','Tendulkar')
) , user_date(04,05,98),10000)) ;
```

The possible queries that can be performed are :

To retrieve all the records from the table

```
SELECT * FROM employee_data;
SELECT e.ename.fname, e.ename.lname FROM employee_data e;
SELECT e.empno, e.dob.day FROM employee_data e;
```

Collection Datatypes

Collection datatypes can hold multiple records. They are of two types:

- VARRAY (Varying Array)
- NESTED TABLE (Multiple records)

Varray

An array is an ordered set of data elements. All elements of a given array are of the same datatype. Each element has an index, which is a number corresponding to the element's position in the array. The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called VARRAYs. The maximum size must be specified while creating varrays.

Declaring a Varray does not occupy space. It defines a type, which can be used as

- The datatype of a column of a relational table.
- An object type attribute.
- A PL/SQL variable, parameter, or function return type.

An array object is stored in line, that is, in the same tablespace as the other data in its row. The following example illustrates this concept.

Example

```
CREATE OR REPLACE TYPE price_list AS VARRAY(5) OF number(9) ;  
  
CREATE TABLE prods  
(  
    pno    number,  
    rate   price_list  
) ;
```

Inserting Values

Inserting records are always done only using Constructor method.

```
INSERT INTO prods VALUES  
(1, price_list (12,34,56,78));  
  
INSERT INTO prods VALUES  
(2, price_list (12,34,56));
```

Here the maximum values that can be specified are only 5. The number of values that the column can hold must be less than or equal to 5. On exceeding this maximum value, the insertion leads to the following error:

```
INSERT INTO prods VALUES  
(4, price_list (12,3,5,6,71,90));
```

ERROR at line 2:
ORA-22909: exceeded maximum VARRAY limit

Querying the data from varrays:

Data can be selected as a complete list from the table containing varrays. Individual manipulation cannot be done in SQL. Also, while performing updation, even if one single value has to be updated the whole collection has to be specified.

Example given below shows how to select and update records.

While issuing SELECT statement,

```
SELECT * FROM prods;
```

Displays,

```
PNO  
-----  
RATE  
-----
```

```
PRICE_LIST(12, 34, 56)
```

```
1
```

```
PRICE_LIST(12, 34, 56, 78)
```

Selecting can be done only as a collection unit.

Nested Tables

A nested table is like an object table without the object identifiers. It has a single column, and the type of that column is a built-in type or an object type. If it is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use

- The datatype of a column of a relational table.
- An object type attribute.

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table. They are also called Multiple Row Object Type. There are no restrictions on the number of records to be created.

Example

The following example creates a nested table and embeds the type with the relational table.

```
MULTIPLE ROW OBJECT TYPE(NESTED TABLE)
```

```
CREATE OR REPLACE TYPE stud_type AS
OBJECT
(
    studno          number,
    studname        char(20),
    sex             char(1)
) ;
```

```
CREATE OR REPLACE TYPE stud_nt AS TABLE OF stud_type ;
CREATE TABLE faculty
(
    factno          number,
    name            varchar2(30),
    students        STUD_NT
) NESTED TABLE students STORE AS stud_nt_tab ;
```

Here , NESTED TABLE nested_item STORE AS storage_table

specifies storage_table as the name of the storage table in which the rows of all nested_item values reside. This clause must be included when creating a table with columns or column attributes whose type is a nested table. The nested_item is the name of a column or a column-qualified attribute whose type is a nested table.

The storage_table is the name of the storage table. The storage table is created in the same schema and the same tablespace as the parent table.

Creating a table with columns of type TABLE implicitly creates a storage table for each nested table column. The storage table is created in the same tablespace as its parent table (using the default storage characteristics) and stores the nested table values of the column for which it is created. You cannot query or perform DML statements on the storage table directly, but you can modify the nested table column storage characteristics by using the name of storage table in an ALTER TABLE statement .

```
INSERT INTO faculty VALUES
( 1, 'Ramesh' ,
  stud_nt (
    stud_type( 10, 'Rajesh', 'M' ) ,
    stud_type( 11, 'Suresh', 'M' ),
    stud_type( 12, 'Uma', 'F' )
  )
) ;
```

Insertion into the table is done using the abstract data type called stud_type. Similarly multiple records can be inserted into the table.

While querying, individual columns can be selected.

If,

```
SELECT * FROM faculty is issued ,
```

The display will be,

FACTNO NAME

STUDENTS(STUDNO, STUDNAME, SEX)

1 Ramesh

```
STUD_NT(STUD_TYPE(10,'Rajesh','M'),STUD_TYPE(11,'Suresh','M'),ST  
_TYPE(12,'Uma','F'))
```

Likewise, multiple records can be inserted.

Using Flattened Subqueries

To manipulate the individual rows of a nested table stored in a database column, use the keyword THE. THE keyword must be prefixed to a subquery that returns a single column value or an expression that yields a nested table. If the subquery returns more than a single column value, a runtime error occurs. Because the value is a nested table, not a scalar value, Oracle must be informed, which is what THE does.

QUERING NESTED TABLE Values alone (Built-in Function Used : THE)

```
SELECT nt.studno, nt.studname, nt.sex FROM  
THE ( SELECT students FROM faculty WHERE  
factno = 1 ) nt ;
```

Note, here only one main subset value can be selected.

The display of the above query will be,

STUDNO	STUDNAME	S
----	-----	-
10	Rajesh	M
11	Suresh	M
12	Uma	F

Here, if a user needs to enter another record, this can be done using the following syntax:

```
INSERT INTO THE ( SELECT students FROM faculty
                  WHERE factno=1) nt
                  VALUES ( 13, 'Hema', 'F' ) ;
```

1 row gets created. If the previous query is issued, the display changes to,

STUDNO	STUDNAME	S
----	-----	-
10	Rajesh	M
11	Suresh	M
12	Uma	F
13	Hema	F

In the case of Nested table, a particular column value can be inserted using,

```
INSERT INTO THE ( SELECT students FROM faculty
                  WHERE factno=1 ) nt
                  (Nt.studname) VALUES ( 'Badri' ) ;
```

The output of the query after insert is ,

```
SELECT nt.studno, nt.studname, nt.sex FROM
THE (SELECT students FROM faculty WHERE
      factno = 1 ) nt ;
```

STUDNO	STUDNAME	S
----	-----	-
10	Rajesh	M

11	Suresh	M
12	Uma	F
13	Hema	F
	Badri	

Data in the Relational table can be inserted into the Nested table using the function called CAST. CAST along with MULTISET operator, converts the relational record to a nested table record.

```
INSERT INTO faculty VALUES
( 2 , 'Geetha' ,
CAST(MULTISET (SELECT * FROM THE (SELECT
Students FROM faculty WHERE factno = 1))
AS stud_nt)
) ;
```

The display when all the records are selected will be,

FACTNO NAME

STUDENTS(STUDNO, STUDNAME, SEX)

1 Ramesh

STUD_NT(STUD_TYPE(10,'Rajesh','M'), STUD_TYPE(11,'Suresh','M'),ST
_TYPE(12,'Uma','F'),STUD_TYPE(13, 'Hema ', 'F'), STUD_TYPE(NULL
'Badri ',NULL), STUD_TYPE(NULL, 'Badri ', NULL))

2 Geetha

STUD_NT(STUD_TYPE(10, 'Rajes ', 'M') , STUD_TYPE(11, 'Suresh ', 'M') ,
ST_TYPE(12, 'Uma ', 'F') , STUD_TYPE(13, 'Hema ', 'F') , STUD_TYPE(NULL
'Badri ', NULL) , STUD_TYPE(NULL, 'Badri ', NULL))

Updating a particular column in a subset:

Updating records can be done using the UPDATE and THE statement. The following example is illustrative of this:

```
UPDATE THE ( SELECT students FROM faculty WHERE factno=1) nt
    SET nt.studname = 'Modified'
WHERE nt.studno = 11 and nt.sex = 'M' ;
```

Updates the record in the faculty 1 and under that, the student 11 is updated.

Deleting Individual Records

Deletion can be done in two ways. Either the whole record gets deleted or the corresponding record in the subset can be deleted.

```
DELETE FROM THE (SELECT students FROM faculty WHERE factno = 1)nt
    WHERE nt.studno=11
```

This deletes the record corresponding to student 11 for the faculty number 1.

Describing Individual types

DESC keyword like describing other tables is used to DESCRIBE any object type.

The following points describe the differences between Nested Table and Varrays:

- *Nested tables* are also singly dimensioned, unbounded collections of homogenous elements. They are initially dense but can become sparse through deletions. Nested tables are available in both PL/SQL and the database (for example, as a column in a table).
- *VARRAYs*, like the other two collection types, are also singly dimensioned collections of homogenous elements. However, they are always bounded and never sparse. Like nested tables, they can be used in PL/SQL and in the database. Unlike nested tables, when you store and retrieve a VARRAY, its element order is preserved.

Using a nested table or VARRAY, you can store and retrieve non-atomic data in a single column. For example, the employee table used by the HR department could store the date of birth for each employee's dependents in a single column.

Row Objects and Column Objects

Objects that appear in object tables are called row objects. Objects that appear only in table columns or as attributes of other objects are called column objects.

Embedded and non Embedded Objects

An embedded object is one that is completely contained within another. A column object is one that is represented as a column in a table. To take advantage of OO capabilities, a database must also support ROW OBJECTS – Objects that are represented as rows instead of columns. The row objects are not embedded Objects; instead, they are referenced objects, accessible via references from other objects.

Object Tables And OID (ROW OBJECTS)

An object table is a special kind of table that holds objects and provides a relational view of the attributes of those objects. In an object table, each row is a row object. Each row within the object table has an OID – an Object Identifier value – assigned by Oracle when the row is created. An OID is an identifier for a row object. Other objects within the database can reference the rows of an object table.

Example

```
CREATE OR REPLACE TYPE deptype AS OBJECT
(
  depno NUMBER,
  dname VARCHAR2(20),
  loc VARCHAR2(20)
);
```

This is a simple object type. To create a table out of this object type, the following syntax must be used.

Syntax:

```
CREATE TABLE <tablename> OF <typename>;
```

The example given below shows this better.

To create an object table of the DEPTYPE datatype,

```
CREATE TABLE deptab OF deptype;
```

Oracle allows you to view this table in two ways:

- A single column table in which each entry is an deptype object.
- A multi-column table in which each of the attributes of the object type deptype, namely depno, dname, and loc occupies a column.

Inserting Records to Object Tables.

Inserting records onto object tables can be done in two ways: either by using the constructor method or by a normal insertion. The following example inserts records in both ways:

1. Insertion using Constructor Method

```
INSERT INTO deptab VALUES(deptype(10,'sales','chennai'));
```

2. A Relational way of Insertion

```
INSERT INTO deptab VALUES (20,'Purchase','Bangalore');
```

Selecting Records

After the records are inserted, selecting can be done in a normal way as how other table records are queried. When records are selected from the table, the output will be:

```
SELECT * FROM deptab;
```

DEPNO	DNAME	LOC
10	sales	chennai
20	Purchase	Bangalore

In an object table, each row is a row object. An object table differs from a normal relational table in the following ways :

- Each row within the object table has an OID – an object identifier value – assigned by ORACLE when the row is created. An OID is an identifier for a row object.
- Other objects within the database can reference the rows of an object table.

If the object table is based on an abstract datatype that uses no other abstract datatype, then the object table would behave as if it were a relational table. In the above example, the deptype datatype does not use any other abstract datatype for any of its columns, so the deptab object table behaves as if deptab were a relational table.

Object Identifiers (OIDS)

Oracle8's object identifiers (OIDs) are essentially the reincarnation of the pointer construct. ROWIDs are used by conventional tables and can change during the life span of that table. OIDs are used only by object tables, and they do not change or repeat.

An OID is a 128-byte base-64 number generated internally by the Oracle engine. OIDs are never reused once they are discarded. Oracle guarantees that OIDs are globally unique, even across a distributed system. OIDs are created implicitly whenever an object table is created.

REFs

In the relational model, foreign keys express many-to-one relationship. Oracle with the objects option provides a more efficient means of expressing many-to-one relationships when the "one" side of the relationship is a row object. Oracle gives every row object a unique, immutable identifier, called an OBJECT IDENTIFIER. REFs only describe a relationship in one direction.

Ref and Deref Constructs

The OID contains no actual data but merely "points" to the object table. An object identifier (OID) allows the corresponding object to be referred to from other objects or from relational tables. A built-in datatype called REF represents such references. A REF is a container for an object identifier. REFs are pointers to objects. The only operation Oracle allows on a REF is to replace its contents with a reference to a different object. A REF encapsulates a reference to a row object of a specified object type.

A table can have top-level REF columns or it can have REF attributes embedded within an object column. Tables can have a combination of REF columns and REF attributes, each referencing a different object table.

Example

Consider the deptab object table created using the deptype abstract datatype :

```
CREATE TABLE EMPLOYEES
(
  EMPNO NUMBER,
  ename VARCHAR2(20),
  deptno REF deptype );
```

In the above example, DEPTNO column references data that is stored elsewhere. The REF operator points the DEPTNO column to the DEPTYPE datatype.

```
INSERT INTO EMPLOYEES SELECT 1,'Kalyani',REF(X) FROM DEPTAB X
WHERE depno=10;
```

The deptno column contains the reference to the row object, not the value of the data stored in the row object.

The REF operator takes a row object as its argument and returns a reference value. The DEREF operator performs the opposite function – it takes a Reference value and returns the value of the row objects. The DEREF operator takes as its argument the OID generated for a reference.

In the above example , the reference OID can be seen by querying EMPLOYEES table.

```
SELECT * FROM EMPLOYEES;
```

Displays,

EMPNO	ENAME	DEPTNO
-----	-----	-----
1	Kalyani	

00002202087867EC0FCE3511D4BDBA008048DB6C207867EC0ECE3511D4BDBA008048DB6C2
0

The deptno column contains the reference to the row object, not the value of the data stored in the row object . The DEREF operator will take the OID from deptno column and find the referenced object; the operator will evaluate the reference and return the values to the user.

To display the row object the deptno column in the EMPLOYEES table refers use:

```
SELECT DEREF(K.deptno) FROM employees K
WHERE ename = 'Kalyani' ;

DEREF(K.DEPTNO)(DEPNO, DNAME, LOC)
DEPTYPE(10, 'sales', 'chennai')
```

To display selective information from the rowobject the deptno column in the EMPLOYEES table refers

```
SELECT k.ENAME,k.DEPTNO.DNAME FROM EMPLOYEES K ;
```

This would display,

ENAME	DEPTNO.DNAME
Kalyani	sales

In general, if a table has a REF column, each REF value in the column could reference a row in a different object table. Refs can be of different types as given below:

SCOPED REFS

In declaring a column type, collection element, or object type attribute to be a REF, you can constraint it to contain only references to a specified object table. Such a REF is called a SCOPED REF. SCOPED REFS require less storage space and allow more efficient access than UNSCOPED REFS.

A SCOPE clause restricts the scope of references to a single table. A scoped reference is used when the intention is to limit the reference to a particular table. If the SCOPE option is

selected when a table is created, then all further instances of data in the table must reference the scoped table.

Example

```
CREATE TABLE EMPLOYEES
(
    ENAME VARCHAR2(20),
    DEPTINF REF DEPTYPE SCOPE IS DEPTAB
) ;
```

The above example indicates that the values must be obtained from the object table DEPTAB, which is described as a type of DEPTYPE. Utilizing the SCOPE clause reduces space requirements and speeds access because the target table is isolated.

The REF values can be stored with or without ROWIDs. Storing REF values with ROWID can enhance the performance of the dereference operation but takes up more space.

EXAMPLE : (REF WITH ROWID)

```
CREATE TABLE employees
(
    ename varchar2(20), deptinf REF deptype SCOPE IS deptab WITH ROWID
) ;
OR
```

```
CREATE TABLE employees
(
    ename varchar2(20), deptinf REF deptype SCOPE IS deptab,
    REF(deptinf) WITH ROWID
) ;
```

The default behavior is to store REF values without the ROWID.

When records are inserted the scope is checked for and if the scope is violated, an error is displayed as shown below:

```
CREATE TABLE deptab1 OF deptype;
```

Table created.

```
insert into deptab1 values(12,'fin','chennai');
```

```
1 row created.
```

```
INSERT INTO employees SELECT 'hema',ref(x) FROM deptab1 x where depno=10;
```

```
insert into employees select 'hema',ref(x) from deptab1 x where depno=10
```

```
*
```

ERROR at line 1:

```
ORA-22889: REF value does not point to scoped table
```

In the case of any restriction, this scope checks for the table.

DANGLING REFS

When a REF value points to a non-existent object, the REF is said to be DANGLING. DANGLING is different from being NULL. This is because it is possible for the object identified by a REF to become unavailable – either through deletion of the object or a change in privileges. Such a REF is called DANGLING REF. To check to see if a REF is dangling or not, Oracle SQL provides the predicate IS [NOT] DANGLING.

Example

A Record in the table deptab is deleted. If any records are further referred, dangling can be checked.

```
INSERT INTO EMPLOYEES SELECT 'Sriram',REF(X) FROM DEPTAB X
```

```
WHERE depno=20;
```

```
1 row created.
```

```
DELETE FROM deptab where depno=20;
```

```
SELECT k.ename, k.deptinf.dname  
FROM EMPLOYEES k WHERE k.DEPTinf IS NOT DANGLING ;
```

ENAME	DEPTINF.DNAME
Sriram	Purchase

DEREFERENCING REFs

Accessing the object referred to by a REF is called DEREFERENCING the REF. Oracle provides the DEREF operator to do this. Dereferencing a Dangling REF results in a null object.

RECURSION

RECURSION is the condition where an entity has a relationship to itself. It is also referred to as SELF - ASSOCIATION .

Example

```
CREATE TYPE EMPLOYEE AS OBJECT
(
    EMPNO  NUMBER,
    ENAME  VARCHAR2(20),
    JOB      VARCHAR2(20),
    MGR      REF EMPLOYEE,
    SAL      NUMBER(10,2),
);
```

To create an object table EMP of type EMPLOYEE with PRIMARY KEY constraint on the EMPNO attribute and to place a SCOPE on the REFS in the MGR column.

```
CREATE TABLE EMP OF EMPLOYEE
(
EMPNO PRIMARY KEY
);
ALTER TABLE EMP ADD (SCOPE FOR (MGR) IS EMP);
OR
CREATE TABLE EMP OF EMPLOYEE ;
ALTER TABLE EMP ADD PRIMARY KEY(EMPNO);
ALTER TABLE EMP ADD( SCOPE FOR (MGR) IS EMP);
```

```
INSERT INTO EMP VALUES (    1000,'LAN','PRESIDENT',NULL,15000);
INSERT INTO EMP SELECT
2000,'PARTHA','MANAGER',REF(E),7000
FROM EMP E WHERE E.EMPNO=1000 ;
```

Oracle provides *implicit dereferencing of REFs* .

Example

To display the employee details along with the information of their manager :

```
SELECT e.empno,e.ename,e.job,deref(e.mgr),e.sal
FROM EMP e WHERE e.mgr IS NOT DANGLING;
```

To display the employees name and his manager name :

```
SELECT e.ename,e.mgr.ename FROM EMP e WHERE e.mgr IS NOT DANGLING ;
```

19.6 Object Views

Although Oracle's object extensions offer rich possibilities for design of new systems, few Oracle shops with large relational databases in place will want or be able to completely re-engineer those systems to use objects. In order to allow established applications to take advantage of these new features over time, Oracle8 provides *object views*. With object views, you can achieve the following benefits:

Efficiency of object access. In PL/SQL, and particularly in Oracle Call Interface (OCI) applications, object programming constructs provide for convenient retrieval, caching, and updating of object data. These programming facilities can provide performance improvements, with the added benefit that application code can be more succinct.

Ability to navigate using REFs. By designating unique identifiers as the basis of an object identifier (OID), you can reap the benefits of object navigation. For example, you can retrieve attributes from related "virtual objects" using dot notation rather than explicit joins.

Easier schema evolution. In early versions of Oracle8, a pure object approach renders almost any kind of schema change at best ugly. In contrast, object views offer more ways that you can change both table structure and object type definitions of an existing system.

Consistency with new object-based applications. If you need to extend the design of a legacy database, the new components can be implemented in object tables; new object-oriented applications requiring access to existing data can employ a consistent programming model. Legacy applications can continue to work without modification.

Other new features of Oracle can improve the expressiveness of any type of view, not just object views. Two features which are not strictly limited to object views are collections and "INSTEAD OF" triggers. Consider two relational tables with a simple master-detail relationship. Using the Oracle objects option, you can portray the detail records as a single non-scalar attribute (collection) of the master, which could be a very useful abstraction. In addition, using INSTEAD OF triggers, you can tell Oracle exactly how to perform inserts, updates, and deletes on any view. These two features are available to both object views and non-object views.

Object View Example

In this example, we look at how object views might be used at a fictitious firm that designs web sites. Their existing relational application tracks JPEG, GIF, and other images that they use when designing client web sites. These images are stored in files, but data about them are stored in relational tables. To help the graphic artists in locating the right image, each image has one or more associated keywords, stored in a straightforward master-detail relationship.

```
CREATE TABLE images (
    image_id INTEGER NOT NULL,
    file_name VARCHAR2(512),
    file_type VARCHAR2(12),
    bytes INTEGER,
    CONSTRAINT image_pk PRIMARY KEY (image_id));
...and one table for the keywords associated with the images:
```

```
CREATE TABLE keywords (
    image_id INTEGER NOT NULL,
    keyword VARCHAR2(45) NOT NULL,
    CONSTRAINT keywords_pk PRIMARY KEY (image_id, keyword),
    CONSTRAINT keywords_for_image FOREIGN KEY (image_id)
```

```
REFERENCES images (image_id));
```

To create a more useful abstraction, we've decided to logically merge these two tables into a single object view. In order to do so, we must first create an object type with appropriate attributes. Since there are usually only a few keywords for a given image, this relationship lends itself to using an Oracle collection to hold the keywords.

Before we can create the top-level type, we will first define a collection to hold the keywords.

```
CREATE TYPE Keyword_tab_t AS TABLE OF VARCHAR2(45);
```

From here, it's a simple matter to define the object type. To keep the example short, we'll define only a couple of methods. In the following object type specification, notice that the keywords attribute is defined on the Keyword_tab_t collection type.

```
CREATE TYPE Image_t AS OBJECT (
    image_id INTEGER,
    file_name VARCHAR2(512),
    file_type VARCHAR2(12),
    bytes INTEGER,
    keywords Keyword_tab_t,

    MEMBER FUNCTION setAttrs (new_file_name IN VARCHAR2,
        new_file_type IN VARCHAR2, new_bytes IN INTEGER)
        RETURN Image_t,
    MEMBER FUNCTION setKeywords (new_keywords IN Keyword_tab_t)
        RETURN Image_t,
    PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS, WNDS, RNPS, WNPS)
);
```

Here is the body:

```
CREATE TYPE BODY Image_t
AS
    MEMBER FUNCTION setAttrs (new_file_name IN VARCHAR2,
        new_file_type IN VARCHAR2, new_bytes IN INTEGER)
        RETURN Image_t
    IS
        image_holder Image_t := SELF;
```

```
BEGIN  
    image_holder.file_name := new_file_name;  
    image_holder.file_type := new_file_type;  
    image_holder.bytes := new_bytes;  
  
    RETURN image_holder;  
END;  
  
MEMBER FUNCTION set_keywords (new_keywords IN Keyword_tab_t)  
RETURN Image_t  
IS  
    image_holder Image_t := SELF;  
    BEGIN  
        image_holder.keywords := new_keywords;  
  
        RETURN image_holder;  
    END;  
  
END;
```

At this point there is no connection between the relational tables and the object type. They are independent organisms. It is when we build the object view that we "overlay" the object definition onto the tables.

Finally, to create the object view, we use the following statement:

```
CREATE VIEW images_v  
    OF Image_t  
    WITH OBJECT OID (image_id)  
AS  
    SELECT i.image_id, i.file_name, i.file_type, i.bytes,  
          CAST (MULTISET (SELECT keyword  
                           FROM keywords k  
                           WHERE k.image_id = i.image_id)  
                AS Keyword_tab_t)  
    FROM images i;
```

Differences Between Object Views and Object Tables

In addition to the obvious difference between a view and a table, more subtle differences exist between an object view and an object table. Areas of difference include:

- ***Uniqueness of object identifiers.*** Whereas object tables *always* have unique OIDs, it is possible to create object views which contain duplicate virtual OIDs. Object views may also duplicate the OIDs of other object views or object tables.
- ***Use of REFs.*** As with an object table, it is possible to use a REF to point to an instance of an object view. However, these REFs must be constructed from foreign keys with the special built-in MAKE_REF, which turns out to be a rather finicky function. In particular, if your foreign key value is null, MAKE_REF returns an error!
- ***Storage of REFs.*** Object tables can store REF values persistently, but REFs to virtual objects in an object view must be constructed dynamically (with MAKE_REF).
- ***REFs to Non-Unique OIDs.*** It is impossible in an object table to have a REF to a non-unique OID; but an object view which has such a REF may have this problem. You have to consider it and deal with it if necessary.

Strategies for Object Views

Here are some of the things that should guide your thinking about object views:

Because of their ability to tolerate schema changes, object views can provide advantages over conventional objects. What is unclear is the degree of performance impact they may cause (when compared against object tables).

Unless you have a very good reason to do otherwise, define the OID of an object view to be a unique value.

Adopt a consistent approach in the localization of DML on object views. INSTEAD OF triggers are cool, but triggers can exhibit confusing interactions with each other. Packages may still be the optimal construct in which to define insert, update, and delete logic.

19.7 Member Functions

When dealing with numbers, common convention dictates that the number with the larger value is greater than the other. But when objects are compared, the attributes based on the comparison may not be determined. In order to compare objects, Oracle allows declaring two types of member functions called MAP and ORDER.

MAP

Map Function is used to specify a single numeric value that is used to compare two objects of the same type. The greater than or less than or equality is based on this value.

ORDER

Order Function is used to write the specific code in order to compare two objects and the return value indicates the equality or greater or lesser comparisons.

Using Map Method

MAP method is used to provide an alternative way to specify comparison semantics of the object type.

A MAP function enables to compute a singular scalar value, based on one or more of the object's attributes, which is then used to compare the object in question to other objects of the same type. Map functions require no parameter because they return a value representing only the object whose MAP method is invoked. The result of the MAP method can be one of the following types:

- NUMBER
- DATE
- VARCHAR2

19.8 Using ORDER method

ORDER method is used to compare the object values and return values such as 1 indicates that the current object is greater than the parameter , 0 indicates equality and -1 if the current object is less than the argument.

Note: Whenever Member Functions are used there needs to be specified a Restrict_References needs to be specified. Either an ORDER or MAP function alone needs to be specified.

The Object Types contain 2 parts like packages namely Specification and Body. Body is a must only when any member function is specified.

If an object type is created without any member function comparison cannot be performed as the following example illustrates:

```
select empno,ename,a.sal from emptab a;
```

```
EMPNO  ENAME
-----
SAL(BASIC, DA)
-----
1 hema
SALTYPE(1000, 2300)
2 magesh
SALTYPE(10000, 20000
select a.empno,a.ename,a.sal from emptab a,emptab b
where a.sal >=b.sal;
where a.sal >=b.sal
*
```

ERROR at line 2:

ORA-22950: cannot ORDER objects without MAP or ORDER method In order to do this operation write a function called compare which compares the two objects.

CREATE OR REPLACE TYPE deptype AS Object

```
(deptno number,dname varchar2(20))
```

Order Member Function compare(dtype In deptype) Return Number;

Create or replace type body deptype as order member function compare (dtype in deptype)
return number is if self.deptno > dtype.deptno then

```
    return 1;
```

```
    else
```

```
        return 0;
```

```
    end if;
```

```
end;
```

This allows the above query to be executed.

19.9 Short Summary

- ❖ Oracle 8 implements the object oriented concepts to exhibit real world objects. They are implemented as datatypes(abstract datatypes).
- ❖ Abstract datatypes are embedded as datatypes to a column in a relational table.
Collection datatypes hold multiple data items. They can be:
 - Varrays (Varying Size Arrays)
 - Nested Tables
- ❖ Object tables are used to establish a master-child relationship between objects.
- ❖ Member Functions implement the behaviour of an object. The Member Functions can be either Map or Order Member Functions.
- ❖ Object Views provide efficiency of access, consistency with new object based applications.

19.10 Brain Storm

1. What is Object Constructor?
2. What is Atomic Null Violation Error?
3. What is Row Object and Column Object?
4. What is the clause used in insert statement to return the object reference?
5. What are the advantages of Object tables?
6. How do define Dangling Reference?
7. Objects can be compared only for equality – True or False?
8. Methods can be added to an object type using
Alter type <typename> replace as object (-----);
9. Object type methods can be overloaded – True or False?
10. Explain Map and Order Member Function?
11. What are scoped Reference and Unscoped Reference?

ANSWER

Lecture 20

Introduction to DBA

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about the Database administrator
- ☒ Discuss about the responsibilities of DBA
- ☒ Describe about the roles of DBA
- ☒ Understand about cursor and its use
- ☒ Understand about the parse and create one simple program

Coverage Plan

Lecture 20

- 20.1 Snap shot
- 20.2 Introduction to DBA
- 20.3 Open cursor
- 20.4 Execute
- 20.5 Oracle DBA
- 20.6 Short summary
- 20.7 Brain Storm

15.1 Snap Shot

This chapter introduces the concept of Dynamic SQL along with roles and responsibilities of a Database administrator.

- Dynamic SQL introduction
- Writing a simple program
- Roles and responsibilities of a DBA

15.2 Introduction to DBA

Before a PL/SQL program can be executed, it must be compiled. The PL/SQL compiler resolves references to Oracle schema objects by looking up their definitions in the data dictionary. Then, the compiler assigns storage addresses to program variables that will hold Oracle data so that Oracle can look up the addresses at run time. This process is called binding.

How a database language implements binding affects runtime efficiency and flexibility. Binding at compile time, called static or early binding, increases efficiency because the definitions of schema objects are looked up then, not at run time. On the other hand, binding at run time, called dynamic or late binding, increases flexibility because the definitions of schema objects can remain unknown until then.

Designed primarily for high-speed transaction processing, PL/SQL increases efficiency by bundling SQL statements and avoiding runtime compilation. Unlike SQL, which is compiled and executed statement-by-statement at run time (late binding), PL/SQL is processed into machine-readable p-code at compile time (early binding). At run time, the PL/SQL engine simply executes the p-code.

Overcoming the Limitations

However, the package DBMS_SQL, which is supplied with Oracle, allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time.

20.3 Open_Cursor

To process a SQL statement, you must have an open cursor. When you call the OPEN_CURSOR function, you receive a cursor ID number for the data structure representing a valid cursor maintained by Oracle.

Parse

Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program

You can parse any data manipulation language or data definition language statements. Data definition language statements are executed on the parse, which performs the implied commit.

20.4 Execute

Call the EXECUTE function to execute your SQL statement.

CLOSE_CURSOR

When you no longer need a cursor for a session, close the cursor by calling CLOSE_CURSOR.

Example 14.1

```
DECLARE
    curid NUMBER;
    str  VARCHAR2(200);
    exeid NUMBER;
    tablename varchar2(20):='&a';

BEGIN
    curid:=DBMS_SQL.OPEN_CURSOR;
    str := 'DROP TABLE '||tablename;
    DBMS_SQLPARSE(curid,str,dbms_sql.native);
    exeid :=DBMS_SQL.EXECUTE (curid);
    DBMS_SQLCLOSE_CURSOR(curid);
END;
```

This block accepts the tablename and drops the table name.

20.5 Oracle DBA

Because an ORACLE database system can be quite large and have many users, someone or some group of people must manage this system. The database administrator (or DBA) is this manager.

Every database requires at least one person to perform administrative duties; if a database is large, these administrative duties can be divided among multiple administrators.

DBA's Roles

As a database administrator, your responsibilities can include:

- installing and upgrading the ORACLE Server and application tools
- allocating system storage and planning future storage requirements for the database system
- creating primary database storage structures (tablespaces) once application developers have designed an application
- creating primary objects (tables, views, indexes) once application developers have designed an application
- modifying the database structure, as necessary, from information given by application developers
- enrolling users and maintaining system security
- ensuring compliance with your ORACLE license agreement
- controlling and monitoring user access to the database
- monitoring and optimizing the performance of the database
- planning for backup and recovery of database information
- maintaining archived data on tape
- backing up and restoring the database
- contacting Oracle Corporation for technical support

To accomplish administrative tasks in ORACLE, you need extra privileges both within the database and possibly in the operating system of the server on which the database runs. This section describes the privileges you need and database security facilities of which you should be aware.

Note: Access to a database administrator's account should be tightly controlled.

20.6 Short Summary

- ❖ Database language implements binding affects runtime efficiency and flexibility. Binding at compile time, called static or early binding.
- ❖ Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program.
- ❖ Responsibilities of DBA Installing, allocating, creating, modifying, enrolling, ensuring, controlling, monitoring, planning, maintaining, backing, contacting.

20.7 Brain Storm

1. Explain the roles of DBA.
2. Explain about open cursor.
3. What is parse?



Lecture 21

Java Strategies

Objectives

After completing this lesson, you should be able to do the following

- ❖ Discuss about Java strategies
- ❖ Describe JDBC
- ❖ Describe Java with SQL
- ❖ Discuss about Java Virtual Machine
- ❖ Describe about Java stored procedures

Coverage Plan

Lecture 21

- 21.1 Snap shot
- 21.2 Java strategy
- 21.3 JDBC
- 21.4 Java with SQL
- 21.5 Java virtual machine
- 21.6 Java stored procedures
- 21.7 Short summary
- 21.8 Brain Storm

21.1 Snap Shot

Oracle recognizes Java as a very powerful language that's quickly gaining acceptance in the development community for deploying enterprise applications. Several important characteristics of Java contribute to its success:

- It's an object-oriented language.
- It uses an open standard for application development.
- It uses JavaBeans and Enterprise JavaBeans (EJB), which can improve productivity by allowing code reuse.
- It enables you to develop portable applications.
- It can execute in browsers, application servers, and databases. Thus, developers can write applications by using just one language. It's widely accepted as the language to develop Internet applications.

In this hour, you will

- ☞ Understand the Java Virtual Machine
- ☞ Create and use Java stored procedures
- ☞ Understand the Internet file system
- ☞ Understand Enterprise JavaBeans

21.2 Oracle's Java Strategy

Java is a key component to Oracle's strategy of network computer models that will ultimately result in a low cost of ownership. Oracle has implemented Java into Oracle8I with two major approaches:

- ☞ Providing an enterprise-class Java sever platform with fast access and manipulation for online transaction processing systems and decision support systems, support for a large number of concurrent users, high availability and fast failover, integration with system

management tools such as Oracle Enterprise Manager, and integration of a Java Virtual Machine (JVM).

- ❖ Providing a set of Java tools that can be used to quickly develop efficient Java applications: JDBC drivers that provide database connectivity from Java; an SQLJ translator, which allows the use of embedded SQL in Java; Jdev3elope, which allows the development of Java programs and integrates JDBC and the SQLJ translator into a complete development environment; CORBA connectivity; Oracle Application Server, the ability to create and load Java stored procedures and triggers; and component-oriented development, supported with the use of JavaBeans and EJB.

21.3 JDBC

Using Java database Connectivity (JDBC)

JDBC is a standard set of classes that allow application developers to access and manipulate relational databases from within Java programs. JDBC supports SQL 92 syntax and allows vendors to provide extensions to improve performance.

JDBC Drivers provided by Oracle

Oracle provides two types of JDBC drivers, which you can use for different types of applications.

This JDBC driver Written completely in Java, this is ideal for java applets that can be used with a browser. The thin driver is only 300 kb and can be downloaded. When the applet is fired, the thin driver establishes a direct Net8 connection between the applet and the Oracle database. Scalability is achieved by the use of the Net8 Connection Manager, which should reside on the same host as the Web server. The Connection Manager multiplexes several inbound physical connections onto a single database connection, thereby saving server memory.

JBBC / OCI driver - This provides OCI calls to access the database, using Oracle client libraries such as OCILIB, CORE, and Net8. These are written in C, so the driver isn't downloadable. You have to perform client installation of the JDBC / OCI driver. It can be

used for client / server Java applications, as well as middle-tier Java applications running in a Java application server.

Both drivers are JDBC complaint and support Oracle-specific features:

- Support for Oracle7 and Oracle8 object-related data types
- Support for manipulating LOB data
- Performance enhancement features such as array interface, prefetching, and batch SQL statement execution.
- Access to PL/SQL and Java stores procedures.
- Support for all Oracle character sets.

21.4 Java with SQL

Embedding SQL in Java with SQLJ

Oracle allows developers to write efficient and compact programs with the use of SQLJ. SQLJ, built on top of JDBC, allows application developers to embed SQL statements in Java programs, and to run a preprocessor to translate SQLJ to Java code that makes JDBC calls to the database by using a JDBC driver from any vendor. SQLJ programs can be developed and debugged by using Oracle's JDeveloper, which is a standard Java development tool. Because the SQLJ runtime environment runs on top of the

JDBC driver, it can be deployed by using the same configuration as JDBC programs-namely, the JDBC/OCI driver with middle-tier applications and the thin JDBC driver for Java applets. The biggest advantage you gain by using SQLJ is increased productivity and manageability of Java code by providing the following:

- Code that's significantly compact compared to JDBC
- Strong types queries with the use of typed cursors

- Compile-time checking of SQL statements to identify errors at an earlier stage and quickly debug and deploy applications.

You can use SQLJ programs by following these generic steps.

1. Write a SQLJ Java program
2. Run the SQLJ program through the SQLJ translator. The translator preprocesses the SQLJ program and generates standard Java code with JDBC calls.
3. Compile the Java code
4. Run the Java program.

A Sample JDBC Program

```
Java.sql.CallableStatement stmt;  
Connection conn;  
ResultSet res;  
  
/* Declare the objects for a callable statement,  
connection, and the result set*/  
  
Conn = DriverManager.getConnection ("jdbc:default");  
  
/* Initialize the connection object with the default connection*/  
Stmt = conn. PrepareStatement  
("SELECT ename FROM emp WHERE ecity = ? AND deptno = ? *");  
  
/* Inituakuze the connection object with the default connection */  
Stmt = conn.prepareStatement  
("SELECT ename FROM emp WHERE ecity = ? AND deptno = ?");  
  
/* Prepare the statement to execute */  
  
Stmt.setString (1, city_p);
```

```
Stmt.setInteger (2,deptno_p);

/* Use positional parameters to set the variables*/

Res = stmt.executeQuery ( );

/* Execute the query and store the results in the result set */
```

A Sample SQLJ Program

```
ResultSet res;

/* Define an object to store the result set */

#sql res = (SELECT ename FROM emp
             WHERE city = :city_p AND deptno = :deptno_p);

/* Pass the program through the SQLJ translator and execute the program. The result set
is stored in "res" */
```

Oracle8I offers several features that make it an open and extensible platform for application development:

- ❖ An open protocol CORBA / IIOP
- ❖ Java as an open server programming language (in addition to PL/SQL)
- ❖ The capability to use data cartridges
- ❖ The Internet File System (IFS)

21.5 Java virtual machine

Understanding the Java Virtual Machine

Oracle's Java virtual machine (JVM) is a very important component of the Oracle Java strategy, providing high performance and scalability while running Java applications. Oracle's JVM is tightly integrated with the database and runs on the Multi-Threaded Server. The JVM has several characteristics.

- ❖ It supports shared Java byte codes and lightweight Java threads.
- ❖ It provides a well-tuned memory manager and garbage collector that optimize the use of the SGA (40KB per user session) and the operating system's virtual memory manager.
- ❖ It allows Java classes to be stored in the database as database library units.
- ❖ It provides the NCOMP Java compiler to translate Java byte code to C executables, which can be run very efficiently.
- ❖ It supports standard Java libraries such as Java.lang, Java.io, and Java.util.
- ❖ It supports JDBC driver and SQLJ translator approaches
- ❖ It provides a CORBA 2.0-compliant ORB (Object Request Broker) that can be used to call in and out of the database with the CORBA/IOP protocol.
- ❖ It provides utilities to load and unload Java programs into the database.
- ❖ It allows you to use Java stored procedures and triggers.
- ❖ It's 100% JDK 1.1.6 compliant
- ❖ It allows the use of Enterprise JavaBeans (EJB 1.0 compliant) by taking various approaches: pure java clients, CORBA/IOP clients, and D/COM clients.
- ❖ It provides a standard JDBC driver, which offers the same interfaces as OCI, and thin JDBC drivers, which allow Java applications to be easily loaded into and used in the database.

- ❖ It efficiently uses Oracle Multi-Threaded Server (MTS). The Net8 Connection Manager also can multiplex CORBA/IOP connections, in addition to Net8 connections.
- ❖ It provides mission-critical high availability and transparent application failover.
- ❖ It supports efficient load balancing on SMP and MPP architectures.
- ❖ It allows ease of management through the use of Oracle Enterprise Manager. OEM allows centralized monitoring and management of distributed Oracle8 servers and Java stored programs, use of standard Oracle utilities such as SQL* Loader, and Export/Import for Java procedures.
- ❖ It allows the use of existing database security mechanisms such as roles and grants for PL/SQL, as well as Java stored procedures.
- ❖ The use of "invoker" rights with PL/SQL and Java stored procedures simplifies security management.
- ❖ It uses security features available through the advanced networking option against Java procedures.

Using Enterprise JavaBeans and CORBA

EJBs allow the use of a component-based model for implementing Java systems. They can be completely developed in Java and deployed by using execution containers such as databases., TP-Monitors (also known as Transaction Processing Monitors), and application servers. The use of EJBs offers several advantages:

- They use a high level of abstraction, making them simple to program
- They are developed completely in Java.
- They are compatible with CORBA, D/COM, and so on.
- They allow the use of transactional applications as modular components.

Oracle8I provides a CORBA 2.0 - complaint ORB, which allows applications to access Java stores procedures and EJBs through the IIOP protocol. It also allows you to write Java applications that can call out of the database and communicate with other ORBs through IIOP. Using CORBA/IIOP with Oracle8I now requires the use of MTS.

The mechanism to use CORBA/IIOP is very simple:

1. Write you Java stored procedures and EJBs.
2. Publish the procedures and EJBs so that they can be accessed as a stored procedure via Net8 and as a CORBA server.
3. Client applications (CORBA and D/COM) can access the published procedures through IIOP.

21.6 Java Stored Procedures

You can build applications in Java by using three methods:

- Write Java stored procedures and triggers.
- Implement CORBA serves in the database in Java.
- Execute Enterprise JavaBeans.

Oracle8i lets you write stored procedures by using PL/SQL and Java. Although PL/SQL is seamless with SQL, using Java provides an open, general-purpose platform to develop and deploy applications. Java stores procedures can be written efficiently by using SQLJ, which allows SQL to be embedded in Java code. Java stored procedures can run in different runtime contexts:

- You can write top-level SQL functions and procedures that allow you to implement business logic in a manner similar to PL/SQL.
- All types of triggers supported by Oracle can be implemented using Java
- Methods associated with objects can be implemented by using Java, in addition to PL/SQL or C/C++. The body of an object type also can be implemented as a Java class.

Java stored procedures implemented in Oracle8I are standards complaint, which consist of two main parts:

Part 0 The specification of SQLJ that lets you develop efficient applications by embedding SQL statements into Java programs.

Part 1 The specification of Java stored procedures that provides standard DDL extensions to "publish" or register Java programs with SQL so that the Java stored procedure is callable from SQL. It also provides standard DML extensions to call Java from SQL.

The use of Java stored procedures provides several benefits to OLTP applications:

- Network traffic can be reduced because result sets can be processed on the database server and only the results are sent across the network. This allows efficient usage of server resources because transfer of large quantities of data across the network isn't necessary.
- Business rules can be enforced centrally and by allowing Java stored procedures to be replicated between, say, the headquarters and the branch offices.
- Various configurations can be supported, such as client/server and multi-tier configuration.
- The use of Java in the development of stored procedures provides an open and ANSI/ISO - complaint standard environment. This allows the stored procedures to be portable across various platforms.
- Java and PL/SQL can inter-operate very well to allow application reuse.

You develop Java stored procedures by following these general steps:

1. By using a standard Java tool, develop a Java program that you want to make a stored procedure. Oracle provides Jdeveloper and SQLJ, which you can use to develop efficient Java programs quickly.
2. After the Java program is written, load it into Oracle8I's JVM. Java source text can be loaded as standard Java.class files or as Java .jar files. To load Java class files in Oracle8I,

issue the CREATE JAVA DDL command, or use the LOADJAVA utility to automate the loading process.

3. Register the Java program with SQL by exposing the top-level entry point, which SQL can use to call Java. These exposed entry points are the only calls that SQL can make to this Java program. Subsequent Java-to-Java calls can be made within the JVM as needed. You should also make sure that the data types are appropriately mapped between Java and SQL.
4. Give appropriate rights to the users so that they can call and run this procedure.
5. Call the Java stored procedure from SQL DML, PL/SQL, or triggers.

21.7 Short Summary

- ❖ Oracle recognizes Java as a very powerful language that's quickly gaining acceptance in the development community for deploying enterprise applications.
- ❖ JDBC is a standard set of classes that allow application developers to access and manipulate relational databases from within Java programs.
- ❖ JDBC / OCI driver used for client / server Java applications, as well as middle-tier Java applications running in a Java application server.

21.8 Brain Storm

1. Explain about JVM?
2. Explain the JDBC concepts?
3. How to create Java stored procedures, Explain?
4. Explain the tools we used to run java class in oracle?



Lecture 22

Oracle Internet File System

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about Loading Java
- ☒ Describe about the Drop Java Utilities
- ☒ How to about Registering Java
- ☒ Discuss about Oracle Internet File System

Coverage Plan

Lecture 22

- 22.1 Snap shot
- 22.2 Loading Java Classes into the Database
- 22.3 Loading Java & Drop Java utilities
- 22.4 Registering Java Programs
- 22.5 Oracle internet file system
- 22.6 Short summary
- 22.7 Brain Storm

22.1 Snap Shot

In this session we learn about how we use java in oracle. For this we know the loading java utility and drop java utilities. In this load java utility we learn the input sources of oracle8i and we know about jdeveloper. We understand this concepts for using simple java program.

22.2 Loading Java Classes into the Database

Java developers write Java programs by using a standard Java tool such as AppBuilder for Java. They debug the programs locally and then load them into the database. Oracle8I accepts two input sources:

- A binary file on the operating system level
- A LOB column in the RDBMS

The two input sources can be obtained in three formats:

- Standard Java source texts automatically recompiled by the Oracle Java byte code compiler
- Java binaries
- Java resources

Java programs are stored as database library units. The JVM's class loader loads Java from any input source in any format into an **RDBMS** libunit. Three types of libunits are provided, one for each type of input.

22.3 Loading Java & Drop Java utilities

Using the LoadJava and DropJava Utilities

LoadJava is a utility that Oracle provides to automate loading Java programs into the database. LoadJava is written in Java and uses the JDBC drivers to communicate with the database. It can take input, which can be Java source texts, Java binaries (.class), and Java archives. It doesn't load Java resources directly, but you can archive (.jar) the Java resources and then load the JARs via the LoadJava utility.

DropJava is a complimentary utility you can use to drop Java binaries and archives from the libunits.

When LoadJava is run, it performs the following tasks:

1. It creates a system-generated table, `create$table`, that can be used to load Java.
2. The Java binaries and JARs are loaded into a BLOB column of this system-generated table.
3. `CREATE JAVA` is invoked to load Java from the BLOB column into database libunits.

The syntax for LoadJava is as follows:

`LoadJava [-user connectstring] [options]`

In this syntax,

- *Connectstring* is of the form `username / password@database` and is used to specify the database schema to load Java into.
- Options that can be specified include the following:
 - o Use the **JDBC/OCI** driver
 - t Use the thin **JDBC** driver
 - v Use verbose mode to indicate the status of the load as it occurs.
 - f Force loading of classes by replacing existing classes. The default is to load only those that don't exist and those that have been modified.
 - r The resolver spec to use during the load
 - s Create public synonyms for the loaded classes.
 - g Grant permissions on loaded classes
 - h Obtain help on the syntax

The following code example loads a Java class named `Test.class` into the schema SCOTT

In the specified server using thin JDBC drivers. The class is forced, and status information is provided as the loading proceeds.

```
LoadJava -user scott/tiger@myserver:5521:orcl -t -f -v Test.class
```

22.4 Registering Java Programs

Unlike PL/SQL procedures, Java procedures aren't automatically available to SQL. You have to register the Java programs by publishing the Java entry points so that SQL can call the Java methods. These Java methods can, in turn, make Java-to-Java calls within the JVM. Publishing Java to SQL can be accomplished in two steps:

1. Create a PL/SQL program that provides a type system wrapper for the Java method. Known as the *call descriptor*, this program specifies the mapping from Java types to SQL types, the mapping of the parameter modes, and so on.
2. At runtime, the SQL and PL/SQL code can call the "wrapper" program. This call is intercepted by the JVM, which executes the Java method in the database's address space.

Oracle8I provides a new SQL DDL syntax that you can use to specify the PL/SQL call descriptor. The PL/SQL call descriptor consists of two parts:

- The PL/SQL procedure PSEC declaration, which represents a proxy for the Java method
- The body of the PL/SQL program, which is composed of the Java method signature

The oracle.sql package provides automatic datatype conversion between Java, SQL, and PL/SQL.

MAPPING JAVA TO SQL AND PL/SQL DATATYPES

JAVA

String, oracle.sql.CHAR
Oracle.sql.DATE

SQL and PL/SQL

CHAR, VARCHAR2
DATE

Int, Integer, float, Float	NUMBER, DECIMAL, INTEGER
Double, Double	FLOAT, REAL, DOUBLE
Oracle.sql.NUMBER	PRECISION
Oracle.sql.BLOB	BLOB
Oracle.sql.CLOB	CLOB
Oracle.sql.BFILE	BFILE
Oracle.sql.RAW	RAW and LONG RAW
Oracle.sql.ROWID	ROWID
Oracle.sql.OracleObject	Oracle8 objects
Oracle.sql.REF	Oracle8 REF

When you use Java stored procedures, you should be careful about using different modes for the parameters. Java accepts only IN arguments by reference, whereas SQL and PL/SQL use IN, OUT, and IN OUT modes of parameter passing. When using IN parameters, consult the Table to determine the data type mapping that can be used between Java and PL/SQL types. While using OUT and IN OUT parameter modes in SQL and PL/SQL, the corresponding Java parameter should be a one-element array of the appropriate Java type so that no mismatch occurs with the use of parameters in the Java stored procedures.

Here's the syntax for declaring the PL/SQL call descriptor:

```
CREATE [OR REPLACE]
PROCEDURE proc_name [ ( [ sql_parameters ] ) ] :
FUNCTION func_name [ ( [ sql_parameters ] ) ]
    RETURN sql_type
AS LANGUAGE JAVA NAME 'Java_fullname ( [ Java_parameters ] )
[ return Java_type_fullname ]'

In this syntax,
sql_parameters := sql_type> [, sql_parameters]
Java_parameters := Java_type_fullname [ , Java_parameters
```

A JAVA STORED PROCEDURE

```
/* The Java method 'greetings' takes a String argument and returns
a string value. This method needs to be published in SQL */
Java Class Welcomejava:
    Public class welcome {
        Static public String
```

```
Greetings (string firstname) {
    Return "Welcome" + firstname;
}
}

/* PL/SQL proxy for welcomejava
It maps the string datatype to SQL VARCHAR2 */
CREATE FUNCTION greetings (fname VARCHAR2) RETURN
VARCHAR2 AS
LANGUAGE JAVA NAME
'Welcome.greetings (Java.lang.String)
return Java.lang.String'
```

Java methods called from SQL or PL/SQL execute under definer's privileges. Java methods called from other Java methods can execute under definer's or invoker's privileges, but by default, they execute under the invoker's privileges. This is a marked difference from execution of PL/SQL procedures that execute under the definer's privileges. In order to execute a Java method, users must have the following privileges:

- EXECUTE rights on the PL/SQL call descriptor for the Java method
- EXECUTE rights on the Java class
- EXECUTE rights on the Java classes and resources used by the Java class being called

To Do: Write a Java Stored Procedure

Now you have a chance to write a Java stored procedure that modifies an employee record to give the employee a raise and change the employee's location. This example uses Jdeveloper, but you can use any Java tool or editor to write the Java program.

Follow these steps:

1. Start Jdeveloper
2. Choose File, New Project
3. Write the Java programs as shown in Listings updjava and empjava one in each Java source file.
4. Compile both source files: updjava and empjava
5. From the toolbar, select the New button.
6. On the Deployment page, choose Stored Procedure Profile.
7. Click OK to start the Stored Procedure Wizard
8. For the wizard's first two dialog boxes, click Next.
9. In the next dialog box, select only the updjava file.

10. Click Next in the Step 3 dialog box
11. In the next dialog box, select Thin JDBC Driver and then Next.
12. Specify the connection parameters: username, password, host ID, Oracle SID, and port number, Click Next.
13. Click Next for the Step 6 dialog box
14. In the next dialog box, select the method that has to be published, and click Next.
15. Click Finish. The Java method is now loaded and published to SQL.
16. Select the Project pull-down menu from the menu bar
17. On the Run/Debug page, provide the appropriate connection parameters, for example,

Jdbc:orcale:thin:@myserver:5521;ORCL scott tiger

18. Choose Send Run Output to Execution Log.
19. Click OK.
20. Compile the empjava file.
21. Run empjava to invoke the procedure

UPDJAVA - USING SQLJ TO WRITE JAVA STORED PROCEDURES

```
package modify_emp;

import sqlj.runtime.*;
import sqlj.runtime.ref.*;

public class update_emp  {

    public static int update_emp (String name, int raise, String location)
    {

        int newsal = -1;
        String newloc = "UNKNOWN";

        try {
            # sql {UPDATE emp
                  SET sal = sal + :raise
                  WHERE      ename = :name } ;
#sql    {UPDATE emp
          SET city = :location
          WHERE ename = :name } ;
#sql {SELECT sal INTO :newsal
```

```
        FROM emp
        WHERE ename = :name} ;
#sql {SELECT city INTO :newloc
        FROM emp
        WHERE ename = :name} ;
}

catch (Java.sql.SQLException e) { }

return newsal;
}

}
```

EMPJAVA-CALLING JAVA STORED PROCEDURES

```
//Title:      Using SQLJ with Java Stored Procedures
//Author:     Megh Thakkar
//Description: This Java program modifies the employee record by
//giving the employee a raise and changing the employee's location package modify_emp;

import Java.sql.*;
import Java.io.*;

class maintain emp
{
    public static void main (String args [])
        throws SQLException, IOException
    {
        String cstring = args[0];
        String uname = args[1];
        String pwd = args[2];

        //Load JDBC Drivers
        DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver( ));

        // Connect to the database using the specified parameters
```

```
Conection conn = DriverManager.getConnection (cstring, uname, pwd);

//Prepare to call the stored procedure
CallableStatement cstmt = conn.prepareCall ( " { ? = call
    modify_emp_update_emp (?, ?, ?, ?) } " );

// We are using positional arguments
// The first argument is declared as an OUT parameter
cstmt.registerOutParameter (1, Types.INTEGER);

//The second argument is SCOTT
//The third argument is the raise
// The fourth argument is the city
cstmt.setString (2, "SCOTT");
cstmt.setInt (3, 50000);
cstmt.setString (4,"MIAMI");

// Execute the command
cstmt.execute ();

//the first argument returns the new salary
int new_salary = cstmt.getInt (1);

System.out.println ("The new salary for SCOTT is: " + new_salary);

}
```

USING THE JDBC DRIVER WITH JAVA APPLICATIONS

```
//Title:      Using JDBC
//Author:     Megh Thakkar
//Description: This Java program quires the emp table to find the
// location of each employee

package package1;
```

```
import Java.sql.*;

public class find_location

{

    public static void main (String args[ ] )

        throws SQLException


    {

        String cstring = args [ 0 ] ;

        String uname = args [ 1 ] ;

        String pwd = args [2];



        // Load JDBC Drivers

        try

        {

            Class.forName ("oracle.jdbc.driver.OracleDriver");

        }

        catch (ClassNotFoundException e)

        {

            System.out.println ("Unable to load driver");

        }





        //Connect to the database using the connection parameters

        Connection conn=DriverManager.getConnection (cstring, uname, pwd);





        // Create the employee locations and store

        Statement stmt = conn.createStatement( ) ;





        //Query the employee locations and store

        //the output in the result set

        ResultSet rset = stmt.executeQuery ("select empno, ename,city from emp");

        while (rset.next( ) )

        {

            //Get Employee Number

            int eno = rset.getInt (1);
```

```
//Get Employee Name  
String ename = rset.getString (2);  
// Get Employee city  
double loc = rset.getString (3);  
System.out.println (eno + " " + ename + " " + city);  
}  
conn.close ();  
}  
}
```

22.5 Oracle Internet File System

It has been found that most companies store very little information in relational databases. Most of the information is now held in file systems. File systems basically fragment information such as documents, Web content, and so on, into separate data stores, which have their own storage and access methods. The current technique of storing information in file systems has several problems:

- *End users* - End users have to install multiple clients, each with its own configuration set to access the information in the separate data stores. Users also find it extremely difficult to keep track of the various datatypes associated with the project they're working on.
- *System administrators* - System administrators have a difficult time maintaining security, performance, and configuration; providing access to users; and maintaining the integrity of the data, which may be spread across multiple machines and data stores.
- *Application developers* - Application development is slow because of the necessity for developers to understand different operating system peculiarities; understand different APIs to access files, Web content, and relational data; and keep track of all the components of the project.

One approach to simplifying the problems associated with storing information in file systems is to move as much data as possible into a relational database such as Oracle.

However, this approach also presents several problems.

- Certain types of information, such as email and simple files, aren't efficiently used and manipulated using a relational database.
- End users have to be more "SQL aware."
- In a regular file system, you double-click a file to access its contents. For a file stored in a relational database, you have to download the file, edit the file, and upload it back to the database.

Oracle8I provides the Internet File System (IFS) to merge the file system with the database. IFS is a database file system (DBFS) that stores the data in the database, rather than leave the data outside the database and provide links from the database to access the information. Placing data within the database provides several benefits:

- *Simplified data security* - Clients can access the information by using multiple protocols and client interfaces. To access IFS, you can use the following protocols:

SMB Allows users to drag and drop files. Files can be edited within IFS. Access is possible through Windows 95, Windows 98, and Windows NT.

HTTP Used for Web browsers.

FTP Access to IFS is through FTP. The contents of IFS are viewed as FTP directories from which such as Microsoft Outlook and Eudora.

SMTP,IMAP4, POP3 Access to IFS is through email clients such as Microsoft Outlook and Eudora.

- Application developers can use a common API for programming against the database.
- Advanced search capabilities can be used to search the contents of the DBFS using SQL and provide much better search performance compared to file systems.
- Database-level security is provided to simplify administration.

- Check in and check out facilities simplify the management of information and increase productivity.
- Enhanced scalability can ease data administration
- Document parsing and rendering make it possible to view the same document in different formats.
- Data within the database is tightly integrated with the Oracle Enterprise Manager, ConText data cartridge and Oracle8I features such as objects.
- IFS messaging can be used to simplify administration by sending email to the system administration when important events occur.
- You can write and use Java stored procedures and EJBs
- Version control is provided so that a new version of a file can be created when you copy or edit a file. One version can be identified as "official" and will be used by all users.
- Automatic purging of files after a specified "expiration" period simplifies data management.

Oracle IFS runs as a Java application that runs inside a Java machine in Oracle8I, providing a flexible and easy-to-use mechanism for storing and manipulating the data. Multiple protocol can be used to access the data. Application development is achieved by using the programming APIs provided by Java, **CORBA**, and **PL/SQL**.

IFS uses a "folder" strategy to make itself appear as a mounted network drive. The contents of IFS appear in folders and subfolders, which can be easily manipulated just like a file system. In addition to the manipulation of folders provided by IFS, you can store relational and non-relational data in the same folder.

Relational data also appears as files, and you can construct a document by merging relational and non-relational data. The documents can further appear as files.

Another important characteristics of an IFS "file" is that it can have multiple parent directories. For example, an email document (which appears as an IFS file) can be part of two separate folders.

Several additional features are planned for the future release of IFS.

- Support for NFS
- Application development tools for IFS
- Portable IFS
- "File system" tablespaces
- Integration with SQL

22.6 Short Summary

- ➲ Java is a key component of the Oracle8i strategy, and this chapter discusses the various ways Oracle has implemented Java. The Java Virtual Machine makes it a very easy environment to run Java applications. You have seen how to write Java stored procedures and the use of Enterprise JavaBeans.
- ➲ Oracle provides two types of drivers to allow Java programs to access the database: thin drivers and JDBC/OCI drivers. No matter which driver you use, the code is still the same, but you use different connect strings.

22.7 Brain Storm

1. Which type of input source we used in oracle8I?
2. How to use Jdeveloper explain?
3. What are the protocols we need to access IFS?



Lecture 23

Introduction to Forms

Objectives

After completing this lesson, you should be able to do the following

- ❖ Identifying the main form builder executables
- ❖ Identifying the main components of form builder
- ❖ Identifying the main objects in a form module
- ❖ Control forms across several windows and database transactions

Coverage Plan

Lecture 23

- 23.1 Snap shot
- 23.2 Introduction to forms
- 23.3 Using forms builder
- 23.4 Form wizards
- 23.5 Creation of forms
- 23.6 Short summary
- 23.7 brain Storm

23.1 Snap Shot

In this session introduces you to the oracle developer. This lesson teaches you to how to build effective and professional form applications using oracle developer form builder. This is only one component of oracle developer, and it is useful for you to understand the whole suite of components and the capabilities of oracle developer in building consistent, integrated applications.

Oracle developer is a suite of oracle components that you can integrate to build comprehensive applications.

Oracle developer is one group of a family of products that support the design, creation, and running of applications across a variety of platforms.

Oracle developer enables you to build high- performance systems that take advantage of graphical user interface, database, client-server, and web technologies.

23.2 Introduction to forms

As commercial application developers, our primary task is to design data-entry forms that look as close to the printed to the printed sheets of paper that the data entry operators use in the current manual system of data collection.

Hence when the data entry operators switch from paper/pen to keyboard/VDU what they see on the VDU will look familiar to them.

Oracle Forms

Oracle Forms Builder provides a powerful 'Graphical User Interface' to design such forms. All objects, properties, triggers can be selected by simply clicking on an appropriate icon. A Forms wizard can be used to quickly create forms. This tool allows commercial application developers to design forms that will capture, validate and store data with the very minimum of coding.

Forms Builder, Oracle's GUI based forms creation tool comprises of the following components:

- Forms Builder
- Forms Compiler
- Forms Runtime

Forms Builder is what is used to create a form. The design and layout of data entry screens, the creation of event driven PL/SQL code used for data validation and navigation can be done via Forms Builder.

Forms Compiler is required to compile the file created in Forms Builder and create a binary file, which can be executed by Forms Runtime.

Forms Runtime is used to run the compiled code created by Forms Compiler. To run an Oracle form only the Forms Runtime module is required. At the time of software deployment only the Forms Runtime module needs to be installed on machines on which forms created by Forms Builder are deployed.

To create, add program code, test/debug on Oracle form, a complete installation of Oracle Forms Builder is required.

Application Development in Forms 5.0

Applications built using Oracle Forms Builder will contain the following components;

1. Form Module
2. Menus
3. PL/SQL Libraries
4. Object Libraries
5. Database Objects

Form Module:

The primary object created using Form Builder is a form. The Form module is nothing but a collection of objects such as blocks, canvas, frames, items and event based PL/SQL code blocks called triggers housed in an MS Windows 'Command' Window.

Menus:

The menu module is a collection of objects such as menu items, sub menus, sub menu items and PL/SQL code blocks.

PL/SQL Libraries:

The library module is a collection of PL/SQL functions and procedures stored in a single library file. This library file is then attached to a form/menu module. All other objects in the form or menu can now access and share the collection of PL/SQL functions and procedures.

Object Libraries:

In a development environment, standards need to be set and maintained. Standards can be maintained by developing common objects, which can be reused throughout a commercial application as well as across applications.

Object Libraries provides an easy method of creating and storing reusable objects and enforcing standards.

An Object Library can be created to:

- Create, store, maintain and distribute standard and reusable objects.
- Rapidly create applications by dragging and dropping predefined objects on to form.

Database Objects:

Oracle's interactive tool. i.e. SQL *Plus allows the creation of Database objects like Stored Procedures, Stored Functions and Database Triggers using appropriate SQL and PL/SQL syntax.

Form Module

A Form module consists of the following components:

- Blocks
- Items
- Frames
- Canvas Views
- Windows
- PL/SQL Code blocks

Blocks:

A form contains one or more blocks. Blocks are logical containers and have no physical representation. Only the items contained in a block are visible in the form interface.

A block can be conceptualized as a parent container object that holds a related group of child objects such as text items, lists, and push buttons, etc for storing, displaying, and manipulating table data. Each block has a set of properties that determine the behavior of the block.

A block connected to a database object is called a '**Data Aware**' block. A block not connected to any database object is called '**Control Block**'. A block can be connected to a database object like a table, view or synonym. A block can also be connected to 'Stored Procedures'

Each Data Block can be directly related to a (single) database table, view or synonym. A table, view or synonym connected to a block is known as the '**Base Table**'. Each column of the base table may have an associated block item bound to it. The association or binding of block items with table columns is done by giving the item on the block the same name as the table column. This direct relationship allows the data in a base table to be manipulated at will.

Blocks have several characteristics that can be defined. Some of these are the attributes that determine how to sort information retrieved into a block from the base table, the number of records that can be displayed in the block, etc.

Blocks can be related to each other by specifying a master-detail relationship. A master-detail relationship corresponds to a primary-foreign key relationship between the base tables of the blocks. Whenever a row is retrieved in the master block from the master table, the master detail relationship automatically displays the corresponding set of rows from the detail table in the detail block, without any special processing code being required.

Items:

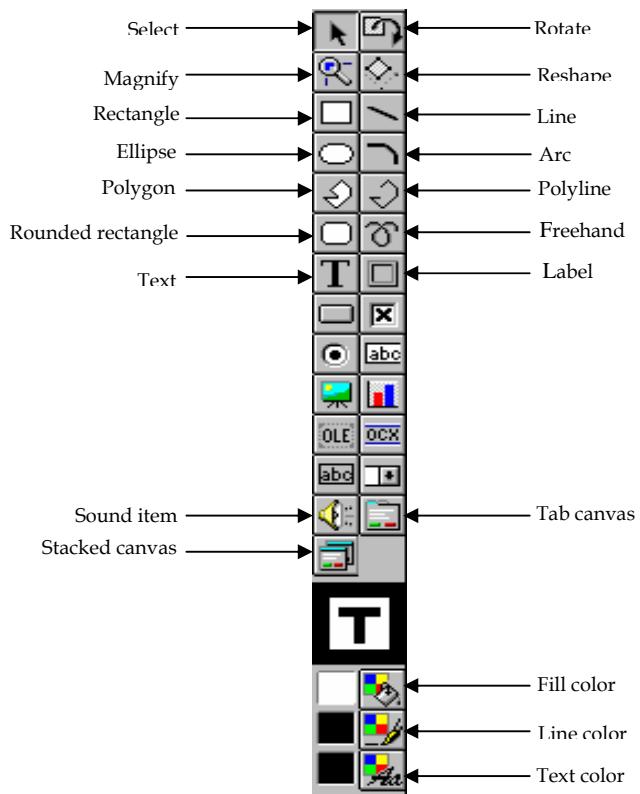
Items are objects contained in blocks. At the most basic level, items serve as containers for data. The user can manipulate values in an item. A procedure or a trigger can also be used to manipulate item values. An item is always associated with a block. Each block normally has one or more items.

The items in a block are usually bound to columns in a base table. Entering data into such items will determine the values entered in associated columns of the base table. Alternatively, items can be filled with data from the base table by performing an SQL query on the base table.

Items need not always be bound to the columns of a base table. They can also hold calculated values, display related information from associated tables or accept operator input for processing later. Items not connected to a base table are called '**Control Items**'.

Items have attributes that can be set via the item's property palette. These attributes may be set dynamically at runtime via suitable code as well. These attributes determine an item's behavior at form run time.

Oracle Forms Builder supports several types of items that can be used to build a form. The items supported by Oracle Forms Builder are described below:



Text Item



Text item displays data values, and can be edited.

Display Item



The display item shows information that must be fetched or assigned programmatically. These items are not navigable to and their contents cannot be directly edited.

List Item



A list item displays a list of choices from which only one value can be selected at a time.

Button



A button is a rectangle with text label or an icon graphic. These items are normally used to initiate some action.

Check Box



A check box is a text label with an indicator that displays the current value as checked or unchecked. It is normally used as an item, which takes in a yes/no or true/false like value. This item is normally bound to a table column, which must hold a single character value or a boolean value.

Radio Group



A radio group of two or more radio buttons, from which only one radio button can be active. This item is used on a form to hold several radio buttons which are bound to a single table column. A technique commonly used when multiple options are available, with only a single value being stored in a table column.

Chart Item



A chart item is a bordered rectangle of any size that displays a chart generated by the Oracle Graphics Tool.

Image Item



An image item is a bordered rectangle of any size that displays images stored in a database or in a file. Image items are dynamic and can change with the image being displayed when required.

OLE Container



An OLE container is an area that stores and displays an OLE object. OLE objects are created for OLE server applications. OLE objects can be embedded or linked in an OLE container.

This feature is available for O/s that supports Object Linking and Embedding. Object Linking and Embedding is currently available on Microsoft Windows and Macintosh platforms.

OCX Control



An OCX control is a custom control that simplifies the building and enhancing of user interfaces. OCX Controls are available for forms running on Microsoft windows, as Microsoft Windows is the only O/s that supports OCX controls.

Frames:

A frame is a graphic object that appears on a canvas. Frames are used to arrange items within a block.

- A frame is an object with properties.
- Each frame can be associated with a block.
- When a frame is associated with a block, the items in the block are automatically arranged within the frame.
- Frames can be sub-classed or included within an Object Library to enforce visual standards.

Canvas View:

The canvas-view is the ‘background’ on which Items, Text and Graphics or Frames are placed. The items in a block can be placed on different canvas-views. Each canvas-view can be displayed in different windows.

The Oracle Forms, Block wizard allows items to be placed on different types of Canvas like ‘Content’ and ‘Tab’.

Window:

Every new form is automatically held in a default window named WINDOW 1. Additional windows can be created as needed by inserting them in the Object Navigator under the ‘Windows’ node.

For each window created, at least one canvas-view must be created. The canvas-view is the background on which items are placed. The canvas-view is linked to a window by setting the ‘Window’ canvas-view property appropriately.

Window properties can be set at design or runtime, which determine the appearance and functionality of the window at runtime.

PL/SQL Code Blocks:

Data manipulation can be done using the default form created by the Oracle Forms wizard. Forms created by the Forms Wizard can be customized to suit a commercial application or to map to the standard set for the application.

The Oracle Forms tool provides suitable facilities to attach PL/SQL code blocks to ‘Events’ built into all Form objects. This allows the form to map to the object oriented standard of ‘Event driven’ form processing. ‘Events’ can be conceptualized as PL/SQL code holders that automatically execute the PL/SQL code when a specific event occurs. Based on the needs of the application, an appropriate event must be selected and PL/SQL code in its rigid format must be written in the selected event. For example if data needs to be validated to ensure input / output or business rules, PL/SQL code must be written in the WHEN-VALIDATE-

ITEM event. PL/SQL code written in this event gets executed when item navigation occurs after data entry.

Every form contains at least one block, one window, one canvas and one or more items. Each item on the form has a set of attributes or characteristics, which determine how the item behaves at run time. Form items are named so that code in PL/SOL blocks used for form processing can reference them.

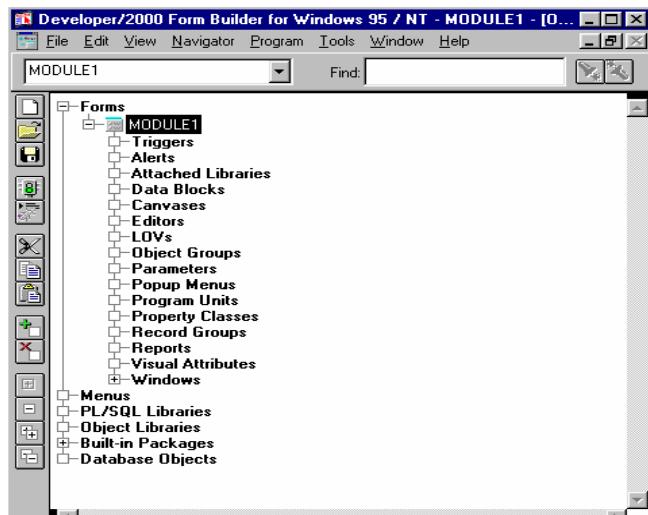
23.3 Using forms builder

A general understanding of the tools available with Forms Builder will make life a lot easier when actually creating a form via Forms Builder. The tools are:

1. Object Navigator
2. Property Palette
3. Layout Editor
4. PL/SOL Editor
5. Menu Editor

Object Navigator:

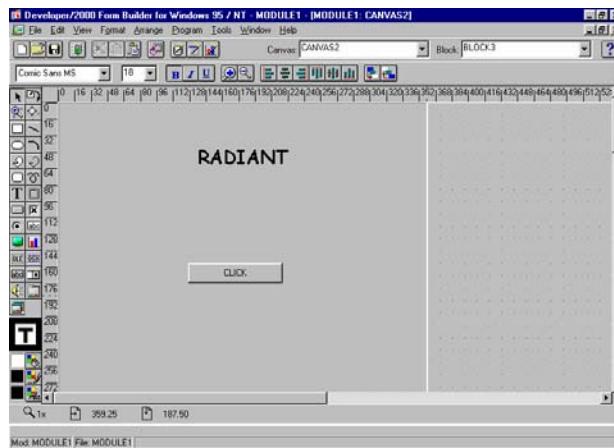
Blocks, items, frames, canvas and windows are constituent parts of a form. They are called objects. While developing forms, several objects are created. Each of these objects may have their own set of objects beneath them. The object Navigator displays each object created and allows navigation through this hierarchy of objects.



Property Palette:

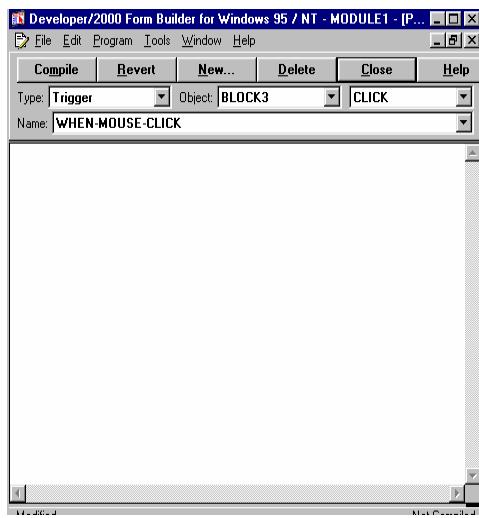
Property Palette allows the examination of the properties of each object. It also allows the setting of properties for the object.

Layout Editor



Forms developed will have different objects like text items, check boxes, radio buttons, labels, graphic to name a few. The Layout editor allows the sizing, positioning and alignment of these objects. Font characteristics can also be set through the layout editor.

PL/SOL Editor:



In Commercial Application, application specific tasks need to be performed. To achieve this appropriate PL/SOL code must be written and attached an event, belonging to a Form object. The PL/SOL editor provides the interface via which this is done.

Menu Editor:

This tool is used to create defined menus. Forms Builder automatically provides a default menu with default functionality like querying, inserting, deleting records and navigating through different records. However for system specific functionality a user defined menu which overrides the default Oracle Menu needs to be created. This is done via the Menu Editor.

23.4 Form Wizards

Forms builder provides a number of wizards for creating different objects on a form. Some of these wizards are:

Data Block Wizard:

The Data Block Wizard allows the creation of data blocks. The data block wizard displays a welcome dialog box and then asks for additional information to create the data block. The information required to create a data block will be Data Source. I.e. Table or Stored Procedure.

If Table is selected as a data source, the Table Name and the Field Names must be specified. If a Stored Procedure is selected as a data source, the Procedure Name must be specified.

The Data Block wizard then leads to a Layout wizard

Layout Wizard

The Layout Wizard works in conjunction with the Data Block wizard. The Layout Wizard allows the layout of the data blocks objects to be designed.

The Layout Wizard displays a welcome dialog box and then asks for information about the canvas that must be used in the layout. The canvas information required is canvas name and

canvas type. A list of data block s and the columns included in the block are displayed. A specific canvas item must be bound to each table column selected.

Each item associated with a table column and displayed on the form generally includes a prompt label that gives information on the type of data to be loaded into the item. A label or prompt along with its height and width can be specified for each table column.

The layout style I e, Form or Tabular is then specified. The Layout style determines how items will be arranged on the form.

Items are grouped inside a frame. The frame title, number of records to be displayed and the distance between the records needs to be given to the layout wizard.

The form layout is then created based on the information specified in the Layout wizard. The form so created so default form. This form can be run to perform data manipulation operations such as View, Insert, Update and Delete.

Property Palette

You can modify the appearance and behavior of a data block after it has been created. To do this use one of the following methods:

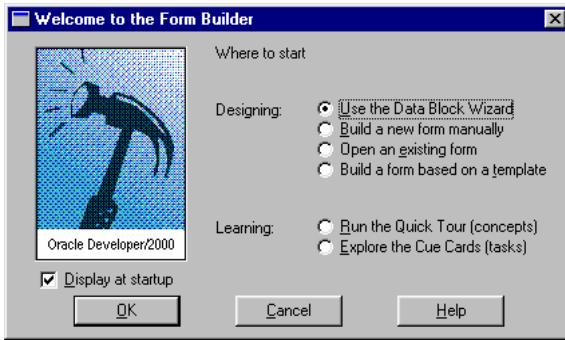
- Reentrant wizards: Reenter the Data Block Wizard or the Layout Wizard as described in the previous lesson to modify the data source and visual presentation of objects within the data block.
- Layout editor: Invoke the layout editor and make your modifications manually.
- Data Block Property Palette: Open the Data Block Property Palette and change individual property values to modify the behavior of the data block at run time.
- Frame Property Palette: Open the associated Frame Property Palette and change individual property values to modify the behavior of the data block at run time.
- Frame Property Palette: Open the associated Frame Property Palette and change individual property values to modify the arrangement of items within the data block.

23.5 Creation of Forms

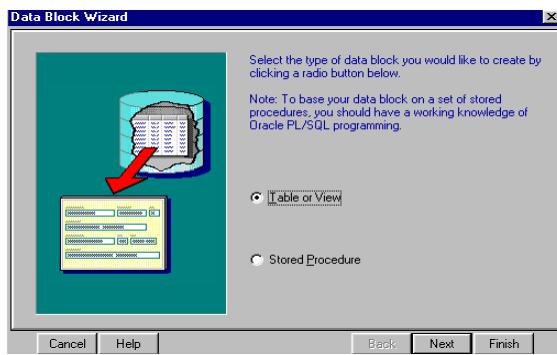
For creating a form when we open a developer 2000. Opening screen allows the use of wizards to

- ▲ Create new forms
- ▲ Open existing forms
- ▲ Build a form from a template or
- ▲ Learn more about forms builder
- ▲ If the data block creation option is selected, an appropriate wizard is displayed
- ▲ If manual form creation is specified, a new form will be created.

The window is like the below figure.



For this when we choose the data block wizard the next screen will be like this.



In this above figure we have two options 'table or view' or 'stored procedures'. For this we choose the table or view option.

In the next window we choose any table for using the browse button. For creating a form we connect the developer 2000 to oracle for giving the user name and password. After connecting the oracle database in available columns it will display all the fields in that table for this we choose the needed columns and place it into database item columns for using the below buttons.



Used to include a single column.



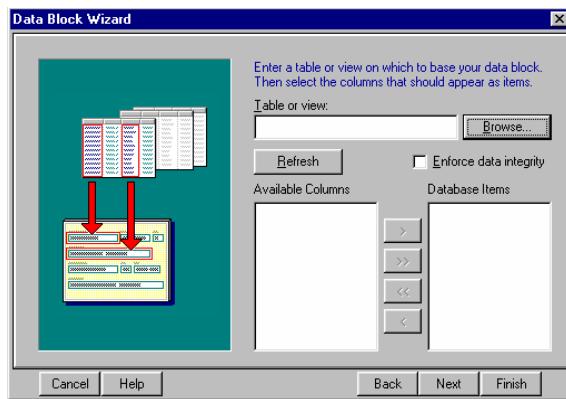
Used to include multiple columns or all columns.



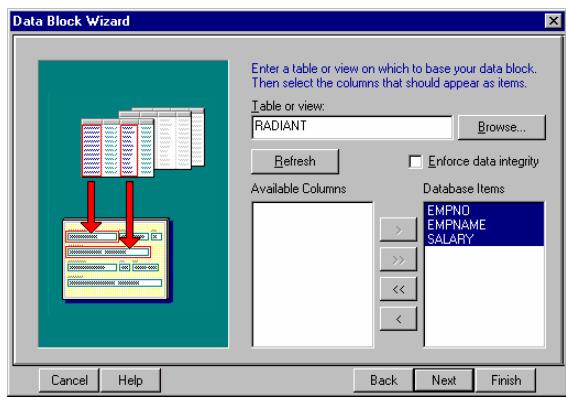
Used to remove single columns.

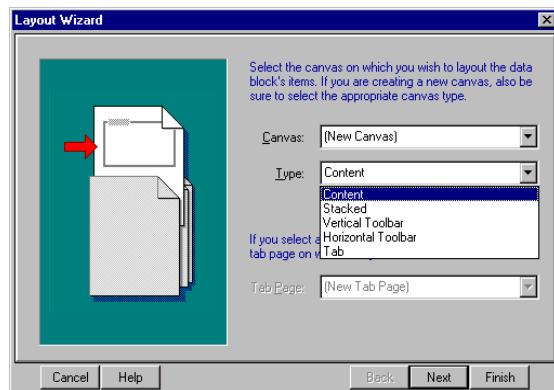
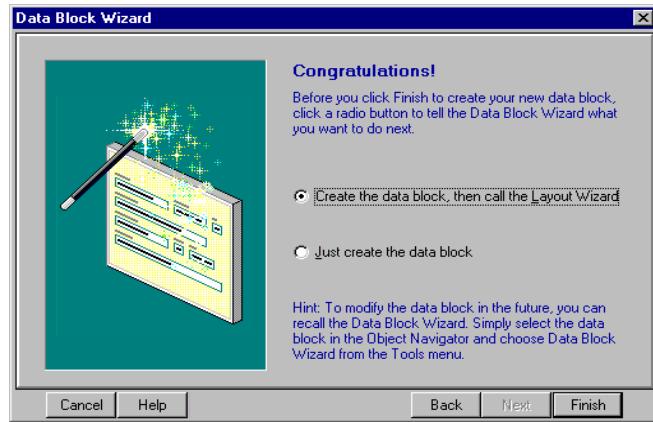


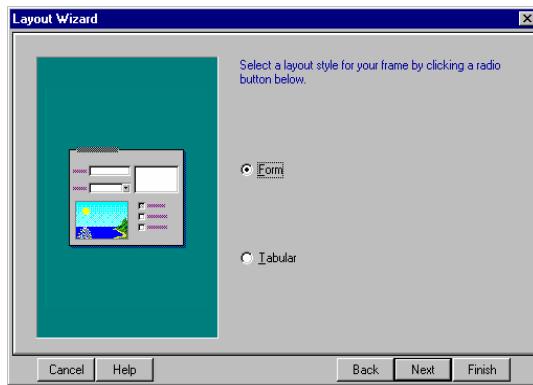
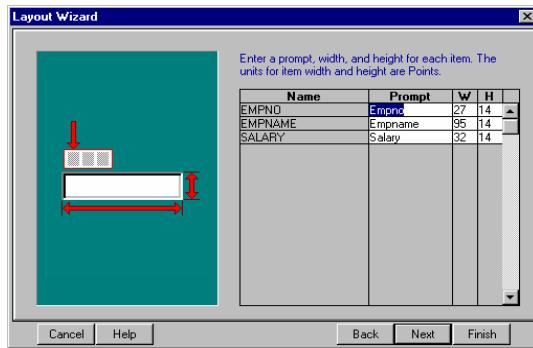
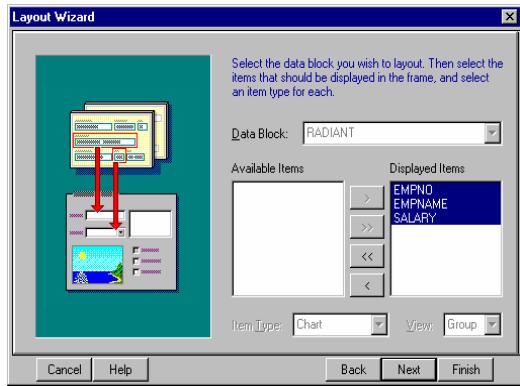
Used to remove multiple columns or all columns.

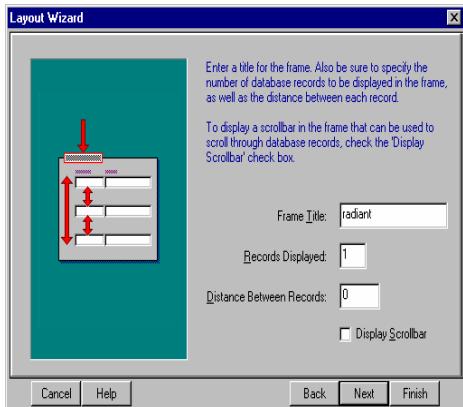


The given figures are denote the creation of form there are nine figures for step by step creation of form for using these windows we can create both data block wizard and layout wizards for this we give the table name and field name for creating the form.









23.6 Short Summary

- ♣ Form builder can present information through textual items, GUI objects, and bitmapped images. User can perform database transactions by interacting with these objects.
- ♣ Wizards help you build simple, standard modules very quickly.
- ♣ Oracle developer provides a suite of components with common features and a common builder interface for form, report and graphic charts.
- ♣ Each oracle developer component has a set of preferences that you can alter for the current and subsequent builder sessions.
- ♣ The product provides a common builder interfaces, including object navigator, layout editor, PL/SQL editor and property palette components, and it offers a comprehensive online help system.

23.7 Brain Storm

1. Explain about Oracle developer?
2. Explain the components of oracle developer?
3. How to create a form explain it?
4. Explain about property palette.

8008

Lecture 24

Property Class

Objectives

After completing this lesson, you should be able to do the following

- Discuss about Property class
- Describe visual attributes
- Describe about library
- Simplifies reuse in complex environments
- Create and control alerts
- Describe about object libraries
- Simplifies the sharing of reusable components
- Create editors and associate them with text items in a form module

Coverage Plan

Lecture 24

- 24.1 Snap shot
- 24.2 Property class
- 24.3 Visual attributes
- 24.4 Library
- 24.5 Alerts
- 24.6 Object libraries
- 24.7 Editors
- 24.8 Short summary
- 24.9 Brain Storm

24.1 Snap Shot

In this session we are working with property classes and visual attributes and differentiate property class and visual attributes. In visual attributes defining a visual attribute object and associating visual attributes objects to the items on the form. In this lesson we learn about assigning property class to other objects on the form.

24.2 Property Class

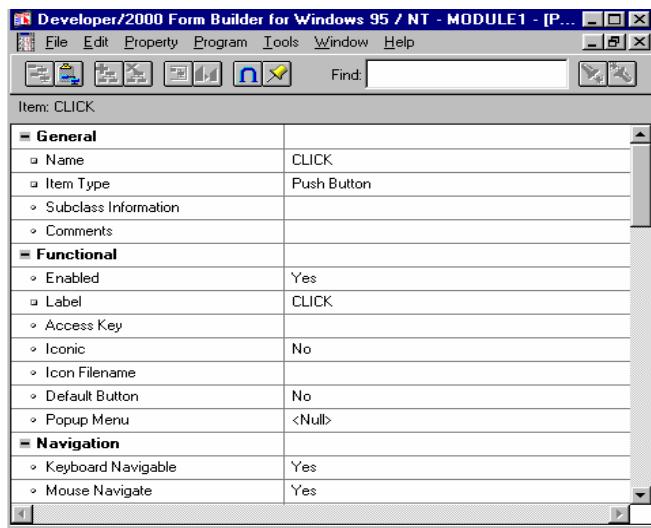
Property class inheritance is powerful feature that allow you to quickly define objects that conform to your own interface and functionality standards. Property classes also allow you to make global changes to applications quickly. By simply changing the definition of a property class, you can change the definition of all objects that inherit properties from that class. A property class is a named object that contains a list of properties and their settings. Once you create a property class you can base other objects on it.

For example, you might define separate property classes for enterable items, required items, required items, read-only items. When you want to change the appearance and functionality of all such items in an application, you need only update the properties in the property class.

There can be any number of properties in a property class, and the properties in a class can apply to different types of objects. For examples, a property class might contain some properties that are common to all types of items, some that apply only to text times, and some that apply only to check boxes.

Property classes are separate objects, and, as such, can be copied between modules as needed. Perhaps more importantly, property classes can be referenced in any number of modules.

Every object in a form module, as well as the form module itself, has properties that dictate the object's behavior. When an object is first created, it is automatically assigned several property values by default. You can change by default. You can change these property values in the Property Palette.



Displaying the Property Palette

To display the Property Palette of an object, use any of the following methods:

- ☒ Select the object in the Object Navigator and then select ->Tools Property Palette from the menu system.
- ☒ Double-click the object icon for the object in the Object Navigator (except for code objects and canvases).
- ☒ Double-click an item in the Layout Editor.
- ☒ Select the object in the editor or the Object Navigator, and then click the right mouse button. From the pop-up menu, select the Property Palette option.

Property Palette Features

Feature	Description
Property list	<p>The property list displays a two-column list of property names and property values. Properties are grouped under functional headings or nodes.</p> <p>You can expand or collapse a node by using the plus and minus</p>

	icons beside the node name.
Find field	The Find field enables you to quickly locate the name of a particular property. The Search Forward and Search Backward buttons enhance your search.
Toolbar	The toolbar consists of a series of buttons that provide quick access to commands.

Using the property palette

Each form object has various types of properties. Properties are manipulated differently, depending on the property type. Here is a summary of controls used in the property palette:

Property Control	Description
Text field	This is displayed when the current property can be set by entering a text value. For longer text values, an iconic button also appears, enabling you to open a text editor.
Poplist	This occurs where the property is yes or no, or where a fixed set of values are allowed. Click the down arrow to open the list and select a value. Alternatively, double-click the property name to cycle through the values.
LOV Window	LOVs occur where a potentially large list of possible values is available. Click the iconic button in the property value column to invoke an LOV.
More button	Use this when more complex settings are needed. Click the More button to open the extra dialog.

Property Palette Icons

Each property in a Property Palette has an icon to its left. Here is a summary of these icons and their description:

Icon	Description
Circle	Specifies that the property value is the default value
Square	Specifies that the property value has been changed from the default

Arrow	Specifies that the property value is inherited
Arrow with across	Specifies that the property value was inherited but has been overridden

Property classes are similar to named visual attributes, but there are important differences you should be aware of:

- Named visual attributes define only font, color, and pattern attributes; property classes can contain these and any other properties.
- You can change the appearance of objects at runtime by changing the named visual attribute programmatically; property class assignment cannot be changed programmatically.
- When an object is inheriting from both a Property class and a named visual attribute, the named visual attribute settings take precedence, and any visual attribute properties in the property class are ignored.

Create Property Classes:

You can define property classes in form and menu modules. In the Object Navigator, property classes appear under the Property Classes node in the appropriate module.

There are two ways to create a property class. You can create a new property class in the Object Navigator and then add properties to it as desired. Or, you can create a property class from an existing list of properties in the Properties window.

To Create a Property class in the Object Navigator:

1. In the Object Navigator, position the cursor on the Property Classes node and choose Navigator → Create

A property class object with a default name is inserted under the node. The new property class does not yet contain any properties.

2. In the properties window, add properties to the class as desired.

To Create a property class in the Properties window:

1. In the Object Navigator or an editor, select one or more objects so that the Properties window displays the properties that you want to be in the property class.
2. In the properties window, click the Property Class button on the toolbar. An alert is displayed to confirm the name of the class being created.

Adding and Removing Properties from a Property Class:

Once you have created a property class, you can add and remove properties from it as desired. You can add properties one at a time, or with the Copy/Paste properties commands.

To add a single property to a property class:

1. In the Object Navigator, select the property class so that its properties are displayed in the Properties window.
2. In the Properties window, click the Add Property button on the toolbar. The Properties LOV appears, showing the list of all Oracle Forms properties.
3. Select the property you want to add to the property class.
4. In the Properties window, set the new property as desired.

To copy properties into a property class:

Use the Copy Properties command to copy the properties of existing objects, then use the Paste Properties command to paste those properties and their settings to the property class.

To remove a property from a property class:

1. In the Object Navigator, select the property class so that its properties are displayed in the properties window.
2. In the properties window, click on the property you want to remove to select it.
3. Click the delete property button on the properties window toolbar. Basing an Object on a property class. Note: you must already created the property class you can base objects on it.
4. In the object navigator, select the property class so that its properties are displayed in the properties window.
5. In the properties window, set the class property by selecting the desired property class from the list of all classes defined in the active module.

When you base an object on a property class, the properties of the object are updated as follows:

- Properties for which there is a corresponding property in the property class are updated to match the property class setting.
- Properties for which there is no corresponding property in the property class are left unchanged.

24.3 Visual Attributes

Visual attributes are the font, color, and pattern properties that you set for form and menu objects.

A visual attribute is another object that you can create in the Object Navigator with properties such as font, color, and pattern combinations.

When creating visual attributes, you can use the Font Picker and Color Picker to select the font and color. When changing a font from the Property Palette, click the Font group itself to invoke the Font Picker.

Every interface object in a forms application has a property called Visual Attribute Group, which determines how the individual visual attribute settings of an object are

derived. The Visual Attribute Group property can be set to Default, NULL, or the name of a named visual attribute object.

Partial Visual Attributes

Partial visual attributes are just like visual attributes, except that you need to set only the properties that you want to be inherited by the objects that use them. This means that you can apply a visual attribute that changes the font color without having to set the font name.

24.4 Library

A PL/SQL library is a collection of program units, including user-named procedures, functions and packages. PL/SQL Libraries provide a convenient means of storing client-side program units and sharing them among multiple applications.

Once a PL/SQL library is created, it can be attached to any form or menu. The program units stored in the PL/SQL library can be called from triggers, menu item commands, and user-named routines written in the form or menu module to which the PL/SQL library attached.

The same PL/SQL libraries can also be attached to other PL/SQL libraries. When a PL/SQL library is attached to another PL/SQL library, program units within the attached PL/SQL library can be referenced in the current library.

PL/SQL libraries support dynamic loading. Program units are loaded into a computer's memory only when an application needs it. This can significantly reduce the runtime memory requirements of an application, but tends to slow an application. The trade off is between speed Vs. memory usage.

Library File Formats

There are three library file formats, .PLL, .PLX, and .PLD.

1. .PLL

A library .PLL file contains both library source code and the compiled, platform specific p-code (executable code). The .PLL file is created or updated when it is saved in the library module. When the library module is saved, the changes are reflected in each module to which the library is attached.

2. .PLX

A library .PLX file is a platform-specific executable. When an application is ready to be deployed, there would be a need to generate a version of library files that contain only compiled p-code.

3. ,PLD

The .PLD file is a text format file, and can be used for technical documentation of the library files.

Creating And Attaching A Library To A Module

Opening, defining, compiling and saving library modules, creates a library.

One can define the following types of PL/SQL subprograms in a library:

- Procedures
- Functions
- Package specifications
- Package bodies.

Creating a new library

1. In the navigator, to create a library, choose menu items **File..New..PL/SQL library** or select the Libraries node and then click on **Navigate..Create**.
2. To create a program unit, expand the desired library node, select the *Program Units* node, and then click **Navigator..Create**. The *New Program Unit* dialog appears.
3. Specify the *Program Unit Name* and its Type (Procedure, Function, Package Spec, or Package Body). Click on OK. The *PL/SQL Editor* is displayed.

4. In the PL/SQL Editor, define appropriate program units and then click on **Compile** to compile and apply modification. Click on **Close** to close the editor.
5. Choose **Program..Compile..Incremental** to compile any uncompiled library program units, or click on **Program..Compile..All** to compile all library program units.
6. Click on **File..Save** to save the library module to a file or to the database.

24.5 Alerts

Commercial applications will generally encounter two types of error conditions. Those that are system generated and those that arise out of business rules being violated by the data being captured. In either case the user of the commercial application must be informed of what occurred and if required be given information on how to correct the error condition. *For example, if the user has entered the cost price of a product as 0, then the system must display an error message as 'Cost Price cannot be 0'.*

Often choices are given on how to actually process an event e.g. if the delete button is clicked the system must ask for a *confirmation to delete the current record*. If the user clicks on the 'Yes' or 'OK' button, the record must be aborted.

Oracle Forms Builder allows creation of an object called an 'Alert' that makes displaying an error message and/or offering different processing choices quite simple and elegant.

An alert is a modal window that displays a message. Alerts are used to inform the user of unusual situations or to give a warning for an action that might have undesirable or unexpected consequences.

There are three styles of alerts: **Stop, Caution, and Note**. Each style denotes a different level of message severity. Message severity is represented visually by an icon that is displayed in an alert window.

Oracle Forms Builder uses many built-in alerts that display pre-defined messages. Custom alerts created by a user can also be displayed in response to application-specific events.

When an event occurs that causes an alert to display, the alert must be responded to by selecting one of its predefined *Alert Buttons*. Selecting any button immediately dismisses the alert and passes control back to the program that called it.

When an alert is created, basic information such as the message to be displayed by the alert and the text labels for the alert buttons must be specified.

After creating an alert, an appropriate trigger or user-named routine must be written to display the alert in response to a particular event. In addition, the action that each button initiates is determined by the PL/SQL code written in the trigger or user-named routine.

An Alert can be displayed by the SHOW_ALERT() function. The SHOW_ALERT () function returns alert_button1, alert_button2 or alert-button3 based on the button clicked by the user. This return value can be used to determine what action that has to be taken.

Creating and Controlling Alerts

- Display a message that the operator cannot ignore, and must acknowledge.
- Ask the operator a question where up to three answers are appropriate (typically Yes, No or Cancel).

You handle the display and responses to an alert by using built-in subprograms. Alerts are therefore managed in two stages:

- Create the alert at design-time, and define its properties in the Property palette.
- Activate the alert at run time by using built-ins, and take action based on the operator's returned response.

Create an Alert

Like other objects you create at design-time, alerts are created from the Object Navigator.

1. Select the Alerts node in the Navigator, and then select Create.
2. Define the properties of the alert in the Property Palette.

Here are the properties that are specific to an alert. This is an abridged list.

Property	Description
Name	Name for this object
Title	Alert title
Alert Style	Defines the symbol that accompanies message: Stop, Caution, or Note
Button1, Button2, Button3	Labels for each of the three possible buttons (Null indicates that the button is not required.)
Default Alert Button	Button 1, Button 2, or Button 3
Message	Message that will appear in the alert (maximum 200 characters)

Planning Alerts: How Many Do You Need?

Potentially, you can create an alert for every separate alert message that you need to display, but this is usually unnecessary.

You can define a message for an alert at run time, before it is displayed to the operator. This means that a single alert can be used for displaying many messages, providing that the available buttons are suitable for responding to each of these messages.

Create an alert for each combination of:

- Alert style required
- Set of available buttons (and labels) for operator response

For example, an application might require one Note-style alert with a single button (**OK**) for acknowledgment, one Caution alert with a similar button, and two Stop alerts that each provide a different combination of buttons for a reply. You can then assign a message to the appropriate alert before its display, through the **SET_ALERT_PROPERTY** built-in procedure.

Controlling Alerts at Run Time

There are built-in subprograms to change an alert message, to change alert button labels, and to display the alert, which returns the operator's response to the calling trigger.

SET_ALERT_PROPERTY Procedure

Use this built-in to change the message that is currently assigned to an alert. At form startup, the default message (as defined in the Property palette) is initially assigned:

`SET_ALERT_PROPERTY ('alert_name', property, 'message')`

Parameter	Description
Alert_name	The name of the alert, as defined in the Designer (You can alternatively specify an alert_id (unquoted) for this argument.)
Property	The property being set (Use ALERT_MESSAGE_TEXT when defining a new message for the alert.)
Message	The character string that defines the message (You can give a character expression instead of a simple quoted string, if required.)

SET_ALERT_BUTTON_PROPERTY Procedure

Use this built-in to change the label on one of the alert buttons:

`SET_ALERT_BUTTON_PROPERTY ('alert_name', button, property, 'value')`

Parameter	Description
Alter_name	The name of the alert, as defined in the Designer (You can alternatively specify an alert_id (unquoted) for this argument.)
Button	The number that specifies the alter button (Use ALERT_BUTTON1, ALERT_BUTTON2, ALERT_BUTTON3 constants.)
Property	The property being; use LABEL
Value	The character string that defines the label.

SHOW_ALERT Function

SHOW_ALERT is how you display an alert at run time, and return the operator's response to the calling trigger:

`Selected_button := SHOW_ALERT ('alert_name');`

Alert_Name is the name of the alert, as defined in the builder. You can alternatively specify an *Alert_Id* (unquoted) for this argument.

SHOW_ALERT returns a NUMBER constant, that indicated which of the three possible buttons the user pressed in response to the alert. These numbers correspond to the values of three PL/SQL constants, which are predefined by the Form Builder:

If the number equals...	The Operator selected is...
ALERT_BUTTON1	Button 1
ALERT_BUTTON2	Button 2
ALERT_BUTTON3	Button 3

After displaying an alert that has more than one button, you can determine which button the operator pressed by comparing the returned value against the corresponding constants.

Example

A trigger that fires when the user attempts to delete a record might invoke the alert, shown opposite, to obtain confirmation. If the operator selects Yes, then the DELETE_RECORD built-in is called to delete the current record from the block.

```
IF SHOW_ALERT ('del_check') = ALERT_BUTTON1 THEN
    DELETE_RECORD;
END IF;
```

Directing Errors to an Alert

You may want to display errors automatically in an alert, through an On-Error trigger. The built-in functions that return error information, such as ERROR_TEXT, can be used in the SET_ALERT_PROPERTY procedure, to construct the alert message for display.

Example

The following user-named procedure can be called when the last form action was unsuccessful. The procedure fails the calling trigger and displays Error_Alert containing the error information.

```
PROCEDURE alert_on_failure IS
    n NUMBER;
BEGIN
    SET_ALERT_PROPERTY (
        'error_alert',
        ALERT_MESSAGE_TEXT,
        ERROR_TYPE / / '-' / / TO_CHAR (ERROR_CODE) / / ':' / /
        ERROR_TEXT);
    n := SHOW_ALERT ('error_alert');
END;
```

Alerts are an alternative method for communicating with the operator. Because they display in a modal window, alerts provide an effective way of drawing attention and forcing the operator to answer the message before processing can continue.

Use alerts when you need to do the following:

Setting alert properties at runtime

An alert may have to be displayed when the exit button on the data entry form is clicked. The functionality of this alert is the same as all other alerts except the alert title and the message in the alert box must be different.

Instead of having two separate alerts, one for delete and one for exit. Oracle forms allows these properties to be set at runtime. The title and the message can be changed depending on the button that is pressed allowing the use of one alert object to deal with multiple situations.

The following are the steps to achieve this.

1. Open the product form in the forms builder. Locate the node labeled alerts and click on navigator..Create
2. In the properties window, set the following properties:

Name : CONFIRM_ALERT
Alert style : Caution
Button 1 : Yes
Button 2 : No
Default Button : Button2

3. To display the alert the following trigger need to be added where it is required.

Trigger Name : WHEN-BUTTON-PRESSED From : Product
Function : button_palette item : pb_delete
Function : Display the alert message box before deleting a record.
Text : DECLARE

```
    Chk_button    number;
    Begin
        Go_block('product_master');
        set_alert_property('confirm_alert',title,'Delete records');
        set_alert_property('confirm_alert',alert_message_text,
                           'do you really want to delete the record?');
        chk_button := show_alert ('alert_delete');
        if chk_button = alert_button1 then
            delete_record;
        endif;
    END;
```

Trigger name : WHEN-BUTTON-PRESSED form :product
Block : button_palette
Function : Quit the product form.
Text : DECLARE

```
    Ans number;
    BEGIN
        If : system.form_status = 'CHANGED' then
            Set_alert_property('confirm_alert',title,'save changes');
            Set_alert_property('confirm_alert',alert_message_text,
                               'would you like to make changes permanent');
            ans := Show_alert('confirm_alert');
            if ans = alert_button1 then
                commit_form;
            End if;
        End if;
        Exit_form(No_commit);
    END;
```

24.6 Object Libraries

Object libraries are convenient containers of objects for reuse. They simplify reuse in complex environments, and they support corporate, project, and personal standards.

An object library can contain simple objects, property classes, object groups, and program units, but they are protected against change in the library. Objects can be used as standards (classes) for other objects.

Object libraries simplify the sharing of reusable components. Reusing components enables you to:

- Apply standards to simple objects, such as buttons and items, for a consistent look and feel
- Reuse complex objects such as a Navigator

In combination with SmartClasses, which are discussed later, object libraries support both of these requirements.

Note: Form Builder opens all libraries that were open when you last closed Form Builder

Why Object Libraries Instead of Object Groups?

- Object libraries can contain individual items; for example, iconic buttons. The smallest unit accepted in an object group is a block.
- Object libraries accept PL/SQL program units.
- If you change an object in an object library, all forms that contain the subclassed object reflect the change.

Creating an Object Group

An object group is a logical container for a set of Form Builder objects.

You define an object group when you want to package related objects for copying or subclassing in another module. You can use object groups to bundle numerous objects into higher-level building blocks that you can use again in another application.

SmartClass

A SmartClass is a special member of an Object Library. Unlike other Object Library members, it can be used to subclass existing objects in a form using a SmartClass option from the right mouse button popup menu. Object Library members which are not SmartClasses can only be used to create new objects in form modules into which they are added.

If you frequently use certain objects as standards, such as standard buttons, date items, and alerts, you can mark them as SmartClasses by selecting each object in the object library and choosing

Object → SmartClass.

You can mark many different objects that are spread across multiple object libraries as SmartClasses.

You can also have many SmartClasses of a given object type; for example:

- Wide_Button
- Narrow_Button
- Small_Iconic_Button
- Large_Iconic_Button

Work with SmartClasses

1. Select an object in the Layout Editor or Navigator.
2. From the pop-up menu, select SmartClasses. The SmartClasses pop-up menu lists all the SmartClasses from all open object libraries that have the same type as the object. When you select a class for the object, it becomes the parent class of the object. You can see its details in the Subclass Information dialog box, just like any other subclassed object.

This mechanism makes it very easy to apply classes to existing objects.

24.7 Editors

With a text editor enabled the user can view multiple lines of a text item simultaneously, search and replace text in it, and generally modify the value of an item from this separate window.

You can use one of three editors at run time:

- Form Builder default editor
- User-named default editor
- System editor

Every text item has the default editor available, but you can design your own replacement editor for those items that have special requirements such as larger editing window, position, color, and title.

By overriding the default editor for a text item, you can provide a larger editing window for items with potentially large textual values. Optionally, use an external system editor.

Editor at Run Time

With the cursor in the text item to be edited, follow these steps:

1. Press the (Edit) key, or select Edit → Edit to invoke the attached editor.
2. Edit the text in the Editor window. Form Builder editors provide a Search button that invokes an additional search-and-replace dialog box for manipulating text
3. Click OK to write your changes back to the text item.

24.8 Short Summary

- ➲ A property class is a named objects that containing a list of properties and they're setting.
- ➲ Forms provide a variety of methods for reusing objects and code.
- ➲ Using property classes define standard properties for several objects at a time.
- ➲ Using object libraries enables drag and drop reuse and provides smart classes for default objects.

- ☞ Display alerts with show_alert
- ☞ Change alert message with set_alert_property.
- ☞ Associate one of three types of alerts with a text item.

24.9 Brain Storm

1. What is an editor?
2. Explain about alerts?
3. Differentiate property classes and visual attributes?
4. Explain the types of library file formats
5. How to create a library explain?

ANSWER

Lecture 25

Triggers

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about Master Detail Form
- ☒ Create a database and run a master detail form module
- ☒ Modify a data block by using the data block wizard
- ☒ Define triggers
- ☒ Identify the different triggers categories
- ☒ Describe the properties that affect the behaviour of a trigger
- ☒ Explain the effects of the validation unit upon a form
- ☒ Control validation by using triggers

Coverage Plan

Lecture 25

- 25.1 Snap shot
- 25.2 Master detail Relation
- 25.3 Master detail form
- 25.4 Triggers
- 25.5 Validations
- 25.6 Short summary
- 25.7 Brain Storm

25.1 Snap Shot

In this session we discuss about building a new form module by using the following methods

- Form builder wizards
- Manually
- Template form

Using the data block wizard to create a new data block with its associated data sources quickly and easily.

25.2 Master Detail Relation

What is a Relation?

A *relation* is a Form Builder object that handles the relationship between two associated blocks.

You can create a relation either:

- ❖ Implicitly with a master-detail form module
- ❖ Explicitly in the Object Navigator

Implicit Relations

When you create a master-detail form module, a relation is automatically created. This relation is named `masterblock_detailblock`, for example, `S_ORD_S_ITEM`.

Explicit Relations

If a relation is not established when default blocks are created, you can create your own by setting the properties in the New Relation dialog box. Like implicitly created relations, PL/SQL program units and triggers are created automatically when you explicitly create a relation.

How to Create a Relation Explicitly

1. Select the master block entry in the Object Navigator
2. Click the Create icon.
The New Relation window is displayed
3. Specify the name of the relation.
4. Specify the name of the master block.
5. Specify the name of the detail block.
6. Choose your master delete property.
7. Choose your coordination property
8. Specify the join condition.
9. Click OK.

The new relation, new triggers, and new program units are highlighted in the Object Navigator.

Modifying a Relation

You can alter the relation properties to affect the way deletes and block coordination are handled.

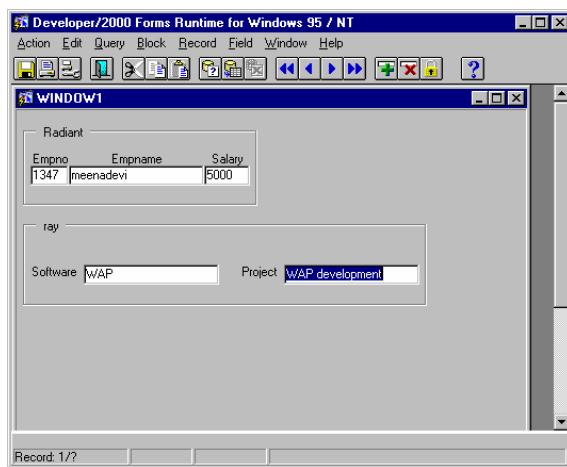
Master Deletes

You can prevent, propagate, or isolate deletion of a record in a master block when corresponding records exist in the detail block by setting the Master Deletes property. For example, you can delete all corresponding line items when an order is deleted.

Property	Use
Non-Isolated	Prevents the deletion of the master record when the detail records exist
Cascading	Deletes the detail records when a master record is deleted
Isolated	Deletes only the master record

When Happens When You Modify a Relation?

- Changing the Master Deletes property from the default of Non-Isolated to Cascading replaces the On-Check-Delete-Master trigger with the Pre-Delete trigger.
- Changing the Master Deletes property from the default of Non-Isolated to Isolated results in the removal of the On-Check-Delete-Master trigger.



Types of Blocks

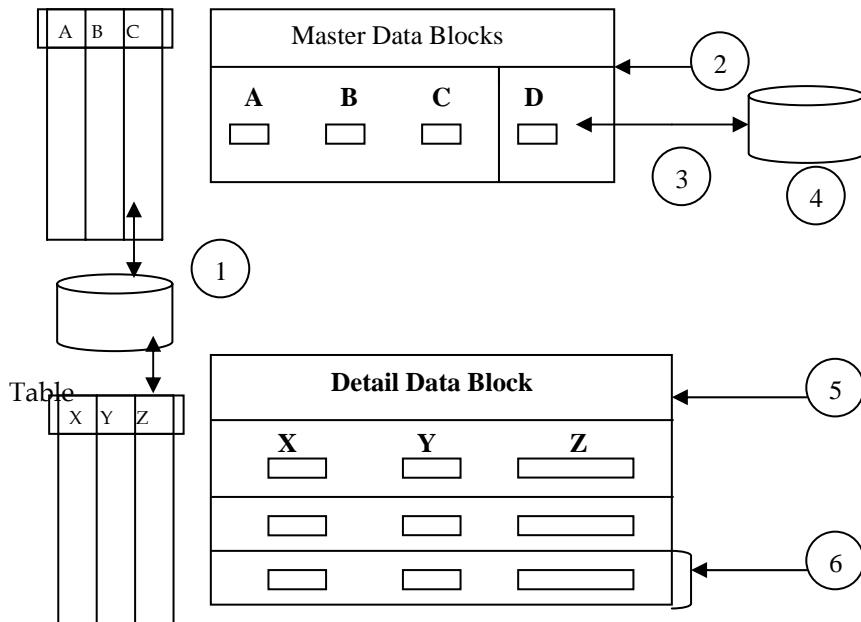
In Form Builder there are two main types of blocks data blocks and control blocks.

Data Blocks When you build database applications with Form Builder, many of the blocks will be data blocks. A data block is associated with a specific database table (or view), a stored procedure, a FORM clause query, or transactional triggers.

If it is based on a table (or view) the data block can be based on only one base table, even though the data block can be programmed to access data from more than one table and data sources. By default, the association between a data block and the database enables the user to automatically access and manipulate data in the database. However, to access data from other tables (non base table) you need to write triggers

- Creates items in the data block to correspond to columns in the table (there items are data items or base table items)
- Produces code in the form to employ the rules of the table's constraints
- Generates SQL at run time (implicit SQL) to insert update, delete, and query rows in the base table, based table, based upon the user's actions.

At run time, you can use standard function keys, buttons, menu options or standard toolbar options to initiate query, insert, update, or delete operations on base tables and the subsequent commit of the transaction.



Table

1	Base table source
2	Single-record data block
3	Triggers access
4	Nonbase table source
5	Multirecord data block
6	record

Control Blocks A control block is not associated with a database and its items do not relate to any columns within any database table. Its items are called control items. For example

you can create many buttons in your module to initiate certain actions and to logically group these buttons in a control block.

Master Versus Detail Blocks

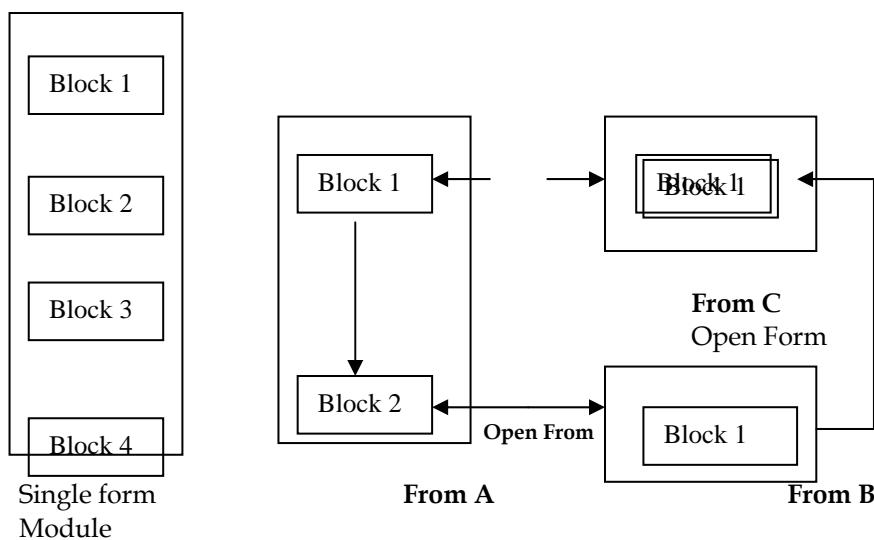
To support the relationship between data blocks and their underlying base tables, you can define one data block as the detail (child) of a master (parent) data block. This links primary key and foreign key values across data blocks, and synchronizes the data that these data blocks display.

Form Builder automatically generates the objects and code needed to support master detail relationships. As the designer, you need only request it.

Note: If your application requires it, you can also create independent dat blocks in which there is no relationship between the two data blocks.

Single-Record Versus Multirecord Blocks

We can design a data block to show one record at a time(Single-Record block) or several records at once(Multi record block).Usually ,we create a single -record data block to show master block data and a multirecord data block to show detail block data. In either case, records in a data block that are currently not visible on the screen are stored in a block buffer.



Note

This slide illustrates multiple data blocks in a single form compared to the multiple form application.

Many Blocks or Many Forms?

Typically, a Form Builder application consists of more than one data block. With more than one data block. We can do the following

- Separate the navigation cycle of one group of items from another
- Map each data block to a different database table (We can have one database per data block.)
- Produce a master-detail form, with a master data block and corresponding detail data blocks that are related to the master.

We can create a large form module with many data blocks. Alternatively , we can create several smaller form modules with fewer data blocks in each.

Generally ,a modular application with several smaller form modules has the following characteristics:

- Modules are loaded only when their components are required, thus conserving memory.
- Maintenance can occur on one module without regenerating or loading the others.
- Forms can call upon one another, as required.
- Parallel development can be carried out by different team members on different components

Here are some points to consider when grouping data blocks in the application:

Data Blocks in the Same Form Module	Data Blocks in Different Form Modules
The data blocks can be directly linked in master-detail relationships.	The Data blocks cannot be linked by the standard interblock relations.

Navigation between data blocks is handled by default functionality.	Navigation between data blocks of different forms is programmed by the designer(although mouse navigation to visible items can be automatic).
---	---

25.3 Creating Master-Detail Form

Creating Data Blocks with Relationships

A form module can contain one or more data blocks. Each data block can stand alone or be related to another data block.

Master-Detail Relationship

A master detail relationship is an association between two data block that reflects a primary foreign key relationship between the database tables on which the two data blocks are based. The master data block is based on the table with the primary key, and the detail data block is based on the table with the foreign key. A master-detail relationship equates to the one to many relationship in the entity relationship diagram.

A Detail Block Can Be a Master

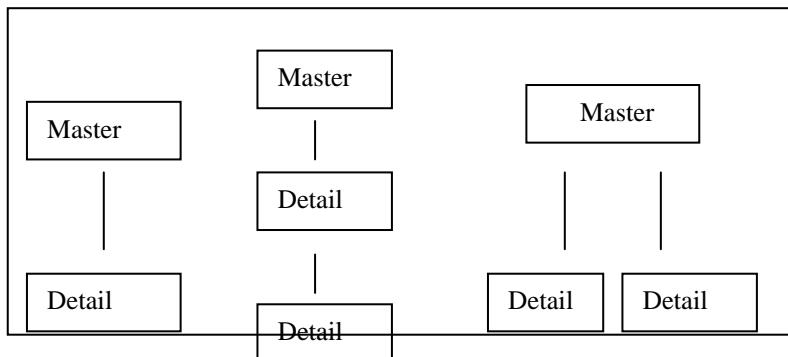
You can create block relationships in which the detail of one master detail link is the master for another link.

A Master Block Can Have More Details

You can create more than one detail blocks for a master block.

Note : The following are examples of the master detail structure:

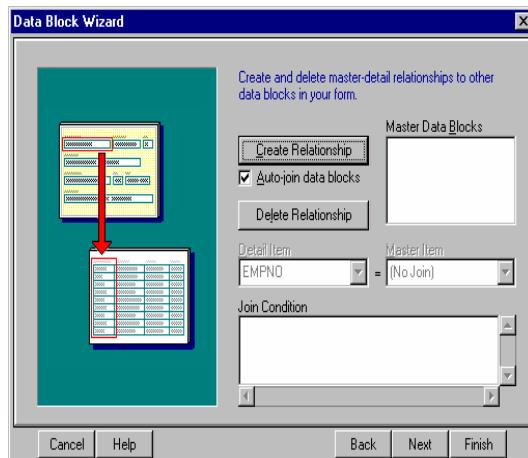
- Master -detail : Order-items
- Master-detail-detail : Customer-order-items
- Master-2 * detail : Customer-order and customer-shipping contacts



Creating a master-detail form module with the data block wizard

You can build a master-detail form modules either by creating a relation between a master and detail block explicitly, or by using the data block wizard.

1. Create the master block as described earlier in this lesson.
2. Invoke the data block wizard in the Object navigator.



3. Follow the same steps as before to create a new data block in the data select the "auto-join data blocks" checkbox and click Create relationship.

Note : if the "auto-join data blocks" check box is clear, the data block dialog is displayed with a list of all data blocks in the form without any foreign key constraint names.

4. Select a master data block in the Data Block dialog and click OK. The wizard automatically creates the join condition between the detail and master data blocks in the

join Condition field and displays the name of the master data block in the Master Data Blocks field.

Note : If the "Auto-join data blocks check box is clear, the wizard does not automatically create the join condition between the detail and master data blocks. You must use the Detail Item and Master Item pop-up lists to create a join condition manually.

5. Click Next and finish the Data Block Wizard steps. Go through the Layout wizard steps as described earlier in this lesson to finish creating and laying out the detail data block.

Note: The master data block must exist in the form module before you create the detail block.

New Relation

Once you create a master-detail form module, the Data Block Wizard automatically creates a form object that handles the relationship between two associated data blocks. This object is called a relation. The following tasks occur automatically.

- The new relation object is created under the master data block node in the Object Navigator with default properties.
- The relation is given the following default name:
- Master Data Block_Detail Block for example ORDER_ITEM
- Triggers and program units are generated to maintain coordination between the two data blocks.

25.4 Triggers

Event Driven PL/SQL code has to be written to add functionality to each push button placed on the form. The code is executed based on a specific event occurring with respect to the push buttons, generally when the push buttons are pressed.

Event Driven Programming:

An ‘Event’ that gets executed at a specific moment in time is a ‘Trigger’. Triggers are named and built-into form items. PL/SQL code blocks when attached to these events are executed when the event(trigger) occurs.

If no PL/SQL code is attached to a trigger, the trigger will fire but no processing of any kind will take place. If PL/SQL code is attached to an event (trigger) when the event occurs the PL/SQL code executes and whatever processing is specified in the code block is carried out.

An Oracle Form Item and is appropriate trigger name must be selected. For example, when push button *pb_view* is clicked, a trigger named ‘*When-button-pressed*’ is fired. PL/SQL code i.e. *execute_query* is written in this trigger using which a call to the underlying base table is made and the records of the base table are displayed on the form.

Type of Triggers based on the User action or programmatic control:

When selecting triggers, it is important to understand precisely when triggers fire, both in relation to other triggers, and in relation to form processing. It is necessary to identify the type of trigger. Form triggers can be categorized into the following basic types:

Interface Triggers:

Triggers associated with user interface objects are called *Interface* triggers. Some of these triggers fire only in response to operator input or manipulation. Others can fire in response to both operator input and programmatic control. Interface triggers can be further classified into:

- ❖ Key Triggers
- ❖ Action Triggers associated with User objects
- ❖ Mouse Triggers.

Key Triggers:

Example:

Trigger Name	Key
Key-Next-Item	Tab or Enter
Key-ExeQry	F8
Key-NxtBlk	Shift F5
Key-Commit	F10

Action Triggers associated with user objects:

Triggers that get executed due to user interaction are called Action Triggers. Action triggers are associated with specific objects. The examples of action triggers are:

Example:

Trigger Name	Action
When-Button-Pressed	Pressing a button
When-CheckBox-Changed	Clicking a check box
When-Image-Pressed	Clicking the image
When-List-Changed	A List Item is selected
When-Radio-Changed	Clicking On a Radio Button
When-Window-Activated	When focus is on a specific Windows
When-Window-Closed	When a window is closed

Mouse Triggers:

The triggers associated with the Mouse Action are called Mouse Triggers. The Examples of Mouse Triggers are as follows:

Example:

Trigger Name	Mouse Action
When-Mouse-Click	Left Click with the Mouse button
When-Mouse-DoubleClick	Double Click with Left Mouse button.

When-Mouse-Enter	When Mouse Cursor Enters in a user Object.
When-Mouse-Leave	When Mouse Cursor is moved out of a user Object.
When-Mouse-Move	When the Mouse Cursor is Moved within an Object.

Navigational Triggers:

Navigational triggers fire in response to navigational events. When the operator clicks on a text item in another block, navigational events occur as Oracle Forms Builder moves its focus from the current item to the target item.

For example when **the cursor navigates** out of a text item to another text item the *Post-Text-Item* trigger for the current text item and *Pre-Text-Item* trigger for the next text item it moves to will both fire in sequence.

Navigational events occur at different levels in the Oracle Forms Builder object hierarchy (Form, Block, Record and Item). Navigational triggers can be further sub-divided into two categories:

- **Pre- and Post-** triggers
- **When-New-Instance** triggers

Pre- and Post- Triggers:

- Pre- and Post- triggers fire as Oracle Forms Builder cursor navigates internally through different levels of the object hierarchy. These c. by an operator, such as pressing <Tab>, <Shift><Tab> or a mouse click.
- Cursor Navigation initiated programmatically by using Oracle's built-in procedures like Go-Item, Go-Block etc.
- Internal navigation that Oracle Forms Builder automatically performs during default processing. Oracle's Internal navigation:

- from one record to another
- from one block to another
- is initiated if records for different blocks in a form are saved using the *commit key sequence* or by programmatic control like *commit_form*.

When-New-Instance-Triggers:

When-New-Instance triggers fires after the form's focus is positioned on a given form, block or item.

To know when these triggers fire, there is a need to understand the Oracle Forms Builder processing model.

Consider a form named '*Product*', with a block named *product_master* with text items associated with the columns of *product_master* table.

As soon as the form is opened in memory, a series of navigational triggers are executed. The sequence of triggers that fire when a form is invoked are as follows:

Sequence	Trigger Name	Remarks
1.	Pre-Form	The first Navigational trigger that gets executed. This trigger gets executed even before the form is displayed to the user.
2.	Pre-Block	This triggers fires while entering the block and also during navigation from one block to another.
3.	Pre-Record	This triggers fires for the first time before entering the first record in the first block on the form. <i>Subsequently</i> it fires before navigation to any item in that block.
4.	Pre-Text-Item	This triggers fires for the first time before entering the first enterable item in the first block on the form. <i>Subsequently</i> it fires before navigation to any item in that block.

5.	When-New-Form-Instance	This trigger fires after the cursor is positioned on the first item of the block and also during navigation from one block to another.
6.	When-New-Block-Instance	This triggers fires after the cursor is positioned on the first item of the block and also during navigation from one block to another.
7.	When-new-record-instance	This trigger fires when the forms focus changes from one record to the other.
8.	When-New-Item-Instance	This triggers fires after the cursor is positioned on the any item on the form and also during navigation from one item to another.

The sequence of triggers that fire when a form is closed are as follows:

Sequence	Trigger Name	Remarks
1.	Post-text-item	This triggers fires after leaving the current item.
2.	Post-Record	This triggers fires after leaving the current record.
3.	Post-Block	This triggers fires after leaving the current block.
4.	Post-Form	This triggers fires just before exiting the form.

Smart Triggers:

Each object in the object navigator, has a set of events (triggers) associated with it. Oracle Forms Builder provides a list of pre-defined triggers for the form, block or specific item. Oracle Forms Builder shows a list of triggers attached to a specific object. When Triggers are classified on the basis of the object, they are termed as **Smart Triggers**.

Example of 'Smart Triggers':

The triggers associated with push button Item are:

- When-New-Item-Instance

- When-Button-Pressed.

Writing Trigger PL/SQL code blocks:

The steps for writing PL/SQL code blocks for triggers are as follows:

1. Select the object for which trigger code is to be written in the *Object Navigator* and right click. A popup menu is displayed.
2. Click on *Smart Trigger* in the popup menu. It displays a list of triggers to select from that are connected to the selected object as shown in below diagram.
3. Select a trigger from the list provided. Oracle Forms Builder displays the PL/SQL Editor where PL/SQL Editor where PL/SQL code blocks can be written as shown in below diagram. In the current example the PL/SQL block of code that makes the 'View' button functional must be written.

Right click on push button *pn_view* to see a list of triggers associated with the push button item.

From the list displayed select the 'When-Button-Pressed' trigger. The PL/SQL Editor will display the following details and allow creation of suitable PL/SQL blocks.

Type : Trigger

Object : pb_view in block Button_Palette

Name : When-button-pressed

4. Type in appropriate PL/SQL code in the text area of the PL/SQL Editor tool. The Completed *When-Button-Pressed* trigger code attached to *button_palette* and the item *pb-view* is displayed in below diagram.

25.5 Validations

Form Builder performs a validation process at several levels to ensure that records and individual values follow appropriate rules. If validation fails, then control is passed back to the appropriate level, so that the operator can make corrections. Validation occurs at:

- Item level: Form Builder records a status for each item to determine whether it is currently valid. If an item has been changed and is not yet marked as valid, then Form Builder first performs standard validation checks to ensure that the value conforms to the item's properties. These checks are carried out before firing any When-Validate-Item triggers that you have defined. Standard checks include the following:
 - Format mask
 - Required (if so, then is the item null?)
 - Data type
 - Range (Lowest-Highest Allowed Value)
 - Validate from List
- Record level: After leaving a record, Form Builder checks to see whether the record is valid. If not, then the status of each item in the record is checked, and a When-Validate-Record trigger is then fired, if present. When the record passes these checks, it is set to valid.
- Block and form level: At block or form level, all records below that level are validated. For example, if you commit (save) changes in the form, then all records in the form are validated, unless you have suppressed this action.

When Does Validation Occur?

Form Builder carries out validation for the validation unit under the following conditions:

- ❖ The [Enter] key is (ENTER command is not necessarily mapped to the key that is physically labeled Enter) pressed or the ENTER built-in procedure is run (whose purpose is to force validation immediately).
- ❖ The operator or a trigger navigates out of the validation unit. This includes when changes are committed. The default validation unit is item, but can also be set to record, block, or form by the designer. The validation unit is discussed in the next section.

The Validation Unit

The validation unit defines the maximum amount of data an operator can enter in the form before Form Builder initiates validation. Validation unit is a property of the form module, and it can be set in the Property Palette to any of the following:

- Default
- Item
- Record
- Block
- Form

The default setting is item level. The default setting is usually chosen.

In practice, an item-level validation unit means that Form Builder validates changes when an operator navigates out of a changed item. This way, standard validation checks and firing the When-Validate-Item trigger of that item can be done immediately. As a result, operators are aware of validation failure as soon as they attempt to leave the item.

At higher validation units (record, block, or form level), the above checks are postponed until navigation moves out of that unit. All outstanding items are validated together, including the firing of When-Validate-Item and When-Validate-Record triggers.

You might set a validation unit above item level under one of the following conditions:

- Validation involves database references, and you want to postpone traffic until the operator has completed a record (record level).
- The application runs in a block-mode environment (block level).

When-Validate-Item Trigger

You have already used this trigger to add item-level validation. The trigger fires after standard item validation, and input focus is returned to the item if the trigger fails.

Example

This When-Validate-Item trigger on :S_ORD.date_ordered ensures that the Order Date is not later than the current (database)date:

```
IF : S_ORD.date_ordered > SYSDATE THEN
    MESSAGE ('Order Date is later than today!');
    RAISE form_trigger_failure;
END IF;
```

When-Validate-Record Trigger

This trigger fires after standard record-level validation, when the operator has left a new or changed record. Because Form Builder has already checked that required items for the record are valid, you can use this trigger to perform additional checks that may involve more than one of the record's items, in the order they were entered.

When-Validate-Record must be defined at block level or above.

Example

The When-Validate-Record trigger on block S_ORD ensures that orders cannot be shipped before they are ordered.

```
IF : S_ORD.date_shipped < : S_ORD.date_ordered THEN
    MESSAGE('Ship Date is before Order Date!');
    RAISE form_trigger_failure;
END IF;
```

25.6 Short Summary

- Ý Validation occurs at three levels
 - ♣ Item level to ensure that the value conforms to the item properties
 - ♣ Record level to ensure that the record is valid
 - ♣ Block and form level to ensure that the all records below the level is validated.
- Standard validation occurs before trigger validation

- Triggers are event activated program units triggers type defines the event that fires the trigger.
- Creating data block with a master detail relationship.

25.7 Brain Storm

1. What are triggers?
2. Explain the types of triggers?
3. Why we need validations?
4. How to create master detail form?
5. What are the three important levels of validations?

808

Lecture 26

List of Values

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about Working with LOV objects
- ☒ Describe the Using Multiple canvases
- ☒ Describe about Types of Canvas
- ☒ Describe about Object Groups
- ☒ Discuss about Parameters
- ☒ Describe about Record Groups

Coverage Plan

Lecture 26

- 26.1 Snap shot
- 26.2 Working with LOV objects
- 26.3 Using multiple canvases
- 26.4 Types of canvas
- 26.5 Object groups
- 26.6 Parameters
- 26.7 Record groups
- 26.8 Short summary
- 26.9 Brain Storm

26.1 Snap Shot

In this session we discuss the following details

- ♠ Describe LOVs and editors
- ♠ Design, create, and associate LOVs with text items in a form module.
- ♠ Create editors and associate them with text items in a form module.
- ♠ Describe windows and content canvases
- ♠ Describe the relationship between windows and content canvases

26.2 Working with LOV objects

What Are LOVs and Editors?

Lists of Values (LOV) and editors are objects in a form module that each opens their own window when activated at run time. They are defined at the form level, which means you can use them to support text items in any block of the form module.

LOVs

An LOV is a scrollable pop-up window that provides a user with a simple mechanism to pick the value of an item from a multicolumn dynamic list. The user can reduce the lines displayed in the list by simple automatic reduction techniques, or by search strings.

Each line in an LOV can present several field values, with column headings above. You can design your LOV to retrieve some or all of the field values from the line chosen by the user, and place them into form items or variables.

LOVs have the following qualities:

- Dynamic: The list entries can change to reflect changes in the source data.
- Independent: The designer can invoke an LOV from any text item, or from outside a text item if called programmatically.

- Flexible: You can use the same LOV to support several items, if appropriate (for example, product_ID, product_name).
- Efficient: You can design LOVs to reuse data already loaded into the form, instead of accessing the database for every call. This is useful where data is relatively static.

Editors

With a text editor enabled the user can view multiple lines of a text item Simultaneously, search and replace text in it and generally modify the value of an item from this separate window.

You can use one of three editors at run time:

- ❖ Form Builder default editor
- ❖ User-named editor
- ❖ System editor

Every text item has the default editor available, but you can design your own replacement editor for those items that have special requirements such as larger editing window, position, color, and title.

By overriding the default editor for a text item, you can provide a larger editing window for items with potentially large textual values. Optionally, use an external system editor.

How to Use an Editor at Run Time

1. Press the [Edit] key, or select Edit → Edit to invoke the attached editor.
2. Edit the text in the Editor window. Form Builder editors provide a Search button that invokes an additional search-and-replace dialog box for manipulating text.
3. Click OK to write your changes back to the text item.

Designing an LOV

When you build an LOV, consider the following objects:

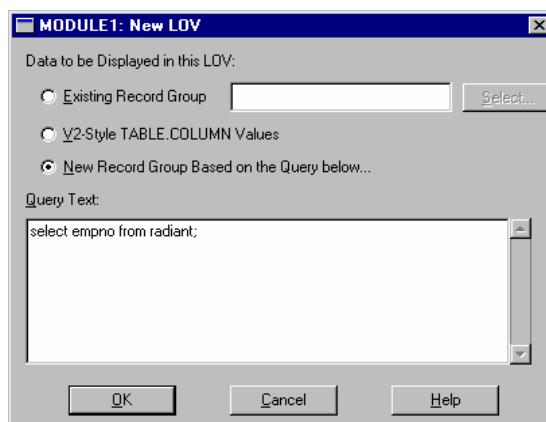
- Record group: A Form Builder object that is used to store the array of values that are presented by an LOV (The record group can be created first or as part of the LOV creation process if based on a query)
- LOV: The list itself, which presents one or more column values from the supporting record group in the LOV window (It enables the user to select values, and then write values back to specified items or variables).

Text items : The main text item that you attach to an LOV is usually one that the LOV returns a value to. You can call the LOV from this item to provide possible to attach it to these items as well in your application.

In fact, you can attach the LOV to any text item from which the same list of values needs to be viewed, whether or not it will receive a value.

Record Groups

A record group is a column-and-row structure stores within Forms Runtime memory and is similar to the structure of a database table. It holds records that can be reused by Oracle Developer applications, hence reducing repeated access to external data.



Record groups can be designed to contain static values. Alternatively, they can be populated programmatically at run time or, most commonly, populated by a SQL query.

Record groups can provide the following:

- Data that is presented by LOVs.
- Data for dynamic list items
- Data to be passed to Report Builder and Graphics Builder
- Other application-defined uses.

Creating an LOV

Select the LOVs node in the Object Navigator, and then click the Create icon. This opens the New LOV dialog box, with the following elements:

Element	Description
Data to be displayed in this LOV (radio group)	Select from: <ul style="list-style-type: none">➤ Existing record group➤ V2 style TABLE.COLUMN values➤ New record group based on the query below
Select (Button)	Use to see list of record groups, if LOV uses one that already exists.
Query Text	Multiline area where you enter SQL statement (Use this if a query-based record group is to be created to support the LOV.)

The Query Text

When you create a record group at the same time as the LOV (the default action), the new record group will be query-based. That is, it will execute a SQL SELECT statement to populate the group from the database.

The query text that you supply in the New LOV dialog box defines the following:

- ✓ The query on which the new record group will be based
- ✓ Optionally, the return items for values in the LOV (This can be defined through the INTO clause, but you can supply this information more conveniently later)

The query text must include a SELECT and FROM clause. Optional clauses include the following:

- INTO
- WHERE
- GROUP BY
- ORDER BY

Query Text Example

The following query text sample displays sales people's names, IDs, and departments from the S_EMP table. (Department numbers are for display purposes only and are not returned by the LOV)

```
SELECT last_name, id, dept_id
FROM s_emp
WHERE title = 'Sales Representative'
ORDER BY last_name
```

Query Text Example

The following New LOV Query Text retrieves and displays the names and IDs of customers. Receiving items will be defined for the LOV later.

```
SELECT name, id
FROM s_customer
ORDER BY name
```

LOV Default Functionality

Once you create the new LOV, it displays in the Object Navigator. You will also see the associated record group. The two objects initially have the same name, if they were created together.

Note: To view the underlying SELECT statement, open the Property Palette of the associated record group and examine the Record Group Query property value.

Setting LOV Properties

After you create an LOV, open its Property Palette to define its properties.

Property	Function
Title	Specifies a title for the LOV
X Position and Y Position	Screen coordinates for the LOV window in the current form coordinate units (Choose a position that is suitable for the items that the LOV supports)
Width and Height	Size of the LOV window in the current form coordinate units (The user can adjust this, but choose a size that is suitable for the data)
Column Mapping Properties (More...)	Opens the LOV Column Mapping window
Filter Before Display	Determines whether users should be prompted with a dialog box that enables them to enter a search value before the LOV is invoked.
Automatic Display	Determines whether the LOV should be invoked automatically when the cursor enters an item to which the LOV is attached.
Automatic Refresh	When this property is set to Yes, the record group reexecutes its query every time the LOV is invoked. When this property is set to No, the record group query fires only the first time the LOV is invoked within a user session. Subsequent LOV calls use current record group data.
Automatic Select	Determines whether the LOV should close and return values automatically when reduced to a singly entry
Automatic Skip	Determines whether the cursor skips to the next navigable item when the operator selects a value from the LOV to populate the text item.
Automatic Position	Determines whether Form Builder automatically positions the LOV near the field from which it was invoked.
Automatic Column Width	Determines whether Form Builder automatically sets the width of each column to display the entire column title when the column title width is longer than the column display width.

Note: More than one LOV can be based on the same record group. When this is the case and you set Automatic Refresh to No, Form Builder will not reexecute the LOV query once any of the LOVs is invoked.

The Column Mapping Properties

When you click the More property control button for Column Mapping Properties, the LOV Column Mapping dialog box opens.

Column Mapping Element	Description
Column Names (List)	Lets you select an LOV column for mapping or defining a column
Return Item	Specifies the name of the form item or variable to which Form Builder should assign the column value. Use one of the following: <code>block_name.item_name</code> <code>GLOBAL.variable_name</code> <code>PARAMETER.parameter_name</code> If null, the column value is not returned from LOV.
Display Width	Width of column display in LOV(A 0 value causes the column to be hidden, although its value remains available for return.)
Column Title	Heading for column in LOV window.

To set a column mapping in this dialog, first select the column from the Column Names list, then set the other mapping values, as required.

Note: The record group columns and LOV columns must remain compatible. You can modify the record group query from its own properties list.

Associating an LOV with a Text Item

So that the user can invoke an LOV from a text item, you must specify the LOV name in the Property Palette of the text item.

1. Select the text item in the Object Navigator from which the LOV is to be accessible.

2. In the item Property Palette, set the List of Values property to the required LOV.

Remember that the List of Values lamp is displayed when the user navigates to this text item, indicating that the LOV is available through the [List of Values] key or menu command.

Creating an LOV by Using the LOV Wizard

1. Launch the LOV Wizard.
2. The Welcome page is displayed. Click Next.
3. Specify the LOV source in the LOV Source page. Choose an existing record group or create a new one based on a query. The New Record Group based on a query radio button is set by default. Click Next to select the default.
4. In the SQL Query page specify the query used to construct the record group.
 - Click the Build SQL Query button to use the Query Builder
 - Click the Import SQL Query button to import a query from a file.
 - To enter the query directly, type the SQL syntax in the SQL Query Statement field. Then click the Check Syntax button.
5. In the Column Selection page, select the record columns that you want to include in the LOV.

Creating an LOV by Using the LOV Wizard (continued)

6. In the Column Properties page, specify the title, width and return value for each LOV column. Note that the Return Value Into item is optional.
7. In the LOV Display page, specify the title, width, and height of the LOV window.

Creating an LOV by Using the LOV Wizard (continued)

8. In the Advanced Options page, set the additional advanced properties. Specify:

- The number of records to be fetched from the database
 - If the user should be presented with a dialog box to add criteria before the LOV is displayed.
 - If the LOV records should be queried each time the LOV is invoked.
9. In the Finish page, click Finish to complete the LOV creating process.

26.3 Using Multiple Canvases

Without Oracle Developer you can take advantage of the GUI environment by displaying a form module across several canvases and in multiple windows. This lesson familiarizes you with the window object and the default canvas type, the content canvas.

Windows and Content Canvases

With form Builder you can display an application in multiple windows by using its display objects windows and canvases.

What is a Window?

A window is a container for all visual objects that make up a Form Builder application. It is similar to an empty picture frame. The window manager provides the controls for the window that enable such functionality as scrolling as scrolling, moving, and resizing. You can minimize a window.

A single form may include several windows.

What is a Canvas?

A canvas is a surface inside a window container on which you place visual objects such as interface items and graphics. It is similar to the canvas upon which a picture is painted. To see a canvas and its content at run time you must display it in a window. A canvas always displays in the window to which it is assigned.

Note: Each item in a form must refer to no more than one canvas. An item displays on the canvas to which it is assigned, through its Canvas property. Recall that if the Canvas property for an item is left unspecified, that item is said to be a Null-canvas item and will not display at runtime.

What is a Viewport?

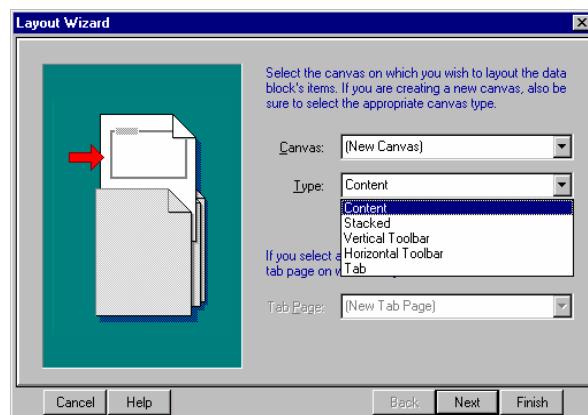
A viewport is an attribute of a canvas. It is effectively the visible portion of, or view onto, the canvas.

26.4 Types of Canvas

What is a Content Canvas ?

Form Builder offers different types of canvases. A content canvas is the base canvas that occupies the entire content pane of the window in which it displays. The content canvas, is the default canvas type. Most canvases are content canvases.

Displaying a Form Module in Multiple Windows



When you create a new form module, Form Builder creates a new window implicitly. Thus, each new form module has one predefined window, which is called WINDOW 1. You can delete or rename Window1, or change its attributes.

Uses and Benefits of a New Window

You can create additional windows in which to display your form application. A new or second window provides the ability to do the following:

- Display two or more content canvases at once.
- Modularize the form contents
- Switch between canvases without replacing the initial one
- Take advantage of window manager functionality such as minimizing.

Window Types

You can create two different window types ; modal and modeless.

- A modal window is a restricted window that the user must respond to before moving the input focus to another window. Modal windows.

Must be dismissed before control can be returned to a modeless window. Become active as soon as they display,

Require a means of exit or dismissal

- A modeless window is an unrestricted window that the user can exit freely. Modeless windows:

Can display many at once

Are not necessarily active when displayed

- Are the default window type

Physical	Function
X Position	Determines the X coordinate for the window
Y Position	Determines the Y coordinate for the window
Width	Determines the width of the window
Height	Determines the height of the window
Bevel	Determines how the window border displays
Show horizontal Scrollbar	Determines whether a horizontal scroll bar displays in the window
Show Vertical Scrollbar	Determines whether a vertical scroll bar displays in the window

Functional Property	Function
Tile	Specifies a window title to appear in the title bar
Primary Canvas	Specifies the name of the canvas to display in this window when it is invoked programmatically
Window Style	Determines whether the window style is document or Dialog (Document style windows are fixed and always remain within the application window frame. Dialog style window are free floating and can be moved outside the application window frame)
Modal	Determines whether the window is modal (requires user response) Or modeless (does not require user response)
Hide on Exit	Specifies whether a modeless window is hidden automatically when the end user navigates to an item in another window.,
Icon File name	Specifies the icon resource name that depicts the minimized window

Note : If you do not specify a window title, Form Builder uses the window object name specified in the Name property for the title.

The canvas you choose as the primary canvas must be a content canvas.

The X and Y Position (0,0) of a window is relative to the top left corner of the screen when you set the Window Style to dialog. If you set the Window is relative to the top left corner of the screen when you set the Window Style to dialog. If you set the Window Style to document, the X and Y Position (0,0) is relative to the top-left corner of the MDI window.

What Are GUI Hints?

GUI Hints are recommendations to the window manager about the window appearance and functionality. There are certain properties under the Functional group that enable you to

make these recommendations. If the current window manager supports the specific GUI Hint property and it is set to Yes, then Form Builder uses it. However if the window manager does not support the property, Form Builder ignores it.

Functional property for GUI Hints	Function
Close Allowed	Enables the mechanism for closing the window, as provided by the window manager specific close command (From Builder responds to user attempts to close the window by firing a WHEN WINDOW CLOSED trigger to actually close it)
Move Allowed	Determines whether the user can move the window by using the means provided by the window manager
Resize Allowed	Determines whether the user can resize the window at run time
Maximize Allowed	Determines whether the user can iconify and minimize the window
Inherit Menu	Determines whether the window displays the current form menu

Note : The Minimize Allowed property must be set to yes in order for Icon file name to be valid

How to Create a New Window

1. Click the Windows node in the Object Navigator
2. click the create icon
3. a new window entry displays, with a default name of WINDOWXX.
4. If the property Palette is not already displayed, double click the window icon to the left of the new window entry.
5. Set the window properties according to your requirements (as described in the tables, earlier in this lesson)

Note: For you new window to display, you must specify its name in the Window property of at least one canvas.

Displaying a Form Module on Multiple Layouts

You can have more than one content canvas in your form application. However, remember that only one content canvas can display in a window at one time. To display more than one content canvas at the same time, you can assign each content canvas to a different window.

Now you can display the form module on multiple layouts or surfaces.

Creating a New Content Canvas implicitly

There are two ways of implicitly creating a new content canvas:

- **Layout Wizard** : When you use the Layout Wizard to arrange data block items on a canvas, the wizard enables you to select a new canvas on its Canvas page. In this case, the wizard creates a new canvas with a default name of CANVASSXX.
- **Layout Editor** : When there are no canvases in a form module and you invoke the Layout Editor; form Builder creates a default canvas on which you can place items.

Creating a New Content Canvas Explicitly

General property	Function
Canvas	Specifies the type of canvas (for a content canvas, this property should be set to Content)

Physical Property	Function
Window	Specifies the window in which the canvas will be displayed
ViewPort X position or Canvas	Specifies the X coordinate of the top-left corner of the view relative to the upper-left corner of the canvas

View port Y position on Canvas	Specifies the Y coordinate of the top-left corner of the view relative to the upper-left corner of the canvas
Width	Specifies the width of the canvas
Height	Specifies the height of the canvas
Bevel	Specifies a sculpted canvas border

You can create a new content canvas explicitly by using the create icon in the Object Navigator.

Note: For a canvas to display at run time, its Window property must be specified.

How to create a new content canvas

1. Click the Canvas node in the Object Navigator.
2. Click the Create icon.

A new canvas entry displays with a default name of a CANVASXX

3. If the property Palette is not already displayed, click the new canvas entry and select Tools Property Palette.
4. Set the canvas properties that are described in the above tables according to your requirements.

Note : Double-clicking the icon for a canvas in the Object Navigator will invoke the Layout Editor instead of the Property Palette.

26.5 Object Groups

An object group is a container for a group of objects. You define an object group when you want to package related objects so you can copy or reference them in another module.

Object groups provide a way to bundle objects into higher level building blocks that can be used in other parts of an application and in subsequent development projects.

For example you might build an appointment scheduler in a form and then decide to make it available from other forms in your applications. The scheduler would probably be build

from several types of objects, including a window and canvas view, blocks, and items that display dates and appointments, and triggers that contain the logic for scheduling and other functionality. If you packaged these objects into an object group, you could then copy them to any number of other forms in one simple operation.

You can create object groups in form and menu modules. Once you create an object group, you can add and remove objects to it as desired.

To create an object groups

1. In the Object Navigator, position the cursor on the Object Groups node and choose **Navigator - → Create**

An object group with a default name is inserted under the node. when you double click to expand the objects, the Object Group Children nod appears. Objects you add to the group appear under this node.

2. In the Object Navigator, drag the desired object(s) under the Object Group Children node.

To remove objects from an object group;

In the Object Navigator, select theobject(s) you want to remove from the group and choose Edit --> Cut, Edit-> Clear, or Navigator -> Delete.

Removing an object from and object group does not delete the object from the module

26.6 Parameters

Interlinking between forms can be achieved by opening required forms from the current form and passing required information to the called form.

When opening related forms, in a commercial application it is appropriate that the mode of operation in the called form and the calling form are the same.

Example:

If the mode of operation in the client form is QUERY, the mode of operation in the order form must also be QUERY and information regarding the sales orders placed by the current client must be retrieved from the table.

Similarly, if the mode of operation in the client form is INSERT, the mode of operation in the order form must also be INSERT and information pertaining to the current client must be set by the system in INSERT mode.

Thus the mode of operation must be passed to the order form. The client_no associated with the current client must also be passed to the order form. The client_no passed from the client form can then be used to retrieve corresponding order information if the mode of operation is QUERY. If the mode of operation is INSERT, client_no can be used to set the value of the text item associated with the client_no column in the sales_order table.

Parameter values from a calling form can be passed to a called form, when any form is invoked by using the OPEN_FORM or CALL_FORM procedures.

The basic steps in parameter passing are:

- Creating a list of parameters in the calling form and passing the list of parameters to the called form using the OPEN_FORM or CALL_FORM procedure.
- Defining parameters in the called form and using them to accept values from the list of parameters passed by the calling form.
- Referencing the values held in the parameters defined in the called form.

The steps in passing these parameters from a calling form to the called form are:

- Create a Parameter list object.
- Add parameters to the parameter list
- Pass the parameter list to the called form via the open_form () or the call_form () oracle procedure.

Creating a Parameter List:

A Built-in package procedure named CREATE_PARAMETER_LIST creates a parameter list and returns the memory address.(i.e. specific location) of the parameter list. A variable of type PARAMLIST must be declared to hold the memory address (i.e. a pointer) returned by the function. The variable declaration is as follows:

```
DECLARE
```

```
    client_param_list paramlist;
```

Each parameter list created using CREATE_PARAMETER_LIST() must be assigned a unique name. The parameter list name must be passed as an argument to the CREATE_PARAMETER_LIST () procedure as explained below. A parameter list can then be referenced by its 'Name' or the 'Variable' that holds the pointer to the parameter list.

Adding Parameters to a Parameter List:

A Built-in package procedure ADD_PARAMETER adds parameters to the parameter list. This procedure accepts the following parameters:

- ❖ Name of the parameter list.
- ❖ The Name of the parameter
- ❖ The Type of parameter
- ❖ The Value of the parameter

Types of Parameters:

Parameters added to the parameter list can be used to pass a single value to the called form. Alternatively parameters can be used to pass a record group or a set of records to the called form. The parameter type determines whether the parameter can hold a single value or can hold a record group. The Parameters type can be any one of the following:

- TEXT_PARAMETER
- DATA_PARAMETER

Text Parameter:

When a **single value** is passed as a parameter to the called form, the parameter type must be set to 'TEXT_PARAMETER'.

Example:

When client_no 'C00001' is passed as a parameter form the client form to the order form the Parameter Type, set for a parameter will be 'TEXT_PARAMETER'.

Data Parameter:

When a record group is being passed as a parameter, the parameter type must be set to 'DATA_PARAMETER'

Example:

Create a parameter list named client_list and assign the memory address of the parameter list to a memory variable called client_param_list. Add a parameter named p_client_no and specify its value as 'C00001'. Since a single value is being passed as a parameter, the type of parameter will be 'TEXT_PARAMETER'.

```
DECLARE
    client_param_list paramlist;
BEGIN
    client_param_list := create_parameter_list
        ('client_list');
    add_parameter (client_param_list, 'p_client_no', text_parameter, 'C00001');

END;
```

Passing the Parameter List to the Called Form:

After creating a parameter list and populating its parameters with appropriate values, the form to which the parameters are to be passed must be opened. A form can be opened using OPEN_FORM () or CALL_FORM () procedures. The name of the form must be passed as an argument to any of these procedures as in

```
OPEN_FORM ('order.fmx');
```

The parameter list is to be passed as the last argument to the OPEN_FORM () or the CALL_FORM () procedure. The name of the form is the only mandatory argument to any of these procedures. To ensure that the last position is maintained, all other parameters, normally optional to the OPEN_FORM() or the CALL_FORM () procedure become mandatory. The values for these arguments are pre-defined by Oracle Forms and thus null values are not allowed. The complete syntax for opening the form and passing parameter list is:

CALL_FORM ('form name', display, switch menu, query mode, parameter list);

The arguments specified in the CALL_FORM () procedure are as follows:

Display:

This argument can be set to HIDE or NO_HIDE. If this argument is set to HIDE, the calling form will be made invisible before calling the called form. If this argument is set to NO_HIDE, the called form will be opened without hiding the calling form.

Switch Menu:

This argument can be set to NO_REPLACE or DO_REPLACE. If this argument is set to NO_REPLACE, the menu of the called form will not be replaced with the menu of the calling form. If this argument is set to DO_REPLACE, the menu of the called form will be replaced with the menu of the calling form.

This argument can be set to NO_REPLACE or DO_REPLACE. If this argument is set to NO_REPLACE, the menu of the called form will not be replaced with the menu of the calling form. If this argument is set to DO_REPLACE, the menu of the called form will be replaced with the menu of the calling form.

Query Mode:

This argument can be set to NO_QUERY_ONLY or QUERY_ONLY. If this argument is set to Query_ONLY, the called form is opened for query purpose only. If this argument is set to NO_QUERY_ONLY, the called form can be used for data insertion, updation, deletion and viewing.

Parameter List:

This argument must be set to the name of the parameter list created using CREATE_PARAMETER() function.

Along with the above-mentioned arguments, if the called form and the calling form use PL/SQL libraries, an additional parameter called Data Mode can be specified.

Data Mode:

If the called form and the calling form share the same set of libraries by default, the form runtime tool loads multiple copies of the library file i.e. one copy for every form. Thus, the memory resources of the system are inefficiently used. If same set of libraries are attached to the called form and the calling form, the Data Mode argument can be set to SHARE_LIBRARY_DATA. This ensures that only one copy of the PL/SQL library is loaded and used by both the forms. The system resources are thus efficiently used.

If different set of libraries are attached to the forms then the Data Mode argument can be set to NO_SHARE_LIBRARY_DATA.

Defining Parameters in the Called Form:

Values passed by the calling form to the Called Form must be stored in special variables that can be accessed by any PL/SQL code used in the Called Form. Variables that hold these values passed from the Calling Form are called **Parameters**.

Each parameter has the following properties:

- Parameter Name
- Data Type
- Maximum Length
- Default Value.

These parameters can be created in the **Parameters Node** of the Called Form. The parameter node is found in the Object Navigator in Forms Builder.

26.7 Working With Record Groups

A record group is an internal Oracle Forms data structure that has a column / row framework similar to a database table. However, unlike database tables, record groups are separate objects that belongs to the form module in which they are defined.

A record group can have an unlimited number of columns of type CHAR, LONG, NUMBER, or DATE provided that the total number of columns does not exceed 64 k..

A record group built from a query can store records from database tables much like a database view, with the added advantage that the record group is local to Oracle Forms, rather than existing in the database.

TYPES OF RECORD GROUPS

There are three of record groups: query record groups, static record groups, and non-query groups.

Query Record A query record group is a record group that Group has an associated SELECT statement. The columns in a query record group derive their default names, data types, and lengths from the database columns referenced in the SELECT statement.

Non-query A non-query record group is a group that Record Group does not have an associated query, but whose structure and values can be modified programmatically at runtime.

Static Record A static record group is not associated with a Group query; rather, you define its structure and row values at design time, and they remain fixed at runtime.

A record group can have a unlimited number of columns of type char, long, number or date. When you a create an LOV, we can specify which of the columns in the underlying records should be displayed in the LOV window. They will be displayed in the order that they occur in the record group. Because LOV's and record groups are separate objects, we can create multiple LOV's based on the same record group.

Record Group Valued

The actual values in a record group come from one of two sources:

- ❖ The execution of a SELECT statement that you associated with the record group at design time (query record group) an array of static values that you associated with the record group at design time (static record group).

26.8 Short Summary

In this lesson, you should have learned:

- ♠ About the relationship between windows and content canvases
- ♠ How to create a new window
- ♠ How to create a new content canvas
- ♠ Describing windows and content canvases
- ♠ Creating a new window
- ♠ Creating a new content canvas

26.9 Brain Storm

- a. Explain about List of values.
- b. What are editors?
- c. Explain the properties of LOVs.
- d. How to create LOVs explain?
- e. Explain the difference between windows and content canvas.
- f. Explain the types of record groups



Lecture 27

Canvases

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about the Stacked Canvas
- ☒ Discuss about the Tab Canvas
- ☒ Describe about the Horizontal Toolbar Canvas
- ☒ Discuss about the List Item

Coverage Plan

Lecture 27

- 27.1 Snap shot
- 27.2 Stacked canvas
- 27.3 Tab canvas
- 27.4 Horizontal toolbar canvas
- 27.5 List item
- 27.6 Short summary
- 27.7 Brain Storm

27.1 Snap Shot

In addition to the content canvas, the Oracle Developer forms component enables you to create three other canvases types. This lesson introduces you to the stacked canvas, which is ideal for creating overlays in your application. It also introduces you to the toolbar canvas and the tabbed canvas and the tabbed canvas, both of which enable you to provide a user-friendly GUI application.

Canvases Overview

Besides the content canvas, From Builder provides three other types of canvases which are:

- Stacked canvas
- Toolbar canvas
- Tab canvas

When you create a canvas, you specify its type by setting the Canvas Type property. The type determines how the canvas how the canvas is displayed in the window to which it is assigned.

27.2 Stacked Canvas

What is a Stacked Canvas?

A stacked canvas is displayed on top of, or stacked on the content canvas assigned to a window. It shares a window with a content canvas and any number of other stacked canvases. Stacked canvases are usually smaller than the window in which they display.

Determining the Size of a Stacked Canvas

Stacked canvases are typically smaller than the content canvas in the same window. Determine the stacked canvas dimensions by setting width and Height properties. Determine the stacked canvas views dimensions by setting Viewport Width and Viewport Height properties.

Uses and Benefits of stacked canvases

- Scrolling views as generated by Oracle Designer
- Creating an overlay effect within a single window
- Displaying headers that display constant information, such as company name
- Creating a cascading or a revealing effect within a single window
- Displaying additional information
- Displaying information conditionally
- Displaying context sensitive help
- Hiding information

Note: If a data block contains more items than the window can display, Form Builder scrolls the window to display items outside the window frame. This can cause important items, such as primary key values, to scroll out of view. By placing important items on a content canvas, and placing the items that can be scrolled out of sight on a stacked canvas, the stacked canvas becomes the scrolling region, rather than the window itself.

Creating a stacked canvas

You can create a staked canvas in either of the following:

- object Navigator
- Layout Editor

Viewport property	Function
View X Position	Specifies the X coordinate of the stacked canvas viewport
View Y Position	Specifies the Y coordinate of the stacked canvas viewport
View Width	Specifies the width of the stacked canvas viewport
View height	Specifies the height of the stacked canvas viewport

Physical Property	Function
Show Horizontal Scrollbar	Determines whether the stacked canvas displays a horizontal scroll bar
Show Vertical Scrollbar	Determines whether the stacked canvas displays a vertical scrollbar.

How to Create a Stacked Canvas in the Object Navigator

1. Click the Canvases node in the Object Navigator.
2. Click the Create icon.
3. A new canvas entry displays with a default name of CANVASXX.
4. If the Property Palette is not already displayed, click the new canvas entry and select Tools Property Palette.
5. Set the Canvas Type property to Stacked. Additionally, set the properties that are described in the above table according to your requirements.

Note: To convert an existing content canvas to a stacked canvas, change its canvas Type property value from content to stacked.

In order for the stacked canvas to display properly, make sure that its position in the stacking order places it in front of the content canvas assigned to the same window. The stacking order of canvases is defined by the sequence in which they appear in the Object navigator.

How to create a stacked canvas in the layout editor

1. In the Object Navigator, double click the object icon for the content Canvas on which you wish to create a stacked canvas.
2. Click the Stacked Canvas tool in the toolbar.
3. Click and drag the mouse in the canvas where you want to position the stacked canvas.

4. Open the Property Palette of the stacked canvas. Set the canvas properties according to your requirements (described earlier in the lesson)

Displaying stacked canvases in the layout editor

You can display a staked canvas as it sits over the content canvas in the Layout Editor. Check the display position of stacked canvass by doing the following.

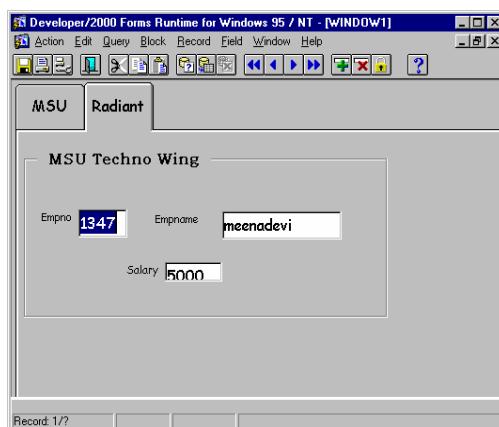
1. Select View Stacked Views in the Layout Editor. The stacked/Tab Canvases dialog box is displayed, with a list of all the stacked canvases assigned to the same window as the current content canvas.
2. select the stacked canvases you want to display in the Layout editor.

Note : [control] + Click to clear a stacked canvas that was previously selected.

27.3 Tab Canvas

What is a tab canvas?

A tab canvas is a special type of canvas that enables you to organize and display related information on separate tabs. Like stacked canvases, tab canvases are displayed on top of a content canvas.



What is a Tab Page?

A tab page is a sub-object of a tab canvas. Each tab canvas is made up of one or more tab pages. A tab page displays a subset of the information in the entire tab canvas. Each tab page has a labeled tab that end users can click to access information on the page.

Each tab page occupies an equal amount of space on the tab canvas.

Uses and Benefits of Tab Canvases

- Create an overlay effect within a single window.
- Display large amounts of information on a single canvas.
- Hide information.
- Easily access required information by clicking the tab.

Creating a Tab Canvas

- Create an empty tab canvas in either of the following:
 - Object Navigator
 - Layout Editor

Define one or more tab pages for the tab canvas.

Place items on the tab pages.

Tab Canvas Related Properties

Once you create a tab canvas and its tab pages, you must set the required properties for both of these objects. Place items on a tab page by setting the required item properties.

Tab Canvas Property	Function
Viewport X Position	Specifies the X coordinate of the tab canvas upper-left corner.
Viewport Y Position	Specifies the Y coordinate of the tab canvas upper-left corner
Viewport Width	Specifies the width of the view for the tab canvas
Viewport Height	Specifies the height of the view for the tab canvas.
Corner Style	Specifies the shape of the labeled tabs on the tab canvas (Select from Chamfered, Square, and Rounded)
Tab Attachment Edge	Specifies the location where tabs are attached to the tab canvas.

Tab Page Property	Function
Label	Specifies the text label that appears on the tab page's tab at run time.

Item Property	Function
Canvas	Specifies the tab canvas on which the item will be displayed
Tab Page	Specifies the tab page on which the item will be displayed.

How to Create a Tab Canvas in the Object Navigator

1. Click the Canvases node in the Object Navigator.
2. Click the Create icon.
3. A new canvas entry displays
4. If the Property Palette is not already displayed, click the new canvas entry and select Tools → Property Palette.
5. Set the Canvas Type property to Tab. Additionally, set the canvas properties according to your requirements.
6. Expand the canvas node in the Object Navigator.
7. The Tab Pages node displays.
8. Click the Create icon.

A tab page displays in the Object Navigator, with a default name of PAGEXX. The Property Palette takes on its context.

9. Set the tab page properties according to your requirements.
10. Create additional tab pages by repeating steps 6 and 7.

How to Create a Tab Canvas in the Layout Editor

1. In the Object Navigator, double-click the object icon for the content canvas on which you want to create a tab canvas.

The Layout Editor displays.

2. Click the Tab Canvas tool in the toolbar.

3. Click the drag the mouse in the canvas where you want to position the tab canvas.

Form Builder creates a tab canvas with two tab pages by default.

4. Open the Property Palette of the tab canvas. Set the canvas properties according to your requirements.
5. Create additional tab pages, if required, in the Object Navigator.
6. Set the tab page properties according to your requirements.

Placing Items on a Tab Page

Once you create a tab canvas and related tab pages, you must place individual items on the tab pages that the end users can interact with at run time. To accomplish this, do the following:

- Open the Property Palette of the item
- Set the item's Canvas and Tab Page properties of the item to the desired tab canvas and tab page.

Note: Display the tab canvas as it sits on top of the content canvas, by selecting View Stacked View in the  Layout Editor.

27.4 Toolbar Canvas

What is a toolbar canvas?

A toolbar canvas is a special type of canvas that you can create to hold buttons and other frequently used GUI elements.

The Three Toolbar Types

- Vertical toolbar: Use a vertical toolbar to position all your tool items down the left or right hand side of your window

- Horizontal toolbar: use a horizontal toolbar to position all your tool items and controls across the top or bottom of your window.
- MDI toolbar : Use an MDI toolbar to avoid creating more than one toolbar for a Form Builder application that uses multiple windows.

Uses and Benefits of Toolbars

- Provide a standard look and feel across canvases displayed in the same window
- Decrease form module maintenance time.
- Increase application usability
- Create applications similar to other used in the same environment
- Provide an alternative to menu or function key driven applications

Toolbar Related properties

Once you create a toolbar canvas, you must set its required properties as well as the required properties of the associated window. For MDI toolbars, you must set the required form module properties.

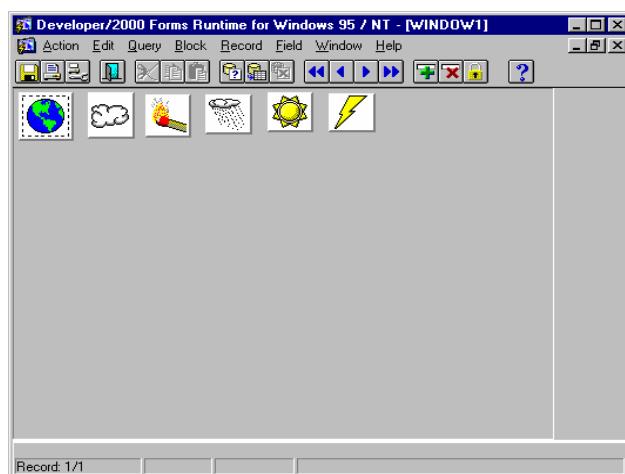
Canvas property	Function
Canvas Type	Specifies the type of canvas; for a toolbar canvas, set to Horizontal Toolbar or vertical Toolbar
Window	Specifies which window the toolbar displays in
Width	Determines the width of the toolbar
Height	Determines the height of the toolbar

Window property	Function
Horizontal Toolbar canvas/vertical Toolbar canvas	Identifies the horizontal / vertical toolbar to display in this window

Form Module Property	Function
Form horizontal toolbar canvas/form vertical toolbar canvas	Identifies the horizontal / vertical toolbar to display in the MDI window

How to create a toolbar canvas

1. Create a new canvas in the Object Navigator
2. If the property palette is not already displayed, click the new canvas entry and select tools property palette.
3. Set the canvas properties that are described in the above table.
4. In the object navigator select one of the following:
 - The window in which you want to display the toolbar (for a form window toolbar)
 - The Form module (for and MDI Toolbar)
 - Set the Horizontal / Vertical Toolbar Canvas properties
 - Add GUI elements boilerplate text, and graphics, as required.



Note: The width of a horizontal toolbar is set to the width of the window (for example, content canvas). Like wise the height of a vertical toolbar is set to the height of the window

27.5 List Item

What is a list item?

A list item is an interface object that displays a predefined set of choices, each corresponding to a specific data value. You use the list item at run time to select a single value. List choices or elements are mutually exclusive one and only one can be selected at a time.

The three List Item Styles

List Style	Description
Poplist	Appears as field with an iconic button attached to the right side (when you click a poplist, all its list elements are displayed)
Tlist	Appears as a rectangular box that displays the list elements (when the display area is not big enough to display all the list elements, a scroll bar is automatically attached to the right side to view the remaining list elements.)
Combo box	Appears as a field with a down arrow next to its right side (use the button to display all the combo box list elements. The combo box accepts user input)

Uses and Benefits of List Items

- ♣ Enable display of a defined set of choices
- ♣ Display a set of choices without using a vast area of canvas
- ♣ Provide an alternative to radio groups
- ♣ Provide a Windows-style list of values.

Setting the Value for a List item

The value for a list item can be set in any of the following ways;

- User selection
- User input (combo box style only)
- A default value
- Programmatic control

Creating a List Item

A list item can be created in three way:

- Converting an existing item
- Using the List Item tool in the Layout editor
- Using the Create icon in the Object Navigator

Item properties specific to the list item

Property	Function
Elements in List	Opens list item elements dialog window(covered later in this lesson)
List Style	Specifies the display style of the list item (choose from poplist, Tlist or combo box)
Mapping of other values	Determines how other values are processed
Mouse Navigate	Determines whether form builder navigates to the item and moves input focus to it when the user activates the item with the mouse

Note : The poplist and combo box take up less space, but end users must open them to see the list elements. A Tlist remains open and end users can see multiple values at a time. Use the attached scroll bar to see more values if the Tlist is not big enough to display all the list elements.

The Elements in List Property

When you click the More property control button for the Elements in List property, the list item elements dialog window opens.

Elements	Description
List Elements	Enables you enter the list elements as they appear at run time
List Item Value	Enables you to specify the actual value that correspond to each of the list elements.

How to convert an existing item into a list item

You can convert an existing item into a list item by changing its item Type property to list item and setting the relevant properties.

1. Invoke the Property Palette for the item that you want to convert.
2. Set the item type property to list item.
3. Select the elements in list property
4. Click more.

The list item elements dialog box appears

5. Enter the element that you want to appear in your list item in the list elements column
6. Enter the value for the currently selected list elements in the list item value field
7. Create additional list elements and values by repeating steps 5 and 6.
8. Click OK to accept and close the list item elements dialog box.
9. Set the other values property to do one of the following
 - Reject values other than those predefined as list values
 - Accept and default all other values to one of the predefined list element values
10. Enter an initial value for the list item.

How to create a list item in the Layout editor

You can also create a list item by using the list item tool in the layout editor.

1. Invoke the Layout Editor
2. Set the canvas and block to those on which you want the list item to be displayed
3. Select the list item tool.
4. Click the canvas in the position where you want the list item to be displayed.
5. Double-click the list item to invoke its property palette
6. Set the properties as required.

Null Values in a List item

If the base table column for a list item accepts NULL values, Form Builder creates a pseudochoice in the list to represent the null.

All three list styles display a blank field if a query returns a NULL value. If the data required property is set to Yes, upon activation list items display a blank element.

- A poplist displays a blank element for a NULL value.
- For Tlists, the user must scroll through to display the blank element.
- A combo box does not display a blank element. The end user must delete the default value if the default value is not NULL.

Handling other values in a list item

If the base table column for a list item accepts values other than those associated with your list elements, you must specify how you want to handle the values. Do this in one of the following ways:

- Ignore other values by leaving the Mapping of Other Values property
- Associate the other values with one of the existing list elements (by naming either the list element or its associated value) in the Mapping of Other Values property.

27.6 Short Summary

- ♣ List items to convert items that are mutually exclusive
- ♣ Describe the different types of canvases and their relationships to each other
- ♣ Identify the appropriate canvas type for different scenarios
- ♣ Create an overlay effect by using stacked canvas
- ♣ List item is an interface object that displays a predefined set of choices each corresponding to specific data values.

27.7 Brain Storm

- Explain the tab canvas.
- What is stacked canvas.
- What is viewport
- How to create a tab canvas explain.
- Explain the properties of list item.

ஈரை

Lecture 28

Menus

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss the Using Menu
- ☒ Discuss about saving compiling and attaching a menu module
- ☒ Describe about checked, enabled menu items
- ☒ Describe master detail menu

Coverage Plan

Lecture 28

- 28.1 Snap shot
- 28.2 Using menu
- 28.3 Saving compiling and attaching
- 28.4 Checked, enabled menu items
- 28.5 Short summary
- 28.6 Brain Storm

28.1 Snap Shot

In this session we discuss about menus. Every form runs with a default menu that is built into every form or a custom menu that you define as a separate module and then attach to the form for run time execution. At runtime, an application can have only one menu module active at a time. Multiple form can share a same menu or each can invoke a different menu.

28.2 Using Menu

Every form runs with one of the following:

- The Default textual menu and a Default Toolbar menu built into the form
- A custom textual menu that the user defines as a separate module and then attaches to the form for runtime execution. A custom toolbar created by the user.

At runtime, a form can have only one menu module active at a time, either the Default menu or a custom menu.

The Default menu is part of the form module. Custom menu modules, however, are separate from form modules. So, when an application, based on a single form that uses a custom menu is delivered, following files must be provided:

- An .FMX form module.
- An .MMX menu module.

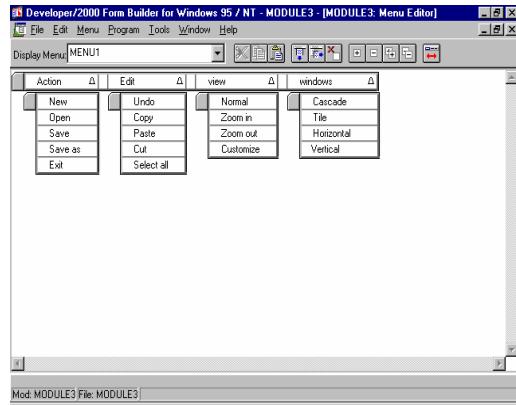
In a multi-form application, several form modules and several menu modules may be delivered to a client. Multiple forms can share the same menu, or each form can invoke a different menu.

Using a Custom Menu

A Custom menu is built by creating a Menu Module in Forms Builder and then defining objects for the menu.

A Menu Module includes different types of objects:

1. The Main Menu encompasses all Sub menu and menu items in a menu module. The Main Menu is the first item in the Menus node in the Object Navigator. The Main Menu includes menu items displayed on the Menu Bar.
2. Sub Menus that encompass menu items.
3. Menu items attached to a Sub Menu with their associated commands and procedures.



28.3 Saving Compiling and Attaching a Menu Module

Saving and Compiling Menu Module:

After creating menu items as required and writing appropriate PL/SQL code, the menu module must be saved by selecting **File..Save As**. A menu module i.e. menu source file is saved with an extension of .MMB

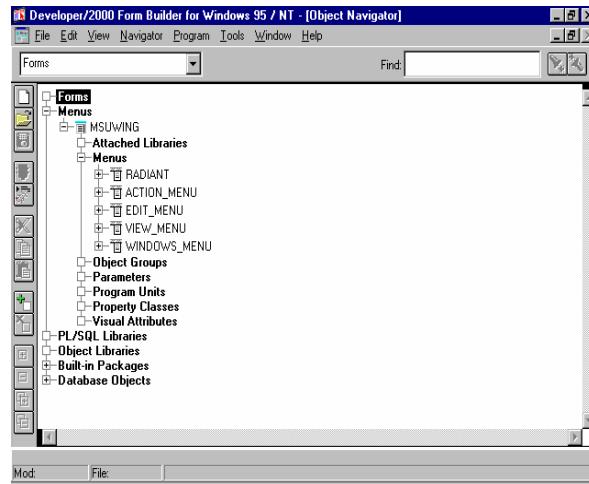
Menu module cannot be executed by itself. Menu module must be compiled and then attached to the form module.

Click on **File..Administration..Compile File** menu options to compile the menu module. When a menu module is compiled Oracle Forms creates an executable file with the same name as the MMB file but with an extension of MMX.

Attaching a Menu Module to a Form Module:

After saving and compiling the menu module, the compiled menu module must be attached to a form. The steps for attaching a menu module to a form are:

1. Open the required form in the Object Navigator. Right click on the form and click on property palette to display the Properties of the form module.
2. In Properties window, set the Menu Module property to the filename of the runtime .MMX menu file . Once a menu is attached to a form, Oracle forms automatically loads the .MMX menu file when the form is run.
3. Select **File..Save** or **File..Save As** to save the form module.
4. Select **File..Administration..Generate** to generate the form module.



28.4 Checked, Enabled and Displayed Property of Menu Items

The displayed, enabled and checked property specifies the state of the menu item. The three properties are explained below.

Checked Property:

Specifies the state of a check-box or radio-style menu item, either CHECKED or UNCHECKED. This property is set programmatically only for the menu items with the **Menu Item Type** property set to **Check** or **Radio**.

Enabled Property:

Specifies whether the menu item should be displayed as an enabled (normal) item or disabled (grayed) item. This property is set programmatically.

Displayed Property:

Determines whether the menu item is visible or hidden at runtime. This property is set programmatically.

Getting the Value of the Menu Property:

The state of the menu item for the above three properties can be obtained using the following functions:

```
get_menu_item_property (menuitem_id, property)
get_menu_item_property (menu_name.menuitem_name, property)
```

The above function returns the state of the menu item given the specific property. The return value is either TRUE or FALSE depending on the state of property (i.e. if above function is used to check the enabled property of menu item, then it returns TRUE if the menu item is enabled and FALSE if it is disabled.)

Creating an Iconic Toolbar

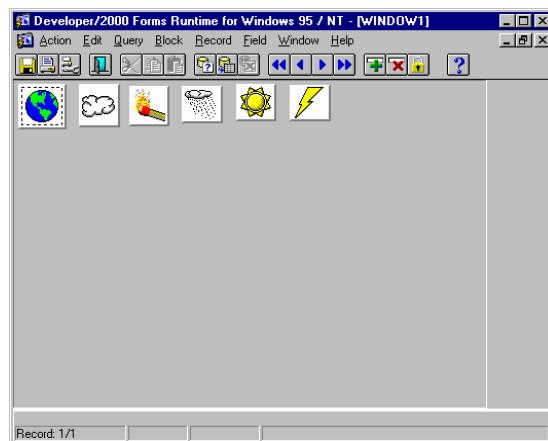
Menu items created in a menu module can also be displayed as iconic buttons on a vertical or horizontal toolbar by setting additional properties for the menu items.

Focus:

Create an iconic horizontal toolbar by altering the properties of the menu items defined in menu module radiant.mmb.

The steps in creating an iconic toolbar are:

1. Click on File..Open to Open menu module named radiant.mmb in c:\student directory.
2. Right click on the menu module and select Menu Editor from the pop up menu. Menu editor displays the menu items defined in the menu module.



3. The menu module displays Action menubar item with the sub menu items Save and Exit.
4. Right click on submenu item Save and select Property Palette from the pop up menu.
5. Set the **Visible in Menu** and **Visible in Horizontal Toolbar** to Yes for all the sub menu items. Set the Icon Filename property as follows for each of the submenu items.
6. Save the Menu module by clicking on **File..Save**. Compile the form by clicking on **File..Administration..Compile File**.
7. Open the Product form and run the form.
8. Run the form after making the changes.

28.5 Short Summary

- ♠ Components of a custom menu
- ♣ Menu bar
- ♣ Menu bar items
- ♣ Sub menu items
- ♠ Creating a custom menu
- ♣ Creating a new menu module
- ♣ Adding menu items
- ♣ Deleting menu items
- ♠ Attaching a PL/SQL library to a menu module
- ♠ Attaching PL/SQL code to menu items
- ♠ Attaching a menu module to a form
- ♠ Checked and enabled property of menu item

28.6 Brain Storm

1. Explain the file format of menu.
2. Explain about the menu.
3. Explain the properties of menu item.
4. Explain the steps for creating iconic toolbar.

©OR

Lecture 29

Reports

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss Introduction to Reports
- ☒ Describe the Types of Reports
- ☒ Discuss about Tabular Reports
- ☒ Describe about Break Reports
- ☒ Describe about Master Detail Reports
- ☒ Describe about Matrix Report
- ☒ Discuss about Report Triggers

Coverage Plan

Lecture 29

- 29.1 Snap shot
- 29.2 Introduction to reports
- 29.3 Types of reports
- 29.4 Tabular reports
- 29.5 Break report
- 29.6 Master detail report
- 29.7 Matrix report
- 29.8 Report triggers
- 29.9 Short summary
- 29.10 Brain Storm

29.1 Snap Shot

In this session we know about the features of reports tool , report data model, report layout, and types of reports.

29.2 Introduction to Reports

Oracle Reports enables creation of a variety of reports, such as a tabular report, form report, master/detail reports, nested matrix reports and mailing labels. Its major features include:

- ❖ Data model and layout editors in which the structure and format of the report can be created.
- ❖ Object Navigator that permits navigation among that data and layout objects in the report.
- ❖ Packaged functions for computations
- ❖ Support for fonts, colors, and graphics
- ❖ Conditional printing capabilities
- ❖ Fully-integrated Previewer for viewing report output.
- ❖ A unique Non-procedural Approach of Oracle Reports helps concentrate design improvements instead of programming. It is easy-to use, fill in the form interface and powerful defaults make developing and maintaining even the most complex reports fast and simple.
- ❖ Oracle Reports adheres to the native look-and -feel of the host environment. Thus, it brings about Portability with GUI Conformance. Oracle Reports can be created on bit-mapped platforms with the guarantee of identical functionality and complete compatibility across all systems.

- ❖ Oracle Reports can be fully integrated with Other Oracle Products such as Oracle Forms, Oracle Graphics and Oracle Mail. For example, graphics and charts can be included in a report, and send output to other users via Oracle mail.
- ❖ Report's open Architecture enables incorporating user defined routines written in COBOL, C and most other programming languages, as well as the powerful, PL/SQL language. Information can be presented exactly as per the report specification provided.

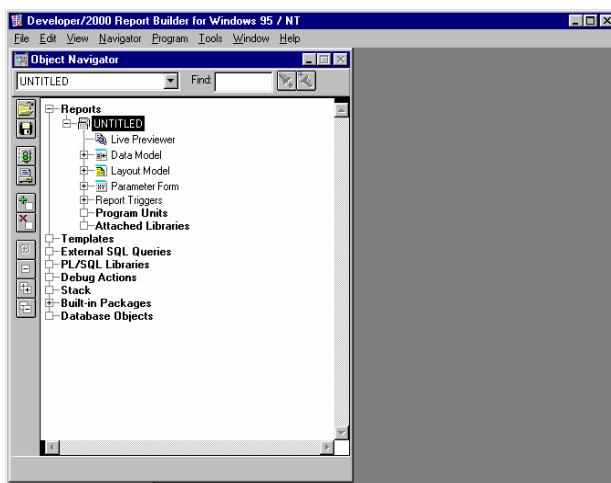
Basic Concepts

In the following few pages, the tool in general and its basic components are explained. There are three steps to building a report with Oracle reports.

- Create a new report definition.
- Define the data model. Data along with their relationships and calculations can be specified to produce report output.
- Specify layout (i.e.design). Default or customized report layouts can be used.

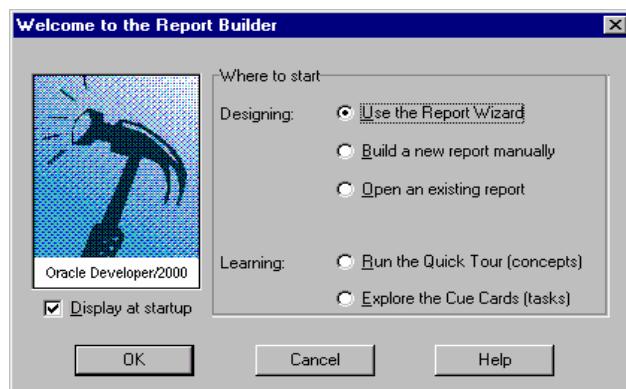
CREATING A REPORT USING ORACLE REPORTS BUILDER

A New report can be created



- By using a Report Wizard

- o Create a report Manually

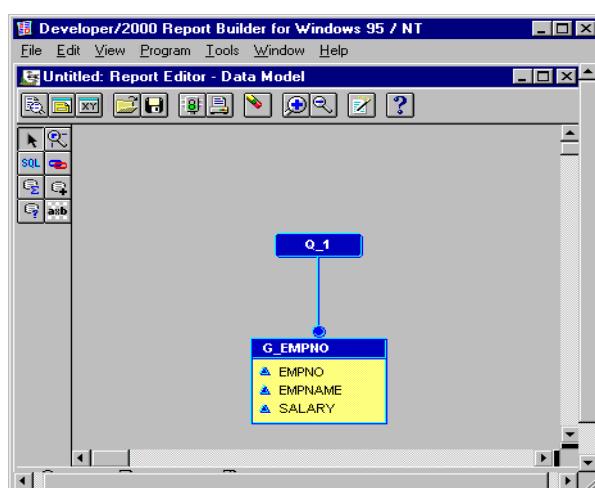


Before a new report is created, the components of an Oracle Report must be understood.

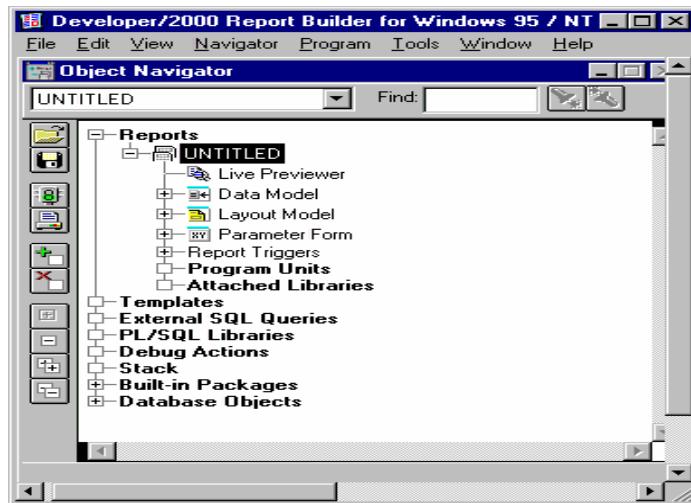
DEFINING A DATA MODEL FOR A REPORT

To specify data for a report, a data model should be defined. A Data model is composed of some or all of the following data definition objects.

- Queries
- Groups
- Columns
- Parameters
- Links



Object navigator window for report builder.



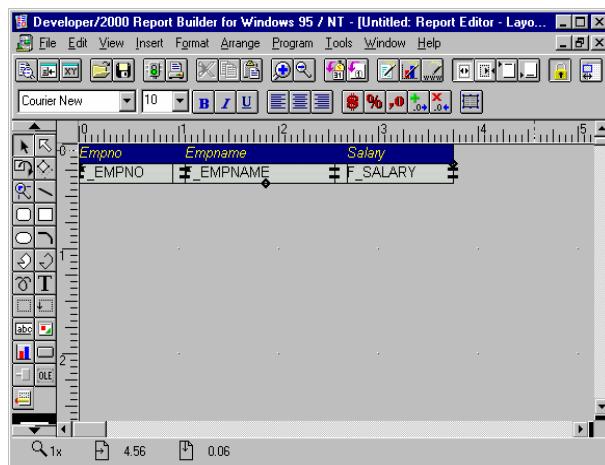
Queries:

As in Queries are SQL SELECT statements that fetch data from the Oracle database. These statements are fired each time the report is run

Groups:

Groups determine the hierarchy of data appearing in the report, and are primarily used to group columns selected in the query as shown. Oracle report automatically creates a group for each query.

Lay out editor for report builder



Data Columns:

Data Columns contain the data values for a report. Default data columns, corresponding to the table columns included in each query's SELECT list are automatically created by Oracle Reports. Each column is placed in the group associated with the query that selected the column.

Formula Columns:

Formulas can be entered in Formula columns to create computed columns. Formulas can be written using PL/SQL syntax. Formula columns names are generally preceded by CF_ to distinguish them from data columns.

Summary Columns:

Summary Columns are used for calculating summary information like sum, average etc. This column uses a set of predefined Oracle aggregate functions that can be applied to data or formula columns. Summary columns named are generally preceded by CS_ to distinguish them from data columns.

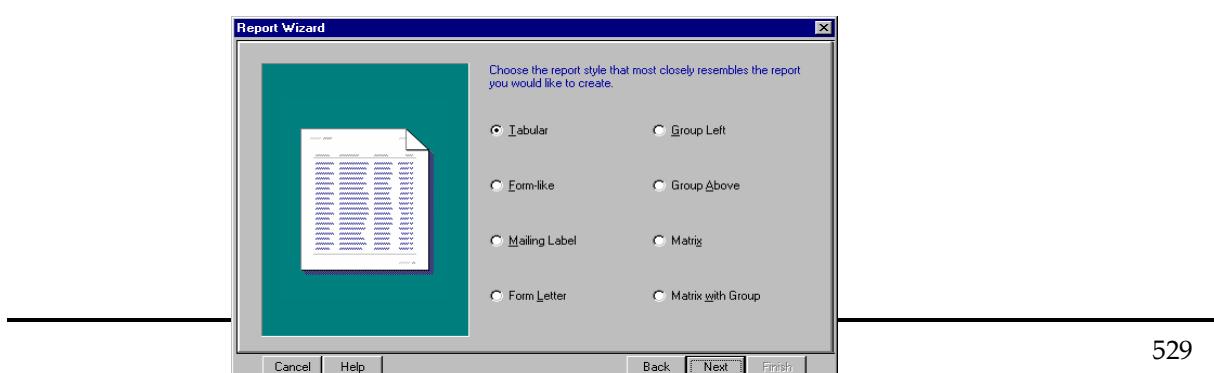
Data Links:

Data links are used to establish parent-child relationships between queries and groups via column matching.

29.3 Types of Reports

We are many types of report they are

1. Tabular report
2. Break report
3. Master/detail report
4. Matrix report



The above figure specifies the types of reports.

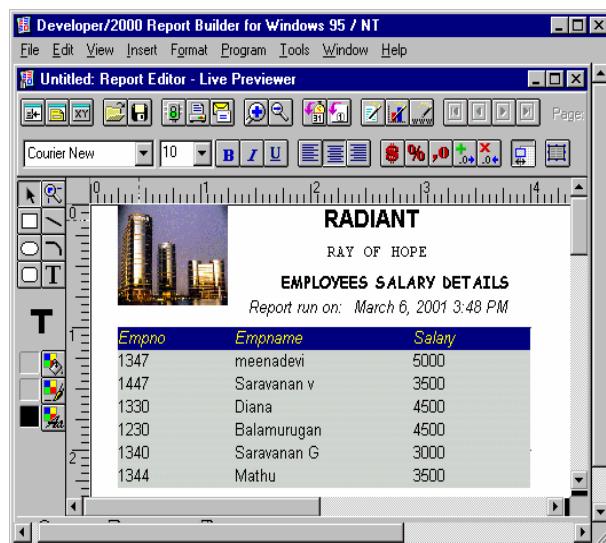
29.4 Creating a Default Tabular Report

For ease of understanding, the tables described below are populated with around 50 records so that pages handling either on the VDU or the printer can be understood.

The following steps are used to create the report shown in below:

- Create a new report definition.
- Connect to the database and select the data columns.
- Specify a default report layout.
- Save and run the report
- View the report output
- Create formula columns, specify a default report layout.
- Create summary columns, specify a default report layout.
- Create user parameters.
- Arrange the layout and run the report.

Creating a new Report Definition:



The screen that will be displayed, as soon as the Oracle Report is invoked is as shown in below. The Object Navigator will display a new report definition.

Connecting to the Database:

1. Select File, Connect menu options. The Connect dialog box will appear, prompting you for your username and password.
2. Click in the User Name field and type your username.
3. Click in the Password field and type password and connect string.

Specifying the Data and Layout for the Report:

1. After invoking the Report Builder and connecting to the database, the next step is to specify the data for the report. Click on **Tools..Report Wizard...** to start the report wizard for a new report. Report Wizard shows a number of Tab pages to enter information required for report creation. The different tab pages are:
 - Style
 - Data
 - Fields
 - Totals
 - Labels
 - Template
2. Any report created using Report Builder requires a query via which data will be retrieved for the report. The Data Tab allows creation of an SQL statement using Query Builder. The SQL statement can also be entered in the multi line edit box provided in the data tab.

This select will actually get the data of the report from a base table. The completed query property sheet will be as shown.

3. Fields Tab is used to specify the fields that must be displayed in tabular format. Select all fields by clicking on icon. Completed screen will be as displayed in following diagram.
4. click on Next. The Totals tab is displayed that allows creation of summary columns using aggregate functions. This report does not include totals for the selected fields and thus click on Next.

5. The Labels for the columns can be changed using the Labels tab. Change the Label of Sum_Quantity column to Quantity and Avg_Product _Rate to Avg. Rate. The completed screen will be displayed as shown.
6. Report can be created from templates. There are a number of pre-determined templates available in Oracle Reports Builder. By default reports are created using pre-determined template named Corporate 1 as shown.
7. Reports Builder includes a Live Previewer that is used to display the report when the Finish button is clicked, the Report Runtime is invoked as shown.
8. Reports created in Report Builder can also be displayed in the Reports Runtime Window. Click on View..Runtime Preview. Report Runtime is displayed as shown.
9. Close Report Runtime Previewer by clicking on icon. Click on View..Data Model to open the Data Model editor. Data model editor displays a query object named Q_1, a group object named as G_description and data column selected by using an SQL Select statement. The query object and the group object must be renamed appropriately.

Right click on query object and select Property Palette from the pop up menu. Set the Name property to radiant. Similarly right click on group object and select Property Palette from the pop up menu. Set the Name property of the group object to radiant. The completed data model editor is shown given below.

10. Click on File..Save to save the report. Specify the report name as radiant.rdf. Specify the path for the report file and click on OK.

The report file will be saved with the specified named.

CREATING COMPUTED COLUMNS

Computed columns calculate values based either on PL/SQL expressions or on data provided by database columns. There are two types of computed columns that can be added to a report:

Formula Columns:

Formula columns compute their values using PL/SQL expressions. Formulas can operate on

multiple values per record.

Summaries Columns:

Summary columns compute their values using build in functions of Oracle Report. Summaries operate on one value over multiple records.

Create a Formula using the Formula Column Tool:

1. A formula column can be created in the data model editor. Click on View..Data Model to open the data model editor. The Data model editor displays a query object and a group object that encloses a number of columns.
2. Increase the height of the group to make place for the formulae column. Select the formula column tool in the palette and place it after the last field in the group radiant.

A new column initially named as CF_1 is created as shown given below. Since a new column within the radiant group is created, it will be displayed as often as the other columns in the same group.

3. Set the following properties for the formulae column.
4. Click on the property PL/SQL formulae property. The program Unit editor is invoked as shown given below.
5. Enter the complete formula for the sale value for each product:

```
function radiant return Number is
begin
    return: avg_product_rate *: sum_quantity;
end;
```

The colons appear before avg_product_rate and sum_quantity because it functions as a bind variable reference; i.e. the values of sum_quantity are substituted into the formula at runtime.

6. Click on Compile. If the function is correctly typed in, the status line reports, "Successfully Compiled." Otherwise the status line reports 'Compiled with Errors,' and the Program Unit editor points out the errors in the Compilation Messages field.

7. Click on Close to close the Program Unit editor, then close the property sheet. *cf_sales_value* is now listed as a column belonging to *radiant*. The column name appears in italic, indicating that it is a user-created column.
8. Right click on the data model editor and select Report Wizard to invoke the report wizard.
9. The newly created field is shown in the available fields list item as shown.
10. Select this field as shown given below.
11. Select Label tab and change the label of the formula field to Sales Value.

Create a Report Summary using Summary Column Tool:

To create a summary that computes the total sale ie. Sum of sales value for each product:

1. Invoke the Report wizard by clicking on **Tools..Report Wizard**. Select the **Totals** tab. A list of available fields for which summary information can be specified is displayed as shown in below diagram.
2. Select *CF_sales_value* from the list and click on icon. Thus indicating that it is the sum of *CF_sales_value*. The completed totals tab screen in the report wizard will be displayed as shown in given below.
3. Click on finish.
4. Open the Data model editor by clicking on **View.. Data Model**. The summary column with a default name will be displayed as shown in the below diagram.

CUSTOMIZING REPORT LAYOUT

Customizing a report includes changing the default layout to improve the overall appearance and readability of the report.

Change Layout Settings:

- ❖ Select the item to be changed and click on Format..Font. Oracle Reports displays a

standard Font Setting dialog box. Specify a font name as Courier New, size as 10, bold.

- ❖ Similarly text for a report can be inserted by creating a new boiler text and entering required Text. Specifying the required text. Arranging the fields and texts inside the corresponding group frame, changing the format of the field, aligning the fields with respect to each other etc. can also be done.
- ❖ Notice the Fill Color, Line Color and Text Color tools located near the bottom of the Tool palette. The Line Color Tool is used to customize borders around layout objects. The Fill Color tool, is used to fill layout objects with colors and patterns and, Text Color tool, enables you to change the default text color.

Deleting Default Header and Inserting Report Header:

1. Select the text marked as '21st Century Products' and press the <Delete> Key. The default header will be deleted. Similarly select 'Report Run on' and 'Date' items and press <Delete> key.
2. Click on Text item on the vertical tool bar i.e. button on it and place it in the header. Enter the name of the company and the address as follows:

RADIANT

M.S.U. WING

TIRUNELVELI

3. Click on the alignment tool and change the text alignment to Left.
4. Similarly change the font to Times New Roman and font size as 10.
5. Select all the fields and labels by using <Shift> <Click> and use the down arrow key to move the report data lower down on the page.
6. Place another text and enter the text as. Change the font to Times New Roman and font size as 14. The completed report will be as shown in the below diagram.
7. Right click on Avg_Rate column and select Property Palette. Set the format Mask

property to NNN.NN0.00 as shown in the following diagram.

29.5 Break Report

Break report is created when repeating values for a column have to be printed only once.

To create the report shown ,

- Create a new report definition.
- Define the data.
- Create a break group
- Create summary column
- Specify a default report layout.
- Save and run the report
- Arrange the layout and run the report.

CREATING A NEW REPORT DEFINITION

1. Invoke Oracle Reports Builder and connect to Oracle. Click on Tools..Report Wizard... to create new report.
2. Select Group Left as a presentation Style as shown.
3. Enter the SQL statement as in the Data Tab. The SQL statement is as follows:
4. Group Left report requires the name of the column that must be used as a break column. In the current example, the is repeating as the multiple items are selected for every order. Thus the break column must be set to.....
5. empno is displayed in the group field list as shown....
6. The list of columns that must be displayed on the report is asked for. Click on to select all the columns.
7. The totals and calculation page is displayed next. Since the totals are not required for

the selected columns, click on text.

8. The Labels page is displayed next.
9. The Templates screen is then displayed. Select 'Cyan Grid' presentation style and click on Finish. A default tabular report with a group break on empno is displayed.....

About Repeating Frames:

Repeating frames hold data owned by their corresponding groups. Repeating frames are called so because repeating frames are repeated as many times as necessary to display all the records.

ARRANGING THE LAYOUT

Deleting Default Header and Inserting Report Header:

1. Select the images displayed on the report and press <Delete> Key. The default header objects will be deleted.
2. Click on Text item on the vertical tool bar i.e., button on it and place it in the header. Enter the name of the company and the address as follows:

**RADIANT
M.S.U. WING
TIRUNELVELI**

3. Click on the alignment tool and change the text alignment to Left. Similarly change the font to Times New Roman and font size as 10.
4. select all the fields and labels by using <Shift> <Click> and use the down arrow key to move the report data lower down on the page.
5. Place another text item and enter the text as ' '. Change the font to Times New Roman and font size as 14.

29.6 Master/Detail Report

A simple master / detail report contains two groups of data i.e. master group and detail group. For each master record fetched, only the related detail records are fetched.

Master / detail reports are similar to break reports in the way data is fetched. For every master or break group, related detail records are fetched.

In addition, if the default Group Above layout style is used format a break report, the output will look like a master/detail report as defined by Oracle Reports. The master record will be displayed in Form-Like format and the detail records will be displayed in Tabular format.

CONCEPTS

Break reports and master / detail reports can result in similar outputs but they require different data model objects. A break report uses one query and two or more groups, while a master/detail report uses two queries, each of which has one group.

Relationship between the two queries must be established in a master / detail report. This relation is set by a **Data Link**.

Data Link Object:

Data link object is a data model object, which joins multiple queries. For a simple master/detail report, two queries will be linked using primary and foreign keys of the tables from which data is selected. The Query with the primary key column i.e, the source for the master records is called the parent Query and the Query with the foreign key column I,e. the source for the detail records is called the Child Query.

Linking two tables via primary and foreign keys is similar to specifying a join condition. Information in the data link object is used to join two queries. The where clause for the join is added to the child query's SELECT statement at run time.

Layout

Master detail report uses Group above layout style in which master records display across the page with the labels to the left of their fields and the detail records appear below the master

records in tabular format.

CREATING A MASTER/DETAIL REPORT

Reports as shown

- Create a new report definition by creating 2 queries for master and detail tables and link them.
- Create formula and summary layout.
- Specify a default report layout.
- Arrange the layout
- Save and run the report

CREATING THE REPORT DEFINITION

Master detail report can be created using multiple queries and selecting 'Group Above' presentation style.

Creating a Master Detail Report using Multiple Queries:

A master detail report can be created using two queries. The first query is used to extract data for the master section and the second query is used to extract data for the detail section.

29.7 Matrix Report

Introduction to matrix report

A matrix report is a summary report that presents the desired data with headings across the top and the left side. Matrix report data is displayed at the intersection of the top and left heading. The totals are displayed across the bottom and rights side. A matrix report is also referred to a 'CROSS-TAB report'.

What is matrix report?

A matrix report is a cross tabulation of four sets of data:

1. one set of data is displayed across the page I,e values are placed one besides the other.

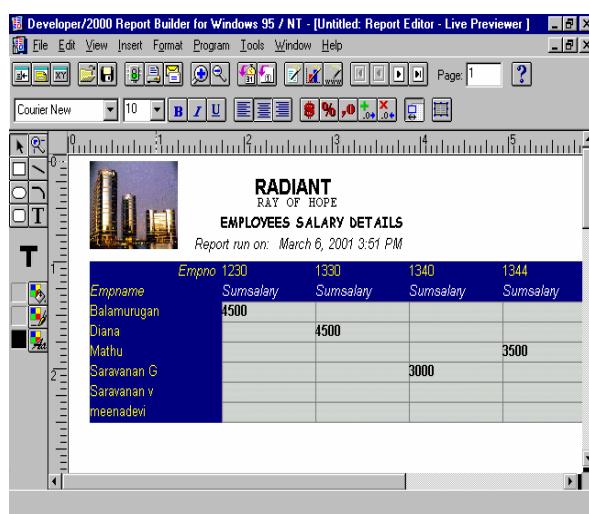
2. one set of data is displayed down the page i.e values are placed one below the other.
3. one set of data is the cross product, which creates data cells at the intersection of the across and down data.
4. onset of data is displayed as the filler of the cells I,e the values are used to fill the cells created in step3.

Types of matrix reports

Different types if matrix reports can be created using Oracle Reports. The four general types of matrix reports are

- simple matrix
- Nested matrix
- Matrix Break
- Multi-Query matrix with break

Before discussing the particulars of matrix building, the objects required in the data model editor and the objects required in the layout editor must be known.



Objects in data model:

In building matrix data model, the following must be taken into consideration:

1. Number of queries
2. Group structure
3. Summary settings

Number of Queries:

Two types of query structures can be used for a matrix data model.

One-Query Matrix

A matrix report can be created from a single query where a single SQL Select statement is used to fetch data from the database. Reports Builder toll then groups the data fetched from the database to construct a matrix report.

Multi-Query Matrix

Multi-query data model can be considered because it results into simple queries as compared to one query data model. In addition, a multi-query data model is required for nested matrix reports.

Groups Structure :

Matrix reports are built with four or more groups:

Two or more dimension groups:

In the layout, the information in at least one groups goes across the page, and the information in at least one group goes down the page, forming a grid. The groups that provide the data to be displayed across or down the page are called **Dimension Groups**

The information in Dimension groups is sometimes referred to as Matrix labels as they provide column and row labels for the matrix. Dimension groups are contained within the

cross product group.

One or more cross product groups:

In the layout, a **Cross Product Group** is represented by the intersection of the across and down dimension groups. Thus the cross product group shows all possible combinations of the values of the across and down dimension groups.

When the report is run, it expands, and each instance of data intersection becomes a separate cell.

One cell or filler groups

The **Cell Group** contains data that is used to fill the cells created by the cross product group. When the report is run, these values appear in the appropriate cells.

Matrix Layout:

A matrix layout model must consist of the following layout objects:

1. At least two repeating frames, one with a Print Direction set as Down and own with a Print Direction set to Across.
2. Several group header, and footer (if summaries are included) frames.
3. A matrix object created for the cross product group that includes cells of the matrix.
4. Boilerplate for each column and row of values, as well as for summaries.

Matrix / Cross Product Object

Matrix object defines the intersection of at least two repeating frames. The repeating frames are the dimensions of the matrix and the matrix object contains at least one field that holds the filler or values of the cell group.

One matrix object must be created for each pair of intersecting repeating frames in the layout.

One of the repeating frames must have a Print Direction of Down and the other must have a Print Direction of Across in order to form a matrix. Matrix object can be created using cross product button on the toolbar.

Matrix reports are different from tabular reports because the number of columns is not known in advance; I,e the number of columns in the report is not determined by the number of columns specified in the SELECT statement plus the columns created. The number of columns in the report depends on the number of values contained in the columns providing the horizontal labels.

The Solution :

- Create a new report
- Create a query
- Create Groups
- Create Default Layout
- Create Summary columns.
- Add zeroes in place of non-existent values.
- Create user parameter.

29.8 Report Triggers

To extend report writing capabilities , Oracle Reports Builder enables the use of PL/SQL code blocks. If conditional logic in required in a report PL/SQL constructs can be used to perform the job.

Examples

If a sales value is below the specified target, then such sales values can be displayed with a different text color. This helps to draw attention to these values.

PL/SQL code blocks can be used in different types of triggers available in Oracle Reports:

1. Formula Triggers
2. Formula Triggers
3. Action Triggers
4. Validation Triggers

5. Report triggers
6. Groups Filters

Each of them are explained below:

Formula triggers:

Formula triggers are PL/SQL functions that populate columns of type formula. A pl/sql function is required for any column of type formula. The data type of the formula column determines the return data type for the formula trigger. An example of a formula trigger is as follows:

Function radiant Return Number is

```
Begin
    Return  :avg_product_rate * :sum_quantity;
End;
```

Formula triggers are created as the name of the formula column followed by the reserve word formula. The value returned by the formula trigger is displayed in the report field connected to the formula column.

Format triggers:

Format triggers are PL/SQL functions executed before the object is formatted. These triggers can be used to dynamically change the formatting attributes(e.g. font, font weight etc.) of any object on the report. They are also used to conditionally print or not print a report column value.

Format triggers return Boolean values TRUE or FALSE. If the return value for the format trigger is false, the value is not displayed in the report.

Action Triggers:

Action triggers are PL/SQL procedures executed when a button is selected in the previewer. This trigger can be used to dynamically call another report or execute any other PL?SQL. it can also be used to execute a multimedia file.

Action triggers are used to perform user-defined action and these triggers do not return any value.

Validation Triggers:

Validation triggers are PL/SQL functions that are executed when a parameter value is entered and the cursor moves to the next parameter. These triggers are used to validate parameter values.

Validation triggers return Boolean value TRUE or FALSE. If the return value is TRUE, the parameter is treated as validated. If the value is FALSE then a default error message is displayed and an exception is raised.

Rules for the validation triggers.

Restrictions defined for validation triggers. These are:

The PL/SQL code in a validation trigger can be a maximum of 32k characters.

In a validation trigger, the values of oracle reports parameters can read or values can be assigned to them. Report column values cannot be read nor can values be assigned to report columns in a validation trigger.

Report triggers:

Report triggers enable execution of PL/SQL functions at specific times during the execution and formatting of a report. Customization of formats, initialization tasks, accessing database etc can be done using the conditional processing capabilities of PL/SQL in these triggers.

Oracle reports have five global report triggers. The trigger names indicate at what point the trigger fires:

1. Before parameter form fires before the runtime parameter form is displayed. From this trigger, the values of parameters, PL/SQL global variables and report-level columns can be accessed and manipulated if required.

2. After parameter form fires after the runtime parameter form is displayed. From this trigger, parameter values can be accessed and their values can be validated. This trigger can be used to change parameter values or, if an error occurs, return to the runtime parameter form.
3. Before report fires before the report is executed but after queries are parsed and data is fetched.
4. Between pages fires before each page of the report is formatted, except the very first page. This trigger can be used for customized page formatting.
5. After report fires after the report previewer is exited, or after report output is sent to a specified destination, such as a file or a printer. This trigger can be used to clean up any initial processing that was done. For example if temporary tables are created during report execution, such tables can be deleted in the After report trigger.

Which trigger to use:

As a general rule, any processing that will effect the data retrieved by the report should be performed in the before parameter form or after parameter form triggers. These are the two report triggers that fire before any SQL statement is parsed or data is fetched.

Any processing that will not effect the data retrieved by the report can be performed in the other triggers.

Return values for the report triggers:

Report triggers return Boolean value TRUE or FALSE. The impact of the return value varies based on the type of report trigger.

Report trigger type	Impact of the Return Value
Before Parameter Form	If the return value is FALSE, an error message is displayed and then the focus returns to the place from where the report was executed.
After parameter form	If the return value is FALSE, the focus returns to the runtime

	parameter form. If the parameter form is not displayed then the focus returns to place from where the report was executed.
Before Report	If the return value is FALSE, an error message is displayed and then returns to the place from which the report was executed.
Between pages	If the return value is FALSE, an error message is displayed and then the focus returns to the place from where the report was executed. <u>If the trigger returns FALSE on the last page, nothing happens because the report formatting is complete.</u> The between page trigger does not fire before the first page is displayed. An error message is displayed when the focus moves to the second page if the report consists of two or more pages.
After report	If the return value is FALSE, it does not affect the report. A message can be displayed to indicate that the report displayed is probably incorrect.

The order of report trigger execution is as follows:

BEFORE PARAMETER FORM TRIGGER is fired



Runtime parameter form appears



AFTER PARAMETER FORM TRIGGER is fired



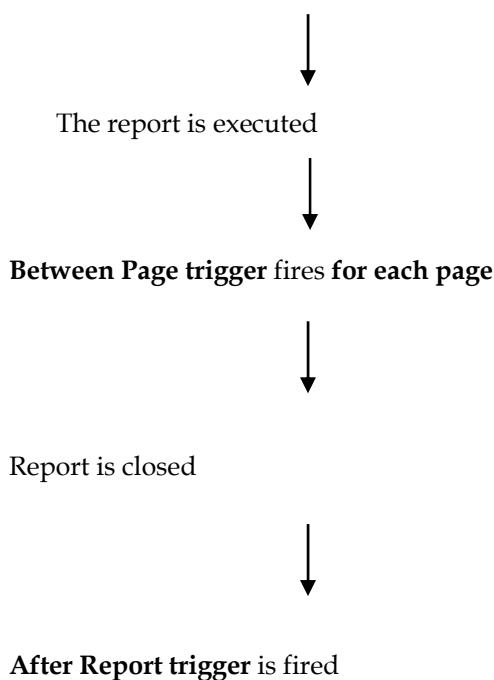
Report is “complied.”



Before Report trigger is fired



Queries are parsed and data is fetched



29.9 Short Summary

- ❖ Define a data model for a report
 - ♣ Queries
 - ♣ Groups
 - ♣ Data links
- ❖ Specify the layout for a report
 - ♣ Report layout formats
 - ♣ Frames
 - ♣ Repeating frames
 - ♣ Fields
- ❖ Creating a default tabular report
 - ♣ Creating a new report definition
 - ♣ Connecting to the database
 - ♣ Specifying data and layout for the report
 - ♣ Creating formula columns
 - ♣ Creating summary columns

- ♣ Customizing report layout
- ❖ Specifying report definition for a break report
 - ♠ Selecting break fields for a break report
 - ♠ Creating a master/detail report
- ❖ Creating a master/ detail report using multiple quires
 - ♣ Creating a data link object
 - ♣ Creating formula columns
- ❖ Types of matrix reports
 - ♣ Single query matrix report
 - ♣ Multiple query matrix report
- ❖ Matrix group structure
 - ♣ Dimension groups
 - ♣ Cross product groups
 - ♣ Filler groups
- ❖ Types of report triggers
 - ♠ Format triggers
 - ♠ Action triggers
 - ♠ Validation triggers
 - ♠ Group filter triggers

29.10 Brain Storm

1. What is report?
2. Explain the types of report
3. What is a trigger?
4. Explain the types of triggers?
5. Explain the different types of matrix report?

©CGR

Lecture 30

Graphics

Objectives

After completing this lesson, you should be able to do the following

- ☒ Discuss about Introduction to Graphics
- ☒ Describe about Using Graphics Builder
- ☒ Describe creating a graph

Coverage Plan

Lecture 30

- 30.1 Snap shot
- 30.2 Introduction to graphics
- 30.3 Using graphics builder
- 30.4 Creating a graph
- 30.5 Short summary
- 30.6 Brain Storm

30.1 Snap Shot

In this session we deal the following

- ♠ Creating a display modules.
- ♠ Graph presentation styles.
- ♠ Chart template.
- ♠ Passing parameters to a display module.
- ♠ Calling display module from forms.
- ♠ About PL/SQL library.

30.2 Introduction to Graphics

A graph is a form of information representation, which can visually display a lot of information in a compact form. A graph is a mechanism that allows organization staff to visually analyze information on-line. Using graphs data can be retrieved from a variety of sources and displayed, using charts, line drawing, bit-mapped images etc.

Working with Charts:

Graphs comprise of the following components:

- ❖ Displays
- ❖ Layout
- ❖ Queries
- ❖ Chart Properties
- ❖ Templates

Display:

The primary object of Graphics Builder is the Display. A display module is nothing but a collection of objects such as Layout, Template, Queries and Triggers.

Layout:

The Layout editor is the main work area in Oracle Graphics Builder. It is where the layout of a display is designed, is designed, by creating and modifying graphical objects (lines, polygons, rectangles, text, etc.). Every display contains exactly one layout, and every layout belongs to exactly one display. The contents of the layout are shown when the Display is executed. The information presented can be viewed or the display can be manipulated by using a mouse or keyboard.

Queries:

Query is a SQL SELECT statement of file that Oracle Graphics needs in order to retrieve data for a chart. A chart can be associated with only one query at a time. When a chart is drawn, the data from the current query is used.

Chart Properties:

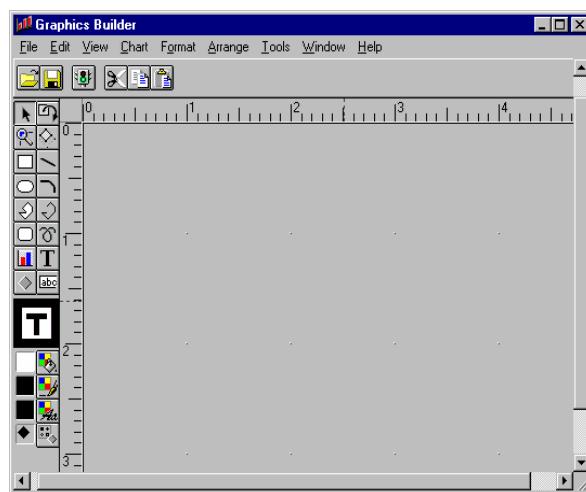


Chart properties define options for the chart as a whole, including its associated chart template and query. Chart properties are accessible through the **Chart property sheet**, which is a multi tab page object. The tab pages are organized as follows:

Chart : Includes the chart's Name, Title, Chart Type and Template.

Data : Includes query related properties such as a Query Filter Function and the range of data to plot.

Categories :Includes the chart categories and any sub-categories to plot in the chart. Category data is plotted at fixed intervals. Category data is not considered to be mathematically related and is usually plotted along what is called the Discrete axis.(ie., X axis)

Values : Includes the chart values. Value dependent data generally starts at one value and continues until another value. Value data is considered to be mathematically related and is usually plotted along what is called the Continuous axis.(i.e. Y axis)

PL/SQL : Includes any mouse events and associated procedures for the chart.

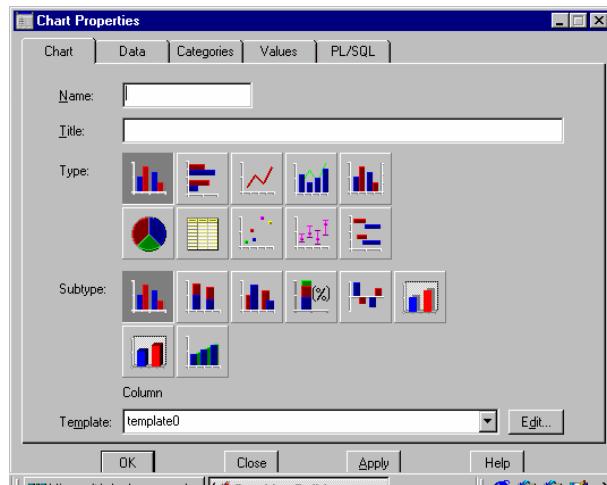


Chart Types:

The following list describes the types of charts available that can be created along with their usage.

Column : Column Charts type is used to compare sets of data ranges vertically. For example, a Column Chart can be used to show the quarterly sales revenue generated by each sales representative.

Bar : Bar Chart type is used to compare sets of data ranges horizontally. For example, a bar chart can be used to compare monthly sales.

Line : Line chart type is used to show vertical changes for a specific set of data. For example, a line chart can be used to gauge daily or weekly changes in a stock's value.

Mixed : Mixed chart combines multiple plot types such as column and line. For example, a mixed chart can be used to show daily sales revenue, with a line plot type to provide a summary view of the data.

Double-Y : Double Y chart type provides two independent Y-axes on which to plot data. Each Y-axis can show a different range of values. For example, one axis can be used to represent revenue and the second axis can be used to represent income, for a specific month or year.

Pie : Pie Charts type is used to compare the ratios or percentages of parts of a whole. For example, a pie chart can be used to compare annual revenue by department or quarter.

Table : Table Chart type is used to show data in a table format. For example, a table chart can be used to show an Employee Organization Chart.

Scatter : Scatter chart type is used to show data along two-value axis. Scatter charts are well suited for showing standard deviation. For example, a scatter chart can be used to plot the target and the actual sales for each salesmen. If there is a correlation between the two sets of data, the points will be grouped together closely. One or more points well outside the group could indicate a disparity.

High-low : HighLow chart type is used to show fields that correspond to high, low, and close (ie.,stock prices). For each row in a query, a high-low range is plotted on the chart.

Gantt : Gantt chart type is used to show sets of project data over a given amount of time. Gantt charts are generally used to show project milestone timelines.

Chart Templates:

Template is the format of a chart and its properties. A chart can be associated with only one chart template at a time. Each chart template can be associated with one or more field templates, which define the properties of the individual fields, such as bars, columns, pie slices, or lines.

Using a single chart template, multiple charts with same set of basic characteristics such as grid line settings or tick label rotation etc. can be created. For example, instead of specifying the properties for each chart individually, a single chart template can be created and

associated it with multiple queries. The Chart Template Editor is used to work with chart templates.

30.3 Using Graphics Builder

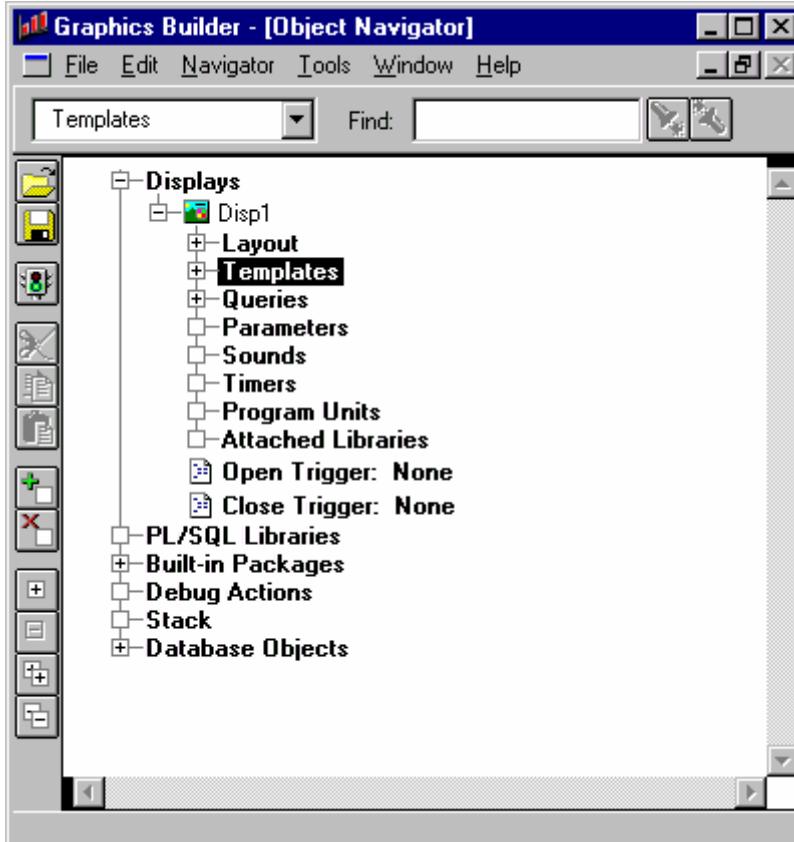
Tools Available with the Graphics Builder:

The tools available with the Graphics Builder must be looked at before attempting to create a chart so that chart creation is made simpler. These are:

1. Object Navigator
2. Layout Editor
3. PL/SQL editor

Object Navigator:

Queries and templates are constituent parts of a Display. The Object Navigator allows navigation through the hierarchy of objects.



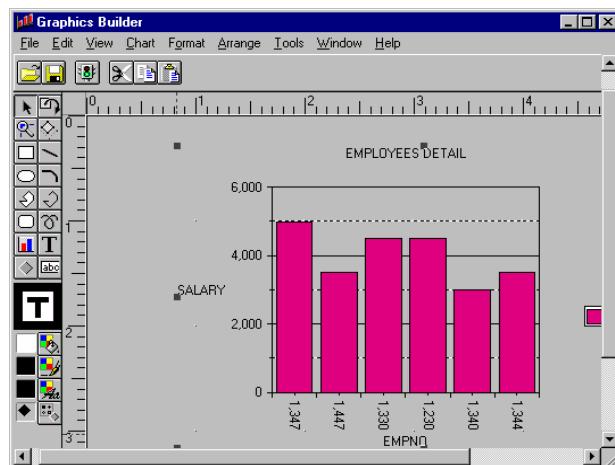
Layout Editor:

The developed charts will be displayed in the layout editor. The sizing, positioning and alignment of the chart objects on the screen are done through this tool.

PL/SQL Editor:

More often than not, there is a need to perform tasks, which are specific to an application. Such tasks can be handled by writing code to customize application processing and to meet these specific requirements. The PL/SQL Editor gives an interface to define code blocks called Trigger Code for appropriate trigger events.

30.4 Creating a Graph



To design a simple Line graph for sales order transactions. The Graph earning for the year grouped by month.

The table definitions are as follows:

Table Name : radiant (Master)

Description : Use to store information about employees salary.

Table Name : Ray

Description : Use to store information about projects.

The Solution:

1. Invoke Oracle Graphics Builder Tool. When the Graphics Builder Tool is invoked, a new display with a default name disp1 is created and opened in the Object Navigator window as shown in the below diagram.
2. Connect to the database by clicking on **File** and **Connect**.

Creating a Query for the Graph:

1. From the Menu bar select **Chart..Create Chart...** Since a graph is created using a query, a **Chart Genie - New Query** dialog box is displayed.

This dialog box includes two tab pages i.e. **Query** and **Data**. The **Query** tab includes query properties like Query Name, Query Type.

Query Name :The default name given to the newly created query is query0.

Query Type :The Query Type property determines the data source for a graph. Different options available for the Query Type are:

- SQL Statement
- SYLK File or a WKS File – These are spreadsheet file formats.
- PRN is a printer file.
- External SQL File.

The default Query Type is SQL Statement and thus an edit box to enter the SQL statement is available.

1. Change the name of the query object to radiant.
2. Enter the SQL Statement in the **SQL Statement** option.

Select * from radiant;

3. Check the data. If the displayed data is correct click on OK. A Chart Properties dialog box is displayed. Enter the Name, Title for the chart and select the required Type of Graph and its Subtype as shown in following diagram.
4. Click on **Apply** and then OK button to accept the values. The completed graph will be displayed as shown in the following diagram
5. Click on **File..Save**. A display can be saved in the database or as a file. Thus, a confirmation to save the file as the 'File System' is asked for. Click on OK. Save As dialog box is displayed. Select appropriate drive and directory and enter the file name as **radiant.odg** as shown in the following diagram.
6. Click on **File..Run** to execute the display. The layout editor will be replaced with the Graphics Debugger window. The values displayed for the Categories axis are displayed vertically. Click on **File..Exit Runtime** to return to the layout editor.

To change to horizontal display, double click on the Category values in the layout editor. The Axis Properties dialog box is displayed.

Change the **Tick Label Rotation** property to horizontal by clicking on the appropriate radio button.

PASSING PARAMETERS IN ORACLE GRAPHICS

The graph created so far is static. It does not have the ability to accept values and display a graph accordingly. To make graphs dynamic, it would be essential to define parameters that accept input values and pass the values to the graph.

30.5 Short Summary

- ♠ Creating a display module
 - ♣ Creating a display module in graphics builder
 - ♣ Using graphics builder tool
- ♠ Graph presentation styles

- ♣ Graph types
 - ♣ Graph sub types
 - ♣ GRAPH CATEGORIES AXES
 - ♣ Graph values axis
- ♠ Chart template
- ♣ Changing graph presentation by using a template
- ♠ Passing parameters to a display module
- ♣ Creating a user parameter
 - ♣ Setting properties of a user parameter
 - ♣ Using user parameter in graph query
- ♠ Calling display module from forms
- ♣ Need to call a display module from a form
 - ♣ Passing parameters to the called display module
 - ♣ Use of PL/SQL library
 - ♣ Calling display module using open() procedure

30.6 Brain Storm

1. What is graph?
2. Why we need graphics builder
3. How to create graph explain.
4. Explain the types of graph
5. What are the properties we used in graphics.

END

MANONMANIAM SUNDARANAR UNIVERSITY
Centre for Information Technology and Engineering
Tirunelveli – 627 012

Syllabus for M. Sc. (IT&EC) / MIT

2.1 RDBMS Concepts and Oracle 8i

- Lecture 1 Relational Database management system definitions - Basic concepts – Data – Database -Database user – Database system - Database Modeling.
- Lecture 2 Entity relationship model – entity – attributes – Relationships - mapping cardinalities.
- Lecture 3 RDBMS concepts - Object based data models – OLTP – DSS – Transaction - large Database management control - Data concurrency – Data Consistency – Data Integrity.
- Lecture 4 Normalization – Redundancy and Inconsistent dependency – first normal form – second normal form – Third normal form - Denormalization.
- Lecture 5 Codd's rule – Data Integrity – Keys.
- Lecture 6 Client server architecture – clients – servers – Types of servers – multithreaded server architecture – advantages of client/server computing.
- Lecture 7 Front ends – open clients - ODBC support – Character based clients – GUI based clients.
- Lecture 8 Introduction to SQL - Generation of Languages - SQL - SQL Command classification - Data types - Create, view, manipulate data.
- Lecture 9 Data integrity - Adding Constraints
- Lecture 10 Operators and functions - Expressions - functions - Scalar functions - Aggregate functions
- Lecture 11 Joins and Sub queries - joins tables - sub queries - Correlated sub query - Inline queries - pseudo columns
- Lecture 12 Database objects - privileges - Database objects - synonyms – sequences Understanding Database structure - Access Methodologies.
- Lecture 13 Introduction to PL/SQL - PL/SQL Architecture - Declares Variables - Control structures - iteration control
- Lecture 14 Understanding cursors - types of cursors - cursor variables - eg. For using constrained cursors.
- Lecture 15 Errors –Exceptions - User-Defined Exceptions – Unhandled Exceptions
- Lecture 16 Introduction to Subprograms - Procedures to Parameters - Functions of Notations - Procedures Vs Functions.

- Lecture 17 Packages - Specification - Package body - Overloading of Package
- Lecture 18 Triggers-parts of triggers-types of triggers-creating triggers-cascading triggers-mutating and constraining tables - enabling and disabling triggers
- Lecture 19 Oops and object concepts - why objects - Oracle supported Data types - Collection types - Object views - member functions - using order method.
- Lecture 20 Introduction to DBA - Open cursor - Execute - Oracle DBA.
- Lecture 21 Java strategies - JDBC - Java with SQL - Java virtual machine - Java stored procedures.
- Lecture 22 Loading Java - Drop Java utilities - registering Java - Oracle Internet file system.
- Lecture 23 Introduction to forms - using forms builder - form wizards - creation of forms.
- Lecture 24 Property class - visual attributes - library - alerts - object libraries - editors.
- Lecture 25 Master detail form - creation of master detail form - triggers - validations
- Lecture 26 Working with LOV objects - using multiple canvases - types of canvas - Object groups - Parameters - record groups.
- Lecture 27 Stacked canvas - tab canvas- Horizontal toolbar canvas- List item
- Lecture 28 Using menu - saving compiling and attaching a menu module - checked, enabled menu items.
- Lecture 29 Introduction to reports - types of reports - tabular reports - break report - master detail report - matrix report - report triggers.
- Lecture 30 Introduction to graphics - using graphics builder - creating a graph.

 Best of Luck