

ORACLE

Developed By:
Jignesh Dhol



Atmiya

Infotech Pvt. Ltd.

Contents

Ch:1	<u>Introduction to RDBMS</u>
Ch:2	<u>SQL, SQL * Plus</u>
Ch:3	<u>Managing Tables and Data</u>
Ch:4	<u>Other ORACLE database objects</u>
Ch:5	<u>Transaction and Decision Control Language</u>
Ch:6	<u>Introduction to PL/SQL</u>
Ch:7	<u>Advanced PL/SQL</u>
Ch:8	<u>Oracle Database Structure</u>

Ch: 1 Introduction to RDBMS

Top:1	<u>What is Database Management System ?</u>
Top:2	<u>Database Models</u> Hierarchical Model - Network Model - Relational Model
Top:3	<u>What is Relational Database Management System ?</u>
Top:4	<u>Difference between DBMS / RDBMS</u>
Top:5	<u>E - R diagram</u>
Top:6	<u>Types of Relationship</u> One to One - One to Many - Many to Many
Top:7	<u>Normalization</u>
Top:8	<u>Codd's Rules</u>

Ch: 2 SQL, SQL *Plus

Top:1	<u>Introduction to SQL</u>
Top:2	<u>SQL Commands and Datatypes</u>
Top:3	<u>Expression, Conditions and Operators</u>
Top:4	<u>SELECT statement</u>
Top:5	<u>Special Operator</u>
Top:6	<u>Join, Subquery</u>

Ch:3 Managing Tables and Data

Top:1	<u>Creating and Altering tables (Including Constraints)</u>
Top:2	<u>Data Definition Language</u>
Top:3	<u>Data Manipulation Commands</u> like Insert, Update, Delete and Alter
Top:4	<u>Functions</u> Aggregate, Date - Time, Arithmetic, Character, Conversion, Miscellaneous
Top:5	<u>SQL * Plus</u>

Ch: 4 Other ORACLE database objects

Top:1	View
Top:2	Sequence
Top:3	Synonyms
Top:4	Index
Top:5	Database Links

Ch:5 TCL and DCL

Top:1	<u>What is transaction ?</u>
Top:2	<u>Starting and Ending of Transaction</u>
Top:3	<u>Commit, Rollback, Save Point</u>
Top:4	<u>Grant, Revoke</u>
Top:5	<u>Role, Creating Users, Change Password</u>

Ch:6 Introduction to PL/SQL

Top:1	<u>SQL v/s PL/SQL</u>
Top:2	<u>PL/SQL Block Structure</u>
Top:3	<u>Language construct of PL/SQL</u> (Variables, Basic Datatypes, Composite Datatypes, Conditions looping etc.)
Top:4	<u>%TYPE and %ROWTYPE</u>
Top:5	<u>Using Cursor (Implicit, Explicit)</u>

Ch: 7 Advanced PL/SQL

Top:1	<u>Creating and Using Procedure</u>
Top:2	<u>Functions</u>
Top:3	<u>Package</u>
Top:4	<u>Trigger</u>
Top:5	<u>Creating Objects</u>
Top:6	<u>PL/SQL Tables</u>
Top:7	<u>Nestead Tables</u>
Top:8	<u>Varrays</u>

Ch:8 Oracle Database Structure

Top:1	<u>Initialization Parameter</u>
Top:2	<u>Control Files, Redo Logs files</u>
Top:3	<u>Processes</u>
Top:4	<u>Tablespace (Create, Alter, Drop)</u>
Top:5	<u>Oracle Blocks</u>
Top:6	<u>Import, Export</u>
Top:7	<u>SQL * Loader</u>
Top:8	<u>Instance Architecture</u> 1. Database Processes 2. Memory Structure

TOP:1 What is Database Management System ?

C. J. Dates define a database system, as a computer base record keeping system whose overall purpose to record and maintain information. In other word, database is a collection of related records and a set of programs to access this data. Because it is an entire system and enables ones to enter, store and manage data it is call Database Management System.

Modern Database Management System comes in many different classifications, and with many different capabilities, but in general they try or accomplish three things.

Data consolidation refers to the combining or unifying of separate data file into centralized structure, and storing data in a non-redundant format. A redundant format is a structure is that stores the same data item two or more location. For instance, as seen in the examples above, if within a company an employee address is stored not only by the HRD Department in the employee history file, but also the account department in the payroll file, the employee own department in a project file, etc. then you have non centralized structure carrying redundant information. An integrated (non-redundant) system stores the employee's address stores only one location.

Data Sharing

Data sharing refer to the ability of the system to aloe multiple user concurrent access to the individual pieces of data in the database. You can think of the database as a 'pool' of sharable information.

Data Protection

Data protection refers to the ability of a database management system to maintain integrity of its data in the face of certain type of processing adversity such as crashes, program failures, etc. if this type of events occurs, the DBMS must have the ability to back out (or undo) incomplete or erroneous changes to data stored in the database.

How is a Database System Beneficial?

- The amount of redundancy in the stored data can be reduced.
- No more inconsistencies.
- The stored data can be shared.
- Standards can be set and followed.
- Data integrity is maintained.
- Security of data can be implemented.
- Data independence.

DBMS Users

- The Database Designers
- The Database Administrator or DBA
- The Application Programmer
- The actual End-Users of the application



Atmiya
Infotech Pvt. Ltd.

TOP:2 Database Models

Database Management Systems organize data in what is known as data model. You can think of a data model as the infrastructure of the data organization, in other words how the data is presented to the user. There are three basic data models:

- The Hierarchical Model
- The Network Model
- The Relational Model

(i) The Hierarchical Model

One of the earliest database management systems was based on the Hierarchical Model. In a hierarchical data model the records have a parent-child relationship. The application used was Production planning for automobile manufacturing companies. The model of database is shown in following figure. An automobile manufacturer may manufacture various models of car. Each car model was decomposed into its assemblies (Engine, Body and Chassis). Each assembly is further decomposed into sub-assemblies (valves, spark plugs, etc.) and so on. If a manufacturer wanted to generate the Bill of Materials for a particular model of an automobile the hierarchical data model would be very suitable because the bill of materials for a product has hierarchical structure. Each record represents a particular part and since the records have a parent-child relationship each part is linked to its sub-part. The hierarchical model supports multiple occurrences

of the same record type.

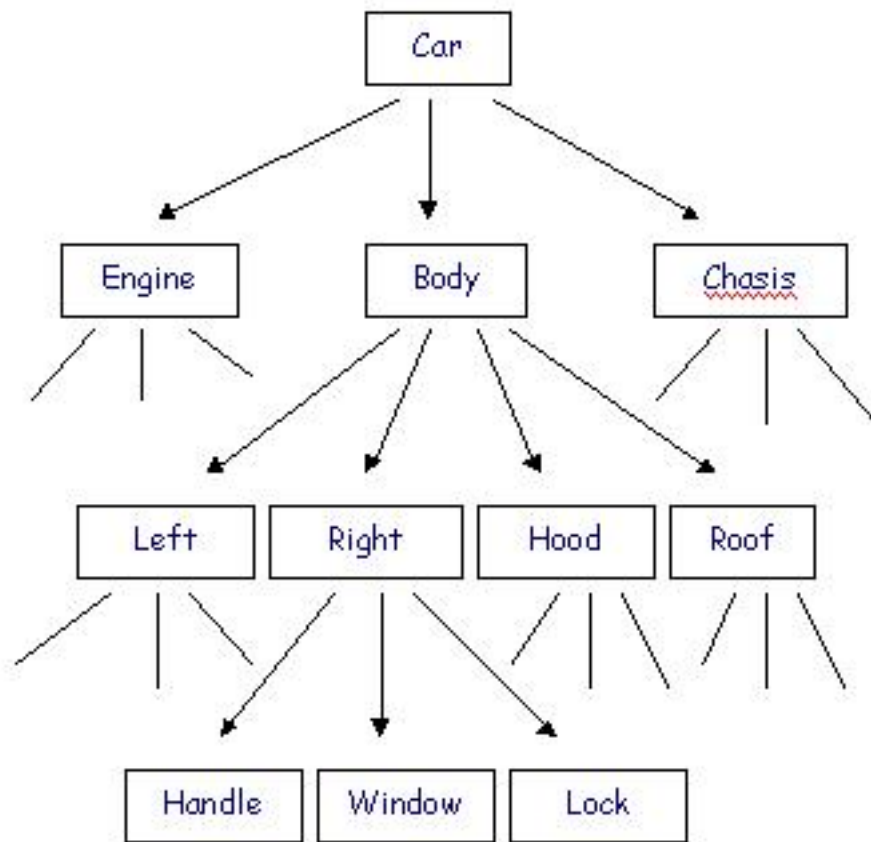
One of the most popular hierarchic database management system was IBM's Information Management System (IMS) introduced in 1968. IMS is still most widely used DBMS in IBM mainframes.

The Characteristics of DBMS are:

- **Data is represented as hierarchical trees.**

The hierarchical database is characterized by parent-child relationship between records. A record type, R_1 , is said to be the parent of record type, R_2 , if R_1 is one level higher than R_2 in the hierarchic tree. The root of the hierarchy is the most important record type and all records at different levels of the hierarchy are dependent of the root. Each child record has only one parent record. The parent record can have one or more children record type.





(Fig. Shows Hierarchical Model)

- **Represents a set of related records.**

There can be one or more than one record occurrences for given record type. When writes into database, one occurrence of record of the record type is written. Similarly, whenever a record is retrieved from the database, one occurrence if the record type is retrieved.

- **Hierarchy is established through pointers.**

In the hierarchic database, the pointers link the records. Pointers determine whether a particular record occurrence is a parent of child record and path from parent to the child.

- **Simple structure**

The database is simple hierarchical tree. The parent and child

records can be stored close to each other on the disk, minimizing disk input and output. The hierarchic data model is simpler than a network model.

- **High performance**

The parent-child relationship is stored as a pointer from one record to another; hence navigation through the database is very fast resulting in high performance.

- **Relationships between record types are pro-defined**

The hierarchical DBMS is based on the hierarchic tree structure in which the parent-child relationship is supported. A record type, R_1 , is said to be the parent of record type, R_2 , if R_1 is one level higher than R_2 in the hierarchical tree. Records types at different level of the hierarchy are dependent on the root, which is most important record type in the hierarchy. Since the relationships are predefined, flexibility is lost but a high performance compared to other data models is achieved.

- **Tedious to reorganize.**

It is tedious to reorganize the database because the hierarchy has to be maintained. Each time a record type is inserted or deleted, the pointer have to be manipulated to maintain the parent-child relationship. The reorganization is static and appropriate changes have to be made to the application programs.

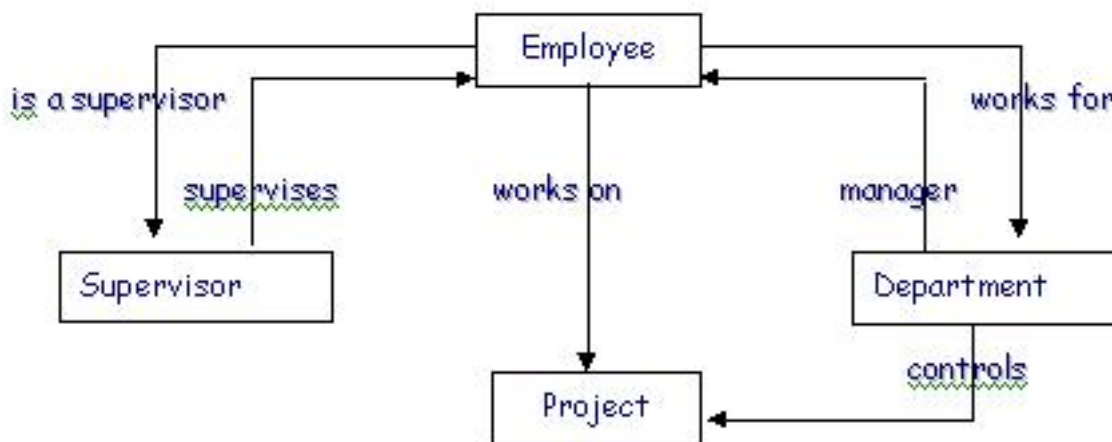
- **Real life requirement are more complex**

The hierarchic DBMS is based on a simple parent-child relationship, but real life applications are more complex and cannot be represented by a hierarchic structure. In an order-processing database, a single order might participate in three

different parent-child relationships linking the order to the customer who placed the order, the items ordered and the sales person who took the order. This complex structure cannot be represented in a hierarchical structure.

(ii) The Network Model

To overcome the problem posed by the hierarchical data model, the network model was developed. The network model modified the hierarchical model by allowing multiple parent-child relationships. This relation is known as set in network model was developed. The network model together with the hierarchical data model was major a data model for implementing numerous commercial DBMSs. The network model structure and language construct were defined by the CODASYL (Conference on Data Systems Language).



(Fig. Shows Network Model)

The characteristics of a network DBMS are:

- Data record types are represented as a network.
- A network is used when hierarchy is not established or when a record participates in more than one relationship.

- Each sub-module can have one or more super-ordinate modules. Since each multiple parent child relationship is supported child record type could have one than more parent record types.
- Represents a set of related records.

The sets that support multiple parent-child relationships and the structure of the record have to be specified in advance.

- Complex structure

Since multiple parent-child relationship is supported, database structure is very complicated. The network database implements sets that support multiple parent-child relationships. The sets have to be specified in advance. In the tradeoff between flexibility and performance, a network model is not very flexible to reorganize but has high performance level.

- Difficult to reorganize

The network database is very difficult to reorganized because insert and deleting a record would trace the pointers and changing the appropriate links.

- Navigation done by the programmer

The programmer will have to write 3-GL programs specifying the relationship and direction in which to navigate in the database.

- 3-GL needed to program database

To access records the programmer has to navigate the database record-by-record. Program will have to be written specifying to which relationship to navigate and the direction.

- 3-GL inadequate for handling sets.

The records network model are processed one set at a time. 3-GLs handle only one record at a time and hence are inadequate for handling sets.

- Query facility not available

Network database management system do not have any query facility and hence 3-GL programs will have to written specifying the path and the relationship.

(iii) The Relational Model

An IBM research scientist Dr. E. F. Codd, was unhappy with the way the DBMSs available in those day handled large volumes of data. He felt the need to apply the rules and decline of mathematics to help address the problems associated with the earlier models as

Data integrity

Data redundancy

In June 1970, he presented his paper titled ' A Relational Model of Data for Large shared Databanks'. This paper actually laid down 12 rules. Which a true RDBMS would have to satisfy.

The term 'Relation' is derived from the set theory of mathematics. The basis characteristics of a relational model are discussed here.

First, in a relation model, data is stored in relations. 'What are relations?' will be the next question that we will answer. Before that, consider the following example. Given below are two different lists. One is a list of countries and their capitals. The other lists countries

and the local currencies used by them.

Country	Capital
Greece	Athens
Italy	Rome
India	New Delhi
China	Beijing
Japan	Tokyo
Australia	Sydney
France	Paris
New Zealand	Auckland
Spain	Madrid
Peru	Lima
Portugal	Lisbon

Country	Currency
Greece	Drachma
Italy	Lira
India	Rupee
China	Remnimbi(Quan)
Japan	Yen
Australia	Australian Dollar
France	Francs
New Zealand	Dollars
Spain	Peso
Peru	Nuevo sols
Portugal	Escudo

You will notice that their two different lists shown here. However, there is a column, which is the common to both lists. This the column, which contains names of the country. Now if someone wants to know the currency used in Rome, first one should find out the name of the country. Next that country should be looked up the next list to find out the currency.

It is possible to get this information because it is possible to establish relation between the two lists through a common column called "country".

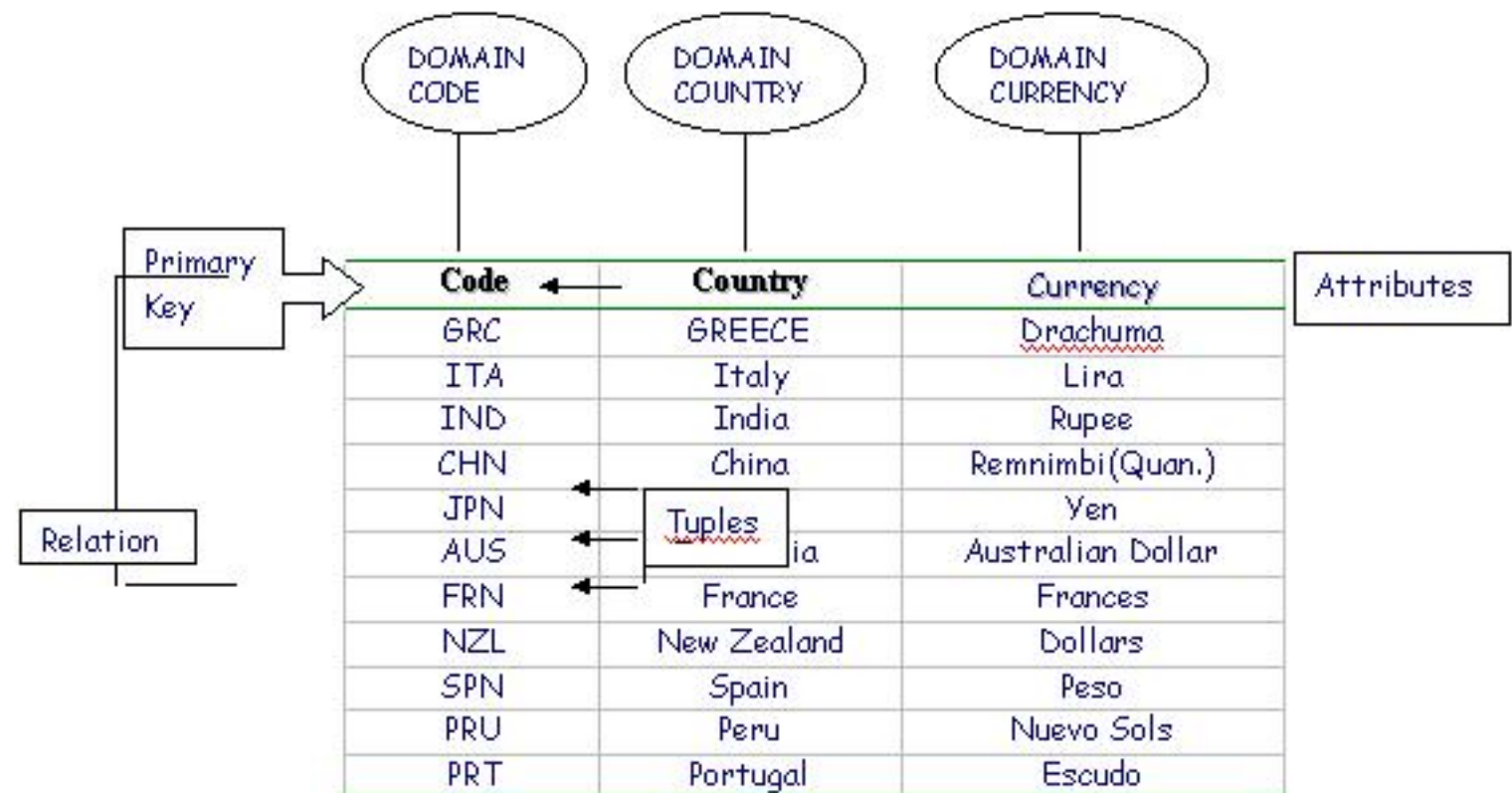
In the relational model data is stored in relations, relation is a formal term for the table. In the example above we have stored information

about countries as a table. A table in a database as a unique table name that identifies its contents. Each table can be called an intersection of row and columns. One of the most important properties of a table is that the rows are unordered. A row cannot be identified by its position in the table. Every table must have a column that uniquely identifies by each row in the table. It is essential that no two rows should contain identical information. This is prevented by the use of primary key.

Now we will exam the relational model in detail.

The relational model - the details

Let us consider any of the tables we have considered in the example above less us take the currency table.



(Fig. Shows Relational Model)

a new column called "codes" has been introduced as the primary key. The table show above consist of the components listed below,

according to a relational model:

Domain

Domain is pool of values from where one or more attributes (columns) can draw their actual values. For example, the values in the field "country" are available from the name of all the countries in the world. Hence, the domain name for this field is country.

tupple

according to the relational model, every relation or table is made up of many tuples. They are called records- a term that we are already familiar with. They are the rows that a table is made up of. Given below are some of the tuples that are part of the currency table.

CHN	China	Remnimbi (quan)
FRN	France	Francs
PRT	Portugal	Escudo

The number of tuples in a table is the cardinality of the tuple.

Attributes

The term "attributes" refers to characteristic. The characteristic of the tuple is reflected by its attributes or field. This simply means that what the column contains will be define by the attributes of that column. The number of attributes is called the degree of that table. Look at some of the attributes shown below.

Peso

Australia
New Zealand

Although relational models prescribe these above terms they do not appear in the daily usage. The terms "records" and "fields" are commonly used.

Advantages of a Relational Database Model

Some of the salient advantages of a relational database model have been listed below:

Built in integrity at various levels.

Allow data integrity to be incorporated at the field level to ensure data accuracy; integrity at the table level to avoid duplication of records and to detect records with missing primary key values;

At the relationship level to ensure that relationships between tables are valid.

Logical and Physical data Independence from database applications

Changes made in the logical design of the database or changes made in the database software will adversely affect the implementation of the database.

Data consistency and accuracy

Due to the various levels, at which data integrity can be built in, data is accurate and consistent.

Easy data retrieval and data sharing

Data can be easily extracted from one or more than one tables. Data can also be easily shared users.



Atmiya
— Infotech Pvt. Ltd.

TOP:3 What is Relational Database Management System ?

A relational database is a database structured on the relational model. A Relational Database Management System or RDBMS is a suite of software programs that can be used for creating, maintaining, modifying and manipulating a relational database. It can also be used to create the application that a user will require for interacting with the data stored within the database. A very important point to note here is that an RDBMS that satisfies the 12 criteria lay down by Dr. Codd is called a true RDBMS. (Refer to Appendix A for Dr. Codd's 12 Rules).

Kinds of Relations

Various kinds of relations (tables) can exist in a relational system. They are listed below:

Base Tables

Query results

Views

We are familiar with the concept of base tables. The other two types will be discussed in subsequent sessions.

Base Tables

A base table is a table with a name that physically exists in a database. It is created by the user, not something that is derived

from another table. A base table can be created, altered and removed from a database. All these tasks are accomplished using SQL statement, which will be covered in detail in subsequent sessions.

Query Results

When a 'question' is asked to a table, the resultant data is also stored in tables. Such tables are called query result tables.

Views

A view is a virtual table. Some columns of a base table may not be required by the user. In such cases, a view is created. This view will consist only of those columns of the base table that the user is interested in seeing. This format can be saved as a view by giving it a name. This concept will be covered in subsequent sessions in further detail.

Now that we have a better understanding of RDBMS concept, we progress and understand how this relational model is represented.

In a previous session, we understood what the relational model is. A database is meant to store data and this data in turn should provide information needs of an organization, a conceptual model has to be designed first.

The first step in this direction will be the collection and gathering of the data that will be required information. The data analysis process is where this will take place. Data analysis entails the collection of bills, reports, forms and other such records. The next step is to assess the uses of the organizations data and removing the data that is not required or is getting repeated. A data analysis also involves identifying tables, their fields and records and establishing relationships between them.

The completion of data analysis is marked by the drawing of the entity - relationship diagram or the ER diagram. In order to understand how

to draw the ER diagram, it is important to first appreciate the ER model. In this section, we will learn about the ER model and ER diagrams.

The ER data model is based on the object-based logical models. Chen introduced this model and its diagramming technique. Let's examine the components of this model here. Thus model comprises of the following components.

Entities

Attributes

Relationship

Entities

An entity can be defined as anything, which can be distinctly identified. A place, person, picture, thing, concept, process, result or data are some of the examples of an entity. As can be seen, the term concept a very broad spectrum.

Entity Set

An entity set of entities of some type. The set of person studying in a class can be defined as an entity set of student. Similarly it is possible to define to customers of a shop, patients of a clinic etc. as entity sets.

Attributes

Each entity has a specific characteristic that is defined by the attribute

Relationships

A relationship is defined as an association among entities. We shall understand this model with the help of an example.

Consider an organization having many departments in it. Each department has several employees are managed by a department head.

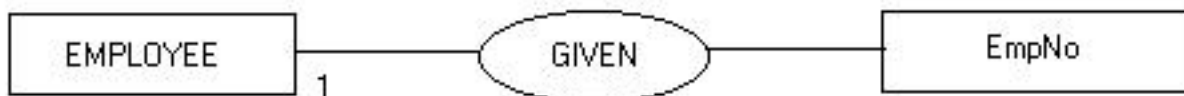
Each entity is related with another relationships. In example that we have considered above, each of the entities i.e. the department are all distinct entities. Relationships in a relational database model are categorized as:

One to One (1:1)

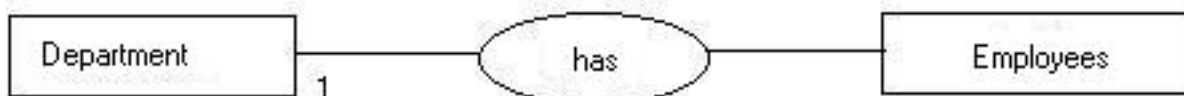
One to Many (or many to one) (1:M)

Many to Many (M:M)

Let us understand how these types of relationships apply in our example. Each employee given a unique employee number. Since any one employee can have only one employee number, this is called a one to one relationship; one department can have only one department head. This is also an example of a one to one relationship.



However, one department can have several employees working it. This is an example of a one to many relationships. Thus, it can be said that the relationship existing between employees and a department is of the 1:M type.



In most real life situations, is difficult to find a many to many relationship. In our example, here we do not have many to many

relationships between any of entities.

If user is a familiar with the relationship among the tables in the database, data can be accessed in a number of ways. Data can be accessed from tables. Which are directly as well as indirectly related.

In the next section, we will learn how to represent these relationships pictorially. The ER diagram is a way of expressing of representing this relationship. We will learn more about this in detail in the next section.



TOP:4 Difference between DBMS v/s RDBMS

DBMS	RDBMS
The concept of relationships is missing in a DBMS. If it exists it is very less	It is based on the concept of relationships.
Speed of operation is very slow.	Speed of operation is very fast.
Hardware and software requirements are less.	Hardware and software requirements are high.
Facilities and Utilities offered are limited.	Facilities and Utilities offered are many.

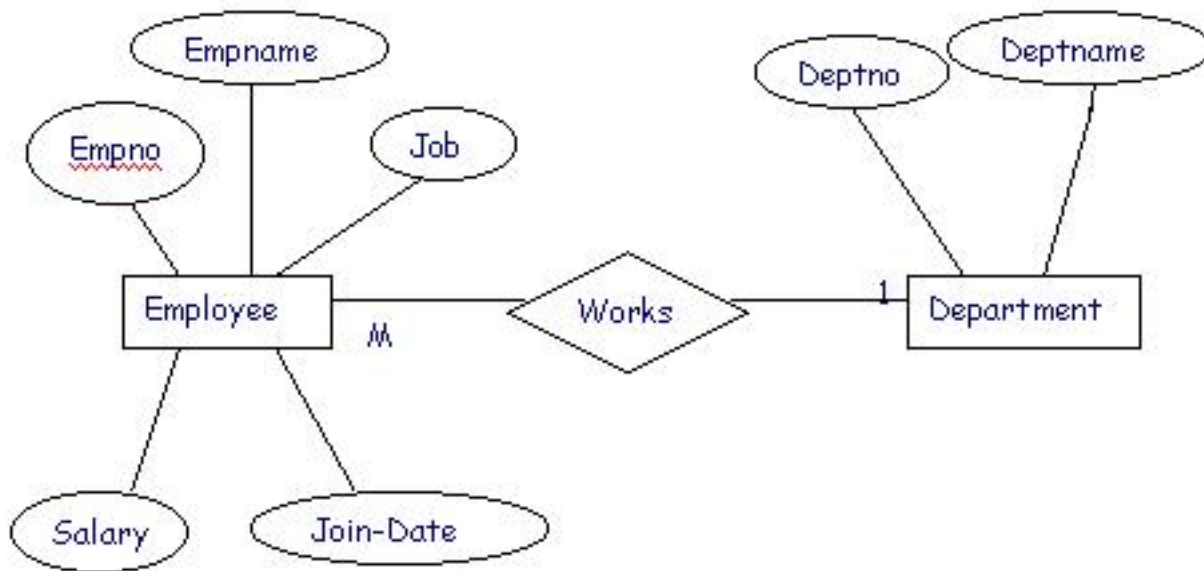
Platform is used is normally DOS	Platform used can by any DOS, UNIX, VAX,VMS etc.
Uses concept of a file.	Uses concept of a table.
DBMS normally use a 3GL.	RDBMS normally use a 4GL.
Examples are dBASE, FOXBASE etc.	Examples are ORACLE, INGERS etc.

TOP:5 Entity Relationship Diagram

Diagrams are one of the better ways to communicate different of a components of a system. They are also too easy to understand by everyone. They offer an overview of the entire system. An E-R diagram is graphical method of representing entity classes, attributes and relationships. An E-R diagram uses six basic symbols:

- A rectangle to denote an entity or entity set.
- A diamond to denote a relationship between two entities.
- An oval to denote attributes.
- A '1' to denote a single occurrence.
- An 'M' to denote multiple occurrences.
- A line which links attributes to an entity or entity set and entity sets to relationships.

When an E-R diagram is built. The first step is defining entities. The next step is to define the relationship between the entities. The final step to identify the attributes that belong to each entity. Once the E-R diagram is completed. The entities will become the files (or table). Figure illustrates a many-to-one relationship between the entity sets Employee and Department. The next process is that of normalization. Which will be covered in detail in the next session.



The importance of relationships

We will understand why relationships are a vital part of the database. The main reasons are listed and described below.

- Relationships establish a connection between a pair of tables that are logically related to each other in some manner. Data in a customer's table and orders tables are logically related. When a customer orders for an item, this order is recorded in the order table. Hence a customer record in the customer's table is related to the record in the order table.
- Relationships help refine and streamline table structures. This helps in further in removing data redundancy.
- Data for many tables can be extracted at the same time if relationship exists among the table.
- A well-defined relationship helps maintain a high level integrity. For example a customer record from the customer table cannot be deleted if a record for that customer exists in the order table.

We will learn as progress with our study of RDBMS, that establish

relationship carefully help in designing database that are easy to use.



TOP:6 Types of Relationship

One to One (1:1)

One to Many (1:M) or Many to One

Many to Many (M:M)

One to One

A pair of table is define as having one to one relationship if one record in the first table is related to only one record in the second table. Let's consider an example to understand this. In our library example, supports we have a books table with details of books as follows:

Book code

Book name

Book Author

Book Publisher

Book Cost etc.

The primary key is the book code. It is required to stored publisher details in separate tables because many books can have the same publisher. This leads to redundancy. Hence, a new table called publisher is created the structure is follows:

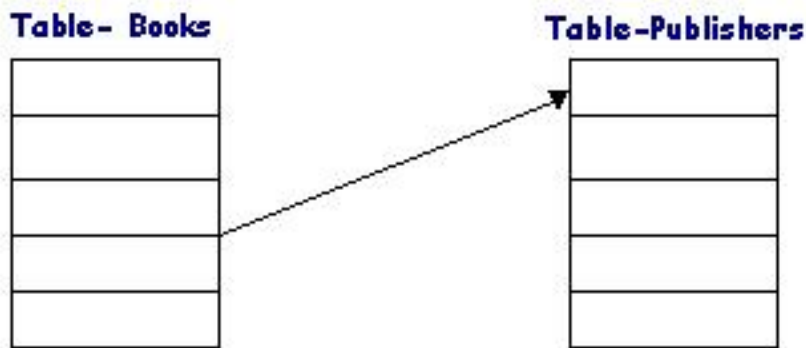
Book Code

Publisher Name

Publisher Add.

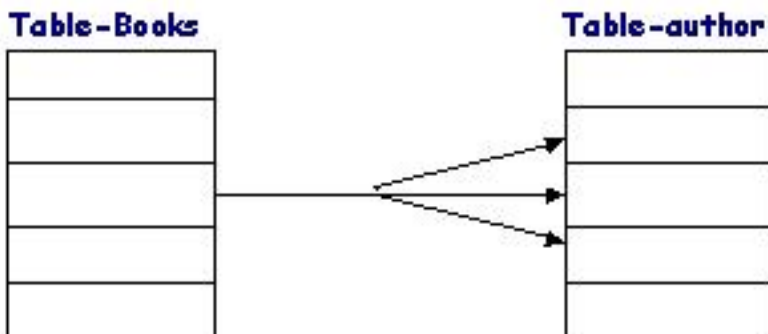
Publisher Tele. / Fax etc.

In this table, the book code is foreign key. Since any book can have only one publisher, this is an example of one to one relationship. Look at the pictorial representation of this relationship.



One to Many

A relationship is defined as one to many when a single record in the first table points to many records in the second table. However, a single record in the second table can only point to one record in the table. Let's consider our library example again. Like our publisher table suppose we create author table. One book title can be written by several authors take the example of the book title-oracle power object-developers guide. A publisher is Mc. Grow Hill and the authors are R. Finkelstern, R. Greenwald and kasu sista

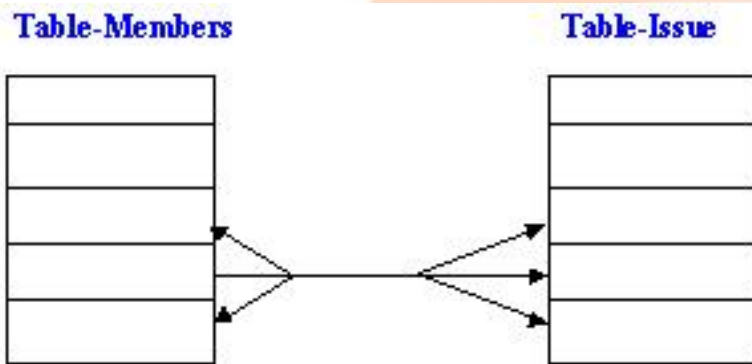


ech Pvt. Ltd.

In the fig. Above, notice that the three arrows from a picture that reassembles a foot of crow. A crowfoot is normally used to signify the one to many relationships.

Many to Many Relationship

A many to relationship exists between two tables if a single record in the first table points to more then one record in the second table and a single record in the second table points to many records in the first table. We will understand this with the help of an example. We have already seen the issue table and the member table in this case one member can be issued (can borrow) many books at one instance. At the same time, one book title can be borrowed by more than one member at any instance. Not the crow feet pointing at both tables



TOP:7 Normalization

It is one of the most important concepts in the study of the RDBMS. The case with which information is stored and retrieved. Depends a lot on the way of the tables have been defined. Tables that are huggled and bulky often defeat the purpose of having an RDBMS all together. This is because such tables may not be easy to maintain

In this section, we will discuss the concept of normalization in detail. It can be defined as a processed of putting data right-making it normal. Normalization is important from the database design. Designer point of view as it enables him to design better. It is concerned with database design. There are two ways of approaching logical database design-

The top down approach

The bottom-up approach

In the top down approach, first entities and relationship are identified; the ER diagram is made and mapped with the tables. The ER modeling technique uses the top down approach.

Normalization uses the bottom up approach. Normalization is the technique that makes the relational data files differ from other data files, which are referred to as flat files we will understand this with the help of an example.

Infotech Pvt. Ltd.

A library maintains a register of all books issued to its members. The register contain the following column

No.	Name of the book	Borrowed by	Date of issue	Date of return

Every time a member failed to return the book, a letter is sent/a telephone call is made to that member's house for reminding. In separate part of the register. A list of all members name along with their address and telephone numbers is maintain.

Member name	Address and telephone number.

Now every time a member defaults in returning a book, the librarian looks of the name of that member in the list and makes a reminder call. In this case we can say that the librarian has a relational database on paper. This is called a normalized data. Normalization is here referring to data being collect and stored in natural grouping. Issue detail of book and member details are stored as separate groups or lists.

Like this library register even in RDBMS, it is imperative to have normalized table. Normalization can be defined as the process of the restructuring a relation (table). For reducing it to a form where each domain would consist of single non composite values

Atmiya
Infotech Pvt. Ltd.

Benefits of normalization

Normalization reduces repetition for example data redundancy. When the same data is reputedly is stored it cases storage and access problems.

- Inconsistency in data retrieval
- error while updating data tables

Suppose the address and the telephone nos. of the members were Grouped together in one table along with book details. How would it make difference? First let take a look at the table below:

No	Name of the book	Borrowed By	Address	Telephone	Date of Issue	Date of Return

As you can see there are no groups everything is put in one table now consider a member who borrow above four books every weeks, it's now easy to see how many times the address and telephone no of this member will be stored. In the table leading to a huge amount of data

Suppose one member is changed his address. The no. of rows were the address will have to be changed will be many. This duplication of effort due to poor database design. as it can be seen data that has not been normalized can be lead to a several problems two of which we have discussed.

Having understood normalization and its benefits let's proceed what happens after a table is normalized.

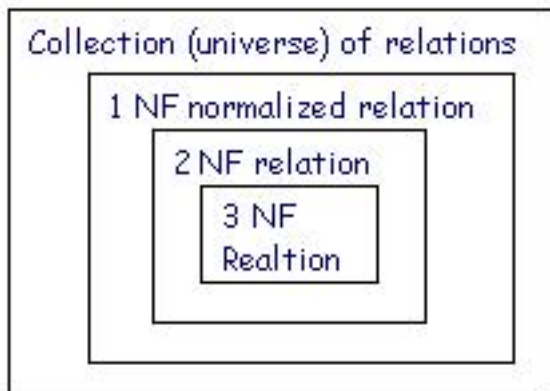
Normal Form

Dr. Codd originally defines three levels of normalization. These three

levels were called first normal form. Second normal form and third normal form respectively. Normalization is usually discussed in terms of forms. Normal forms are table structures with minimum redundancy. Normal forms that have been identified are:

- First normal form (1st NF)
- Second normal form (2nd NF)
- Third normal form (3rd NF)
- Boyce-Codd normal form

The first three forms were defined by Dr. Codd. Later Dr. Codd and Boyce introduced one more normal form, which called the Boyce-Codd normal form. Theory of normalization is based on the concept of function dependency. The diagram below illustrates the levels of normalization.



In order to understand more about normal form we must have understand is meant by functional dependency.

Establish deletion rules

Data integrity is one of the main advantages that relationships offer in an RDBMS. Deletion rules must be established for a relationship. This rule states what will happen if a record has to be deleted. Defining a deletion rule prevents records from being operand; i.e. the record will

exist in a subordinate table but will not exist in the main table. There can be two option applicable for deletion rule.

Restrict: when deletion is restrict, a record in a subordinate table of a one to one or many to one relationship can not be deleted for example a publisher whose books are still found in the books table can not be deleted. There are several examples where significance of such a rule can be appreciated further.

In an employ table, an employ cannot be deleted because a salary table still hold his record. In a pending order table, pending order cannot be deleted until the order has been serviced. Customer who has not paid these dues cannot be deleted from the customer table.

Cascade: in this type of rule when a record is deleted, all related records in all subordinate tables will also be deleted. In the first session we had talk about RDBMS maintaining data integrity. This is how it is possible.



TOP:8 Codd's Rules

E.F.TED CODD'S LAWS For a fully functional Relational Database Management System

Relational Database Management

A relational database management system uses only its relational capabilities to manage the information stored in its database.

Information Representation

All information stored in a relational database is represented only by data item values, which are stored in the tables that make up the database. Associations between data items are not logically represented in any other way, such as, by the use of pointers from one table to the other.

Logical Accessibility

Every data item value stored in a relational database is accessible by stating the name of the table it is stored in, the name of the column under which it is stored and the value of the primary key that defines the row in which it is stored.

Representation of null values

The database management system has a consistent method for representing null values. For example, null values for numeric data must be distinct from zero or any other numeric value and for character data it must be different from a string of blanks or any other character value.

Catalog facilities

The logical description of a relational database is represented in the same manner as ordinary data. This is done so that the facilities of the relational database management system itself can be used to maintain database description.

Data Language

A relational database management system may support many types of languages for describing data and accessing the database. However, there must be at least one language that uses ordinary character strings to support the definition of data, the definition of views, the manipulation of data, constraints on data integrity, information concerning authorization and the boundaries for recovery of units.

View Updatibility

Any view that can be defined combinations of base tables, which are theoretically updateable, is capable of being updated by the relational database management system.

Insert, Update and Delete

Any operand that describes the results of a single retrieval operation is capable of being applied to an insert, update or delete operation as well.

Physical Data independence

Changes made to physical storage representations or access methods do not require changes to be made to application programs.

Logical data independence

Changes made to tables, that do not modify any data stored in that

table, do not require changes to be made to application programs.

Integrity Constraints

Constraints that apply to entity integrity and referential integrity are specifiable by the data language implemented by the database management system and not by the statements coded into the applications program.

Database Distribution

The data language implemented by the relational database management system supports the ability to distribute the database without requiring changes to be made to application programs. This facility must be provided in the data language, Whether or not the database management system itself supports distributed databases.

Non-Subversion

If the relational database management system supports facilities that allow application programs to operate on the tables a row at a time, an application program using this type of database access is prevented from bypassing entity integrity or referential integrity constraints that are defined for the database.



TOP:1 Introduction to SQL (Structured Query Language)

Every data table that we have created so far us. All this data would data serve very little purpose if it could not be retrieved. In order to retrieve this data, one needs to be able to 'talk' to the tables. The structure query language allows users of the database communicate with the database.

Data in a relational database can be retrieved using a standard language like SQL. In an English like computer language, which makes interaction between user and the database very simple. Let us first look at its evolution.

SQL - A Brief History

SQL was first introduced by Dr. Codd in his pioneering work 'A relational Model for Large Shared Data Banks'. Since introduction many researchers at the San Jose Research Laboratories made efforts in implementing Dr. Codd's ideas, the mid of 70s saw the development of many computer programming language based on this relational model. One of these was called Structure English Query Language of SEQUEL language.

System T did well with relational database. One of such organization called relational software from Belmont, California made it software commercially available. It was a relational database called ORACLE. This company later changed ins name to oracle corporation because of the success of this software. Today oracle manufactures a wide range of SQL products along with Relational Database Management System software.

Since many software vendors started offering a large number of SQL

base products, there was a dire need to standardize this language. The first SQL standards were later adopted by the international standard organization (ISO) in 1987. The database language SQL was formed under the approval both ANSI and ISO.

Why SQL?

Let us understand that why there is a need to SQL. Although other programming languages exist. The primary reason for doing so is that SQL was created for the relational database. The relation model was considered when creating a SQL. SQL therefore is like a co-worker. Assisting the RDBMS achieve the user's requirement. SQL simplifies the task of creating, Manipulating and communicating the database. A traditional and general programming language like C would be difficult to use for this purpose.

Characteristics of SQL

SQL is Non Procedural Language

In the conventional programming language, coding is essential to achieve a given task. In addition to specifying the task to be achieved, needs to be specifying the tasks to be achieved. How to go about doing it has also to be specified. In SQL only the task that has to be achieved needs to be specified. For example, to retrieve rows from a table we simply use the SELECT command.

A Data Sub language

SQL does not support programming language constructs. Conditional statements like 'If Then', 'While' can not be used in SQL.

Infotech Pvt. Ltd.

Not a database management system

SQL is an important tool for communication with the DBMS and supports database management statements.

Can be embedded in third generation languages like 'C' or 'COBOL' to facilities database access.

Use of SQL

SQL is language used for communication with the DBMS. It is a language that can be used by all users such as

System Administrator

Database Administrator

Application Programmer

Management Personnel

End Users

it can used for the following:

It is an interactive query language that allows users to use SQL statements to retrieve the data and display on it screen. SQL is an important tool that allows adhoc queries on the database.

A database programming language that allows programmer to embed SQL statement in third generation language program to access data from the database.

A database Administration Language that define the structure of database, controls the user access to data and also the level of user access.

Client/Server Language that allows application programs on PCs connected via LAN to communicate with the database server using SQL. Application using client/server Language make optimum use of PCs and servers and also reduce traffic

over the LAN.

Database Gateway Language uses SQL to communicate with the distributed database.

Distributed Database language is used when data is distributed over many machine. The Distributed Database Management System uses SQL to communicated with the distributed database.

Types of SQL

SQL is two types-interactive and embedded. While both operate exactly the same way their usage differs. Interactive SQL is used to interactive directly with the database there the output of the operation is used of human consumption. Once a command is specific, it is execute and the user can immediately view the output.

In the case embedded SQL, commands are SQL commands that are written in some other languages, such as COBOL or Pascal. This makes to programs very fast and powerful. In this course, we will use SQL only in its interactive form.



TOP:2 SQL Command and DataTypes

SQL Commands can classified as under:

DDL (Data Definition Language)

CREATE	to create table or objects
ALTER	to alter existing database
DROP	to drop existing objects
TRUNCATE	to remove whole data at a time

DML (Data Manipulation Language)

INSERT	to insert data in table
UPDATE	to update existing data in table
SELECT	to view database
DELETE	to delete particular records in database



TCL (Transaction Control Language)

COMMIT	to save buffer data to storage device
---------------	---------------------------------------

ROLLBACK	to undo save
SAVEPOINT	to keep break in save action

DCL (Decision Control Language)

GRANT	to provide rights for user on database
REVOKE	to revoke user rights

DATATYPES

NUMERIC

The NUMBER datatype is used to store zero, negative, positive, fixed and floating point numbers with up to 38 digits of precision. numbers range between $1.0 * 10^{-130}$ and $1.0 * 10^{126}$.

NUMBER(p , s)

Where p is the precision up to 38 digits and s is the scale (number of digits to the right of the decimal point). The scale can range between -84 and 127.

NUMBER(p)

This is a fixed point number with a scale of zero and a precision of p.

NUMBER

This is a floating point number with a precession of 38.

The following list shows how Oracle stores different scales and precisions.:

Actual Data	Defined as	Stored as
123456.789	NUMBER(6,2)	123456.79
123456.789	NUMBER(6)	123457
123456.789	NUMBER(6,-2)	123400
123456.789	NUMBER	123456.789

DATE

Instead of storing date and time information in a character or numeric format. IBM created a separate datatype. for each DATE datatype, the following information is stored.

Century - Year - Month - Day - Hour - Minute - Second

You can easily retrieve the current date and time by using the function **SYSDATE**.

Date arithmetic is possible using number constants or other dates. Only addition and subtraction are supported. For example, **SYSDATE + 7** returns oneweek from today. Every database system has a default date format that is defined by the initialization parameter **NLS_DATE_FORMAT**. This parameter is usually set to **DD-MON-YY**, where **DD** is the day of the month (the first day of the month

is 01), MON is the abbreviated month name, and YY is a two-digit designation. If you do not specify a time, the default time is 12:00:00 a.m. if only the time component is captured, the default date is the first day of the current month.

CHARACTER

There are six character types available:

- The **CHAR** datatype is used where fixed-length fields are necessary. Any length up to 2,000 characters can be specified. The default length is 1. When Data is entered any space left over is filled with blanks. All alphanumeric characters are allowed.
- The **VARCHAR2** is used for variable-length fields. A length component must be supplied when you use this datatype. The maximum length is 4000 characters. All alphanumeric characters are allowed.
- The **LONG** datatype is used to store large amounts of variable-length. Any length up to 2GB can be specified. Be aware that there are some restrictions to using this datatype:

Only one column per table can be defined as LONG.

A LONG column cannot be indexed.

A LONG column cannot be passed as an argument to a procedure.

You cannot use a function to return a LONG column.

You cannot use a LONG column in WHERE, ORDER BY, GROUP BY or DISTINCT by clauses.

- The **VARCHAR** datatype is synonymous with VARCHAR2. Oracle Corporation is reserving this for future use. Do not use this datatype.

BINARY

Two datatypes, **RAW** and **LONG RAW**, are available for storing binary type data such as digitized sound and images. These datatypes take on characteristics similar to the VARCHAR2 and LONG datatypes already mentioned.

Use the RAW datatype to store binary data up to 2,000 bytes and use the LONG RAW datatype to store binary data up to 2GB. Oracle stores and retrieves only binary data; no string manipulations are allowed. Data is retrieved as hexadecimal character values.



TOP:3 Operators and Expressions

EXPRESSION

The definition of an expression is simple: An expression returns a value. Expression types are very broad, covering different data types such as String, Numeric, and Boolean. In fact, pretty much anything following a clause (SELECT or FROM, for example) is an expression. In the following example amount is an expression that returns the value contained in the amount column.

```
SELECT amount FROM checks;
```

In the following statement NAME, ADDRESS, PHONE and ADDRESSBOOK are expressions:

```
SELECT NAME, ADDRESS, PHONE FROM ADDRESSBOOK;
```

Now, examine the following expression:

```
WHERE NAME = 'BROWN'
```

It contains a condition, NAME = 'BROWN', which is an example of a Boolean expression. NAME = 'BROWN' will be either TRUE or FALSE, depending on the condition =.

CONDITIONS

If you ever want to find a particular item or group of items in your

database, you need one or more conditions. Conditions are contained in the WHERE clause. In the preceding example, the condition is

NAME = 'BROWN'

To find everyone in your organization who worked more than 100 hours last month, your condition would be

NUMBEROFHOURS > 100

Conditions enable you to make selective queries. In their most common form, conditions comprise a variable, a constant, and a comparison operator. In the first example the variable is NAME, the constant is 'BROWN', and the comparison operator is =. In the second example the variable is NUMBEROFHOURS, the constant is 100, and the comparison operator is >. You need to know about two more elements before you can write conditional queries: the WHERE clause and operators.

The WHERE Clause

The syntax of the WHERE clause is

SYNTAX:

WHERE <SEARCH CONDITION>

SELECT, FROM, and WHERE are the three most frequently used clauses in SQL. WHERE simply causes your queries to be more selective. Without the WHERE clause, the most useful thing you could do with a query is display all records in the selected table(s). For example:

INPUT:

SQL> SELECT * FROM BIKES;

lists all rows of data in the table BIKES.

OUTPUT:

<u>NAME</u>	<u>FRAMESIZE</u>	<u>COMPOSITION</u>	<u>MILES</u>	<u>RIDDER</u>	<u>TYPE</u>
TREK 2300	22.5	CARBON FIBER	3500		RACING
BURLEY	22	STEEL	2000		TANDEM
GIANT	19	STEEL	1500		COMMUTER
FUJI	20	STEEL	500		TOURING
SPECIALIZED 16		STEEL	100		MOUNTAIN
CANNONDALE	22.5	ALUMINUM	3000		RACING

6 rows selected.

If you wanted a particular bike, you could type

INPUT/OUTPUT:

```
SQL> SELECT * FROM BIKES WHERE NAME = 'BURLEY';
```

which would yield only one record:

<u>NAME</u>	<u>FRAMESIZE</u>	<u>COMPOSITION</u>	<u>MILES</u>	<u>RIDDER</u>	<u>TYPE</u>
BURLEY	22	STEEL	2000		TANDEM

ANALYSIS:

This simple example shows how you can place a condition on the data that you want to retrieve.

OPERATORS

Arithmetic	Logical	Like	Relational	Miscellaneous
+	AND	LIKE	<	Is
-	OR		>	In
*	NOT		=	Any
/			< =	All
%			> =	The
			!=	
			< >	

The precedence of the operators are:

- The following have equal precedence

=, !=, >, <, >=, <=

BETWEEN&ldots;&ldots;AND , IN

LIKE, IS NULL

- NOT

- AND

OPERATORS in Details:

Operators are the elements you use inside an expression to articulate how you want specified conditions to retrieve data. Operators fall into six groups: arithmetic, comparison, character, logical, set, and miscellaneous.

Arithmetic Operators

The arithmetic operators are plus (+), minus (-), divide (/), multiply (*), and modulo (%). Modulo returns the integer remainder of a division. Here are two examples:

$$5 \% 2 = 1$$

$$6 \% 2 = 0$$

If you place several of these arithmetic operators in an expression without any parentheses, the operators are resolved in this order: multiplication, division, modulo, addition, and subtraction. The following sections examine the arithmetic operators in some detail and give you a chance to write some queries.

Plus (+)

You can use the plus sign in several ways. Type the following statement to display the PRICE table:

INPUT:

```
SQL> SELECT * FROM PRICE;
```

OUTPUT:

ITEM	WHOLESALE
TOMATOES	0.34
POTATOES	0.51
BANANAS	0.67

3 rows selected.

Now type:

INPUT/OUTPUT:

SQL> SELECT ITEM, WHOLESALE, WHOLESALE + 0.15 FROM PRICE;

Here the + adds 15 percents to each price to produce the following:

ITEM	WHOLESALES	WHOLESALE + 0.15
TOMATOES	0.34	0.49
POTATOES	0.51	0.66
BANANAS	0.67	0.82

3 rows selected.

ANALYSIS:

What is this last column with the unattractive column heading WHOLESALE+0.15 ? It's not in the original table. (Remember, you used * in the SELECT clause, which causes all the columns to be shown.) SQL allows you to create a virtual or derived column by

combining or modifying existing columns.

Retype the original entry:

INPUT/OUTPUT:

```
SQL> SELECT * FROM PRICE;
```

The following table results:

ITEM	WHOLESALE
TOMATOES	0.34
POTATOES	0.51
BANANAS	0.67

3 rows selected.

ANALYSIS:

The output confirms that the original data has not been changed and that the column heading `WHOLESALE+0.15` is not a permanent part of it. In fact, the column heading is so unattractive that you should do something about it.

Atmiya
Infotech Pvt. Ltd.

NOTE: It is simple to use all the other arithmetic operators like Plus (+).

Comparison Operators

Comparison operators compare expressions and return one of three values: TRUE, FALSE, or Unknown. Wait a minute! Unknown? TRUE and FALSE are self-explanatory, but what is Unknown? To understand how you could get an Unknown, you need to know a little about the concept of NULL. In database terms NULL is the absence of data in a field. It does not mean a column has a zero or a blank in it. A zero or a blank is a value. NULL means nothing is in that field. If you make a comparison like `Field = 9` and the only value for Field is NULL, the comparison will come back Unknown. Because Unknown is an uncomfortable condition, most flavors of SQL change Unknown to FALSE and provide a special operator, `IS NULL`, to test for a NULL condition.

Here's an example of NULL: Suppose an entry in the PRICE table does not contain a value for WHOLESALE. The results of a query might look like this:

INPUT:

```
SQL> SELECT * FROM PRICE;
```

OUTPUT:

ITEM	WHOLESALE
TOMATOES	0.34
POTATOES	0.51
ORANGES	

Notice that nothing is printed out in the WHOLESALE field position for oranges. The value for the field WHOLESALE for oranges is NULL. The NULL is noticeable in this case because it is in a numeric column. However, if the NULL appeared in the ITEM column, it would be impossible to tell the difference between NULL and a blank.

Try to find the NULL:

INPUT/OUTPUT:

```
SQL> SELECT * FROM PRICE WHERE WHOLESALE IS NULL;
```

ITEM	WHOLESALE
ORANGES	

ANALYSIS:

As you can see by the output, ORANGES is the only item whose value for WHOLESALE is NULL or does not contain a value. What if you use the equal sign (=) instead?

INPUT/OUTPUT:

```
SQL> SELECT * FROM PRICE WHERE WHOLESALE = NULL;
```

no rows selected

ANALYSIS:

You didn't find anything because the comparison `WHOLESALE = NULL` returned a FALSE--the result was unknown. It would be more appropriate to use an `IS NULL` instead of `=`, changing the `WHERE` statement to `WHERE WHOLESALE IS NULL`. In this case you would get all the rows where a NULL existed.

This example also illustrates both the use of the most common comparison operator, the equal sign (`=`), and the playground of all comparison operators, the `WHERE` clause. You already know about the `WHERE` clause, so here's a brief look at the equal sign.



Atmiya
— Infotech Pvt. Ltd.

TOP:4 SELECT Statement

The general syntax for a SELECT statement:

SYNTAX:

```
SELECT [DISTINCT | ALL] { * | { [schema.]{ table | view |
snapshot}.* | expr } [ [AS] c_alias ] [, { [schema.]{ table | view |
snapshot}.* | expr } [ [AS] c_alias ] ] ... }
```

```
FROM [schema.]{ table | view | snapshot} [@dblink] [t_alias] [,
[schema.]{ table | view | snapshot} [@dblink] [t_alias] ] ...
```

```
[WHERE condition ]
```

```
[GROUP BY expr [, expr] ... [HAVING condition] ]
```

```
[{ UNION | UNION ALL | INTERSECT | MINUS} SELECT command ]
```

```
[ORDER BY { expr|position} [ASC | DESC] [, { expr|position} [ASC |
DESC]] ...]
```

Query:1

The simple first query to perform, which yields all records available in table

INPUT:

```
SQL> select * from dept;
```

OUTPUT:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ANALYSIS:

This output looks just like the code in the example. Notice that columns 1 in the output statement are right-justified and that columns 2 and 3 are left-justified. This format follows the alignment convention in which numeric data types are right-justified and character data types are left-justified.

The asterisk (*) in select * tells the database to return all the columns associated with the given table described in the FROM clause. The database determines the order in which to return the columns.

Terminating an SQL Statement

In implementations of SQL, the semicolon at the end of the statement tells the interpreter that you are finished writing the query.

Query:2 Changing the Order of the Columns or selecting

particular columns

The preceding example of an SQL statement used the * to select all columns from a table, the order of their appearance in the output being determined by the database. To specify the order of the columns, you could type something like:

INPUT:

```
SQL> SELECT dname, deptno from dept;
```

Notice that each column name is listed in the SELECT clause. The order in which the columns are listed is the order in which they will appear in the output. Notice both the commas that separate the column names and the space between the final column name and the subsequent clause (in this case FROM). The output would look like this:

OUTPUT:

DNAME	DEPTNO
ACCOUNTING	10
RESEARCH	20
SALES	30
OPERATIONS	40

Another way to write the same statement follows.

INPUT:

```
SQL> SELECT dname, DEPTNO
```

```
FROM dept;
```

Notice that the FROM clause has been carried over to the second line. This convention is a matter of personal taste when writing SQL code. The output would look same as above. Now you have the columns you want to see. Notice the use of upper- and lowercase in the query. It did not affect the result.

Query:3 Queries with Distinction

If you look at the original table, EMP, you see that some of the data repeats. For example, if you looked at the JOB column using

INPUT:

```
SQL> select JOB from emp;
```

you would see

OUTPUT:

JOB

CLERK

SALESMAN

SALESMAN

MANAGER

SALESMAN



MANAGER

MANAGER

ANALYST

PRESIDENT

SALESMAN

CLERK

CLERK

ANALYST

CLERK

14 rows selected.

Notice that the job 'salesman' is repeated. What if you wanted to see how many different jobs were in this column? Try this:

INPUT:

SQL> select DISTINCT job from emp;

The result would be

OUTPUT:

JOB

CLERK

SALESMAN

MANAGER

ANALYST

PRESIDENT

5 rows selected.

ANALYSIS:

Notice that only five rows are selected. Because you specified **DISTINCT**, only one instance of the duplicated data is shown. **ALL** is a keyword that is implied in the basic **SELECT** statement. You almost never see **ALL** because **SELECT <Table>** and **SELECT ALL <Table>** have the same result.

Query:4 WHERE Clause

With **WHERE** in your vocabulary, you can be more selective. To find all the employee having salary more than 2500

INPUT:

SQL> SELECT ename, job, sal FROM emp WHERE SAL > 2500;

The **WHERE** clause returns the five instances in the table that meet the required condition:

OUTPUT:

ENAME	JOB	SAL
-----	-----	-----
JONES	MANAGER	2975
BLAKE	MANAGER	2850
SCOTT	ANALYST	3000
KING	PRESIDENT	5000
FORD	ANALYST	3000

Query:5 The LIKE Clause

LIKE is an addition to the WHERE clause that works as a helping hand to WHERE. Compare the results of the following query:

INPUT:

SQL> SELECT empno, ename FROM emp WHERE ename LIKE 'S%';

OUTPUT:

EMPNO	ENAME
-----	-----
7369	SMITH

7788

SCOTT

Query:6 The ORDER BY Clause

From time to time you will want to present the results of your query in some kind of order. As you know, however, SELECT FROM gives you a listing, and unless you have defined a primary key, your query comes out in the order the rows were entered. Consider a beefed-up DEPT table:

INPUT:

```
SQL> select * from dept order by dname;
```

OUTPUT:

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK
40	OPERATIONS	BOSTON
20	RESEARCH	DALLAS
30	SALES	CHICAGO

INPUT:

```
SQL> select * from dept order by deptno;
```

OUTPUT:

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ANALYSIS:

The ORDER BY clause gives you a way of ordering your results. For example, to order the preceding listing by check DNAME in alphabetic order, you would use ORDER BY clause:

Query:6 The GROUP BY Clause

You have/had learned how to use aggregate functions (COUNT, SUM, AVG, MIN, and MAX). If you wanted to find the total amount of money spent on salary for employee, you would type:

INPUT:

```
SQL> SELECT SUM(sal) from emp;
```

OUTPUT:

```
SUM(SAL)
```

29025

You might get different result for this, it depends on salary column value in your database. Now if you want to calculate sum of salary department wise then you have to perform this way,

INPUT:

SQL> SELECT deptno, sum(sal) from emp group by deptno;

OUTPUT:

DEPTNO	SUM(SAL)
10	8750
20	10875
30	9400

NOTE: One simple rule, if you wish any table column along with aggregate function then you must use to write that column with group by clause else you'll get error like this;

INPUT:

SQL> SELECT deptno, job, avg(sal) from emp group by deptno;

OUTPUT:

SELECT deptno, job, avg(sal) from emp group by deptno

*

ERROR at line 1:

ORA-00979: not a GROUP BY expression

ANALYSIS:

Here job is not included in group by expression, now let's correct this,

INPUT:

SQL> SELECT deptno, job, avg(sal) from emp group by deptno, JOB;

OUTPUT:

DEPTNO	JOB	AVG(SAL)
-----	-----	-----
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	3000
20	CLERK	950
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850

30	SALESMAN	1400
----	----------	------

9 rows selected.

ANALYSIS:

This provides group value for each deptno and available job for that department. AVG function yields result for each department and each job.

Query:7 The HAVING Clause

How can you qualify the data used in your GROUP BY clause? Use the table EMP and above example and try this: The following statement qualifies this query to return only those departments with average salaries more than 1300:

INPUT:

SQL> SELECT deptno, job, avg(sal) from emp group by deptno, JOB
HAVING AVG(sal) > 1300;

OUTPUT:

DEPTNO	JOB	AVG(SAL)
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	3000

20	MANAGER	2975
30	MANAGER	2850
30	SALESMAN	1400

6 rows selected.

ANALYSIS:

On sort, to provide condition on aggregate function or group by function one must require HAVING clause.

FINAL EXAMPLE:

INPUT:

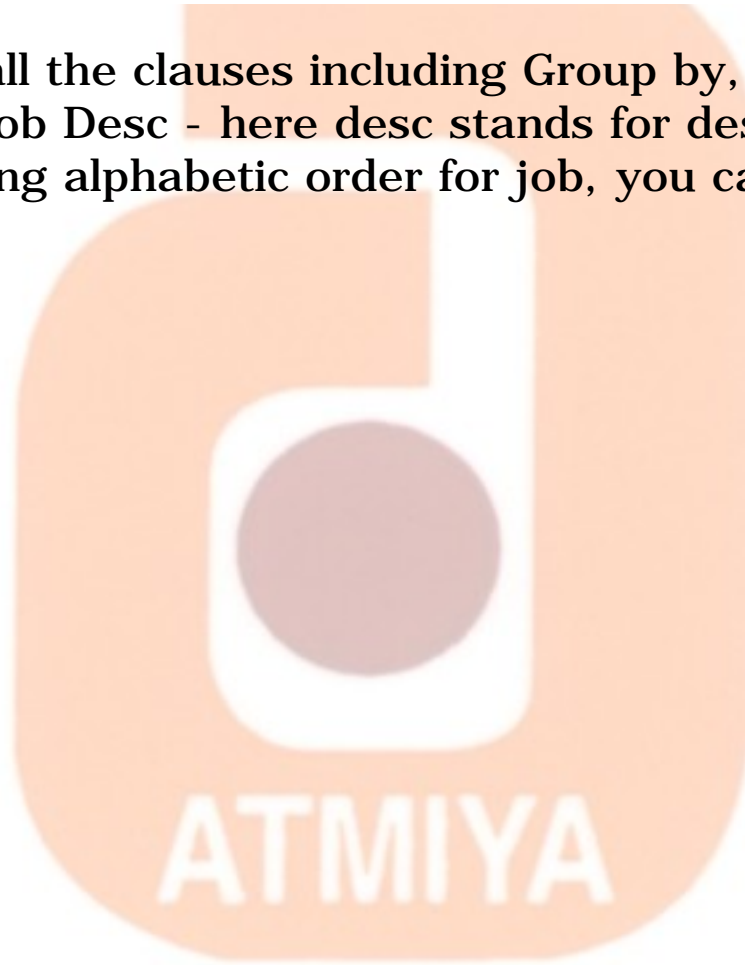
SQL> SELECT deptno, job, avg(sal) from emp group by deptno, JOB
HAVING AVG(sal)<6000 and deptno= 10 order by job desc;

OUTPUT:

DEPTNO	JOB	AVG(SAL)
10	PRESIDENT	5000
10	MANAGER	2450
10	CLERK	1300

ANALYSIS:

This query shows all the clauses including Group by, Order by and Having. Order By job Desc - here desc stands for descending. it pays results in descending alphabetic order for job, you can mark in above table.



TOP:5 Special Operators

Operator:1 The LIKE Clause

LIKE is an addition to the WHERE clause that works as a helping hand to WHERE. Compare the results of the following query:

INPUT:

```
SQL> SELECT empno, ename FROM emp WHERE ename LIKE 'S%';
```

OUTPUT:

EMPNO	ENAME
-----	-----
7369	SMITH
7788	SCOTT

INPUT:

```
SQL> SELECT empno, ename FROM emp WHERE ename LIKE '%N';
```

OUTPUT:

EMPNO	ENAME
-------	-------

-----	-----
7499	ALLEN
7654	MARTIN

INPUT:

SQL> SELECT empno, ename FROM emp WHERE ename LIKE '%A%';

OUTPUT:

EMPNO	ENAME
-----	-----
7499	ALLEN
7521	WARD
7654	MARTIN
7698	BLAKE
7782	CLARK
7869	ADAMS
7900	JAMES

Underscore (_)

The underscore is the single-character wildcard. Using a modified version of the table EMP, type this:

INPUT:

SQL> SELECT empno, ename FROM emp WHERE ename LIKE '_LA%';

OUTPUT:

EMPNO	ENAME
-----	-----
7698	BLAKE
7782	CLARK

ANALYSIS:

Result shows the ename like any one character at the place of underscore (_) then two words as 'LA' and rest any (%) character. You can use several underscores in a statement.

Concatenation (||)

The || (double pipe) symbol concatenates two strings. Try this:

INPUT:

SQL> SELECT empno || ename FROM emp WHERE job like 'MANAGER';

OUTPUT:

EMPNO ENAME



7566JONES

7698BLAKE

7782CLARK

ANALYSIS:

Notice that || is used instead of +. If you use + to try to concatenate the strings, the SQL interpreter used for this example (Personal Oracle8) returns the following error:

INPUT/OUTPUT:

```
SQL> SELECT EMPNO + ENAME FROM emp;
```

ERROR:

ORA-01722: invalid number

It is looking for two numbers to add and throws the error invalid number when it doesn't find any. NOTE: Some implementations of SQL use the plus sign to concatenate strings. Check your implementation.

Here's a more practical example using concatenation:

INPUT/OUTPUT:

```
SQL> SELECT deptno || ' - ' || dname NAME FROM dept;
```

NAME

10 - ACCOUNTING

20 - RESEARCH

30 - SALES

40 - OPERATIONS

ANALYSIS:

This statement inserted a des (-) between the department no and the name. Notice the extra spaces between the first name and the last name in these examples. These spaces are actually part of the data. With certain data types, spaces are right-padded to values less than the total length allocated for a field.

Operator:2 Logical Operators

Logical operators separate two or more conditions in the WHERE clause of an SQL statement.

INPUT/OUTPUT:

SQL> SELECT empno, ename, deptno, sal FROM emp WHERE deptno = 10 **AND** sal > 2000;

EMPNO	ENAME	DEPTNO	SAL
-----	-----	-----	-
7782	CLARK	10	2450

7839

KING

10

5000

ANALYSIS:

This query performs data search in a way that department no must be 10 as well salary must be greater than 2000. This means this query use logical operator and relational operator also.

AND

AND means that the expressions on both sides must be true to return TRUE. If either expression is false, AND returns FALSE. For example, to find out which employees have salary greater than 2000 and have registration in department 10.

OR

You can also use OR to sum up a series of conditions. If any of the comparisons is true, OR returns TRUE. To illustrate the difference, conditions run the last query with OR instead of with AND:

INPUT/OUTPUT:

SQL> SELECT empno, ename, deptno, sal FROM emp WHERE deptno = 10 **OR** sal > 2000;

EMPNO	ENAME	DEPTNO	SAL
7566	JONES	20	2975
7698	BLAKE	30	2850

7782	CLARK	10	2450
7788	SCOTT	20	3000
7839	KING	10	5000
7902	FORD	20	3000
7934	MILLER	10	1300

ANALYSIS:

The original names are still in the list, but you have three new entries. These five new names made the list because they satisfied one of the conditions. OR requires that only one of the conditions be true in order for data to be returned.

NOT

NOT means just that. If the condition it applies to evaluates to TRUE, NOT make it FALSE. If the condition after the NOT is FALSE, it becomes TRUE. For example, the following SELECT returns the only two names not beginning with S in the table:

INPUT:

```
SQL> SELECT ename, job FROM emp WHERE job NOT LIKE 'S%';
```

ENAME	JOB
-----	-----
SMITH	CLERK
JONES	MANAGER

BLAKE	MANAGER
CLARK	MANAGER
SCOTT	ANALYST
KING	PRESIDENT
ADAMS	CLERK
JAMES	CLERK
FORD	ANALYST
MILLER	CLERK

10 rows selected.

NOT can also be used with the operator IS when applied to NULL.

Operator:3 Set Operators

The following sections examine set operators.

UNION and UNION ALL

Infotech Pvt. Ltd.

UNION returns the results of two queries minus the duplicate rows. The following two tables represent the rosters of teams:

INPUT:

```
SQL> SELECT ename, deptno FROM emp UNION SELECT dname,  
deptno FROM dept;
```

OUTPUT:

ENAME	DEPTNO
--------------	---------------

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

ACCOUNTING	10
------------	----

ADAMS	20
-------	----

ALLEN	30
-------	----

BLAKE	30
-------	----

CLARK	10
-------	----

FORM	20
------	----

JAMES	30
-------	----

JONES	20
-------	----

KING	10
------	----

MARTIN	30
--------	----

MILLER	10
--------	----

OPERATIONS	40
------------	----

RESEARCH	20
----------	----

SALES	30
-------	----

SCOTT	20
-------	----

SMITH	20
TURNER	30
WARD	30

18 rows selected.

ANALYSIS:

The combined list--courtesy of the UNION ALL statement--has 18 names. UNION ALL works just like UNION except it does not eliminate duplicates.



INTERSECT

INTERSECT returns only the rows found by both queries. The next SELECT statement shows the list of deptno who are available on both tables:

INPUT:

```
SQL> SELECT deptno FROM emp INTERSECT SELECT deptno FROM dept;
```

OUTPUT:

DEPTNO

10

20

30

ANALYSIS:

In this example INTERSECT returns only those record which are available in both the tables. Graphical representation is shown below.



MINUS (Difference)

Minus returns the rows from the first query that were not present in the second. For example:

INPUT:

SQL> SELECT deptno FROM dept **MINUS** SELECT deptno FROM emp;

OUTPUT:

DEPTNO

40

The preceding examples shows department no available in DEPT table but not available in table EMP. Graphical representation is shown below.



Operators:4 IN and BETWEEN

The two operators IN and BETWEEN provide a shorthand for functions you already know how to do. If you wanted to find employee having job any from ANALYST, MANAGER or PRESIDENT then rather using two or more OR statement, one should use IN clause.

INPUT:

```
SQL> SELECT ename, job FROM emp WHERE job = 'ANALYST' OR job  
= 'MANAGER' OR job = 'PRESIDENT' ;
```

or

```
SQL> SELECT ename, job FROM emp WHERE job IN ('ANALYST',  
'MANAGER', 'PRESIDENT');
```

OUTPUT:

ENAME	JOB
-----	-----
JONES	MANAGER
BLAKE	MANAGER
CLARK	MANAGER
SCOTT	ANALYST
KING	PRESIDENT
FORD	ANALYST

6 rows selected.

ANALYSIS:

The second example is shorter and more readable than the first. You never know when you might have to go back and work on something you wrote months ago.

INPUT:

```
SQL> SELECT ename, sal FROM emp WHERE sal >= 2000 AND sal
<= 3000;
```

or

```
SQL> SELECT ename, sal FROM emp WHERE sal BETWEEN 2000
```

AND 3000;

OUTPUT:

ENAME

SAL

JONES

2975

BLAKE

2850

CLARK

2450

SCOTT

3000

FORD

3000



TOP:6 Join, Subquery

Join

Today you will learn about joins. This information will enable you to gather and manipulate data across several tables. You will understand and be able to do the following:

- **Perform an equi-join**
- **Join a table to itself (self join)**
- **Perform an outer join**
- **Perform a Between Join**

Introduction

One of the most powerful features of SQL is its capability to gather and manipulate data from across several tables. Without this feature you would have to store all the data elements necessary for each application in one table. Without common tables you would need to store the same data in several tables. Imagine having to redesign, rebuild, and repopulate your tables and databases every time your user needed a query with a new piece of information. The JOIN statement of SQL enables you to design smaller, more specific tables that are easier to maintain than larger tables.

A Few Join Considerations

1. You may join up to 15 tables in one SELECT, But this limit would have to be considered a bit impractical. Four tables is more sensible - especially if you are using medium to large sized tables (10000 rows or more).
2. If there are identical column names referred to in the join, those column names must be predefined by a table name (or alias) to ensure uniqueness.
3. A join will retrieve all possible combinations of rows that satisfy the join condition. The datatypes for the columns in the join condition should be compatible (either both numeric or both character). Note that NULL values never satisfy a join condition. Even if both rows from both tables match with NULL values in them, the rows will not be selected.
4. The condition after the WHERE creates the vital link between the tables, that is necessary to restrict the selection to useful rows.
5. The WHERE clause may specify multiple selection criteria.
6. You must have authority to access the tables that you are going to join.

Multiple Tables in a Single SELECT Statement (Equi-join)

INPUT:

```
SQL> SELECT emp.empno, emp.ename, emp.deptno, dept.dname  
FROM emp, dept WHERE emp.deptno=dept.deptno;
```

OUTPUT:

EMPNO	ENAME	DEPTNO	DNAME
-----	-----	-----	-----
7369	SMITH	20	RESEARCH
7499	ALLEN	30	SALES
7521	WARD	30	SALES
7566	JONES	20	RESEARCH
7654	MARTIN	30	SALES
7698	BLAKE	30	SALES
7782	CLARK	10	ACCOUNTING
7788	SCOTT	10	ACCOUNTING
7839	KING	10	ACCOUNTING
7844	TURNER	30	SALES
7876	ADAMS	20	RESEARCH
7900	JAMES	30	SALES
7902	FORD	20	RESEARCH
7934	MILLER	10	ACCOUNTING

ANALYSIS:

Check the query, in that table name is used to refer column before all the column. Now say dept table have 4 rows and emp table have 14

rows. If you omit WHERE clause in query then you will get $14 * 4 = 56$ records as a result. So it is compulsory to use WHERE clause in join query. This way equi join example. In place of table name as a reference of column you can generate table alias, so you need not to write long name of tables. To generate alias of table one need to write alias name in FROM clause. Example of the same is given here.

```
SQL> SELECT e.empno, e.ename, e.deptno, d.dname FROM emp e,
dept d WHERE e.deptno=d.deptno;
```

Self Join (Join to it self)

A self join is the most powerful illustrator of the complex queries that can be written using the simple SELECT statement. A self join is a join of a table with itself. This query is executed by logically making two copies of the same table. To do this the same table has to be given two aliases, which can then be compared to one another. Suppose you need to find employee name along with name of manager instead of MGR (manager code), one must use self join query as :

```
SQL> SELECT e1.empno, e1.ename, e1.mgr, e2.ename FROM emp
e1, emp e2 WHERE e1.mgr=e2.empno;
```

EMP NO	ENAME	MGR	ENAME
-----	-----	-----	-----
7369	SMITH	7902	FORD
7499	ALLEN	7698	BLAKE
7521	WARD	7698	BLAKE
7566	JONES	7839	KING

7654	MARTIN	7698	BLAKE
7698	BLAKE	7839	KING
7782	CLARK	7839	KING
7788	SCOTT	7566	JONES
7844	TURNER	7698	BLAKE
7876	ADAMS	7788	SCOTT
7900	JAMES	7698	BLAKE
7902	FORD	7566	JONES
7934	MILLER	7782	CLARK

13 rows selected.

ANALYSIS:

Here, one can mark that KING is not available as employee but available in list of managers. That is because of KING does not have MGR value, and what we perform is equality concept as $e1.mgr = e2.empno$.



Infotech Pvt. Ltd.

Outer Join

An outer join is a join similar to a simple join. A simple join is a join of two based on a common column. A special kind of simple join is the equi join. We have already discussed this type of join. An outer join is different from all other types of joins as it returns:

All the rows that are returned by a simple join and

All those rows of one table that do not match with the rows in the other table.

An outer join is used to join unmatched rows of a table. A table can be outer joined to at the most one table.

```
SQL> SELECT c.client_no FROM client_master c, sales_order s WHERE
s.client_no(+)=c.client_no;
```

CLIENT

C00001

C00001

C00002

C00003

C00004

C00005

C00006



As you can see there is a small '+' sign after the sales_order table. This sign is used to indicate an outer join. It can be put on either side of WHERE clause. It is however, append to that table which does not have matching rows. Here all customer may not available in sales_order table, but you are getting that record also because of outer join applies on client_master table.

Between Joins

SQL has a special join that allows you to match column values in one table with a range of values in another table. A 'between join' is used to match rows of one table to rows in a second table by specifying that a column value in one table falls into a range of values specified in the other table.

If you want to display the employee number, emp name, basic salary, grade and the salary limits for each employee, you will have to write the following query:

```
SQL> SELECT empno, ename, sal, grade, losal, hisal FROM emp,
salgrade WHERE sal BETWEEN losal AND hisal;
```

EMP NO	ENAME	SAL	GRADE	LOSAL	HISAL
-----	-----	-----	-----	-----	-----
7369	SMITH	800	1	700	1200
7499	ALLEN	1600	1	700	1200
7521	WARD	1250	1	700	1200
7566	JONES	2975	2	1201	1400
7654	MARTIN	1250	2	1201	1400
7698	BLAKE	2850	2	1201	1400
7782	CLARK	2450	3	1401	2000
7788	SCOTT	3000	3	1401	2000

7839	KING	5000	4	2001	3000
7844	TURNER	1500	4	2001	3000
7876	ADAMS	1100	4	2001	3000
7900	JAMES	950	4	2001	3000
7902	FORD	3000	4	2001	3000
7934	MILLER	1300	5	3001	9999

Subquery

A subquery is a query whose results are passed as the argument for another query. Subqueries enable you to bind several queries together. You will understand and be able to use the keywords EXISTS, ANY, and ALL with your subqueries

Building a Subquery

Simply put, a subquery lets you tie the result set of one query to another. The general syntax is as follows:

SYNTAX:

```
SQL> SELECT * FROM TABLE1 WHERE TABLE1.SOMECOLUMN =
(SELECT SOMEOTHERCOLUMN FROM TABLE2 WHERE
SOMEOTHERCOLUMN = SOMEVALUE)
```

Notice how the second query is nested inside the first.

INPUT:

```
SQL> SELECT client_no, name FROM client_master WHERE
```

```
client_no= (SELECT client_no FROM sales_order WHERE
s_order_no= 'O19001');
```

OUTPUT:

CLIENT	NAME
-----	-----
C00001	Ivan Bayross

ANALYSIS:

Above query can be explain in a way, Find the customer no, name for the client who has placed order no 'O19001'. This query can be evaluated with the use of joins also as,

```
SQL> select c.client_no, c.name from client_master c, sales_order s
where c.client_no=s.client_no and s.s_order_no= 'O19001';
```

The difference between join query and subquery is nothing but some time it is compulsory to use sub query. This can be evaluated from material given below.

Nested SubQuery

Atmiya
Infotech Pvt. Ltd.

Nesting is the act of embedding a subquery within another subquery. For example:

```
Select * FROM SOMETHING WHERE (SUBQUERY (SUBQUERY
(SUBQUERY)));
```

Find the department no which does not have any employee..

```
SQL> SELECT deptno FROM dept WHERE deptno NOT IN (SELECT
DISTINCT(deptno) from emp);
```

DEPTNO

40

IN clause provides multiple records available in EMP table. And NOT clause makes result and returns value to DEPT table. Here deptno 40 is not available in EMP table, so result is 40.

In nested subquery there can be used more than one nesting or more than one table to evaluate appropriate result. Example: Find out the products which has been sold to 'Ivan Bayross'.

INPUT:

SQL> SELECT sod.product_no, p.description FROM sales_order_details sod, product_master p WHERE sod.product_no=p.product_no AND sod.s_order_no IN (SELECT so.s_order_no FROM sales_order so, client_master c WHERE so.client_no=c.client_no AND c.client_no IN (SELECT client_no FROM client_master WHERE name LIKE 'Ivan Bayross')) ;

OUTPUT:

PRODUC	DESCRIPTION
-----	-----
P00001	1.44 Floppies
P07885	CD Drive
P07965	540 HDD

Atmiya
Infotech Pvt. Ltd.

P03453 Monitors

P06734 Mouse

5 rows selected.

Example:2 Find the product_no and description of moving product

```
SQL> SELECT product_no, description FROM product_master WHERE  
product_no IN (SELECT DISTINCT(product_no) FROM  
sales_order_details);
```

Example:3 Find the product_no and description of non-moving product

```
SQL> SELECT product_no, description FROM product_master WHERE  
product_no NOT IN (SELECT DISTINCT(product_no) FROM  
sales_order_details );
```



Top:1 Creating and Altering Tables (with constraints)

Creating Tables

A) Simple Create Table Command Syntax:

CREATE TABLE tablename (ColumnName DataType(Size) Column Level Constraint, ColumnName DataType(Size), Table level Constraints);

Example:1

<u>Column Name</u>	<u>Data Type</u>	<u>Size</u>
client_No	Varchar2	6
name	Varchar2	20
address1	Varchar2	30
address2	Varchar2	30
city	Varchar2	15
state	Varchar2	15

pincode	Number	6
remarks	Varchar2	60
bal_due	Number	10,2

CREATE TABLE client_master (client_no varchar2(6), name varchar2(20), address1 varchar2(30), address2 varchar2(30), city varchar2(15), state varchar2(15), pincode Number(6), remarks varchar2(60), bal_due number(10,2));

B) Creating Table from existing Table Command Syntax:

CREATE TABLE tablename [(columnname, columnname)] AS SELECT columnname, columnname FROM tablename;

Example:2

Create table supplier_master from table client_master, select all fields and rename client_no with supplier_no and name with supplier_name.

CREATE TABLE supplier_master (supplier_no, supplier_name, address1, address2, city, state, pincode, remarks) AS SELECT client_no, name, address1, address2, city, state, pincode, remarks FROM client_master;

NOTE: If the Source table from which the Target table is being created, has records in it then the Target table is populated with these records as well. To eliminate this use where condition which is false in its sense, that will create structure of table only. Below is a example of that, (Where 1=2 is false condition)

CREATE TABLE supplier_master (supplier_no, supplier_name,

address1, address2, city, state, pincode, remarks) AS SELECT client_no, name, address1, address2, city, state, pincode, remarks FROM client_master WHERE 1=2;

Constraints in Create Tables

The Create table statement enforce you several different kinds of constraints on a table: candidate key, primary key, foreign key, check conditions. A constraint clause can contain a single column or a group of columns in a table. It maintains integrity on your database. The more constraint you add on table less you work in application. On other hand more constraint on a table, slower data to update in table. There are two ways to define constraints: as part of column definition (a column constraint) , at the end of Create Table command (a table level constraint)

CANDIDATE KEY

Candidate key is a combination of one or more columns, the value of which uniquely identify each row of a table. The following listing shows the creation of a UNIQUE constraint for the BILL table.

CREATE TABLE bill (bill_no number(6), bill_date date, client_no varchar2(6), remarks varchar2(60), Constraint uq_bill UNIQUE (bill_no, bill_date));

The key of this table is the combination of bill_no and bill_date. Notice that both the column are declared as NOT NULL. This feature allows you to prevent data from being entered into the table without certain columns having data into them.

PRIMARY KEY

You can have only one primary key and a primary key column does not contain NULL values,

CREATE TABLE bill (bill_no number(6), bill_date date, client_no varchar2(6), remarks varchar2(60), Constraint pk_bill PRIMARY KEY (bill_no, bill_date));

Above create table has the same effect as the previous one, except the you can have several UNIQUE constraint but only one PRIMARY KEY constraint.

For, Single-column primary or candidate keys, you can define the key on the column with a column name constraint instead of a table constraint:

CREATE TABLE client_master (client_no varchar2(6) PRIMARY KEY, name varchar2(20), address1 varchar2(20));

In this case client_no is primary key, and oracle will generate name for the PRIMARY KEY constraint.

FOREIGN KEY

A foreign key is a combination of columns with values based on the primary key values from another table. A Foreign key constraint, also known as a referential integrity constraint, specifies that the value of foreign key correspond to actual value of the primary key in the other table.

Example: Create table Sales_Order_Details with primary key as s_order_no and product_no and foreign key as s_order_no referencing column s_order_no in the sales_order table.

FOREIGN KEY as a column constraint

```
CREATE TABLE sales_order_details (s_order_no varchar2(6)
REFERENCES sales_order, product_no varchar2(6), qty_ordered
number(8), qty_disp number(8), product_rate number(8,2), PRIMARY
KEY (s_order_no, product_no));
```

FOREIGN KEY as a table constraint

```
CREATE TABLE sales_order_details (s_order_no varchar2(6),
product_no varchar2(6), qty_ordered number(8), qty_disp number(8),
product_rate number(8,2), PRIMARY KEY (s_order_no, product_no),
FOREIGN KEY (s_order_no) REFERENCES sales_order);
```

You can refer to a **primary key** or **unique key**, even in a same table. However, you can not refer to a table in a remote database in the **reference** clause. You can use table form instead of the column form to specify foreign keys with multiple columns.

CHECK CONSTRAINT

Many column must have values that are within a certain range or that satisfy certain conditions. With a **CHECK** constraint, you can specify an expression that must always be true for every row in a table. Never use CHECK constraint if the constraint can be defined using the not null, primary key or foreign key constraint.

following are a few examples of appropriate CHECK constraints:

- a CHECK constraint on the client_no column of the client_master so that no client_no value starts with 'C'.
- a CHECK constant on name column of the client_master so that the name is entered in upper case.
- a CHECK constraint on the city column of the client_master so that only the cities "BOMBAY", "NEWDELHI", "MADRAS", and

"CULCATTA" are allowed.

```
CREATE TABLE client_master (client_no varchar2(6)
CONSTRAINT ck_clientno CHECK (client_no like 'C%'), name
varchar2(20) CONSTRAINT ck_cname CHECK
(name=upper(name)), address1 varchar2(30), address2
varchar2(30), city varchar2(15) CONSTRAINT ck_city CHECK
(city IN ('NEWDELHI', 'BOMBAY', 'MADRAS', 'CULCATTA')),
state varchar2(15), pincode number(6), remarks varchar2(60),
bal_due number(10,2));
```

Restriction on CHECK Constraint:

A CHECK integrity constraint requires that a condition be true or unknown for every row of the table. If a statement causes the condition to evaluate to false; the statement is rolled back. The condition of a CHECK constraint has the following limitations;

- The condition must be a Boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition can not contain subqueries or sequences.
- The condition can not include the SYSDATE, UID, USER or USERENV SQL functions.



NOT NULL CONSTRAINT — Infotech Pvt. Ltd.

NOT NULL constraint can be used to restrict field for having no data. This means field having NOT NULL constraint must be entered by the user. Example of the same is given below.

```
CREATE TABLE sales_order_details (s_order_no varchar2(6),
product_no varchar2(6), qty_ordered number(8) NOT NULL, qty_disp
number(8), product_rate number(8,2) NOT NULL, PRIMARY KEY
```


(s_order_no, product_no));

Altering Tables

Tables can be altered in one of three ways : By adding a column to an existing table - by changing column definition - or by dropping column of table. Adding a column is straightforward, and similar to creating table.

- ALTER TABLE supplier_master ADD (state varchar2(20), country varchar2(20));

You can drop column in Oracle 8i, for the you have to type simple command like,

- ALTER TABLE supplier_master DROP column state;

To modify column several way are defined as under,

- ALTER TABLE supplier_master ADD PRIMARY KEY (supplier_no);
- ALTER TABLE supplier_master MODIFY (state varchar2(30));

Some other examples for alter table commands are given under, that can be used alternatively as an when required.

- ALTER TABLE supplier_master DROP PRIMARY KEY; (this command drops the primary key constraint from supplier_master)
- ALTER TABLE sales_order_details DROP CONSTRAINT product_fkey; (this command drop foreign key constraint on column product_no in table sales_order_details)
- ALTER TABLE sales_order_details ADD CONSTRAINT

order_fkey FOREIGN KEY (s_order_no) REFERENCES sales_order
MODIFY (qty_ordered number(8) NOT NULL);

RULES for Adding or Modifying a Column

These are rules for adding a column to table

1. You may add a column at any time if NOT NULL isn't specified.
2. You may add a NOT NULL column in three steps:
 - Add a column without NOT NULL specified.
 - Fill every row in that column with data.
 - Modify the column to be NOT NULL.

These are rules for modifying a column to table

1. You can increase a character column's width at any time.
2. You can increase the number of digits in a NUMBER column at any time.
3. You can increase or decrease the number of decimal places in a NUMBER column at any time.

In case only if whole column to be modify is NULL, you can

1. You can change the Column's DataType.
2. You can decrease a character column's width.
3. You can decrease the number of digits in a NUMBER column.

Disabling Constraints:

Constraints can be temporarily removed by using the following statement:

```
ALTER TABLE <tablename> DISABLE CONSTRAINT  
<constraintname>;
```

Example:

```
alter table emp disable constraint pk_en;
```

After disabling a primary key constraint if you try to add a duplicate row in the table, it will allow you to, since the primary key constraint has been disabled.

Enabling Constraints:

To enable the constraint again the following statement can be coded:

```
ALTER TABLE <tablename> ENABLE CONSTRAINT <constraintname>;
```

example:

```
alter table emp enable constraint pk_en;
```

Removing Constraints:

Constraints can be permanently removed if not required by coding the

following statement:

```
ALTER TABLE <tablename> DROP CONSTRAINT <constraintname>;
```

example:

```
alter table emp DROP CONSTRAINT pk_en;
```

Note: A primary key constraint can not be removed if some foreign keys are referencing that primary key, but you can forcefully drop the primary key with the cascade option as shown below.

```
ALTER TABLE <tablename> DROP PRIMARY KEY CASCADE;
```

The above statement will drop the primary key as well as all the foreign keys that reference it.

Another way to enable, disable and drop primary key constraints:

```
ALTER TABLE <tablename> DISABLE PRIMARY KEY;
```

```
ALTER TABLE <tablename> ENABLE PRIMARY KEY;
```

```
ALTER TABLE <tablename> DROP PRIMARY KEY;
```

Here the name of the constraint is not mentioned. But if you are enabling, disabling and dropping any other constraint the name of the constraint is required.

Rename Table

SQL *Plus Data Definition Language allows us to easily rename the existing tables. Only the owner of the table can rename the table. The

RENAME statement is very easy to code. As you can see in syntax:

RENAME <tablename> **TO** <newtablename>;

Example:

RENAME book **TO** books;



Top:2 Data Definition Language

Data Definition Language Comprises of these commands:

DDL (Data Definition Language)

CREATE	to create table or objects
ALTER	to alter existing database
DROP	to drop existing objects
TRUNCATE	to remove whole data at a time

CREATE COMMAND

ALTER COMMAND

DROP COMMAND

DROP is the SQL verb used to delete tables from the database. It is easy to understand, and almost too easy to use. All that is necessary to delete entire tables is execute the DROP TABLE statement. The DROP TABLE statement is very easy to code. As you can see from figure,

Infotech Pvt. Ltd.

DROP TABLE <tablename>

1. DROP TABLE - This keyword tells the DBMS that it has to delete the table.

2. `<tablename>` - The table name of the table you wish to delete.
3. A semicolon must be the last item in the statement. A semicolon tells SQL that statement is complete and it should now be executed.

For, Example, to drop a table temp, you would code:

```
DROP TABLE dept;
```

Once executed, the table is gone - all rows, any INDEXes associated with the table, and any VIEWs associated with the table. The system does not give you a 'last chance to bail out' prompt. Your data is gone for good. The table definition is removed from the system and your table does not exist anymore. Similarly, you can also drop primary keys, constraints, and other database objects, using the DROP command.

SQL, however, does ensure that you have the authority to DROP tables (usually only the ones that you are created), before it execute the statement. However this is a dangerously powerful verb. It is particularly important to check and recheck the spelling of the table name to be deleted. You might inadvertently delete the wrong one.

If you are faint of heart, you might want to delete all the rows from a table instead of dropping the table. The DELETE command can be undone with the ROLLBACK command, whereas the DROP command can not be undone. (The DELETE and ROLLBACK commands are covered later during the course.)

Infotech Pvt. Ltd.

For example, to delete all the rows in the member table, you would code:

```
DELETE FROM member;
```

TRUNCATE COMMAND

Truncate command is useful to empty table records, Which can be state as

TRUNCATE TABLE <tablename>

On one click of ENTER key you will find your table with no data. For example,

```
SQL> TRUNCATE TABLE emp;
```

Now to check records for emp table, let us use.....

```
SQL> SELECT * FROM emp;
```

no rows selected.

RENAME COMMAND

SQL * Plus Data Definition Language allows us to easily rename the existing tables. Only the owner of the table can rename the table. The RENAME statement is very easy to code. As you can see from figure,

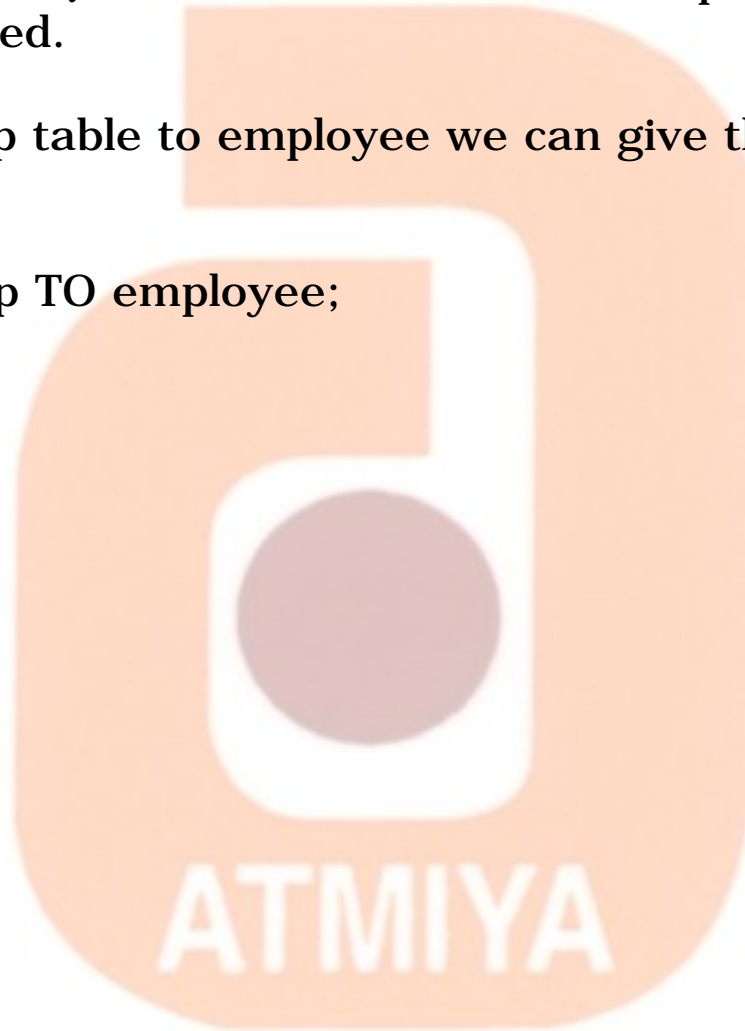
RENAME <oldtablename> TO <newtablename>;

1. RENAME - This tells the RDBMS that it has to rename the table.
2. <oldtablename> - The table name of the table you wish to rename.
3. TO - The keyword to

4. **<newtablename>** - The new table name that you wish the table to have.
5. A semicolon must be the last item in the statement. A semicolon tells SQL that the statement is complete and it should now be executed.

To rename the emp table to employee we can give the following statement:

```
SQL> RENAME emp TO employee;
```



TOP:3 Data Manipulation Commands

You may have used a PC-based product such as Access, dBASE IV, or FoxPro to enter your data in the past. These products come packaged with excellent tools to enter, edit, and delete records from databases. One reason that SQL provides data manipulation statements is that it is primarily used within application programs that enable the user to edit the data using the application's own tools. The SQL programmer needs to be able to return the data to the database using SQL. In addition, most large-scale database systems are not designed with the database designer or programmer in mind. Because these systems are designed to be used in high-volume, multiuser environments, the primary design emphasis is placed on the query optimizer and data retrieval engines.

DML command have 4 category commands like, are explained below

INSERT Command
UPDATE Command
DELETE Command
SELECT Command

INSERT Command

The SQL Command INSERT lets you place a row of information directly into a table. Here three possible way of writing INSERT statement is shown below,

INSERT INTO <table name> values <values for all column>
INSERT INTO <table name> (column names) values (values for column)
INSERT INTO <table name> as SELECT (column name) from <table name>

IN Detail:

The INSERT statement enables you to enter data into the database. It can be broken down into two statements:

INSERT...VALUES

and

INSERT...SELECT

The INSERT...VALUES Statement

The INSERT...VALUES statement enters data into a table one record at a time. It is useful for small operations that deal with just a few records. The syntax of this statement is as follows:

SYNTAX:

INSERT INTO table_name (col1, col2...) VALUES (value1, value2...);

The basic format of the INSERT...VALUES statement adds a record to a table using the columns you give it and the corresponding values you instruct it to add. You must follow three rules when inserting data into a table with the INSERT...VALUES statement:

The values used must be the same data type as the fields they are being added to.

The data's size must be within the column's size. For instance, you cannot add an 80-character string to a 40-character column.

The data's location in the VALUES list must correspond to the location in the column list of the column it is being added to. (That is, the first value must be entered into the first column, the second value into the second column, and so on.)

Example : 1 If you wanted to add a new record to EMP table, you would write

INPUT:

```
SQL> INSERT INTO emp (empno, ename, job, mgr, hiredate, sal,
comm, deptno)
```

```
VALUES (1111, 'JIGNESH', 'DIRECTOR', 1112, '26-JUN-75', 6000,
4000, 10);
```

OUTPUT:

1 row created.



You can execute a simple SELECT statement to verify the insertion:

INPUT:

```
SQL> SELECT * FROM EMP;
```

ANALYSIS:

The INSERT statement does not require column names. If the column names are not entered, SQL lines up the values with their corresponding column numbers. In other words, SQL inserts the first value into the first column, the second value into the second column, and so on.

Example : 2 The following statement inserts the values from Example 1 into the table:

INPUT:

```
SQL> INSERT INTO emp VALUES (1111, 'JIGNESH', 'DIRECTOR',  
1112, '26-JUN-75', 6000, 4000, 10);
```

1 row created.

ANALYSIS:

By issuing the same SELECT statement as you did in Example 1, you can verify that the insertion worked as expected:

Inserting NULL Values

For now, all you need to know is that when a column is created, it can have several different limitations placed upon it. One of these limitations is that the column should (or should not) be allowed to contain NULL values. A NULL value means that the value is empty. It is neither a zero, in the case of an integer, nor a space, in the case of a string. Instead, no data at all exists for that record's column. If a column is defined as NOT NULL (that column is not allowed to contain a NULL value), you must insert a value for that column when using the INSERT statement. The INSERT is canceled if this rule is broken, and you should receive a descriptive error message concerning your error. In case, if you require to keep some column empty then you must specify columns to which you want to add data and can omit rest

column where you are not intended to add data.

INPUT:

```
SQL> INSERT INTO emp (empno, ename, job, sal, deptno)
VALUES (1112, 'NAMRATA', 'CEO', 6000, 20);
```

OUTPUT:

1 row created.

ANALYSIS:

Data will be added to specified columns only and rest will be NULL.

Inserting Unique Values

Database management systems also allow you to create a UNIQUE column attribute. This attribute means that within the current table, the values within this column must be completely unique and cannot appear more than once. This limitation can cause problems when inserting or updating values into an existing table, as the following exchange demonstrates:

INPUT:

```
SQL> INSERT INTO emp VALUES (1111, 'JIGNESH', 'DIRECTOR',
1112, '26-JUN-75', 6000, 4000, 10);
```

OUTPUT:

```
INSERT INTO emp VALUES (1111, 'JIGNESH', 'DIRECTOR', 1112, '26-
JUN-75', 6000, 4000, 10)
```

*

ERROR at line 1:

ORA-00001: unique constraint (SCOTT.UNQ_EMP_ITEM) violated

ANALYSIS:

In this example you tried to insert another EMPNO called 1111 into the EMP table. Because this table was created with EMPNO as a unique value, it returned the appropriate error. A properly normalized table should have a unique, or key, field. This field is useful for joining data between tables, and it often improves the speed of your queries when using indexes.

The INSERT...SELECT Statement

The INSERT...VALUES statement is useful when adding single records to a database table, but it obviously has limitations. Would you like to use it to add 25,000 records to a table? In situations like this, the INSERT...SELECT statement is much more beneficial. It enables the programmer to copy information from a table or group of tables into another table. You will want to use this statement in several situations. Lookup tables are often created for performance gains. Lookup tables can contain data that is spread out across multiple tables in multiple databases. Because multiple-table joins are slower to process than simple queries, it is much quicker to execute a SELECT query against a lookup table than to execute a long, complicated joined query. Lookup tables are often stored on the client machines in client/server environments to reduce network traffic.

The INSERT...SELECT statement can take the output of a SELECT statement and insert these values into a table. Here is an example:

INPUT:

```
SQL> INSERT INTO EMP1 select * from EMP;
```

OUTPUT:

14 rows inserted.

ANALYSIS:

You are selecting all the rows that are in table and inserting them into EMP.

The syntax of the INSERT...SELECT statement is as follows:

SYNTAX:

```
INSERT INTO table_name (col1, col2...) SELECT col1, col2... FROM  
tablename WHERE search_condition;
```

Essentially, the output of a standard SELECT query is then input into a database table. The same rules that applied to the INSERT...VALUES statement apply to the INSERT...SELECT statement.

The INSERT...SELECT statement requires you to follow several new rules:

- The SELECT statement cannot select rows from the table that is being inserted into.
- The number of columns in the INSERT INTO statement must equal the number of columns returned from the SELECT statement.
- The data types of the columns in the INSERT INTO statement must be the same as the data types of the columns returned from the SELECT statement.

Another use of the INSERT...SELECT statement is to back up a table that you are going to drop, truncate for repopulation, or rebuild. The process requires you to create a temporary table and insert data that is contained in your original table into the temporary table by selecting

everything from the original table.

UPDATE Command

The SQL Command UPDATE modify existing database values. Below is a list of way, you deal with database.

Update <table name> set (column name) = (value)
Update <table name> set (column name) = (value) where (condition)

IN Detail:

The purpose of the UPDATE statement is to change the values of existing records. The syntax is

SYNTAX:

UPDATE table_name SET columnname1 = value1 [, columnname2 = value2]... WHERE search_condition;

This statement checks the WHERE clause first. For all records in the given table in which the WHERE clause evaluates to TRUE, the corresponding value is updated.

Example : 4 This example illustrates the use of the UPDATE statement:

INPUT:

SQL> UPDATE emp SET sal= 7000 WHERE empno= 1111;

OUTPUT:

1 row updated.

To confirm the change, the query

INPUT:

```
SQL> SELECT * FROM emp;
```

Here is a multiple-column update:

INPUT:

```
SQL> UPDATE emp SET sal=sal+ 100, comm=comm- 100;
```

14 row updated.

ANALYSIS:

If you omit WHERE clause, all the record will be updated.

DELETE Command

TO delete existing data in your table, you can use these syntax

Delete from <table name>
Delete from <table name> where (condition)

IN Detail:

In addition to adding data to a database, you will also need to delete data from a database. The syntax for the DELETE statement is

SYNTAX:

DELETE FROM tablename WHERE condition

The first thing you will probably notice about the DELETE command is that it doesn't have a prompt. Users are accustomed to being prompted for assurance when, for instance, a directory or file is deleted at the operating system level. Are you sure? (Y/N) is a common question asked before the operation is performed. Using SQL, when you instruct the DBMS to delete a group of records from a table, it obeys your command without asking. That is, when you tell SQL to delete a group of records, it will really do it!

Depending on the use of the DELETE statement's WHERE clause, SQL can do the following:

- Delete single rows
- Delete multiple rows
- Delete all rows
- Delete no rows

Here are several points to remember when using the DELETE statement:

- The DELETE statement cannot delete an individual field's values (use UPDATE instead). The DELETE statement deletes entire records from a single table.
- Like INSERT and UPDATE, deleting records from one table can cause referential integrity problems within other tables. Keep this potential problem area in mind when modifying data within a database.

- Using the DELETE statement deletes only records, not the table itself. Use the DROP TABLE statement to remove an entire table.

Example :

This example shows you how to delete all the records from EMP where SAL is less than 2750.

INPUT:

SQL> DELETE FROM emp WHERE sal < 2750;

OUTPUT:

6 row deleted.

INPUT:

SQL> DELETE FROM emp;

OUTPUT:

14 row deleted.

ANALYSIS:

First input delete searched records on basis of where condition and Second input delete all the records available in table because on where condition is applied. It is advisable to use delete command very carefully.

SELECT Command

To Display database records in your expected way, you need to write

any of your required way which are shown below.

Select * from <table name>

Select (column name) from <table name>

Select (column name) from <table name> where (condition)

**Select (column name) from <table name> where (condition)
Group by (column name) Order by (column name)**

NOTE: This section is discussed in detail in previous top. To switch there [CLICK ME.](#)



Top:4 Built in Functions

Now we talk about functions. Functions in SQL enable you to perform feats such as determining the sum of a column or converting all the characters of a string to uppercase. By the end of the day, you will understand and be able to use all the following:

- [**Aggregate functions**](#)
- [**Date and time functions**](#)
- [**Arithmetic functions**](#)
- [**Character functions**](#)
- [**Conversion functions**](#)
- [**Miscellaneous functions**](#)

These functions greatly increase your ability to manipulate the information you retrieved using the basic functions of SQL.



Top:5A Introduction to SQL * Plus

Objectives

By the end topic, you will understand the following elements of SQL*Plus:

- **How to use the SQL*Plus buffer**
- **How to format reports attractively**
- **How to manipulate dates**
- **How to make interactive queries**
- **How to construct advanced reports**
- **How to use the powerful DECODE function**

Introduction

We are presenting SQL*Plus today because of Oracle's dominance in the relational database market and because of the power and flexibility SQL*Plus offers to the database user.

SQL*Plus commands can enhance an SQL session and improve the format of queries from the database. SQL*Plus can also format reports, much like a dedicated report writer. SQL*Plus supplements both standard SQL and PL/SQL and helps relational database programmers gather data that is in a desirable format.

The SQL*Plus Buffer

The SQL*Plus buffer is an area that stores commands that are specific to your particular SQL session. These commands include the most recently executed SQL statement and commands that you have used to customize your SQL session, such as formatting commands and variable assignments. This buffer is like a short-term memory. Here are some of the most common SQL buffer commands:

1. LIST line_number--Lists a line from the statement in the buffer and designates it as the current line.
2. CHANGE/old_value/new_value--Changes old_value to new_value on the current line in the buffer.
3. APPEND text--Appends text to the current line in the buffer.
4. DEL-- Deletes the current line in the buffer.
5. SAVE newfile--Saves the SQL statement in the buffer to a file.
6. GET filename--Gets an SQL file and places it into the buffer.
7. /--Executes the SQL statement in the buffer.

1. LIST

INPUT:

```
SQL> SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3* WHERE cost_price > 1525;
```



OUTPUT:

PRODUC	DESCRIPTION	COST_PRICE	SELL_PRICE
-----	-----	-----	-----
P03453	Monitors	11280	12000
P07868	Keyboards	3050	3150
P07885	CD Drive	5100	5250
P07965	540 HDD	8000	8400

The LIST command lists the most recently executed SQL statement in the buffer. The output will simply be the displayed statement.

SQL> LIST

```

1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3* WHERE cost_price > 1525;
```

ANALYSIS:

Notice that each line is numbered. Line numbers are important in the buffer; they act as pointers that enable you to modify specific lines of your statement using the SQL*PLUS buffer. The SQL*Plus buffer is not a full screen editor; after you hit Enter, you cannot use the cursor to move up a line. NOTE: As with SQL commands, you may issue SQL*Plus commands in either uppercase or lowercase.

TIP: You can abbreviate most SQL*Plus commands; for example, LIST can be abbreviated as l. You can move to a specific line from the

buffer by placing a line number after the L:

INPUT:

SQL> L3;

3* WHERE cost_price > 1525;

ANALYSIS:

Notice the asterisk after the line number 3. This asterisk denotes the current line number. Pay close attention to the placement of the asterisk in examples. Whenever a line is marked by the asterisk, you can make changes to that line. Because you know that your current line is 3, you are free to make changes.

2. CHANGE

You can change your old values to new values with the use of change command, so you need not to work hard in changing queries. Simple rules for that, you go to line no you want to change with the use of LIST (L) command say L3 goes to line 3, then apply change command. The syntax for the CHANGE command is as follows:

SYNTAX:

CHANGE/old_value/new_value **or** C/old_value/new_value

INPUT:

SQL> CHANGE/1525/3525

OUTPUT:

3* WHERE cost_price > 3525

INPUT:

SQL> L

OUTPUT:

```
1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3* WHERE cost_price > 3525;
```

ANALYSIS:

The cost_price 1525 has been changed to 3525 on line 3. Notice after the change was made that the newly modified line was displayed. If you issue the LIST command or L, you can see the full statement.

3. / (SLASH)

Now execute the statement:

INPUT:

SQL> /

OUTPUT:

PRODUC	DESCRIPTION	COST_PRICE	SELL_PRICE
P03453	Monitors	11280	12000
P07885	CD Drive	5100	5250
P07965	540 HDD	8000	8400

ANALYSIS:

The forward slash at the SQL> prompt executes any statement that is in the buffer.

INPUT:

```
SQL> L
```

OUTPUT:

```
1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3* WHERE cost_price > 3525;
```

Now, you can add a line to your statement by typing a new line number at the SQL> prompt and entering text. After you make the addition, get a full statement listing. Here's an example:

INPUT:

```
SQL> 4 order by cost_price
```

```
SQL> L
```

OUTPUT:

```
1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3 WHERE cost_price > 3525
4* order by cost_price
```



ANALYSIS:

Deleting a line is easier than adding a line. Simply type DEL 4 at the SQL> prompt to delete line 4. Now get another statement listing to verify that the line is gone.

4. DEL

INPUT:

```
SQL> DEL4
```

```
SQL> L
```

OUTPUT:

```
1 SELECT product_no, description, cost_price, sell_price
```

```
2 FROM product_master
```

```
3* WHERE cost_price > 3525
```

Another way to add one or more lines to your statement is to use the INPUT command. As you can see in the preceding list, the current line number is 3. At the prompt type input and then press Enter. Now you can begin typing text. Each time you press Enter, another line will be created. If you press Enter twice, you will obtain another SQL> prompt. Now if you display a statement listing, as in the following example, you can see that line 4 has been added.

5. INPUT

INPUT:

```
SQL> INPUT
```

```
4 and sell_price<12000
```

5 order by cost_price

6

SQL> L

OUTPUT:

1 SELECT product_no, description, cost_price, sell_price

2 FROM product_master

3 WHERE cost_price > 3525

4 AND sell_price < 12000

5* ORDER BY cost_price

To append text to the current line, issue the APPEND command followed by the text. Compare the output in the preceding example--the current line number is 5--to the following example.

6. APPEND

INPUT:

SQL> APPEND desc 

OUTPUT:

5* ORDER BY cost_price desc

Now get a full listing of your statement:

INPUT:

SQL> l

OUTPUT:

```
1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3 WHERE cost_price > 3525
4 AND sell_price < 12000
5* ORDER BY cost_price desc
```

Suppose you want to wipe the slate clean. You can clear the contents of the SQL*Plus buffer by issuing the command CLEAR BUFFER. As you will see later, you can also use the CLEAR command to clear specific settings from the buffer, such as column formatting information and computes on a report.

7. CLEAR**INPUT:**

```
SQL> CLEAR BUFFER
```

OUTPUT:

buffer cleared

INPUT:

```
SQL> L
```

OUTPUT:

No lines in SQL buffer.

ANALYSIS:

Obviously, you won't be able to retrieve anything from an empty buffer.

8. The DESCRIBE Command

The handy DESCRIBE command enables you to view the structure of a table quickly without having to create a query against the data dictionary.

SYNTAX:

DESC[RIBE] table_name

INPUT:

SQL> DESCRIBE dept

Name	Null?	Type
-----	-----	-----
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

ANALYSIS:

DESC displays each column name, which columns must contain data (NULL/NOT NULL), and the data type for each column. If you are writing many queries, you will find that few days go by without using this command. Over a long time, this command can save you many hours of programming time. Without DESCRIBE you would have to

search through project documentation or even database manuals containing lists of data dictionary tables to get this information.

9. The SHOW Command

The SHOW command displays the session's current settings, from formatting commands to who you are. SHOW ALL displays all settings. This discussion covers some of the most common settings.

INPUT:

```
SQL> show all
```

OUTPUT:

appinfo is ON and set to "SQL*Plus"

arraysize 15

autocommit OFF

autoprint OFF

autotrace OFF

shiftinout INVISIBLE

blockterminator "." (hex 2e)

bttitle OFF and is the 1st few characters of the next SELECT statement

cmdsep OFF

colsep " "

compatibility version NATIVE



concat "." (hex 2e)

copycommit 0

COPYTYPECHECK is ON

define "&" (hex 26)

echo OFF

editfile "afiedt.buf"

embedded OFF

escape OFF

FEEDBACK ON for 6 or more rows

flagger OFF

flush ON

heading ON

headsep "|" (hex 7c)

linesize 100

lno 6

loboffset 1

long 80

longchunksize 80

newpage 1



Infotech Pvt. Ltd.

null ""

numformat ""

numwidth 9

pagesize 24

PAUSE is OFF

pno 1

recsep WRAP

recsepchar " " (hex 20)

release 800040000

repfooter OFF and is NULL

repheader OFF and is NULL

serveroutput OFF

showmode OFF

spool OFF

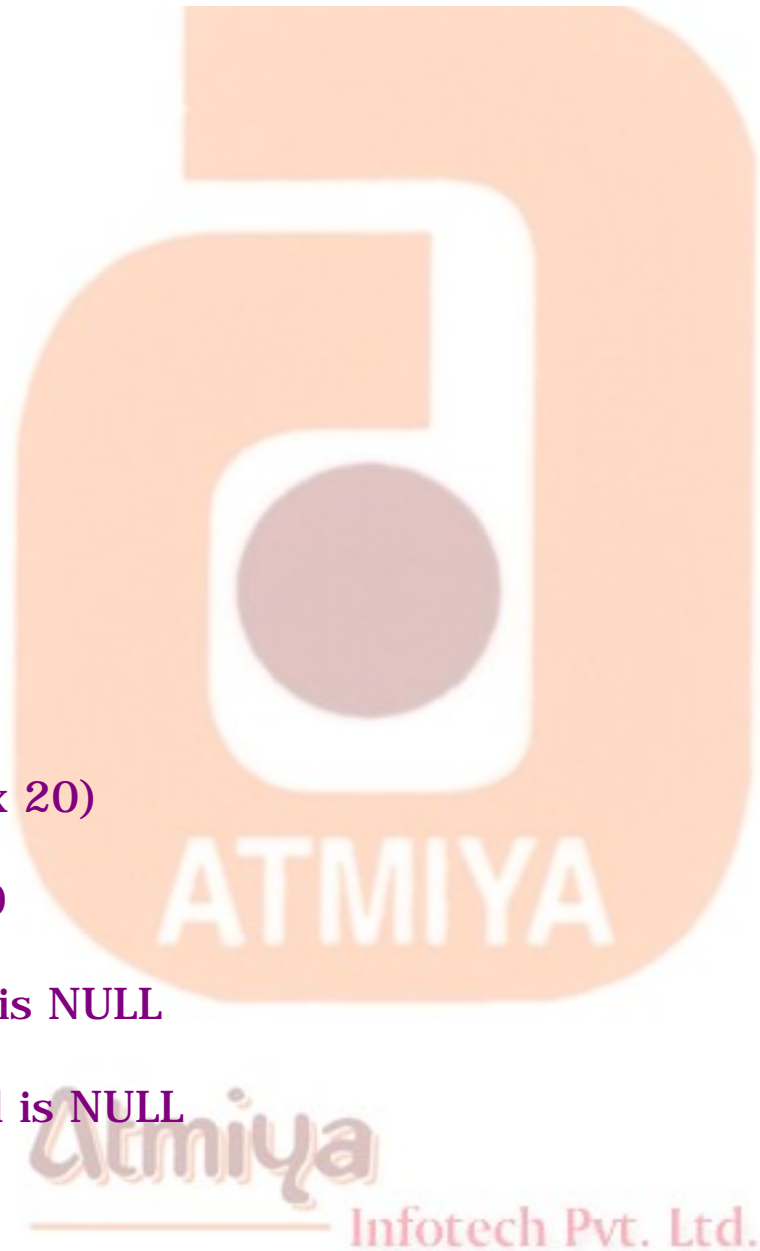
sqlcase MIXED

sqlcode 904

sqlcontinue "> "

sqlnumber ON

sqlprefix "#" (hex 23)



sqlprompt "SQL> "

sqlterminator ";" (hex 3b)

suffix "SQL"

tab ON

termout ON

time OFF

timing OFF

trimout ON

trimspace OFF

tttitle OFF and is the 1st few characters of the next SELECT statement

underline "-" (hex 2d)

USER is "SCOTT"

verify ON

wrap : lines will be wrapped

The SHOW command displays a specific setting entered by the user. Suppose you have access to multiple database user IDs and you want to see how you are logged on. You can issue the following command:

INPUT:

SQL> **show user**



OUTPUT:

user is "SCOTT"

To see the current line size of output, you would type:

INPUT:

SQL> **show linesize**

OUTPUT:

linesize 100

10. File Commands

Various commands enable you to manipulate files in SQL*Plus. These commands include creating a file, editing the file using a full-screen editor as opposed to using the SQL*Plus buffer, and redirecting output to a file. You also need to know how to execute an SQL file after it is created.

The SAVE, GET, START and EDIT Commands

The SAVE command saves the contents of the SQL statement in the buffer to a file whose name you specify. For example:

INPUT:

SQL> SELECT product_no, description, cost_price, sell_price

2 FROM product_master

3 WHERE cost_price > 3525

SQL> save query1.sql

OUTPUT:

Created file query1.sql

ANALYSIS:

After a file has been saved, you can use the GET command to list the file. GET is very similar to the LIST command. Just remember that GET deals with statements that have been saved to files, whereas LIST deals with the statement that is stored in the buffer.

INPUT:

SQL> get query1

OUTPUT:

1 SELECT product_no, description, cost_price, sell_price
2 FROM product_master
3* WHERE cost_price > 3525

You can use the EDIT command either to create a new file or to edit an existing file. When issuing this command, you are taken into a full-screen editor, more than likely Notepad in Windows. You will find that it is usually easier to modify a file with EDIT than through the buffer, particularly if you are dealing with a large or complex statement.

INPUT:

SQL> edit query1.sql

Starting a File, Now that you know how to create and edit an SQL file, the command to execute it is simple. It can take one of the following forms: (Note: Commands are not case sensitive.)

SYNTAX:

START filename **or** STA filename **or** @filename

INPUT:

SQL> **start query1.sql**

OUTPUT:

PRODUC	DESCRIPTION	COST_PRICE	SELL_PRICE
-----	-----	-----	-----
P03453	Monitors	11280	12000
P07885	CD Drive	5100	5250
P07965	540 HDD	8000	8400

NOTE: You do not have to specify the file extension .sql to start a file from SQL*Plus. The database assumes that the file you are executing has this extension. Similarly, when you are creating a file from the SQL> prompt or use SAVE, GET, or EDIT, you do not have to include the extension if it is .sql.

Other similar INPUTs:

SQL> **@ query1**

SQL> **sta query1**

INPUT:

SQL> **run query1**

OUTPUT:

1 SELECT product_no, description, cost_price, sell_price

2 FROM product_master

3* WHERE cost_price > 3525

PRODUC	DESCRIPTION	COST_PRICE	SELL_PRICE
-----	-----	-----	-----
P03453	Monitors	11280	12000
P07885	CD Drive	5100	5250
P07965	540 HDD	8000	8400

Notice that when you use RUN to execute a query, the statement is echoed, or displayed on the screen.

11. Spooling Query Output

Viewing the output of your query on the screen is very convenient, but what if you want to save the results for future reference or you want to print the file? The SPOOL command allows you to send your output

to a specified file. If the file does not exist, it will be created. If the file exists, it will be overwritten.

INPUT:

```
SQL> SPOOL department.lst
```

```
SQL> SELECT * FROM dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

4 rows selected.

INPUT:

```
SQL> SPOOL OFF
```

```
SQL> EDIT department.lst
```

ANALYSIS:

The output in this case is an SQL*Plus file. You must use the SPOOL OFF command to stop spooling to a file. When you exit SQL*Plus, SPOOL OFF is automatic. But if you do not exit and you continue to work in SQL*Plus, everything you do will be spooled to your file until you issue the command SPOOL OFF.

12. SET Commands

SET commands in Oracle change SQL*Plus session settings. By using these commands, you can customize your SQL working environment and invoke options to make your output results more presentable. You can control many of the SET commands by turning an option on or off.

To see how the SET commands work, perform a simple select:

INPUT:

```
SQL> SELECT * FROM dept;
```

OUTPUT:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

4 rows selected.

ANALYSIS:

The last line of output

4 rows selected.

is called feedback, which is an SQL setting that can be modified. The settings have defaults, and in this case the default for FEEDBACK is on. If you wanted, you could type

SET FEEDBACK ON

before issuing your select statement. Now suppose that you do not want to see the feedback, as happens to be the case with some reports, particularly summarized reports with computations.

INPUT:

SQL> SET FEEDBACK OFF

SQL> SELECT * FROM dept;

OUTPUT:

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ANALYSIS:

SET FEEDBACK OFF turns off the feedback display.

In some cases you may want to suppress the column headings from

being displayed on a report. This setting is called **HEADING**, which can also be set **ON** or **OFF**.

INPUT:

```
SQL> SET HEADING OFF
```

```
SQL> SELECT * FROM dept;
```

OUTPUT:

10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ANALYSIS:

The column headings have been eliminated from the output. Only the actual data is displayed.

You can change a wide array of settings to manipulate how your output is displayed. One option, **LINE SIZE**, allows you to specify the length of each line of your output. A small line size will more than likely cause your output to wrap; increasing the line size may be necessary to suppress wrapping of a line that exceeds the default 80 characters. Unless you are using wide computer paper (11 x 14), you may want to landscape print your report if you are using a line size greater than 80. The following example shows the use of **LINE SIZE**.

INPUT:

```
SQL> SET LINE SIZE 20
```


SQL> SELECT * FROM dept;

OUTPUT:

10

ACCOUNTING

NEW YORK

20

RESEARCH

DALLAS

30

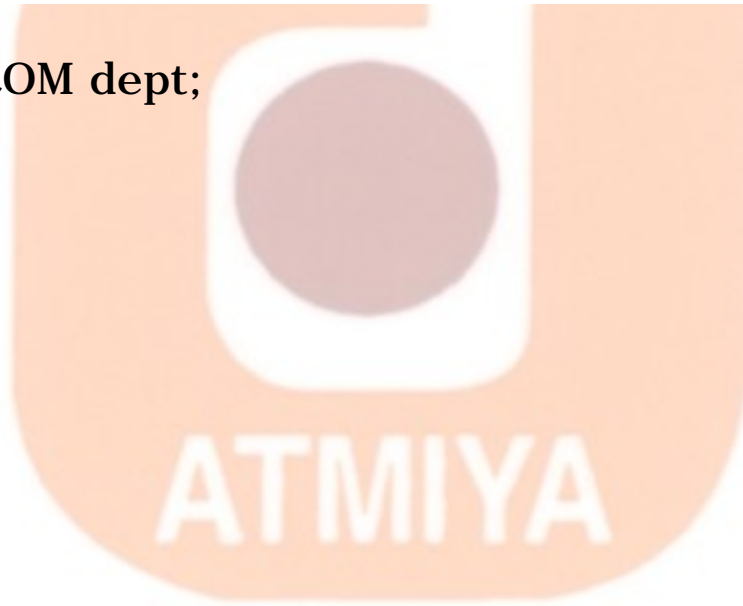
SALES

CHICAGO

40

OPERATIONS

BOSTON



You can also adjust the size of each page of your output by using the setting PAGESIZE. If you are simply viewing your output on screen, the best setting for PAGESIZE is 23, which eliminates multiple page breaks per screen. In the following example PAGESIZE is set to a low number to show you what happens on each page break.

INPUT:

SQL> **set linesize 80**

SQL> **set heading on**

SQL> **set pagesize 5**

SQL> /

OUTPUT:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
DEPTNO	DNAME	LOC
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ANALYSIS:

Using the setting of PAGESIZE 5, the maximum number of lines that may appear on a single page is FIVE. New column headings will print automatically at the start of each new page.

The TIME setting displays the current time as part of your SQL> prompt.

INPUT:

SQL> **SET TIME ON**

OUTPUT:

08:52:02 SQL>



These were just a few of the SET options, but they are all manipulated in basically the same way. As you saw from the vast list of SET commands in the earlier output from the SHOW ALL statement, you have many options when customizing your SQL*Plus session. Experiment with each option and see what you like best. You will probably keep the default for many options, but you may find yourself changing other options frequently based on different scenarios.

13. LOGIN.SQL File

When you log out of SQL*Plus, all of your session settings are cleared. When you log back in, your settings will have to be reinitialized if they are not the defaults unless you are using a login.sql file. This file is automatically executed when you sign on to SQL*Plus. This initialization file is similar to the autoexec.bat file on your PC. In Personal Oracle8 you can use the EDIT command to create your Login.sql file. Type this in Login.sql file:

SET TIME ON

```
SELECT "HELLO!" FROM dual;
```

When you log on to SQL*Plus, here is what you will see:

```
SQL*Plus: Release 8.0.4.0.0 - Production on Tue Sep 4 18:55:13 2001
```

```
(c) Copyright 1997 Oracle Corporation. All rights reserved.
```

```
Enter password: ****
```

```
Connected to:
```

```
Oracle8 Personal Edition Release 8.0.4.0.0 - Production
```

```
PL/SQL Release 8.0.4.0.0 - Production
```

```
With the distributed and replication options
```

```
PL/SQL Release 2.3.2.0.0 - Production
```

```
'HELLO!'
```

```
-----
```

```
HELLO !
```

```
20:38:02 SQL>
```

14. CLEAR Commands

In SQL*Plus, settings are cleared by logging off, or exiting SQL*Plus. Some of your settings may also be cleared by using the CLEAR command, as shown in the following examples.

INPUT:

SQL> clear col

OUTPUT:

columns cleared

INPUT:

SQL> clear break

OUTPUT:

breaks cleared

INPUT:

SQL> clear compute

OUTPUT:

computes cleared



Top:1 Creating VIEW

In this chapter the focus shifts to two features of SQL that enable you to view or present data in a different format than it appears on the disk. This feature is the view. By the end this chapter, you will know the following:

- How to create views
- How to modify data using views

A view is often referred to as a virtual table. Views are created by using the CREATE VIEW statement. After the view has been created, you can use the following SQL commands to refer to that view:

SELECT - INSERT - INPUT - UPDATE - DELETE

Using Views

You can use views, or virtual tables, to encapsulate complex queries. After a view on a set of data has been created, you can treat that view as another table. However, special restrictions are placed on modifying the data within views. When data in a table changes, what you see when you query the view also changes. Views do not take up physical space in the database as tables do.

The syntax for the CREATE VIEW statement is

SYNTAX:

CREATE VIEW <view_name> [(column1, column2...)] AS SELECT


```
<table_name column_names> FROM <table_name>;
```

As usual, this syntax may not be clear at first glance, but material contains many examples that illustrate the uses and advantages of views. This command tells SQL to create a view (with the name of your choice) that comprises columns (with the names of your choice if you like). An SQL SELECT statement determines the fields in these columns and their data types. Yes, this is the same SELECT statement that you have used repeatedly.

Before you can do anything useful with views, you need to populate the EMP and DEPT database with a little more data.

A Simple View

Let's begin with the simplest of all views. Suppose, for some unknown reason, you want to make a view on the DEPT table that looks identical to the table but has a different name. (We call it DEPARTMENT.) Here's the statement:

INPUT:

```
SQL> CREATE VIEW department AS SELECT * FROM dept;
```

OUTPUT:

view created.

To confirm that this operation did what it should, you can treat the view just like a table:

INPUT/OUTPUT:

```
SQL> SELECT * FROM department;
```

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

You can even create new views from existing views. Be careful when creating views of views. Although this practice is acceptable, it complicates maintenance. Suppose you have a view three levels down from a table, such as a view of a view of a view of a table. What do you think will happen if the first view on the table is dropped? The other two views will still exist, but they will be useless because they get part of their information from the first view. Remember, after the view has been created, it functions as a virtual table.

INPUT:

```
SQL> CREATE VIEW salesman AS SELECT * FROM emp WHERE job = 'SALESMAN';
```

view created.

```
SQL> SELECT * FROM salesman;
```

OUTPUT:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----	-----

7499	ALLEN	SALESMAN	7698	20-FEB-81	1700	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1350	500	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1350	1400	30
7844	TURNER	SALESMAN	7698	08-FEB-81	1600	0	30

4 row selected.

The CREATE VIEW also enables you to select individual columns from a table and place them in a view. The following example: selects the ENAME and JOB fields from the EMP table.

INPUT:

```
SQL> CREATE VIEW EMP_INFO (ename, job) AS SELECT ename, job
FROM EMP;
```

view created.

Now, let us check out work....

```
SQL> SELECT * FROM emp_info;
```

OUTPUT:

ENAME

JOB

SMITH

CLERK

ALLEN

SALESMAN



WARD	SALESMAN
JONES	MANAGER
MARTIN	SALESMAN
BLAKE	MANAGER
CLARK	MANAGER
SCOTT	ANALYST
KING	PRESIDENT
TURNER	SALESMAN
ADAMS	CLERK
JAMES	CLERK
FORD	ANALYST
MILLER	CLERK

14 rows selected.

NOTE: Users may create views to query specific data. Say you have a table with 50 columns and hundreds of thousands of rows, but you need to see data in only 2 columns. You can create a view on these two columns, and then by querying from the view, you should see a remarkable difference in the amount of time it takes for your query results to be returned.

Renaming Columns

Views simplify the representation of data. In addition to naming the

view, the SQL syntax for the CREATE VIEW statement enables you to rename selected columns. Like in EMP table you have MGR column for Manager and you want to show user as MANAGER, and also you want to add COMM to SAL and want to display column as TOTAL_AMT then you must use the SQL + operator to combine the fields into one.

INPUT:

```
SQL> CREATE VIEW emp_sal (empno, ename, manager, tot_sal) AS
SELECT empno, ename, mgr, comm+sal FROM emp WHERE job like
'SALESMAN';
```

view created.

INPUT:

```
SQL> SELECT * FROM emp_sal;
```

EMPNO	ENAME	MANAGER	TOT_SAL
-----	-----	-----	-----
7499	ALLEN	7698	2000
7521	WARD	7698	1850
7654	MARTIN	7698	2750
7844	TURNER	7698	1600

ANALYSIS:

The SQL syntax requires you to supply a virtual field name whenever the view's virtual field is created using a calculation or SQL function. This procedure makes sense because you wouldn't want a view's column name to be SUM(*) or AVG(PAYMENT).

SQL View Processing

Views can represent data within tables in a more convenient fashion than what actually exists in the database's table structure. Views can also be extremely convenient when performing several complex queries in a series (such as within a stored procedure or application program).

Restrictions on Using SELECT

SQL places certain restrictions on using the SELECT statement to formulate a view. The following two rules apply when using the SELECT statement:

- You cannot use the UNION operator.
- You cannot use the ORDER BY clause. However, you can use the GROUP BY clause in a view to perform the same functions as the ORDER BY clause.

Modifying Data in a View

As you have learned, by creating a view on one or more physical tables within a database, you can create a virtual table for use throughout an SQL script or a database application. After the view has been created using the CREATE VIEW...SELECT statement, you can update, insert, or delete view data using the UPDATE, INSERT, and DELETE commands you learned about. We discuss the limitations on modifying a view's data in greater detail later. The next group of examples illustrates how to manipulate data that is in a view.

INPUT/OUTPUT:

```
SQL> UPDATE emp_sal SET manager = 9754;
```

4 rows updated.

To check whether UPDATE on VIEW (emp_sal) updates to base table or not, use

```
SQL> SELECT * FROM emp;
```

you will find all the related record to emp_sal view are updated now. It means update on view updates base table.

Problems with Modifying Data Using Views

Because what you see through a view can be some set of a group of tables, modifying the data in the underlying tables is not always as straightforward as the previous examples. Following is a list of the most common restrictions you will encounter while working with views:

- You cannot use DELETE statements on multiple table views.
- You cannot use the INSERT statement unless all NOT NULL columns used in the underlying table are included in the view.
- This restriction applies because the SQL processor does not know which values to insert into the NOT NULL columns.
- If you do insert or update records through a join view, all records that are updated must belong to the same physical table.
- If you use the DISTINCT clause to create a view, you cannot update or insert records within that view.
- You cannot update a virtual column (a column that is the result of an expression or function or multiple column).

Common Applications of Views

- Here are a few of the tasks that views can perform:

- Providing user security functions
- Converting between units
- Creating a new virtual table format
- Simplifying the construction of complex queries
- Views and Security

All relational database systems in use today include a full suite of built-in security features. Users of the database system are generally divided into groups based on their use of the database. Common group types are database administrators, database developers, data entry personnel, and public users. These groups of users have varying degrees of privileges when using the database. The database administrator will probably have complete control of the system, including UPDATE, INSERT, DELETE, and ALTER database privileges. The public group may be granted only SELECT privileges and perhaps may be allowed to SELECT only from certain tables within certain databases. Views are commonly used in this situation to control the information that the database user has access to. For instance, if you wanted users to have access only to the ENAME field of the EMP table, you could simply create a view called EMP_NAME:

INPUT/OUTPUT:

Infotech Pvt. Ltd.

```
SQL> CREATE VIEW EMP_NAME AS SELECT ENAME FROM EMP;
```

View created.

Someone with system administrator-level privileges could grant the public group SELECT privileges on the EMP_NAME view. This group would not have any privileges on the underlying EMP table. As you might guess, SQL has provided data security statements for your use also. Keep in mind that views are very useful for implementing database security.

Using Views to Convert Units

Views are also useful in situations in which you need to present the user with data that is different from the data that actually exists within the database. For instance, if the SAL field is actually stored in Rupees and you don't want Canadian users to have to continually do mental calculations to see the SAL total in Canadian dollars, you could create a simple view called

CANADIAN_SAL:

INPUT/OUTPUT:

```
SQL> CREATE VIEW CANADIAN_SAL (name, can_sal) AS SELECT  
ename, sal/25 FROM emp;
```

view created.

The DROP VIEW Statement

In common with every other SQL CREATE... command, CREATE VIEW has a corresponding DROP... command. The syntax is as follows:

SYNTAX:

```
SQL> DROP VIEW view_name;
```

The only thing to remember when using the DROP VIEW command is that all other views that reference that view are now invalid.

INPUT/OUTPUT:

```
SQL> DROP VIEW emp_sal;
```

View dropped.



Atmiya
— Infotech Pvt. Ltd.

Top:2 Sequence

Sequences are a great way to have the database automatically generate unique integer primary keys. The CREATE SEQUENCE system privilege is needed to execute this command. The DBA is responsible for administering these privileges. The syntax to create a sequence is:

CREATE SEQUENCE schema.name

INCREMENT BY x

START WITH x

MAXVALUE x **NOMAXVALUE**

MINVALUE x **NOMINVALUE**

CYCLE NOCYCLE

CACHE x **NOCACHE**

ORDER NOORDER

In this syntax, **SCHEMA** is an optional parameter that identifies which database schema to place this sequence in. The default is your own.

NAME is mandatory because it is the name of the sequence.

INCREMENT BY is optional. The default is one. Zero is not allowed. If a negative integer is specified, the sequence will descend in order. A positive integer will make the sequence ascend (the default).

START WITH is an optional integer that enables the sequence to begin anywhere.

MAXVALUE is an optional integer that places a limit on the sequence.

NOMAXVALUE is optional. It causes the maximum ascending limit to be 10 27 and -1 for descending sequences. This is the default.

MINVALUE is an optional integer that determines the minimum a sequence can be.

NOMINVALUE is optional. It causes the minimum ascending limit to be 1 and -(10 26) for descending sequences. This is the default.

CYCLE is an option that enables the sequence to continue even when the maximum has been reached. If the maximum is reached, the next sequence that will be generated is whatever the minimum value is.

NOCYCLE is an option that does not enable the sequence to generate values beyond the defined maximum or minimum. This is the default.

CACHE is an option that enables sequence numbers to be preallocated that will be stored in memory for faster access. The minimum value is 2.

NOCACHE is an option that will not enable the preallocation of sequence numbers.

ORDER is an option that ensures the sequence numbers are generated in order of request.

NOORDER is an option that does not ensure that sequence numbers are generated in the order they are requested.

If you want to create a sequence for your deptno column in the DEPT table, it could look like the following example:


```
CREATE SEQUENCE seq_deptno
```

```
INCREMENT BY 10
```

```
START WITH 10;
```

To generate a new sequence number, use the pseudo column **NEXTVAL**. This needs to be preceded with your sequence name. For example, **seq_deptno.nextval** would return 10 for the first access and 20 for the second. If determining the current sequence number is necessary, use **CURRVAL**. Therefore, **seq_deptno.currval** will return the current value of the sequence.

Example:

INPUT:

```
SQL> SELECT seq_deptno.nextval FROM dual;
```

OUTPUT:

```
NEXTVAL
```

```
-----
```

```
10
```

INPUT:

```
SQL> SELECT seq_deptno.currval FROM dual;
```

OUTPUT:

```
CURRVAL
```

```
-----
```

Modifying Sequences

You can modify the sequence when you want to:

- Set or eliminate the minimum or maximum value
- Change the increment value
- Change the number of cached sequence numbers

You can use the following format to modify a sequence:

ALTER SEQUENCE <sequence name>

INCREMENT BY n

[MAXVALUE n] [MINVALUE n]

[CYCLE|NOCYCLE]

[CACHE n|NOCACHE];

The statement below modifies the sequence 'seq_deptno', and sets the maximum value to 990.

ALTER SEQUENCE seq_deptno

INCREMENT BY 10

MAXVALUE 990

MINVALUE 10;

However, if you need to reset a sequence to a starting number higher than the current value of the sequence, you must delete the sequence

and recreate it.

Deleting Sequence

You can delete a sequence just as you delete other database objects. The format of the statement is as follows:

```
DROP SEQUENCE <sequence name>;
```

The statement below deletes the sequence 'seq_deptno' created earlier.

```
DROP SEQUENCE seq_deptno;
```



Top:3 Synonyms

The main reasons for creating synonyms are:

1. The true name of the owner or table needs to be hidden
2. The original location of the table needs to be hidden. This is in case the database is a large one with many installations at different places.
3. Users have to be provided with a simple and easily remembered tablename.

The format of the statement of creating a synonyms is as follows:

**CREATE SYNONYM <synonyms name> FOR
<tablename/viewname>;**

For example if you want to access the 'emp' table created by a user 'Scott', you can create a synonym 'employee' to refer to the 'emp' table created by Scott. To do this you would code the following statement.

```
CREATE SYNONYM employee FOR scott.emp;
```

you can see all the database objects (including synonyms) that you have created, give the following command;

```
SELECT * FROM tab;
```

Similarly, synonyms can also be deleted, if no longer required by using

the DROP command. The format of the statement is as follows;

DROP SYNONYM <synonyms name>;

The following command drops the synonym 'employee' created earlier.

DROP SYNONYM employee;



Top:4 Index

One of the physical storage structure that are provided by most SQL based RDBMS systems is an Index. An index is a structure that provides rapid access to the rows of a table based on the values of one or more columns.

A database index is like an index to a book. If you want to lookup every reference to a work that appears in a book, you would turn to the alphabetically arranged index at the back of the book and look up the term along with the page numbers on which the command is referred to. If however, there were no index, or if your chosen term was not indexed, you would have to scan the entire book to find the piece of information you want.

In the same way, a database index is a listing of keywords accompanied by the location of information on a subject.

While indexes are not strictly necessary to running a RDBMS, they do speed up the process of data retrieval. Indexing enables an RDBMS to locate rows in a large table more quickly than it otherwise would.

In several situations, indexes can dramatically improve database performance. Two of these situations are: primary keys and foreign keys in joined tables.

Primary keys

Perhaps the most important index is on the table's primary key. Because tables should have identified, unique primary keys, each row of any given table can be identified uniquely. Thus, indexing the

primary key using the UNIQUE option guarantees that the key is unique within a table. For example, the primary key in the 'client_master' table is the key 'client_no'. 'client_no' is the client's identifier and is unique for each row of the table. The other columns, such as 'name', 'address' are not unique and cannot be used to select a single client.

Foreign Keys in Joined tables

Performance can also be improved by indexing the join columns of tables. For example, the 'client_master' table and 'sales_order' table are joined on the common column (primary key/ foreign key) 'client_no'. The 'sales_order' table and 'sales_order_details' are joined on the common column 's_order_no'. You will be frequently joining pairs of these tables with SELECT statements. Indexing the 'sales_order' table on the 's_order_no' and 'client_no' columns separately will, speed the join operations.

The real power of indexing is apparent only when the table is large. It will do little to speed up search of a smaller table. However, in the case of a very small table, indexing might even slow you down. The computer must first search the index file for the address of the term you seek, then find that address in the main table. As your tables grow, the time needed for this two step search will become much less than the time needed to scan the entire table without the benefit of an index.

The advantage of having an index is that it greatly speeds the execution of SQL statements with search conditions that refer to the indexed column(s). One disadvantage of having an index is that it consumes additional disk space. Another disadvantage is that the index must be updated every time a row is added to the table and every time the indexed column is updated in an existing row. This imposes additional overheads on INSERT and UPDATE statements for the table.

Oracle automatically determines whether or not to use existing

indexes to boost performance. The way you write SELECT statements can affect that determination. Indexes are used to precess a SELECT statement only if a WHERE clause is part of the SELECT statement and only if the predicates reference one or more indexed columns.

Therefore, in general it is a good idea to create an index for columns that are used frequently in search conditions. Indexing is also more appropriate when queries against a table are more frequent than inserts and updates.

Guidelines for INDEX usage

The following are some guidelines for creating and using indexes in your database:

- If a table has more than a few hundred rows, index it: Indexes are most useful on larger tables. The larger a table, the more an index will improve response time. Additionally, the more indexes available, the better are the RDBMS's chances of finding a "short cut" to the requested information.
- Try not to create more than two or three indexes on a table. Creating too many indexes on a table is not a good idea. This is because first, indexes take up disk storage space; second, although indexes speed up queries, they may slow down data manipulation operations. This is because when you insert or delete a row in an indexed table, or change the value of an indexed column, all the indexes have to be updated.
- Index frequently used columns: Index the columns you use most often in the WHERE clauses. There is no need to index the columns you use only to display values. Moreover, indexes are most useful on columns with a significant amount of variety in their data.

Creating an index on object and relational tables

The indexes can also be defined on the attributes of a type. Let us see how indexes can be created on object and relational tables.

On object tables

An index is created on an object table in the same way as it is created for a normal table. The format of the statement is as follows:

**CREATE [UNIQUE] INDEX <indexname> ON
<objectname> (attributename [ASC|DESC]);**

The optional keyword UNIQUE ensures that the values in the indexed columns for all the rows will be distinct. Indexes can also be created on more than one column. Using the optional keyword ASC or DESC created an index on the column in the specified sort order (i.e. Ascending or Descending).

The statement below creates an index 'idx_client' on the object table 'client_master' for the attribute 'client_no'

```
CREATE INDEX idx_client ON client_master(client_no);
```

On relational tables

To index a relational table on the attributes of a type the following format is used.

**CREATE [UNIQUE] INDEX <indexname> ON
<tablename> (columnname [ASC|DESC]);**

The following statement creates an index 'idx_product' on the table 'product_master' on the column 'product_no' in descending way.

```
CREATE INDEX idx_product ON product_master(product_no DESC);
```

Validating an Index

You can check a specified index to determine whether or not it is accurate and consistent. The format of the statement is as follows.

```
VALIDATE INDEX <indexname> [ON <tablename> ];
```

If the index is valid, oracle displays a message "Index analyzed.". If you receive any message other than this, it means that the index is corrupt and should be discarded. You can delete and recreate the index in this case.

For example, to check if the index 'idx_product' is valid, you can code the following statement.

```
VALIDATE INDEX idx_product;
```

Deleting an Index

To remove an index from the database, the format is as follows:

```
DROP INDEX <indexname> [ON <tablename> ];
```

Only the creator of an index can drop the index. The ON <tablename> is optional and need not be specified. The statement below removes the index 'idx_product' created earlier.

```
DROP INDEX idx_product;
```



Top:5 Database Links

As your database grow in size and number, you will very likely need to share data among them. Sharing data requires a method of locating and accessing the data. In Oracle, remote data accesses such as queries and updates are enabled through the use of database links. As described in this chapter, database links allow users to treat a group of distributed databases as if they were a single, integrated database. In this chapter, you will also find information about direct connections to remote databases, such as those used in client-server applications.

Database Links

Database links tell oracle how to get from one database to another. If you will frequently use the same connection to a remote database, then a database link is appropriate.

How a Database Link Works

A database link requires that Net8 (the current version of the SQL*Net product) be running on each of the machines (hosts) involved in the remote database access. Net8 is usually started by the database administrator (DBA) or the system manager. A sample architecture for a remote access using a database link is shown in Figure 1. This figure shows two hosts, each running Net8. There is a database on each of the hosts. A database link establishes a connection from the first database (named LOCAL, on the Branch host) to the second database (named REMOTE, on the Headquarters host). The database link shown in Figure 1 is software that is located in the local database.

Database links specify the following connection information:

- The communications protocol (such as TCP/IP) to use during the connection
- The host on which the remote database resides
- The name of the database on the remote host
- The name of a valid account in the remote database
- The password for that account

When used, a database link actually logs in as a user in the remote database, and then logs out when the remote data access is complete. A database link can be private, owned by a single user, or public, in which case all users in the local database can use the link.



Syntax for database links

You can create a database link with the following command:

```
create [public] database link REMOTE_CONNECT connect to  
[current_user | username identified by password] using 'connect  
string';
```

The specific syntax to use when creating a database link depends upon two criteria:

- The "public" or "private" status of the database link
- The use of default or explicit logins for the remote database

Using a Database Link for Remote Queries

If you are a user in the local database shown in figure 1, you can access objects in the Remote database via a database link. To do this, simply append the database link name to the name of any table or view that is accessible to the remote account. When appending the database link name to a table or view name, you must precede the database link name with an @ sign.

For local tables, you reference the table name in the from clause;


```
select * from WORKER;
```

For remote tables, use a database link named REMOTE_CONNECT. In the from clause, reference the table name followed by @REMOTE_CONNECT:

```
select * from WORKER@REMOTE_CONNECT;
```

When the database link in the preceding query is used, Oracle will log in to the database specified by the database link, using the username and password provided by the link. It will then query the WORKER table in that account and return the data to the user who initiated the query. This is shown graphically in figure 2. The REMOTE_CONNECT database link shown in figure 2 is located in the local database.


Infotech Pvt. Ltd.



As shown in figure 2, logging into the Local database and using the `REMOTE_CONNECT` database link in your from clause returns the same results as logging in directly to the remote database and executing the query without the database link. It makes the remote database seem local.

NOTE: The maximum number of database links that can be used in a single query is set via the `OPEN_LINKS` parameter in the database's `init.ora` initialization file. This parameter usually defaults to four.

There are restrictions to the queries that are executed using database links. You should avoid using database links in queries that use the `connect by`, `start with`, and `prior` keywords. Some queries using these keywords will work (for example, if `prior` is not used outside of the `connect by` clause, and `start with` does not use a subquery), but most uses of tree-structured queries will fail when using database links.

Using a Database Link for Synonyms and Views

You may create local synonyms and views that reference remote objects. To do this reference the database link name, preceded by an `@` sign, wherever you refer to a remote table. The following example shows how to do this for synonyms. The create synonym command in this example is executed from an account in the local database.

```
create synonym WORKER_SYN for WORKER@REMOTE_CONNECT;
```

In this example, a synonym called `WORKER_SYN` is created for the `WORKER` table accessed via the `REMOTE_CONNECT` database link. Every time this synonym is used in a `from` clause of a query, the remote database will be queried. This is very similar to the remote queries shown earlier; the only real change is that the database link is now defined as part of a local object (in this case a synonym).

What if the remote account that is accessed by the database link does not own the table being referenced? In that event, any synonym that exist for a table that the remote account has been granted access to, then you must specify the table owner's name in the query, as shown in the following example:

```
create synonym WORKERSKILL_SYN from Talbot.  
WORKERSKILL@REMOTE_CONNECT;
```

In this example, the remote account used by the database link does not own the `WORKERSKILL` table, nor does the remote account have a synonym called `WORKERSKILL`. It does, however, have privileges on the `WORKERSKILL` table owned by the remote user `Talbot` in the Remote database. Therefore, the owner and table name are specified; both are interpreted in the remote database. The syntax for these queries and synonyms is almost the same as if everything were in the local database; the only addition is the database link name.

To use a database link in a view, simply add it as a suffix to table names in the `create view` command. The following example creates a view in the local database of a remote table using the `REMOTE_CONNECT` database link: *Infotech Pvt. Ltd.*

```
create view LOCAL_EMPLOYEE_VIEW as select * from  
WORKER@REMOTE_CONNECT where lodging = 'ROSE HILL';
```

The `from` clause in this example refers to `WORKER@REMOTE_CONNECT`. Therefore, the base table for this view is not in the same database as the view. Also note that a `where` clause

is placed on the query, to limit the number of records returned by it for the view.

This view may now be treated the same as any other view in the local database. Access to this view can be granted to other users, provided those users also have access to the REMOTE_CONNECT database link.



Top:1 What is Transaction?

You have spent time learning virtually everything that you can do with data within a relational database. For example, you know how to use the SQL SELECT statement to retrieve data from one or more tables based on a number of conditions supplied by the user. You have also had a chance to use data modification statements such as INSERT, UPDATE, and DELETE. As of today, you have become an intermediate-level SQL and database user. If required, you could build a database with its associated tables, each of which would contain several fields of different data types. Using proper design techniques, you could leverage the information contained within this database into a powerful application.

Objectives

If you intend to (or are currently required to) develop a professional application using any type of relational database, the advanced topics like transaction control, security, embedded SQL programming, and database procedures--will help you a great deal. We begin with transaction control. By the end of the couple of topics, you will know the following:

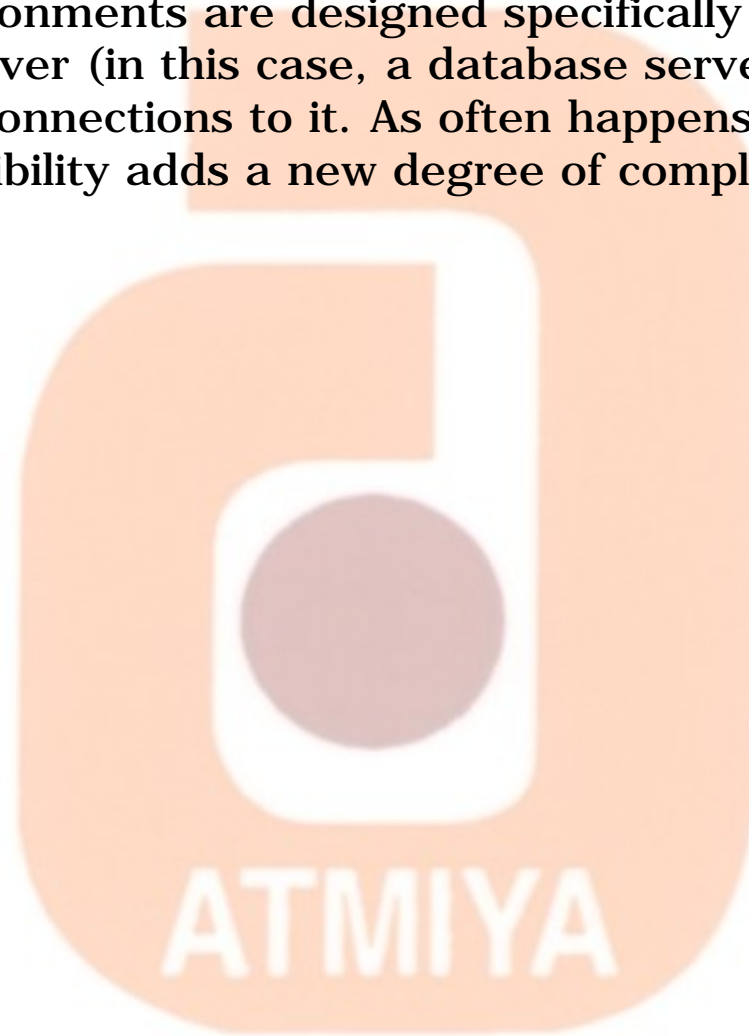
- The basics of transaction control
- How to finalize and or cancel a transaction

Transaction Control

Transaction control, or transaction management, refers to the capability of a relational database management system to perform

database transactions. Transactions are units of work that must be done in a logical order and successfully as a group or not at all. The term unit of work means that a transaction has a beginning and an end. If anything goes wrong during the transaction, the entire unit of work can be canceled if desired. If everything looks good, the entire unit of work can be saved to the database.

Client/server environments are designed specifically for this purpose. Traditionally, a server (in this case, a database server) supports multiple network connections to it. As often happens with technology, this newfound flexibility adds a new degree of complexity to the environment.



Top:2 Starting and Ending of Transaction?

Let us consider one case of manufacturing organization. Say company have departments as store, sales, marketing and administration. One database is being maintains by these all departments. Tables for these are :

Client_master, Product_master, Sales_order, Sales_order_details, Salesman_master, Challan_Header, Challan_Details.

As being employee of Sales department you are doing transaction on sales bill and data are being send to table sales_order, sales_order_Details as well you are retrieving data from product_master table for sales_rate of product and client_master table for information of clients. In such case you will not allow any other user to disturb your transaction. And this is the problem, how to work efficiently on single database by many user on network on an organization. For this reason one has to develop transaction area so that no other user disturb your work.

Let us first check tables available to us, [CLICK ME](#).

On say you make on sales bill with product information out from product_master for sell_price of product. Parallel some user change product_master table and even sell_price of the same product you are accessing as:

Infotech Pvt. Ltd.

```
SQL> UPDATE product_master SET sell_price= 1200 WHERE
description= 'Mouse';
```

As you can see, the information you retrieved earlier could be invalid if

the update occurred during the middle of your SELECT. If your application fired off a letter to be sent to Mr. user, the sell_price it used would be wrong. Obviously, if the letter has already been sent, you won't be able to change the sell_price. However, if you had used a transaction, this data change could have been detected, and all your other operations could have been rolled back.

Beginning a Transaction

Transactions are quite simple to implement. You will examine the syntax used to perform transactions using the Oracle RDBMS SQL syntax. All database systems that support transactions must have a way to explicitly tell the system that a transaction is beginning. (Remember that a transaction is a logical grouping of work that has a beginning and an end.) Using Personal Oracle8, the syntax looks like this:

SYNTAX:

```
SET TRANSACTION { READ ONLY | USE ROLLBACK SEGMENT  
segment }
```

The SQL standard specifies that each database's SQL implementation must support statement-level read consistency; that is, data must stay consistent while one statement is executing. However, in many situations data must remain valid across a single unit of work, not just within a single statement. Oracle enables the user to specify when the transaction will begin by using the SET TRANSACTION statement. If you wanted to examine other user's information and make sure that the data was not changed, you could do the following:

INPUT:

```
SQL> SET TRANSACTION READ ONLY;
```

```
SQL> SELECT * FROM product_master WHERE description = 'Mouse';
```

---Do Other Operations---

SQL> COMMIT;

We discuss the COMMIT statement in next topic. The SET TRANSACTION READ ONLY option enables you to effectively lock a set of records until the transaction ends. You can use the READ ONLY option with the following commands:

SELECT

LOCK TABLE

SET ROLE

ALTER SESSION

ALTER SYSTEM

The option USE ROLLBACK SEGMENT tells Oracle which database segment to use for rollback storage space. SQL Server's Transact-SQL language implements the BEGIN TRANSACTION command with the following syntax:

SYNTAX:

begin { transaction | tran } [transaction_name]

Finishing a Transaction

The Oracle syntax to end a transaction is as follows:

SYNTAX:

COMMIT [WORK];

The COMMIT command saves all changes made during a transaction.

Executing a COMMIT statement before beginning a transaction ensures that no errors were made and no previous transactions are left hanging.



Atmiya
— Infotech Pvt. Ltd.

Top:3 Commit, Rollback, Save Point (TCL - Transaction Control Language)

1. COMMIT

A COMMIT statement is used to end a transaction. It makes any changes made to the database permanent. This statement also erases any savepoints in the transaction. The format of the statement is as follows:

COMMIT [WORK];

A COMMIT statement indicates the following:

All SQL statements have been executed

The successful end of a transaction

The database is in a consistent state

For example, after entering the details of a customer, the user wants to make the change in the database permanent, one can code the following:

```
SQL> INSERT INTO client_master(client_no, name, bal_due) VALUES ('C00008', 'Jignesh', 0);
```

1 row inserted.

```
SQL> COMMIT;
```

commit complete.

2. ROLLBACK

A ROLLBACK statement is used to undo the work done in the current transaction. You are either rollback the entire transaction so that all changes made by the SQL statement are ignored or rollback a transaction to a savepoint so that SQL statement after the savepoint are ignored. The format of the statement is as follows.

ROLLBACK [WORK] [TO [SAVEPOINT] savepoint];

To rollback to a particular stage in a transaction (a savepoint) the format is as follows:

ROLLBACK TO SAVEPOINT A;

The above statement rolls back the current transaction to the specified savepoint A. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

A ROLLBACK statement does the following:

- Discard changes made to the database
- Indicates the unsuccessful end of the transaction
- Restores the database to the state before the transaction

For example, if after entering the details of a client, the user realizes that the person does not want to be a client, user can rollback the transaction that just made as follows:

```
SQL> INSERT INTO client_master(client_no, name, bal_due) VALUES  
( 'C00008', 'Jignesh', 0);
```


1 row inserted.

```
SQL> ROLLBACK;
```

3. SAVEPOINT

A SAVEPOINT is like a marker to divide a lengthy transaction into smaller ones. It is used to identify a point in a transaction to which we can later rollback. It is used in conjunction with rollback, to rollback portions of the current transaction. The format of the statement is as follows:

SAVEPOINT <savepoint name>;

For example, what could happen in the earlier case is, if the user had already entered two records and after entering the third, realizes that the person does not want to become a client, then the rollback statement would rollback the entire transaction the he just made, including the first two records that are actually required. To overcome this problem he can use savepoint as follows:

```
SQL> INSERT INTO client_master(client_no, name, bal_due) VALUES ('C00008', 'Jignesh', 0);
```

1 row inserted.

```
SQL> SAVEPOINT client1;
```

```
SQL> INSERT INTO client_master(client_no, name, bal_due) VALUES ('C00008', 'Namrata', 0);
```

1 row inserted.

```
SQL> SAVEPOINT client2;
```

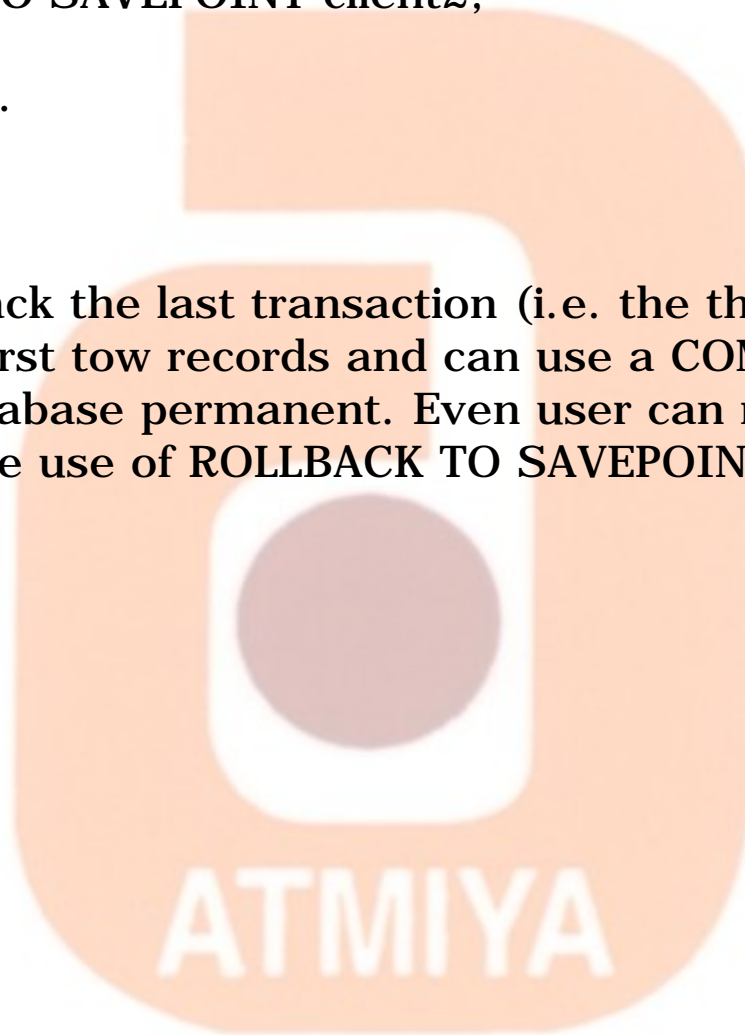
```
SQL> INSERT INTO client_master(client_no, name, bal_due) VALUES  
( 'C00008', 'Amit', 0);
```

1 row inserted.

```
SQL> ROLLBACK TO SAVEPOINT client2;
```

rollback completed.

This will only rollback the last transaction (i.e. the third record.). User will still have the first tow records and can use a COMMIT to make the changes to the database permanent. Even user can rollback two transaction with the use of ROLLBACK TO SAVEPOINT client1.



Top:4 Grant, Revoke (DCL - Data Control Language)

Anything as important as business data has to be well protected. SQL provides protection of data by allocating rights and privileges to authorized users. Their login identification and passwords identify authorized users. Users can be allocated rights and privileges on various database objects. They can have rights like:

- Rights of selecting data (SELECT)
- Rights of adding data (INSERT)
- Rights of updating data (UPDATE)
- Rights of deleting data (DELETE)

These privileges, once granted to a user, can also be revoked. Certain privileges will grant utilities when given to another user and also make him eligible to pass on these privileges to other users. Depending upon the nature of work assigned to a user they can be given the required privileges and rights. These commands come in the category of Data Control Language (DCL). Let us discuss two of these commands.

1. GRANT Privilege Command

Infotech Pvt. Ltd.

The creator of a table automatically becomes the owner of the table. For example, after you create the client_master table, you become the owner of the 'client_master' table. There is no need to be granted any privilege to use the 'client_master' table. In case, you want to share a

database objects, that you have created, with others, the appropriate privileges can be granted on that particular database object to the others. Objects privileges can be granted, to others, using the SQL command GRANT. The format of the statement is as follows:

GRANT <privilege(s)> ON <objectname> TO <username> [WITH GRANT OPTION];

The 'WITH GRANT OPTION' allows that user to pass on the privileges on that database objects to other users. By default, users who have been granted a privilege on a database object, can only use the privilege themselves; they cannot grant any privileges on that database object to other users.

For example, to grant a user 'Scott' the privilege of select and insert on your 'client_master' table, you would code the following statement.

GRANT SELECT, INSERT ON client_master TO Scott;

This will allow Scott to view the data in your 'client_master' table and also insert data into your 'client_master' table. But Scott will not be able to update or delete any rows from your 'client_master' table, as he has not been granted those privileges.

Now if you want to grant Scott all the privileges of select, insert, update and delete, you would have to code the following statement.

GRANT SELECT, INSERT, UPDATE, DELETE ON client_master TO Scott;

Or instead of typing out all the privileges (SELECT, INSERT, UPDATE and DELETE), since you want to grant Scott all the privileges you can code the following statement.

GRANT ALL ON client_master TO Scott;

This will grant all the privileges to Scott. But Scott will not be able to grant any privileges on your 'client_master' table to any of the other

users.

If you want Scott to be able to grant some or all of the privileges on your 'client_master' table to another user, code the following:

Scott will now be able to grant any of the privileges on your 'client_master' table to any other user that he wants to.

2. REVOKE Privilege command

To withdraw the privilege(s) which have been granted to a user, you can use the REVOKE command. The format, similar to that of the GRANT command, is as follows:

REVOKE <privilege(s)> ON <objectname> FROM <username>;

To revoke only the update and delete privileges on the 'client_master' table, that you have given Scott, code the following:

REVOKE UPDATE, DELETE ON client_master FROM Scott;

To revoke all the privileges on the 'client_master' table, that you have given Scott, code the following:

REVOKE ALL ON client_master FROM Scott;



Top:5 Role and Creating Users, Change Password

Database Security

Today we discuss database security. We specifically look at various SQL statements and constructs that enable you to administer and effectively manage a relational database. We will discuss the following:

- Create users
- Change passwords
- Create roles

Wanted: Database Administrator

Security is an often-overlooked aspect of database design. Most computer professionals enter the computer world with some knowledge of computer programming or hardware, and they tend to concentrate on those areas. For instance, if your boss asked you to work on a brand-new project that obviously required some type of relational database design, what would be your first step? After choosing some type of hardware and software baseline, you would probably begin by designing the basic database for the project. This phase would gradually be split up among several people--one of them a graphical user interface designer, another a low-level component builder. Perhaps you, after reading this book, might be asked to code the SQL queries to provide the guts of the application. Along with this task comes the responsibility of actually administering and maintaining the database.

Many times, little thought or planning goes into the actual production phase of the application. What happens when many users are allowed

to use the application across a wide area network (WAN)? With today's powerful personal computer software and with technologies such as Microsoft's Open Database Connectivity (ODBC), any user with access to your network can find a way to get at your database. (We won't even bring up the complexities involved when your company decides to hook your LAN to the Internet or some other wide-ranging computer network!) Are you prepared to face this situation?

Fortunately for you, software manufacturers provide most of the tools you need to handle this security problem. Every new release of a network operating system faces more stringent security requirements than its predecessors. In addition, most major database vendors build some degree of security into their products, which exists independently of your operating system or network security. Implementation of these security features varies widely from product to product. Oracle8 relational database management system supports nearly the full SQL standard. In addition, Oracle has added its own extension to SQL, called PL*SQL. It contains full security features, including the capability to create roles and assign permissions and privileges on objects in the database.

How Does a Database Become Secure?

Has it occurred to you that you might not want other users to come in and tamper with the database information you have so carefully entered? What would your reaction be if you logged on to the server one morning and discovered that the database you had slaved over had been dropped (remember how silent the DROP DATABASE command is)? We examine in some detail how one popular database management system (Personal Oracle8) enables you to set up a secure database. Keep the following questions in mind as you plan your security system:

Who gets the DBA role?

How many users will need access to the database?

Which users will need which privileges and which roles?

How will you remove users who no longer need access to the database?

Personal Oracle8 implements security by using three constructs:

- Users
- Roles
- Privileges

Creating Users

Users are account names that are allowed to log on to the Oracle database. The SQL syntax used to create a new user follows.

SYNTAX:

CREATE USER user IDENTIFIED { BY password | EXTERNALLY }

[DEFAULT TABLESPACE tablespace]

[TEMPORARY TABLESPACE tablespace]

[QUOTA { integer [K|M] | UNLIMITED } ON tablespace]

[PROFILE profile]

If the BY password option is chosen, the system prompts the user to enter a password each time he or she logs on. As an example, create a username for yourself:

INPUT/OUTPUT:

```
SQL> CREATE USER jignesh IDENTIFIED BY dhol;
```

User created.

Each time I log on with my username jignesh, I am prompted to enter my password: dhol.

If the EXTERNALLY option is chosen, Oracle relies on your computer system login name and password. When you log on to your system, you have essentially logged on to Oracle. As you can see from looking at the rest of the CREATE USER syntax, Oracle also allows you to set up default tablespaces and quotas. You can learn more about these topics by examining the Oracle documentation. As with every other CREATE command you have learned about in this book, there is also an ALTER USER command. It looks like this:

SYNTAX:

```
ALTER USER user [IDENTIFIED { BY password | EXTERNALLY} ]
```

```
[DEFAULT TABLESPACE tablespace]
```

```
[TEMPORARY TABLESPACE tablespace]
```

```
[QUOTA {integer [K|M] | UNLIMITED} ON tablespace]
```

```
[PROFILE profile]
```

```
[DEFAULT ROLE { role [, role] ..| ALL [EXCEPT role [, role] ...] |  
NONE} ]
```

You can use this command to change all the user's options, including the password and profile. For example, to change the user jignesh's password, you type this:

INPUT/OUTPUT:

```
SQL> ALTER USER jignesh IDENTIFIED BY jd;
```

User altered.

To change the default tablespace, type this:

INPUT/OUTPUT:

```
SQL> ALTER USER jignesh DEFAULT TABLESPACE USERS;
```

User altered.

To remove a user, simply issue the DROP USER command, which removes the user's entry in the system database. Here's the syntax for this command:

SYNTAX:

```
DROP USER user_name [CASCADE];
```

If the CASCADE option is used, all objects owned by username are dropped along with the user's account. If CASCADE is not used and the user denoted by user_name still owns objects, that user is not dropped. This feature is somewhat confusing, but it is useful if you ever want to drop users.



Creating Roles

A role is a privilege or set of privileges that allows a user to perform certain functions in the database. To grant a role to a user, use the following syntax:

SYNTAX:

GRANT role TO user [WITH ADMIN OPTION];

If WITH ADMIN OPTION is used, that user can then grant roles to other users. Isn't power exhilarating?

To remove a role, use the REVOKE command:

SYNTAX:

REVOKE role FROM user;

When you log on to the system using the account you created earlier, you have exhausted the limits of your permissions. You can log on, but that is about all you can do. Oracle lets you register as one of three roles:

Connect

Resource

DBA (or database administrator)

These three roles have varying degrees of privileges. If you have the appropriate privileges, you can create your own role, grant privileges to your role, and then grant your role to a user for further security.

The Connect Role

The Connect role can be thought of as the entry-level role. A user who has been granted Connect role access can be granted various privileges that allow him or her to do something with a database.

INPUT/OUTPUT:

SQL> GRANT CONNECT TO jignesh;

Grant succeeded.

The Connect role enables the user to select, insert, update, and delete records from tables belonging to other users (after the appropriate permissions have been granted). The user can also create tables, views, sequences, clusters, and synonyms.

The Resource Role

The Resource role gives the user more access to Oracle databases. In addition to the permissions that can be granted to the Connect role, Resource roles can also be granted permission to create procedures, triggers, and indexes.

INPUT/OUTPUT:

```
SQL> GRANT RESOURCE TO jignesh;
```

Grant succeeded.

The DBA Role

The DBA role includes all privileges. Users with this role are able to do essentially anything they want to the database system. You should keep the number of users with this role to a minimum to ensure system integrity.

INPUT/OUTPUT:

```
SQL> GRANT DBA TO jignesh;
```

Grant succeeded.

After the three preceding steps, user jignesh was granted the Connect, Resource, and DBA roles. This is somewhat redundant because the DBA role encompasses the other two roles, so you can

drop them now:

INPUT/OUTPUT:

```
SQL> REVOKE CONNECT FROM jignesh;
```

Revoke succeeded.

```
SQL> REVOKE RESOURCE FROM jignesh;
```

Revoke succeeded.

jignesh can do everything he needs to do with the DBA role.

User Privileges

After you decide which roles to grant your users, your next step is deciding which permissions these users will have on database objects. (Oracle8 calls these permissions privileges.) The types of privileges vary, depending on what role you have been granted. If you actually create an object, you can grant privileges on that object to other users as long as their role permits access to that privilege. Oracle defines two types of privileges that can be granted to users: system privileges and object privileges. (See Tables two tables given below.)

System privileges apply systemwide. The syntax used to grant a system privilege is as follows:

SYNTAX:

```
GRANT system_privilege TO {user_name | role | PUBLIC} [WITH  
ADMIN OPTION];
```

WITH ADMIN OPTION enables the grantee to grant this privilege to someone else.

User Access to Views

The following command permits all users of the system to have CREATE VIEW access within their own schema.

INPUT:

SQL> GRANT CREATE VIEW TO PUBLIC;

OUTPUT:

Grant succeeded.

ANALYSIS:

The public keyword means that everyone has CREATE VIEW privileges. Obviously, these system privileges enable the grantee to have a lot of access to nearly all the system settings. System privileges should be granted only to special users or to users who have a need to use these privileges. Table 1 shows the system privileges you will find in the help files included with Personal Oracle8.

Table 1. System privileges in Oracle8.

System Privilege	Operations Permitted
ALTER ANY INDEX	Allows the grantees to alter any index in any schema
ALTER ANY PROCEDURE	Allows the grantees to alter any stored procedure, function, or package in any schema.
ALTER ANY ROLE	Allows the grantees to alter any role in the database.

ALTER ANY TABLE	Allows the grantees to alter any table or view in the schema.
ALTER ANY TRIGGER	Allows the grantees to enable, disable, or compile any database trigger in any schema.
ALTER DATABASE	Allows the grantees to alter the database.
ALTER USER	Allows the grantees to alter any user. This privilege authorizes the grantee to change another user's password or authentication method, assign quotas on any tablespace, set default and temporary tablespaces, and assign a profile and default roles.
CREATE ANY INDEX	Allows the grantees to create an index on any table in any schema.
CREATE ANY PROCEDURE	Allows the grantees to create stored procedures, functions, and packages in any schema.
CREATE ANY TABLE	Allows the grantees to create tables in any schema. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
CREATE ANY TRIGGER	Allows the grantees to create a database trigger in any schema associated with a table in any schema.
CREATE ANY VIEW	Allows the grantees to create views in any schema.

CREATE PROCEDURE	Allows the grantees to create stored procedures, functions, and packages in their own schema.
CREATE PROFILE	Allows the grantees to create profiles.
CREATE ROLE	Allows the grantees to create roles.
CREATE SYNONYM	Allows the grantees to create synonyms in their own schemas.
CREATE TABLE	Allows the grantees to create tables in their own schemas. To create a table, the grantees must also have space quota on the tablespace to contain the table.
CREATE TRIGGER	Allows the grantees to create a database trigger in their own schemas.
CREATE USER	Allows the grantees to create users. This privilege also allows the creator to assign quotas on any tablespace, set default and temporary tablespaces, and assign a profile as part of a CREATE USER statement.
CREATE VIEW	Allows the grantees to create views in their own schemas.
DELETE ANY TABLE	Allows the grantees to delete rows from tables or views in any schema or truncate tables in any schema.
DROP ANY INDEX	Allows the grantees to drop indexes in any schema.

DROP ANY PROCEDURE	Allows the grantees to drop stored procedures, functions, or packages in any schema.
DROP ANY ROLE	Allows the grantees to drop roles.
DROP ANY SYNONYM	Allows the grantees to drop private synonyms in any schema.
DROP ANY TABLE	Allows the grantees to drop tables in any schema.
DROP ANY TRIGGER	Allows the grantees to drop database triggers in any schema.
DROP ANY VIEW	Allows the grantees to drop views in any schema.
DROP USER	Allows the grantees to drop users.
EXECUTE ANY PROCEDURE	Allows the grantees to execute procedures or functions (standalone or packaged) or reference public package variables in any schema.
GRANT ANY PRIVILEGE	Allows the grantees to grant any system privilege.
GRANT ANY ROLE	Allows the grantees to grant any role in the database.
INSERT ANY TABLE	Allows the grantees to insert rows into tables and views in any schema.

LOCK ANY TABLE	Allows the grantees to lock tables and views in any schema.
SELECT ANY SEQUENCE	Allows the grantees to reference sequences in any schema.
SELECT ANY TABLE	Allows the grantees to query tables, views, or snapshots in any schema.
UPDATE ANY ROWS	Allows the grantees to update rows in tables.

Object privileges are privileges that can be used against specific database objects. Table 2 lists the object privileges in Oracle8.

Table 2. Object privileges enabled under Oracle8.

ALL

ALTER

DELETE

EXECUTE

INDEX

INSERT

REFERENCES

SELECT

UPDATE

You can use the following form of the GRANT statement to give other

users access to your tables:

SYNTAX:

```
GRANT {object_priv | ALL [PRIVILEGES]} [ (column [, column]...) ] [,
{object_priv | ALL [PRIVILEGES]} [ (column [, column] ...) ] ] ... ON
[schema.]object TO {user | role | PUBLIC} [, {user | role | PUBLIC} ]
... [WITH GRANT OPTION]
```

To remove the object privileges you have granted to someone, use the REVOKE command with the following syntax.

SYNTAX:

```
REVOKE {object_priv | ALL [PRIVILEGES]} [, {object_priv | ALL
[PRIVILEGES]} ] ON [schema.]object FROM {user | role | PUBLIC} [,
{user | role | PUBLIC} ] [CASCADE CONSTRAINTS]
```

From Creating a Table to Granting Roles Create a table named SALARIES with the following structure:

INPUT/OUTPUT:

```
SQL> CREATE TABLE SALARIES ( NAME CHAR(30), SALARY NUMBER,
AGE NUMBER);
```

Table created.

Now, create two users--Jack and Jill:

INPUT/OUTPUT:

```
SQL> create user Jack identified by Jack;
```

User created.

```
SQL> create user Jill identified by Jill;
```

User created.

SQL> **grant connect to Jack;**

Grant succeeded.

SQL> **grant resource to Jill;**

Grant succeeded.

ANALYSIS:

So far, you have created two users and granted each a different role. Therefore, they will have different capabilities when working with the database. First create the SALARIES table with the following information:

INPUT/OUTPUT:

SQL> SELECT * FROM SALARIES;

NAME	SALARY	AGE
-----	-----	-----
JACK	35000	29
JILL	48000	42
JOHN	61000	55

You could then grant various privileges to this table based on some arbitrary reasons for this example. We are assuming that you currently have DBA privileges and can grant any system privilege. Even if you do not have DBA privileges, you can still grant object

privileges on the SALARIES table because you own it (assuming you just created it). Because Jack belongs only to the Connect role, you want him to have only SELECT privileges.

INPUT/OUTPUT:

```
SQL> GRANT SELECT ON SALARIES TO JACK;
```

Grant succeeded.

Because Jill belongs to the Resource role, you allow her to select and insert some data into the table. To liven things up a bit, allow Jill to update values only in the SALARY field of the SALARIES table.

INPUT/OUTPUT:

```
SQL> GRANT SELECT, UPDATE(SALARY) ON SALARIES TO Jill;
```

Grant succeeded.

Now that this table and these users have been created, you need to look at how a user accesses a table that was created by another user. Both Jack and Jill have been granted SELECT access on the SALARIES table. However, if Jack tries to access the SALARIES table, he will be told that it does not exist because Oracle requires the username or schema that owns the table to precede the table name.

Qualifying a Table

Make a note of the username you used to create the SALARIES table (mine was Bryan). For Jack to select data out of the SALARIES table, he must address the SALARIES table with that username.

INPUT:

```
SQL> SELECT * FROM SALARIES;
```

SELECT * FROM SALARIES

*

OUTPUT:

ERROR at line 1:

ORA-00942: table or view does not exist

Here Jack was warned that the table did not exist. Now use the owner's username to identify the table:

INPUT/OUTPUT:

SQL> SELECT * FROM jignesh.SALARIES;

NAME	SALARY	AGE
-----	-----	-----
JACK	35000	29
JILL	48000	42
JOHN	61000	55

ANALYSIS:

You can see that now the query worked. Now test out Jill's access privileges. First log out of Jack's logon and log on again as Jill (using the password Jill).

INPUT/OUTPUT:

SQL> SELECT * FROM jignesh.SALARIES;

NAME	SALARY	AGE
-----	-----	-----
JACK	35000	29
JILL	48000	42
JOHN	61000	55

That worked just fine. Now try to insert a new record into the table.

INPUT/OUTPUT:

```
SQL> INSERT INTO jignesh.SALARIES VALUES('JOE',85000,38);
```

```
INSERT INTO jignesh.SALARIES
```

*

ERROR at line 1:

ORA-01031: insufficient privileges

ANALYSIS:

This operation did not work because Jill does not have INSERT privileges on the SALARIES table.

INPUT/OUTPUT:

```
SQL> UPDATE jignesh.SALARIES SET AGE = 42 WHERE NAME = 'JOHN';
```

```
UPDATE jignesh.SALARIES
```

*

ERROR at line 1:

ORA-01031: insufficient privileges

ANALYSIS:

Once again, Jill tried to go around the privileges that she had been given. Naturally, Oracle caught this error and corrected his quickly.

INPUT/OUTPUT:

```
SQL> UPDATE jignesh.SALARIES SET SALARY = 35000 WHERE NAME  
= 'JOHN';
```

1 row updated.

```
SQL> SELECT * FROM jignesh.SALARIES;
```

NAME	SALARY	AGE
-----	-----	-----
JACK	35000	29
JILL	48000	42
JOHN	61000	55

ANALYSIS:

You can see now that the update works as long as Jill abides by the

privileges she has been given.

Using the **WITH GRANT OPTION** Clause

What do you think would happen if Jill attempted to pass his UPDATE privilege on to Jack? At first glance you might think that Jill, because he was entrusted with the UPDATE privilege, should be able to pass it on to other users who are allowed that privilege. However, using the GRANT statement as you did earlier, Jill cannot pass his privileges on to others:

```
SQL> GRANT SELECT, UPDATE(SALARY) ON jignesh.SALARIES TO Jill;
```

Here is the syntax for the GRANT statement that was introduced earlier today:

SYNTAX:

GRANT {object_priv | ALL [PRIVILEGES]} [(column [, column]...)] [, {object_priv | ALL [PRIVILEGES]} [(column [, column] ...)]] ... ON [schema.]object TO {user | role | PUBLIC} [, {user | role | PUBLIC}] ... [WITH GRANT OPTION]

What you are looking for is the WITH GRANT OPTION clause at the end of the GRANT statement. When object privileges are granted and WITH GRANT OPTION is used, these privileges can be passed on to others. So if you want to allow Jill to pass on this privilege to Jack, you would do the following:

INPUT:

```
SQL> GRANT SELECT, UPDATE(SALARY) ON jignesh.SALARIES TO JILL  
WITH GRANT OPTION;
```

OUTPUT:

Grant succeeded.

Jill could then log on and issue the following command:

INPUT/OUTPUT:

```
SQL> GRANT SELECT, UPDATE(SALARY) ON Bryan.SALARIES TO JACK;
```

Grant succeeded.

Changing User Passwords

One of the most common tasks users ask administrators (DBA or system administrator) is to reset user passwords. Quite often, a user of the HelpDesk contacts the DBA with this request. You accomplish this task using the ALTER USER command:

```
SQL> alter user scott identified by lion;
```

User altered.

Users can reset their own personal passwords, but they often forget how to access the account and ask the DBA to do it.

TOP:1 SQL v/s PL/SQL

SQL	PL/SQL
SQL stands for Structured Query Language, which does not have procedural programming capability.	PL/SQL stands for Procedural Structured Query Language. Which have advantage over SQL.
SQL is the language that enables relational database users to communicate with the database in a straightforward manner.	PL/SQL is Oracle's procedural language; it comprises the standard language of SQL and a wide array of commands that enable you to control the execution of SQL statements according to different conditions.
You can use SQL commands to query the database and modify tables within the database.	PL/SQL can also handle runtime errors.
If you send a series of SQL statements to the server in standard SQL, the server executes them one at a time in chronological order.	PL/SQL allows you to write interactive, user-friendly programs that can pass values into variables. SQL statements can be processed simultaneously for better overall performance.

No Programming flexibility available with SQL.

Programmers can divide functions into logical blocks of code. Modular programming techniques support flexibility during the application development.



Atmiya
Infotech Pvt. Ltd.

TOP:2 PL/SQL Block Structure

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Within a PL/SQL block of code, processes such as data manipulation or queries can occur. The following parts of a PL/SQL block are discussed in this section:

- The **DECLARE** section contains the definitions of variables and other objects such as constants and cursors. This section is an optional part of a PL/SQL block.
- The **PROCEDURE** section contains conditional commands and SQL statements and is where the block is controlled. This section is the only mandatory part of a PL/SQL block.
- The **EXCEPTION** section tells the PL/SQL block how to handle specified errors and user-defined exceptions. This section is an optional part of a PL/SQL block.

Here is the basic structure of a PL/SQL block:

SYNTAX:

BEGIN -- optional, denotes beginning of block

DECLARE -- optional, variable definitions

BEGIN -- mandatory, denotes beginning of procedure section

EXCEPTION -- optional, denotes beginning of exception section

END -- mandatory, denotes ending of procedure section

END -- optional, denotes ending of block

Notice that the only mandatory parts of a PL/SQL block are the second BEGIN and the first END, which make up the PROCEDURE section. Of course, you will have statements in between. If you use the first BEGIN, then you must use the second END, and vice versa.

COMMENTS

What would a program be without comments? Programming languages provide commands that allow you to place comments within your code, and PL/SQL is no exception. The comments after each line in the preceding sample block structure describe each command. The accepted comments in PL/SQL are as follows:

SYNTAX:

-- This is a one-line comment.

/* This is a

multiple-line comment.*/*

The DECLARE Section

The DECLARE section of a block of PL/SQL code consists of variables, constants, cursor definitions, and special data types. As a PL/SQL programmer, you can declare all types of variables within your blocks of code. However, you must assign a data type, which must conform to Oracle's rules of that particular data type, to every variable that you define. Variables must also conform to naming standards.

Variable Assignment

Variables are values that are subject to change within a PL/SQL block. PL/SQL variables must be assigned a valid data type upon declaration and can be initialized if necessary. The following example defines a set of variables in the DECLARE portion of a block:

DECLARE

```
owner char(10);  
tablename char(30);  
bytes number(10);  
today date;
```

Notice that each variable declaration ends with a semicolon. Variables may also be initialized in the DECLARE section. For example:

DECLARE

```
customer char(30);  
fiscal_year number(2) := '97';
```

You can use the symbol `:=` to initialize, or assign an initial value, to variables in the DECLARE section. You must initialize a variable that is defined as NOT NULL.

DECLARE

```
customer char(30);  
fiscal_year number(2) NOT NULL := '97';
```

The PROCEDURE Section

The PROCEDURE section is the only mandatory part of a PL/SQL block. This part of the block calls variables and uses cursors to manipulate data in the database. The PROCEDURE section is the main part of a block, containing conditional statements and SQL commands.

BEGIN...END

In a block, the BEGIN statement denotes the beginning of a procedure. Similarly, the END statement marks the end of a procedure. The following example shows the basic structure of the PROCEDURE section:

SYNTAX:

BEGIN

condition1;

statement1;

condition2;

statement2;

.....

END



The EXCEPTION Section

The EXCEPTION section is an optional part of any PL/SQL block. If this section is omitted and errors are encountered, the block will be

terminated. Some errors that are encountered may not justify the immediate termination of a block, so the EXCEPTION section can be used to handle specified errors or user-defined exceptions in an orderly manner. Exceptions can be user-defined, although many exceptions are predefined by Oracle.

Raising Exceptions

Exceptions are raised in a block by using the command RAISE. Exceptions can be raised explicitly by the programmer, whereas internal database errors are automatically, or implicitly, raised by the database server.

SYNTAX:

DECLARE

exception_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception_name;

END IF;

EXCEPTION

WHEN exception_name THEN

statement;

END;

This block shows the fundamentals of explicitly raising an exception. First exception_name is declared using the EXCEPTION statement. In

the PROCEDURE section, the exception is raised using RAISE if a given condition is met. The RAISE then references the EXCEPTION section of the block, where the appropriate action is taken.

Handling Exceptions

The preceding example handled an exception in the EXCEPTION section of the block. Errors are easily handled in PL/SQL, and by using exceptions, the PL/SQL block can continue to run with errors or terminate gracefully.

SYNTAX:

EXCEPTION

WHEN exception1 THEN

statement1;

WHEN exception2 THEN

statement2;

WHEN OTHERS THEN

statement3;

This example shows how the EXCEPTION section might look if you have more than one exception. This example expects two exceptions (exception1 and exception2) when running this block. WHEN OTHERS tells statement3 to execute if any other exceptions occur while the block is being processed. WHEN OTHERS gives you control over any errors that may occur within the block.

Executing a PL/SQL Block

PL/SQL statements are normally created using editor and are executed like normal SQL script files. PL/SQL uses semicolons to terminate each statement in a block--from variable assignments to data manipulation commands. The forward slash (/) is mainly associated with SQL script files, but PL/SQL also uses the forward slash to terminate a block in a script file. The easiest way to start a PL/SQL block is by issuing the START command, abbreviated as STA or @. Your PL/SQL script file might look like this:

SYNTAX:

```
/* This file is called proc1.sql */
```

```
DECLARE
```

```
...
```

```
BEGIN
```

```
...
```

```
statements;
```

```
...
```

```
EXCEPTION
```

```
...
```

```
END;
```

```
/
```

You execute your PL/SQL script file as follows:

```
SQL> start proc1 or
```

```
SQL> sta proc1 or
```

```
SQL> @proc1
```

Displaying Output to the User

Particularly when handling exceptions, you may want to display output to keep users informed about what is taking place. You can display output to convey information, and you can display your own customized error messages, which will probably make more sense to the user than an error number. Perhaps you want the user to contact the database administrator if an error occurs during processing, rather than to see the exact message.

PL/SQL does not provide a direct method for displaying output as a part of its syntax, but it does allow you to call a package that serves this function from within the block. The package is called DBMS_OUTPUT.

EXCEPTION

```
WHEN zero_divide THEN
```

```
DBMS_OUTPUT.put_line('ERROR: DIVISOR IS  
ZERO. SEE YOUR DBA.');
```

ZERO_DIVIDE is an Oracle predefined exception. Most of the common errors that occur during program processing will be predefined as exceptions and are raised implicitly (which means that you don't have to raise the error in the PROCEDURE section of the block).

If this exception is encountered during block processing, the user will see:

INPUT:


```
SQL> @block1
```

ERROR: DIVISOR IS ZERO. SEE YOUR DBA.

PL/SQL procedure successfully completed.

Doesn't that message look friendly than:

INPUT/OUTPUT:

```
SQL> @block1
```

```
begin
```

```
*
```

ERROR at line 1:

ORA-01476: divisor is equal to zero

ORA-06512: at line 20



Example:1 of PL/SQL

Declare

Pi constant NUMBER (9,7) := 3.1415926;

Radius number(5);

Area number(14,2);

Begin

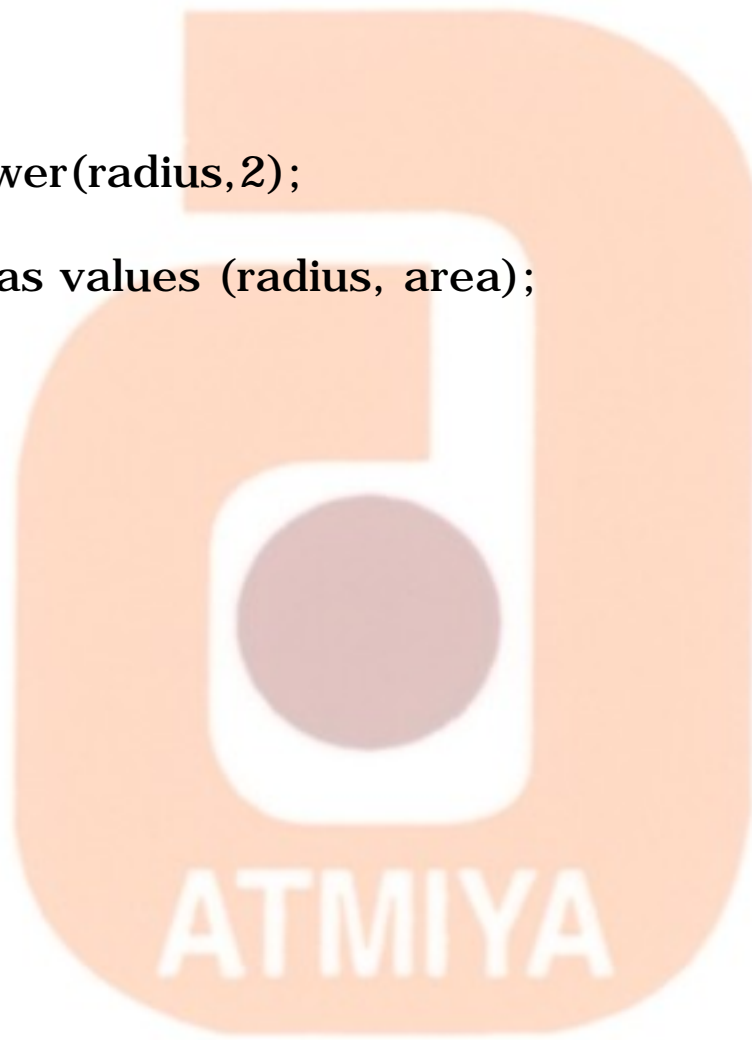
Radius := 3;

Area := pi*power(radius,2);

Insert into areas values (radius, area);

End;

/



TOP:3 Language Construct of PL/SQL

Data Types in PL/SQL

Most data types are obviously similar, but each implementation has unique storage and internal-processing requirements. When writing PL/SQL blocks, you will be declaring variables, which must be valid data types. The following subsections briefly describe the data types available in PL/SQL.

In PL/SQL Oracle provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs, such as a COBOL program, particularly if you are embedding PL/SQL code in another program. Subtypes are simply alternative names for Oracle data types and therefore must follow the rules of their associated data type.

Note: To check available Data Types [CLICK ME](#). Rest of the data types are explain in brief here with.

Binary Data Types

Binary data types store data that is in a binary format, such as graphics or photographs. These data types include RAW and LONGRAW.

BOOLEAN

BOOLEAN stores the following values: TRUE, FALSE, and NULL. Like DATE, BOOLEAN requires no parameters when defining it as a

column's or variable's data type.

ROWID

ROWID is a pseudo column that exists in every table in an Oracle database. The ROWID is stored in binary format and identifies each row in a table. Indexes use ROWIDs as pointers to data.

Conditional Statements

Now we are getting to the good stuff--the conditional statements that give you control over how your SQL statements are processed. The conditional statements in PL/SQL resemble those in most third-generation languages.

IF...THEN

The IF...THEN statement is probably the most familiar conditional statement to most programmers. The IF...THEN statement dictates the performance of certain actions if certain conditions are met. The structure of an IF...THEN statement is as follows:

SYNTAX:

```
IF condition1 THEN
```

```
    statement1;
```

```
END IF;
```



If you are checking for two conditions, you can write your statement as follows:

SYNTAX:

IF condition1 THEN

statement1;

ELSE

statement2;

END IF;

If you are checking for more than two conditions, you can write your statement as follows:

SYNTAX:

IF condition1 THEN

statement1;

ELSIF condition2 THEN

statement2;

ELSE

statement3;

END IF;

ANALYSIS:

The final example states: If condition1 is met, then perform statement1; if condition2 is met, then perform statement2; otherwise, perform statement3. IF...THEN statements may also be nested within other statements and/or loops.

Declare

Pi constant NUMBER (9,7) := 3.1415926;

Radius number(5);

Area number(14,2);

Begin

Radius := 3;

Area := pi*power(radius,2);

If Area > 3 then

Insert into areas values (radius, 3);

Else

Insert into areas values (radius, 2);

End If;

End;

/

LOOPS

Loops in a PL/SQL block allow statements in the block to be processed continuously for as long as the specified condition exists. There are three types of loops.

Simple loops

A loop that keeps repeating until an exit or exit when statement is reached within the loop

FOR loops	A loop that repeats a specified number of times
WHILE loops	A loop that repeats until a condition is met

In the following sections, you will see examples of each type of loop. The loop examples will use as their starting point the PL/SQL blocks used previously in this chapter.

Simple Loops

In the following listing, a simple loop is used to generate multiple rows in the AREAS table. The loop is started by the loop keyword, and the exist when clause determines when the loop should be exited. An end loop clause signals the end of the loop.

Declare

Pi constant NUMBER (9,7) := 3.1415926;

Radius number(5);

Area number(14,2);

Begin

Radius := 3;

loop

Area := pi*power(radius,2);

Insert into areas values (radius, area);

radius := radius+ 1;

exit when area > 100;

Atmiya

Infotech Pvt. Ltd.

```
end loop;
```

```
End;
```

```
/
```

The loop section of the example establishes the flow control for the commands in the executable commands section of the PL./SQL block. The steps within the loop are described in the following commented version of the loop commands:

```
loop
```

```
/* Calculate the area, based on the radius value. */
```

```
Area := pi*power(radius,2);
```

```
/* Insert the current values into the AREAS table. */
```

```
Insert into areas values (radius, area);
```

```
/* Increment the radius value by 1 */
```

```
radius := radius+ 1;
```

```
/* Evaluate the last calculated area. If the value exceeds 100,  
then exit. Otherwise, repeat the loop using the new radius  
value. */
```

```
exit when area > 100;
```

```
/* Signal the end of the loop. */
```

```
end loop;
```

The loop should generate multiple entries in the AREAS table. The first

record will be the record generated by a Radius value of 3. Once an area value exceeds 100, no more records will be inserted into the AREAS table.

Sample output following the execution of the PL/SQL block is shown in the following listing:

```
SQL> SELECT * FROM areas ORDER BY Radius;
```

RADIUS	AREA
-----	-----
3	28.27
4	50.27
5	78.54
6	113.1

Since the area value for a Radius value of 6 exceeds 100, no further Radius values are processed and the PL/SQL block completes.

FOR Loops

In simple loops, the loop executes until an exit condition is met. In a FOR loop, the loop executes a specified number of times. An example of a FOR loop is shown in the following listing. The FOR loop's start is indicated by the keyword `for`, followed by the criteria used to determine when the processing should exit the loop. Since the number of times the loop is executed is set when the loop is begun, an exit command isn't needed within the loop.

In the following example, the areas of circles are calculated based on Radius values ranging from 1 through 7, inclusive:

Declare

```
Pi constant NUMBER (9,7) := 3.1415926;
```

```
Radius number(5);
```

```
Area number(14,2);
```

Begin

```
for Radius in 1..7 loop
```

```
    Area := pi*power(radius,2);
```

```
    Insert into AREAS values (radius, area);
```

```
end loop;
```

End;

```
/
```

The steps involved in processing the loop are shown in the following commented listing:

```
/* Specify the criteria for the number of loop executions */
```

```
for radius in 1..7 loop
```

```
/* Calculate the area using the current Radius value. */
```

```
Area := pi*power(radius,2);
```

```
/* Insert the area and radius values into the AREAS table. */
```

```
Insert into AREAS values (radius, area);
```

```
/* Signal the end of the loop */
```

```
end loop;
```

Note that there is no line that says

```
radius := radius+ 1;
```

in the FOR loop. Since the specification of the loop specifies

for radius in 1..7 loop

the radius values are already specified. For each value, all of the commands within the loop are executed (these commands can include other conditional logic, such as if conditions). Once the loop has completed processing a Radius value, the limits on the for clause are checked, and either the next Radius value is used or the loop execution is complete.

```
SQL> SELECT * FROM areas ORDER BY Radius;
```

RADIUS

AREA

1	3.14
2	12.57
3	28.27
4	50.27
5	78.54
6	113.1

Atmiya
Infotech Pvt. Ltd.

7

153.94

7 rows selected.

WHILE Loops

In a WHILE loop, the loop is processed until an exit condition is met. Instead of specifying the exit condition via an exit command within the loop, the exit condition is specified in the While command that initiates the loop.

In the following listing, a WHILE loop is created so that multiple Radius values will be processed. If the current value of the Radius variable meets the while condition in the loop's specification, then the loop's commands are processed. Once a Radius value fails the while condition in the loop's specification, the loop's execution is terminated.

Declare

Pi constant NUMBER (9,7) := 3.1415926;

Radius number(5);

Area number(14,2);

Begin

radius := 3;

While radius <= 7

loop

Area := pi*power(radius,2);

Insert into AREAS values (radius, area);


```
radius := radius+ 1;
```

```
end loop;
```

```
End;
```

```
/
```

The WHILE loop is similar in structure to the simple loop, since it terminates the loop based on a variable's value. The following listing shows the steps involved in the loop, with embedded comments:

```
/* Set an initial value for the Radius variable */
```

```
radius := 3;
```

```
/* Establish the criteria for the termination of the loop. if the
condition is met, execute the commands within the loop. if
the condition is not met, then terminate the loop. */
```

```
While radius<= 7
```

```
/* Begin the commands to be executed. */
```

```
loop
```

```
/* Calculate the area based on the current radius value and
insert a record in the AREAS table. */
```

```
Area := pi*power(radius,2);
```

```
Insert into AREAS values (radius, area);
```

```
/* Set a new value for the Radius variable. The new
value of Radius will be evaluated against the
termination criteria and the loop commands will be
executed for the new radius value or the loop will
```

```
terminate. */
```

```
radius := radius+1;
```

```
/* Signal the end of the commands within the loop.
*/
```

```
end loop;
```

When executed, the PL/SQL block in the previous listing will generate records in the AREAS table. The output of the PL/SQL block is shown in the following listing:

```
SQL> SELECT * FROM areas ORDER BY Radius;
```

RADIUS

AREA

3

28.27

4

50.27

5

78.54

6

113.1

7

153.94

Because of the value assigned to the Radius variable prior to the loop, the loop is forced to executed at least once. You should verify that your variable assignments meet the conditions used to limit the loop executions.

TOP:4 %TYPE and %ROWTYPE

The %TYPE Attribute

%TYPE is a variable attribute that returns the value of a given column of a table. Instead of hard-coding the data type in your PL/SQL block, you can use %TYPE to maintain data type consistency within your blocks of code.

INPUT:

DECLARE

```
CURSOR cur_emp IS SELECT empno, ename from emp;
```

```
xempno emp.empno%TYPE;
```

```
xename emp.ename%TYPE;
```

ANALYSIS:

The variable xempno is declared to have the same data type as empno in the EMP table. %TYPE declares the variable name to have the same data type as the column ename in the EMP table.

DECLARE

```
cursor cur_emp is select empno, ename from emp;
```

```
xempno emp.empno%TYPE;
```

```
xename emp.ename%TYPE;
```

BEGIN

open cur_emp;

loop

fetch cur_emp into xempno, xename;

end loop;

close cur_emp;

END;

/

NOTE: Attend this topic after you complete next topic say 'Using Cursor'

The %ROWTYPE Attribute

Variables are not limited to single values. If you declare a variable that is associated with a defined cursor, you can use the %ROWTYPE attribute to declare the data type of that variable to be the same as each column in one entire row of data from the cursor. In Oracle's lexicon the %ROWTYPE attribute creates a record variable.

INPUT:

DECLARE

CURSOR cur_emp IS SELECT empno, ename from emp;

employee_record cur_emp%ROWTYPE;



ANALYSIS:

This example declares a variable called `employee_record`. The `%ROWTYPE` attribute defines this variable as having the same data type as an entire row of data in the `cur_emp`. Variables declared using the `%ROWTYPE` attribute are also called aggregate variables.

DECLARE

```
cursor cur_emp is select empno, ename from emp;
```

```
employee_record employee_cursor%ROWTYPE;
```

BEGIN

```
open cur_emp;
```

```
loop
```

```
    fetch cur_emp into employee_record;
```

```
    update emp set sal = 1000 where  
    empno= employee_record.empno;
```

```
    commit;
```

```
end loop;
```

```
close employee_cursor;
```

```
END;
```

```
/
```

Atmiya

Infotech Pvt. Ltd.

TOP:5 Using Cursors

Database cursors enable you to select a group of data, scroll through the group of records (often called a recordset), and examine each individual line of data as the cursor points to it. You can use a combination of local variables and a cursor to individually examine each record and perform any external operation needed before moving on to the next record.

One other common use of cursors is to save a query's results for later use. A cursor's result set is created from the result set of a SELECT query. If your application or procedure requires the repeated use of a set of records, it is faster to create a cursor once and reuse it several times than to repeatedly query the database.

Follow these steps to create, use, and close a database explicit cursor:

1. Create the cursor.
2. Open the cursor for use within the procedure or application.
3. Fetch a record's data one row at a time until you have reached the end of the cursor's records.
4. Close the cursor when you are finished with it.

Creating a Cursor

The Oracle8 SQL syntax used to create a cursor looks like this:

SYNTAX:

CURSOR cursor_name IS SELECT_statement;

By executing this statement, you have defined the cursor result set that will be used for all your cursor operations. A cursor has two important parts: the cursor result set and the cursor position.

Opening a Cursor

The simple command to open a cursor for use is

SYNTAX:

OPEN cursor_name;

Where cursor_name identifies a cursor that has previously been declared. When a cursor is opened, the following things happen:

- The values of the bind variables are examined.
- Based on the values of the bind variables, the active set is determined.
- The active set pointer is set to the first row.

Now you can use the cursor to scroll through the result set.

Fetching from a Cursor

The INTO clause for the query is part of the FETCH statement. The FETCH statement has two forms,

SYNTAX:

FETCH cursor_name INTO list_of_variables;

and

```
FETCH cursor_name INTO PL/SQL_record;
```

Each time the FETCH command is executed, the cursor pointer advances through the result set one row at a time. If desired, data from each row can be fetched into the fetch_target_list variables.

Closing a Cursor

Closing a cursor is a very simple matter. The statement to close a cursor is as follows:

SYNTAX:

```
CLOSE cursor_name;
```

When all of the active set has been retrieved, the cursor should be closed. This tells PL/SQL that the program is finished with the cursor, and the resources associated with it can be freed. These resources include the storage used to hold the active set, as well as any temporary space used for determining the active set. Where cursor_name identifies a previous opened cursor, Once a cursor is closed, it is illegal to fetch from it.

There are two type of cursors in ORACLE. They are Implicit and Explicit cursor. Now let us check individually.

Explicit Cursor

You can explicitly declare a cursor to process the rows individually. A cursor declared by the user is called Explicit Cursor. For queries that return more than one row, you must declare a cursor explicitly.

```
DECLARE
```

```
/* Declare cursor */
```

```
CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE  
deptno= 10;
```

```
/* Declaration of memory variables that holds data fetched  
from the cursor */
```

```
xempno emp.empno%TYPE;
```

```
xsal emp.sal%TYPE;
```

```
BEGIN
```

```
/* Open cursor */
```

```
OPEN cur_emp;
```

```
/* Infinite loop to fetch data from cursor cur_emp one row at  
a time */
```

```
Loop
```

```
/* Fetch records from cursor one by one */
```

```
FETCH cur_emp INTO xempno, xsal;
```

```
/* Update emp table with new data with the use of  
variables */
```

```
UPDATE emp SET sal = sal + (sal * 0.10) WHERE  
empno=xempno;
```

```
End Loop;
```

```
Commit;
```

```
/* Close cursor, now job is over. */
```

```
CLOSE cur_emp;
```

```
END;
```

ANALYSIS:

Above mentioned example fetches records one by one for emp table where department no is 10. Then it updates one by one records with increment of salary to 10 percent. After all the records are over, we close the cursor.

Oracle provides cursor variables to control the execution of the cursor by default. Whenever any cursor is opened and used Oracle creates a set of four system variables via which Oracle keeps track of the 'Current' status of the cursor. You can access these cursor variables. They are:

%FOUND	A record can be fetched from the cursor
%NOTFOUND	No more records can be fetched from the cursor
%ISOPEN	The cursor has been opened
%ROWCOUNT	The number of rows fetched from the cursor so far

The %FOUND, %NOTFOUND, and %ISOPEN cursor attributes are Boolean; they are set to either TRUE or FALSE. Because they are boolean attributes, you can evaluate their setting without explicitly matching them to values of TRUE or FALSE.

Syntax: cursorname%NOTFOUND

DECLARE

CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE
deptno= 10;

xempno emp.empno%TYPE;

xsal emp.sal%TYPE;

BEGIN

OPEN cur_emp;

Loop

FETCH cur_emp INTO xempno, xsal;

/* If no. of records retrieved is 0 or if all the records
are fetched then exit the loop. */

EXIT when cur_emp%NOTFOUND;

UPDATE emp SET sal = sal + (sal * 0.10) WHERE
empno=xempno;

End Loop;

Commit;

CLOSE cur_emp;

END;

Syntax: cursorname%FOUND

DECLARE



```
CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE  
deptno= 10;
```

```
xempno emp.empno%TYPE;
```

```
xsal emp.sal%TYPE;
```

```
BEGIN
```

```
OPEN cur_emp;
```

```
Loop
```

```
FETCH cur_emp INTO xempno, xsal;
```

```
/* If no of records received > 0 then process the  
data else exit the loop */
```

```
IF cur_emp%FOUND THEN
```

```
UPDATE emp SET sal = sal + (sal * 0.10)  
WHERE empno=xempno;
```

```
ELSE
```

```
exit;
```

```
END IF;
```

```
End Loop;
```

```
Commit;
```

```
CLOSE cur_emp;
```

```
END;
```

The logo for Atmiya, featuring the word "Atmiya" in a stylized, rounded font with a light blue and orange color scheme.

Infotech Pvt. Ltd.

Syntax: cursorname%ISOPEN**DECLARE****CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE
deptno= 10;****xempno emp.empno%TYPE;****xsal emp.sal%TYPE;****BEGIN****OPEN cur_emp;****/* If the cursor is open continue with the data processing else
display an appropriate error message */****IF cur_emp%ISOPEN THEN****Loop****FETCH cur_emp INTO xempno, xsal;****UPDATE emp SET sal = sal + (sal *
0.10) WHERE empno=xempno;****ELSE****exit;****END IF;****End Loop;****ELSE**

```
dbms_output.put_line('Unable to open cursor');
```

```
END IF;
```

```
Commit;
```

```
CLOSE cur_emp;
```

```
END;
```

Syntax: cursorname%ROWCOUNT

```
DECLARE
```

```
CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE  
deptno= 10;
```

```
xempno emp.empno%TYPE;
```

```
xsal emp.sal%TYPE;
```

```
BEGIN
```

```
OPEN cur_emp;
```

```
Loop
```

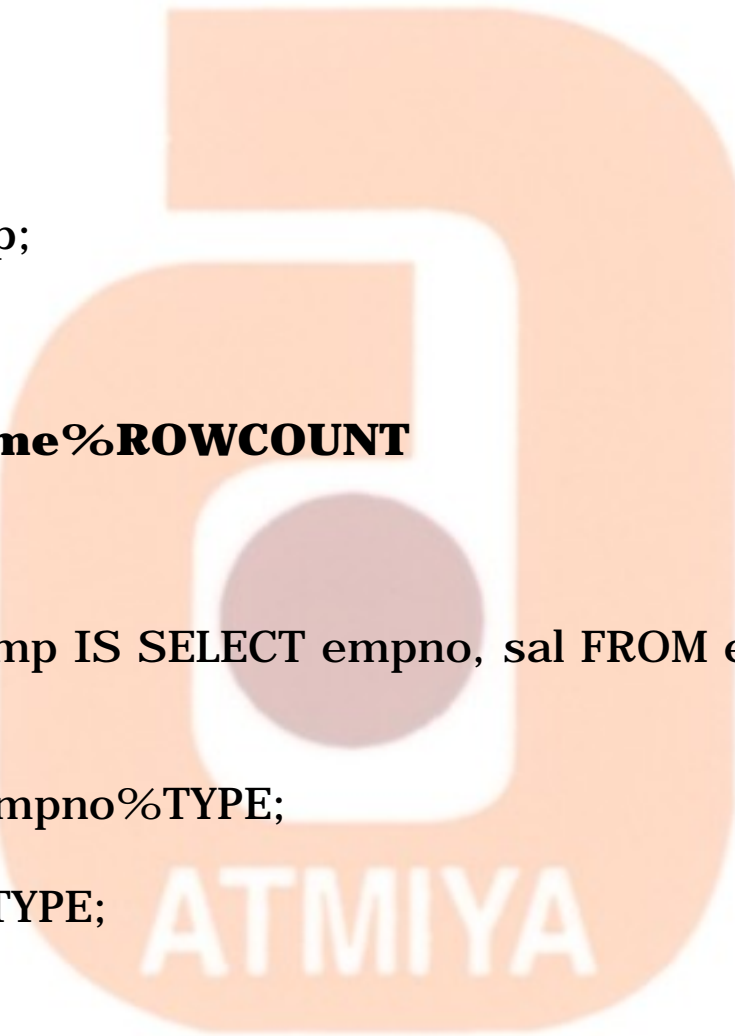
```
FETCH cur_emp INTO xempno, xsal;
```

```
UPDATE emp SET sal = sal + (sal * 0.10)  
WHERE empno=xempno;
```

```
ELSE
```

```
exit;
```

```
END IF;
```



```
exit when cur_emp%ROWCOUNT=3;
```

```
End Loop;
```

```
Commit;
```

```
CLOSE cur_emp;
```

```
END;
```

ANALYSIS:

%ROWCOUNT returns the number of rows fetched from the active set. It is set to zero when the cursor is opened. No if we want to increment salary for first 3 employee, above cursor statement can be used.

In most situations that require an explicit cursor, you can simplify coding by using a cursor for loop instead of the open, fetch and close statement. A cursor for loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor repeatedly fetches rows of values from the active set into items in the record, and closes the cursor when all rows have been processed. As example,

```
DECLARE
```

```
CURSOR cur_emp IS SELECT empno, sal FROM emp WHERE  
deptno=10;
```

```
BEGIN
```

```
for emp_rec in cur_emp
```

```
loop
```

```
update emp set sal = sal + (sal * 0.10) where  
empno=emp_rec.empno;
```

```
end loop;
```

```
commit;
```

```
END;
```

Notice here that when you leave the loop, the cursor is closed automatically. This is true even if you use an exit or goto statement to leave the loop prematurely, or if an exception is raised inside the loop.

Implicit Cursor

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the SQL cursor. So, although you can not use the open, fetch and close statements to control an implicit cursor, you can still use cursor attributes to access information about the most recently executed SQL statement. The variables are same with implicit cursor as explicit have, that we discussed earlier. The SQL cursor has four attributes as described below. When appended to the cursor name, these attributes let you access information about the execution of insert, update, delete and single row select statement. Implicit cursor attributes return the boolean null value, until they are set by a cursor operation. The value of the cursor attributes always refer to the most recently executed SQL statement whenever the statement appears. It might be in a different scope (in a sub-block). So, if you want to save an attribute value for later use, assign it a boolean variable immediately.

%NOTFOUND: evaluates to true, if an insert, update or delete affected no rows, or a single row select returns no rows, Otherwise, it evaluates to false.

SYNTAX: SQL%NOTFOUND

BEGIN

```
UPDATE emp SET sal = sal + (sal * 0.10) WHERE  
empno= &xempno;
```

```
IF sql%NOTFOUND THEN
```

```
    dbms_output.put_line('Employee no does not exist');
```

```
ELSE
```

```
    dbms_output.put_line('Employee record modified  
successfully');
```

```
END IF;
```

```
END;
```

%FOUND: is the logical opposite of %NOTFOUND. %FOUND evaluates to true if an insert, update or delete affected one or more rows, or a single-row select statement returned one or more rows. Otherwise, it evaluates to false.

SYNTAX: SQL%FOUND

Infotech Pvt. Ltd.

BEGIN

```
UPDATE emp SET sal = sal + (sal * 0.10) WHERE  
empno= &xempno;
```

```
IF sql%FOUND THEN
```

```
    dbms_output.put_line('Employee record modified  
successfully');
```

```
ELSE
```

```
dbms_output.put_line('Employee no does not exist');
```

```
END IF;
```

```
END;
```

%ROWCOUNT: returns the number of rows affected by an insert, update or delete, or select into statement.

```
DECLARE
```

```
rows_affected char(4);
```

```
BEGIN
```

```
UPDATE emp SET sal = sal + (sal * 0.10) WHERE  
empno= &xempno;
```

```
rows_affected := to_char(sql%ROWCOUNT)
```

```
IF sql%ROWCOUNT>0 THEN
```

```
dbms_output.put_line(rows_affected || 'Employee  
record modified successfully');
```

```
ELSE
```

```
dbms_output.put_line('Employee no. does not  
exist');
```

```
END IF;
```

```
END;
```

%ISOPEN: Oracle automatically closes the SQL cursor after executing its associated SQL statement. As a result, sql%isopen always evaluates to false.

Cursor Parameters

You can specify parameters for cursors in the same way you do for subprograms. The following example illustrates the syntax for declaring parameter cursors:

```
CURSOR cur_emp (xempno IN NUMBER) IS SELECT empno, sal FROM emp WHERE deptno = 10;
```

The parameter mode is always IN for cursor parameters, but the data type can be any valid data type. You can reference a cursor parameter, whose value is set when the cursor opens, only during the cursor's declared SQL query.

Flexibility within cursor parameters enables the developer to pass different numbers of parameters to a cursor by using the parameter default mechanism. This is illustrated in the following example:

```
CURSOR cur_emp (xempno IN NUMBER, job IN VARCHAR2(20)) IS SELECT empno, sal FROM emp WHERE deptno = 10;
```

By using the INTEGER DEFAULT declaration, you can pass all, one, or none of the parameters to this cursor depending on the logic flow of your code.

Top:1 Procedure

The concept of stored procedures is an important one for the professional database programmer to master. Stored procedures are functions that contain potentially large groupings of SQL statements. These functions are called and executed just as C, FORTRAN, or Visual Basic functions would be called. A stored procedure should encapsulate a logical set of commands that are often executed (such as a complex set of queries, updates, or inserts). Stored procedures enable the programmer to simply call the stored procedure as a function instead of repeatedly executing the statements inside the stored procedure. However, stored procedures have additional advantages.

These procedures are created and then stored as part of a database, just as tables and indexes are stored inside a database. One of the biggest advantages to stored procedures lies in the design of their execution. When executing a large batch of SQL statements to a database server over a network, your application is in constant communication with the server, which can create an extremely heavy load on the network very quickly. As multiple users become engaged in this communication, the performance of the network and the database server becomes increasingly slower. The use of stored procedures enables the programmer to greatly reduce this communication load.

After the stored procedure is executed, the SQL statements run sequentially on the database server. Some message or data is returned to the user's computer only when the procedure is finished. This approach improves performance and offers other benefits as well. Stored procedures are actually compiled by database engines the first

time they are used. The compiled map is stored on the server with the procedure. Therefore, you do not have to optimize SQL statements each time you execute them, which also improves performance.

Procedures are named PL/SQL blocks that can take parameters and perform an action and can be invoked. A procedure is generally used to perform an action and to pass values. Procedures are made up of:

- **Declarative Part:** The declarative part may contain declaration of cursors, constants, variables, exception and subprograms.
- **Executable Part:** The executable part contains a PL/SQL block consisting of statements that assign values, control execution and manipulate ORACLE data. The action to be performed is coded here and data that is to be returned back to the calling environment. Variables declared are put to use in this block.
- **Exception Handling Part:** This part contains code that performs an action to deal with exceptions raised during the execution of the executable part. This block can be used to handle oracle's own exceptions or the exceptions that are declared in the declarative part. One can not transfer the flow of execution from the exception handling part to the executable part or vice versa.

When a procedure is created, ORACLE automatically performs the following steps:

1. Compiles the procedure
2. Stores the compiled code
3. Stores the procedure in the database

Oracle performs the following steps to execute a procedure :

1. Verify user access

2. Verifies procedure validity
3. Executes the procedure

Advantages of procedures:

1. **Security:** Stored procedures can help enforce data security. For e.g. you can grant users access to a procedure that can query a table, but not grant them access to the table itself.
2. **Performance:** It improves database performance in the following ways:
 - Amount of information sent over a network is less.
 - No compilation step is required to execute the code.
 - As procedure is present in the shared pool of SGA retrieval from the disk is not required.
3. **Memory Allocation:** Reduction in memory as stored procedures have shared memory capabilities so only one copy of procedure needs to be loaded for execution by multiple users.
4. **Productivity:** Increases development productivity, by writing a single procedure we can avoid redundant coding and increase productivity.
5. **Integrity:** Improves integrity, a procedure needs to be tested only once to guarantee that it returns an accurate result. Hence coding errors can be reduced.

Procedure Syntax:

CREATE OR REPLACE PROCEDURE [schema.]procedurename

(argument { IN, OUT, IN OUT}
datatype,.....) { IS, AS}

variable declaration;

constant declaration;

BEGIN

PL/SQL subprogram body;

EXCEPTION

exception PL/SQL block;

END;

Keywords and Parameters:

REPLACE	recreates the procedure if it already exists. You can use this option to change the definition of an existing procedure without dropping, recreating and regranting object privileges previously granted on it. If you redefine a procedure ORACLE recompiles it.
schema	is the schema to contain the procedure. ORACLE takes the default schema to be the current schema, if it is omitted.
procedure	is the name of the procedure to be created.
argument	is the name of an argument to the procedure. Parentheses can be omitted if no arguments are present.

IN	specifies that you must specify a value for the argument when calling the procedure.
OUT	specifies that the procedure passes a value for this argument back to its calling environment after execution.
IN OUT	specifies that you must specify a value for the argument when calling the procedure and that the procedure passes a value for this argument back to its calling environment after execution. By default it takes IN.
datatype	is the datatype of an argument. It supports any datatype supported by PL/SQL.
PL/SQL	subprogrambody is the definition of procedure consisting of PL/SQL statements.

Example: 1

Write a procedure to accept the ticket_no as input and display an error message if the total_fare is null otherwise display the ticket_header.

```
CREATE OR REPLACE PROCEDURE ticket (t_no char) IS
```

```
    a ticket_header.ticket_no%type;
```

```
    b ticket_header.origin%type;
```

```
    c ticket_header.destination%type;
```

```
    d ticket_header.dot%type;
```



```
e ticket_header.total_fare%type;
```

```
total exception;
```

```
BEGIN
```

```
SELECT ticket_no, origin, destination, dot, total_fare
```

```
INTO a, b, c, d, e FROM ticket_header WHERE ticket_no =  
t_no;
```

```
if e is null then
```

```
    raise total;
```

```
else
```

```
    dbms_output.put_line(a||' '||b||' '||c||' '||d||' '||e);
```

```
end if;
```

```
EXCEPTION
```

```
when total then
```

```
    dbms_output.put_line('The total fare is null');
```

```
END;
```

Example: 2

Write a procedure to accept the route_id as input and update the capacity to be 25 if the cat_code is 1 and 50 if the cat_code is 2.

CREATE OR REPLACE PROCEDURE route(r_id number) IS

c_code number;

BEGIN

**SELECT category_code INTO c_code FROM route_header
WHERE route_id = r_id;**

if c_code = '1' then

**update route_header set capacity = 25 where
route_id = r_id;**

elsif c_code = '2' then

**update route_header set capacity = 50 where
route_id = r_id;**

end if;

dbms_output.put_line('table updated');

END;



Example: 3

Similar to variable declarations, the formal parameters to a procedure or function can have default values.

CREATE OR REPLACE PROCEDURE AddNewStudent (

```

p_FirstName students.first_name%TYPE,

p_LastName students.Last_name%TYPE,

p_Major students.majot%TYPE DEFAULT 'Computer') AS

```

```

BEGIN

```

```

/* Insert a new row in the students table. Use
student_sequence to generate the new student ID, and 0 for
current_credits */

```

```

INSERT INTO students VALUES (student_sequence.nextval,
p_firstName, p_LastName, p_major, 0);

```

```

COMMIT;

```

```

END;

```

Example: 4

To create a procedure to perform an item id check operation. p_itemidchk is the name of the procedure which accepts a variable itemid and returns a variable valexists to the host environment. The value of valexists changes from 0 (itemid does not exist) to 1 (itemid exists) depending on the records retrieved.

```

CREATE PROCEDURE p_itemidchk (vitemidno IN number, valexists
OUT NUMBER) AS

```

```

/* variable that hold data from the itemmast table */

```

```

dummyitem NUMBER(4);

```

BEGIN

```
select itemid into dummyitem from itemmast where itemid =  
vitemidno;
```

```
/* if the select statement retrieves data, valexists is set to 1  
*/
```

```
valexists := 1;
```

EXCEPTION

```
/* if the select statement does not retrieve data, valexists is  
set to 0 */
```

```
when no_data_found then
```

```
valexists := 0;
```

END;

Any PL/SQL block can be used to call this procedure to perform the check. To do this the contents of the variable vitemidno is passed on as an argument to the procedure p_itemidchk. The return value is then checked and appropriate action is taken. The following PL/SQL code takes care of what needs to be done as expressed in above example:

DECLARE

```
/* Cursor scantable retrieves all the records of table itemtran  
*/
```

```
cursor scantable is select itemid, operation, qty, description
```

```
from itemtran;

/* variables that hold data from the cursor scantable */

vitemidno number(4);

descrip varchar2(30);

oper char(1);

quantity number(3);

/* variable that stores 1 or 0. It is set in the procedure
p_itemidchk */

valexists number(1);

BEGIN

open scantable;

loop

    fetch scantable into vitemidno, oper, quantity,
    descrip;

    /* call procedure p_itemidchk to check if item_id is
    present in itemmast table */

    p_itemidchk(vitemidno, valexists);

    /* if itemid does not exists */

    if valexists=0 then

        /* if mode is insert then insert a record in
        itemmast table and set the status in the
```

```
itemtran table to 'SUCCESSFUL' */
```

```
if oper = 'I' then
```

```
    insert into itemmast(itemid,  
        bal_stock, description) values  
        (vitemidno, quantity, descrip);
```

```
    update itemtran set  
    itemtran.status='SUCCESSFUL'  
    where itemid = vitemidno;
```

```
end if;
```

```
else
```

```
    /* if the record is found and the operation is  
    insert then set the status to 'item already  
    exists' */
```

```
if oper = 'I' then
```

```
    update itemtran set itemtran.status  
    = 'ITEM EXIST' where itemid =  
    vitemidno;
```

```
end if;
```

```
endif;
```

```
exit when scantable%NOTFOUND;
```

```
end loop;
```

```
close scantable;
```

```
commit;
```


END;

DELETING a stored PROCEDURE

A procedure can be deleted from the database by using the following command:

DROP PROCEDURE <procedureName>;

Example: DROP PROCEDURE p_itemidchk;



Top:2 Functions

Unlike procedures, functions can return a value to the caller (procedures cannot return values). This value is returned through the use of the return keyword within the function. One or other way procedure and functions are same except one difference that is mentioned above. Functions can be executed explicitly like procedures. It has the same benefits like procedures as :

Functions are named PL/SQL blocks that can take parameters an action and can be invoked. A functions is generally used to perform an action and to pass values. Functions are made up of:

- Declarative Part
- Executable Part
- Exception Handling Part

When a function is created, ORACLE automatically performs the following steps:

1. Compiles the functions
2. Stores the compiled code
3. Stores the function in the database

Oracle performs the following steps to execute a functions :

1. Verify user access

2. Verifies function validity
3. Executes the function

Advantages of functions:

1. Security
2. Performance
3. Memory Allocation
4. Productivity
5. Integrity

Function Syntax:

CREATE OR REPLACE FUNCTION [schema.]functionname

(argument IN datatype,.....)

RETURN datatype {IS, AS}

variable declaration;

constant declaration;

BEGIN

PL/SQL subprogram body;

EXCEPTION

exception PL/SQL block;

END;

Keywords and Parameters:

REPLACE	recreates the function if it already exists. If you redefine a function ORACLE recompiles it.
schema	is the schema to contain the function. ORACLE takes the default schema to be the current schema, if it is omitted.
function	is the name of the function to be created.
argument	is the name of an argument to the function. Parentheses can be omitted if no arguments are present.
IN	specifies that you must specify a value for the argument when calling the function.
RETURN datatype	is the datatype of the function's return value. Because every function must return a value, this clause is required. It supports any datatype supported by PL/SQL.
PL/SQL	subprogram body is the definition of function consisting of PL/SQL statements.

Example: 1

Write function, which will accept the ticket_no as the input and return total_fare as the output. [hint: use ticket_header]

CREATE OR REPLACE FUNCTION fun1 (t_no number) RETURN number

IS

```
t_fare ticket_header.total_fare%type;
```

BEGIN

```
select total_fare into t_fare from ticket_header where
ticket_no = t_no;
```

```
return(t_fare);
```

END;

To execute the function the following program is used...

SQL> declare

```
tot number;
```

```
begin
```

```
tot := fun1('04');
```

```
dbms_output.put_line(to_char(tot));
```

```
END;
```



Example: 2

Write function which will accept the fleet_id as the input and return the day as the output. [hint: use fleet_header]

```
CREATE OR REPLACE FUNCTION fun2 (f_no number) return date IS

    dayt fleet.day%type;

BEGIN

    select day into dayt from fleet_header where fleet_id = r_no;

    return(dayt);

END;
```

TO execute the above function the following program is used...

```
SQL> declare

    a fleet_header.day%type;

BEGIN

    a:= fun2('01');

    dbms_output.put_line(a);

END;
```



Example: 3

The following function returns TRUE if the specified class is more than 90 percent full and FALSE otherwise:


```

CREATE OR REPLACE FUNCTION AlmostFull ( p_Department
classes.department%TYPE, p_Course classes.course%TYPE)

RETURN BOOLEAN IS

    v_CurrentStudents NUMBER;

    v_MaxStudents NUMBER;

    v_ReturnValue BOOLEAN;

    v_FullPercent CONSTANT NUMBER := 90;

BEGIN

    /* Get the current and maximum students for the requested
    course. */

    SELECT current_students, max_students INTO
    v_CurrentStudents, v_MaxStudents FROM classes WHERE
    department = p_Department AND course = p_Course;

    /* If the class is more full than the percentage given by
    v_FullPercent, return TRUE. Otherwise, return FALSE. */

    IF (v_CurrentStudents / v_MaxStudents * 100) >
    v_FullPercent THEN

        v_ReturnValue := TRUE;

    ELSE

        v_ReturnValue := FALSE;

    END IF;

    RETURN v_ReturnValue;

```

END AlmostFull;

This pl/sql block illustrates how to call a function, as described in above example

DECLARE

**CURSOR c_Classes IS SELECT department, course FROM
classes;**

BEGIN

FOR v_ClassRecord IN c_Classes LOOP

**/* Record all classes which don't have very much
room left in temp_table. */**

**IF AlmostFull(v_ClassRecord.department,
v_ClassRecord.course) THEN**

**INSERT INTO temp_table (char_col) VALUES
(v_ClassRecord.department || ' ' ||
v_ClassRecord.course || 'is almost full!');**

END IF;

END LOOP;

END;

Example: 4

Example shows five different return statements in the function, only one of them is executed. which one is executed depends on how full the class specified by p_Department and p_Course is.

```

CREATE OR REPLACE FUNCTION ClassInfo (

/* Returns 'Full' if the class is completely full, 'Some Room' if the class
is over 80% full, 'More Room' if the class is over 60% full, 'Lots of
Room' if the class is less than 60% full, and 'Empty' if there are no
students registered. */

p_Department classes.department%TYPE,

p_Course classes.course%TYPE)

RETURN VARCHAR2 IS

    v_CurrentStudents NUMBER;

    v_MaxStudents NUMBER;

    v_PercentFull NUMBER;

BEGIN

    /* Get the current and maximum students for the requested
    course. */

    SELECT current_students, max_students INTO
    v_CurrentStudents, v_MaxStudents FROM classes WHERE
    department = p_Department AND course = p_Course;

    -- Calculate the current percentage.

```

```
v_PercentFull := v_CurrentStudents / v_MaxStudents * 100;

IF v_PercentFull = 100 THEN

    RETURN 'Full';

ELSIF v_PercentFull > 80 THEN

    RETURN 'Some Room';

ELSIF v_PercentFull > 60 THEN

    RETURN 'More Room';

ELSIF v_PercentFull > 0 THEN


    RETURN 'Lots of Room';

ELSE

    RETURN 'Empty';

END IF;

END ClassInfo;
```

The logo for Atmiya Infotech Pvt. Ltd. is located at the bottom center of the page. It features the word "Atmiya" in a stylized, rounded font with a blue-to-purple gradient. Below it, "Infotech Pvt. Ltd." is written in a smaller, black, sans-serif font. A thin blue horizontal line separates the two parts of the logo.

Example: 5

`f_itemidchk` is the name of the function which accepts a variable `itemid` and returns a variable `valexists` to the host environment. The value of `valexists` changes from 0 (`itemid` does not exist) to 1 (`itemid` exists) depending on the records retrieved.

**CREATE FUNCTION f_itemidchk (vitemidno IN number) RETURN
number IS**

/* variable that hold data from the itemmast table */

dummyitem NUMBER(4);

BEGIN

**select itemid into dummyitem from itemmast where itemid =
vitemidno;**

**/* if the select statement retrieves data, valexists is set to 1
*/**

return 1;

EXCEPTION

**/* if the select statement does not retrieve data, valexists is
set to 0 */**

when no_data_found then

return 0;

END;

Any PL/SQL block can be used to call this function to perform the check. To do this the contents of the variable vitemidno is passed on as an argument to the function p_itemidchk. The return value is then checked and appropriate action is taken. The following PL/SQL code takes care of what needs to be done as expressed in above example:

DECLARE

```
/* Cursor scantable retrieves all the records of table itemtran
*/
```

```
cursor scantable is select itemid, operation, qty, description
from itemtran;
```

```
/* variables that hold data from the cursor scantable */
```

```
vitemidno number(4);
```

```
descrip varchar2(30);
```

```
oper char(1);
```

```
quantity number(3);
```

```
/* variable that stores 1 or 0. It is set in the procedure
p_itemidchk */
```

```
valexists number(1);
```

```
BEGIN
```

```
open scantable;
```

```
loop
```

```
    fetch scantable into vitemidno, oper, quantity,
    descrip;
```

```
    /* call function f_itemidchk to check if item_id is
    present in itemmast table */
```

```
    valexists := f_itemidchk(vitemidno);
```

```
    /* if itemid does not exists */
```


if valexists=0 then

/* if mode is insert then insert a record in
itemmast table and set the status in the
itemtran table to 'SUCCESSFUL' */

if oper = 'I' then

insert into itemmast(itemid,
bal_stock, description) values
(vitemidno, quantity, descrip);

update itemtran set
itemtran.status='SUCCESSFUL'
where itemid = vitemidno;

end if;

else

/* if the record is found and the operation is
insert then set the status to 'item already
exists' */

if oper = 'I' then

update itemtran set itemtran.status
= 'ITEM EXIST' where itemid =
vitemidno;

end if;

endif;

exit when scantable%NOTFOUND;

end loop;

```
close scantable;
```

```
commit;
```

```
END;
```

DELETING a stored FUNCTION

A function can be deleted from the database by using the following command:

```
DROP FUNCTION <functionName>;
```

Example: DROP FUNCTION f_itemidchk;



Top:3 Package

Packages are groups of procedures, functions, variables and SQL statements grouped together into a single unit. To execute a procedure within a package, you must first list the package name, and then list the procedure name, as shown in the following example:

```
execute Client_Package.New_Client('Jignesh Dhol');
```

Here, the New_client procedure within the Client_Package package was executed. Packages allow multiple procedures to use the same variables and cursors. Procedures within packages may be either available to the public or private, in which case they are only accessible via commands from within the package (such as calls from other procedures). Package may also include commands that are to be executed each time the package is called, regardless of the procedure or function called within the package. Thus, packages not only group procedures but also give you the ability to execute commands that are not procedure-specific.

Create package syntax

When creating packages, the package specification and the package body are created separately. Thus, there are two commands to use: create package for the package specification, and create package body for the package body. Both of these commands require that you have the CREATE PROCEDURE system privilege. If the package is to be created in a schema other than your own, then you must have the CREATE ANY PROCEDURE system package. The following is the syntax for creating package specifications:

CREATE OR REPLACE PACKAGE packagename

AS

(package specifications)

END packagename;

CREATE OR REPLACE PACKAGE BODY packagename

AS

(package body specifications)

END packagename;

Example:

CREATE OR REPLACE PACKAGE order_total

AS

(package specifications)

END order_total;

CREATE OR REPLACE PACKAGE BODY order_total

AS

(package body specifications)

END order_total;

CREATE OR REPLACE PACKAGE name is the command that starts the procedure build in the database. Declarations of objects and subroutines within the package area are visible to your applications.

Think of this area as the application interface to your PL/SQL code; at the very least, you must define the procedure entry routine here. Modifications to any specifications in this area require rebuilding your applications. The END statement signifies the end of the package specification area.

Next is the CREATE OR REPLACE PACKAGE BODY name statement that begins the specification area for declarations of PL/SQL objects and subroutines that only the procedure can "see." This area is invisible to your application but is not required in designing package procedures. However, designing procedures in this manner enables you to modify package body specifications without altering the application interface. As a result, applications do not require recompilation when these internal specifications change. Once again, the END statement marks the end of package body specifications.

The name order_total was selected for both the package and package body names in this example, but these names need not be the same.

Creating Package Subprograms

Creating subprograms within a package is the next step in developing a packaged procedure. You must decide which routines will be application-interface routines and which routines will be available only within the package. This determines where the subprogram specification will reside—in the package or in the package body.

7.3.1 Example of Package:

Definition:

Write a package containing (a) A procedure, which will accept the route_id as input and update the stop to be 'n'. [hint: use route_detail] (b) A procedure, which will accept the route_id as the input and delete that particular row. [hint: use route_details]

Answer:

```
SQL> CREATE OR REPLACE PACKAGE jack is
```

```
2> procedure first (routes char);
```

```
3> procedure second (route char);
```

```
4> END jack;
```

```
SQL> CREATE OR REPLACE PACKAGE BODY jack AS
```

```
    procedure first (routes char) is
```

```
        nstop char;
```

```
    BEGIN
```

```
        select nonstop into nstop from route_detail
        where
```

```
        route_id = routes;
```

```
        If nstop = 's' then
```

```
            update route_detail set nonstop =
            'n'
```

```
            where route_id = routes;
```

```
        else
```

```
            dbms_output.put_line('No updation
            required');
```

```
        end if;
```


end first;

procedure second (route char) is

BEGIN

**delete from route_details where route_id =
route;**

dbms_output.put_line('Deletion Complete');

END second;

END jack;

Execute the body and the specification of the procedure separately. To execute the procedure give "exec procedure name" at the SQL prompt. Give "exec jack.first('101');" and exec jack.second(105); at the SQL prompt to execute the procedures first and second.



Top:4 Trigger

Database triggers are procedures that are stored in the database and are implicitly executed (fired) when the contents of a table are changed. Triggers are executed when an insert, update or delete is issued against a table from SQL * Plus or through an application. The major point that make these triggers stand alone is that they are fired implicitly (i.e. internally) by Oracle itself and not explicitly called by the user, as done in normal procedures.

Benifits of Triggers:

Database triggers support oracle to provide a highly customized database management system. Some of the uses to which the database triggers can be put to customize management information in Oracle are as follows:

- A trigger can permit DML statements against a table only if they are issued during regular business hours or on predetermined weekdays.
- A trigger can also be used to keep an audit trail of a table (i.e. store the modified and deleted records of the table) along with the operation performed and the time on which the operation was performed.
- It can be used to prevent invalid transactions.
- Enforce complex security authorizations.

Precaution for triggers:

- When a trigger is fired, a SQL statement inside the trigger can also fire the same or some other trigger, called cascading, which must be considered.
- Excessive use of triggers for customizing the database can result in complex interdependencies between the triggers, which may be difficult to maintain in large applications.

Some differences are between procedure and trigger which have to be considered first. In procedures it's possible to pass parameters which is not the case with triggers. A trigger is executed implicitly by the Oracle itself upon modification of an associated table whereas to execute a procedure, it has to be explicitly called by the user. Triggers as well as declarative integrity constraints can be used to constraint data input. However both have significant difference as mentioned below:

- A declarative integrity constraint is a statement about a database that is always true. A constraint applies to existing data in the table and any statement that manipulates the table. Triggers constraint what transaction can do. A trigger does not apply to data loaded before the trigger was created, so it does not guarantee all data in table conforms to the rules established by an associated trigger.
- Also a trigger enforces transitional constraint which can not be enforced by a declarative integrity constraint.

A trigger has three basic parts:

- A triggering event or statement

It is a SQL statement that causes a trigger to be fired. It can INSERT, UPDATE or DELETE statement for a specific table. A triggering statement can also

specify multiple DML statements.

- A trigger restriction

A trigger restriction specifies a Boolean expression that must be TRUE for the trigger to fire. It is an option available for triggers that are fired for each row. Its function is to conditionally control the execution of a trigger. A trigger restriction is specified using a WHEN clause.

- A trigger action

A trigger action is the procedure that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluated to TRUE. It can contain SQL and PL/SQL statements; can define PL/SQL language constructs and can call stored procedures.

Additionally, for row triggers, the statements in a trigger action have access to column values (new and old) of the current row being processed.

Types of triggers:

When you define a trigger, you can specify the number of times the trigger action is to be executed; once for every row affected by the triggering statement (such as might be fired by an UPDATE statement that updates many rows), or once for the triggering statement, no matter how many rows it affects. The types of triggers are explained below.

Row triggers:

A row trigger is fired each time the table is affected by the triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the

UPDATE statement. If the triggering statement affects the rows, the trigger is not executed at all. Row trigger should be used when all the trigger action code depends on the data provided by the triggering statement or rows that are affected. e.g. if the trigger is keeping the track of the affected records.

Statement triggers:

A row trigger is fired once on behalf of the triggering statement, independent of the number of rows the triggering statement affects (even if no rows are affected). Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected. e.g. if the trigger makes the security check on the time or the user.

Before Vs. After triggers:

When defining a trigger you can specify the triggering timing, i.e. you can specify when the triggering action is to be executed in relation to the triggering statement. BEFORE and AFTER apply to both row and the statement triggers.

Before triggers:

BEFORE triggers execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situation:

- BEFORE triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using a BEFORE trigger, you can eliminate unnecessary processing of the triggering statement.
- BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

After triggers:

AFTER trigger execute the trigger action after the triggering statement is executed. These types of triggers are commonly used in the following situation:

- **AFTER** triggers are used when you want the triggering statement to complete before executing the trigger action.
- If a **BEFORE** trigger is already present, an **AFTER** trigger can perform different actions on the same triggering statement.

Combinations of triggers can be created as:

- **BEFORE** statement trigger:

Before executing the triggering statement, the trigger action is executed.

- **BEFORE** row trigger:

Before modifying each row affected by the triggering statement and before appropriate integrity constraints, the trigger is executed if the trigger restriction either evaluated to TRUE or was not included.

- **AFTER** statement trigger:

After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.

- **AFTER** row trigger:

After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the

current row if the trigger restriction either evaluates to TRUE or was not included. Unlike BEFORE row triggers, AFTER row triggers have rows locked.

Syntax for creating trigger:

```
CREATE OR REPLACE TRIGGER [schema.]triggername
    { BEFORE, AFTER }
    { DELETE, INSERT, UPDATE [OF column,.....] }
    ON [schema.]tablename
    [FOR EACH ROW [WHEN condition]]
DECLARE
    variable declaration;
    constant declaration;
BEGIN
    PL/SQL subprogram body;
EXCEPTION
    exception PL/SQL block;
END;
```

Keywords and parameters:

OR REPLACE	recreates the trigger if it already exists. You can use this option to change the definition of an existing trigger without dropping it.
schema	is the schema to contain the trigger. If you omit the schema, Oracle creates the trigger in your own schema.
triggername	is the name of the trigger to be created.
BEFORE	indicates that Oracle fires the trigger before executing the triggering statement.
AFTER	indicates that Oracle fires the trigger after executing the triggering statement.
DELETE	indicates that Oracle fires the trigger whenever a DELETE statement removes a row from the table.
INSERT	indicates that Oracle fires the trigger whenever a INSERT statement adds a row to the table.
UPDATE	indicates that Oracle fires the trigger whenever an UPDATE statement changes a value in one of the columns specified in the OF clause. If you omit the OF clause, Oracle fires the trigger whenever an UPDATE statement changes a value in any column of the table.
ON	Specifies the schema and name of the table on which the trigger is to be created. If you omit schema, Oracle assumes the table is in your own schema. You cannot create a trigger on a table in the schema SYS.

REFERENCING	specifies correlation names. You can use correlation names in the PL/SQL block and WHEN clause of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, you can use this clause to specify different correlation names to avoid confusion between table name and the correlation name.
FOR EACH ROW	designates the trigger to be a row trigger. Oracle fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the when clause. If you omit this clause, the trigger is a statement trigger.
WHEN	Specifies the trigger restriction. The trigger restriction contains a SQL condition that must be satisfied for Oracle to fire the trigger. This condition must contain correlation names and cannot contain a query. You can specify trigger restriction only for the row triggers. Oracle evaluates this conditions for each row affected by the triggering statement.
PL/SQL_block	is the PL/SQL block that oracle executes to fire the trigger.

Example: 1

Write a database trigger before insert for each row on the table route_detail not allowing transaction on Saturday and Sundays.

```
CREATE OR REPLACE TRIGGER sun_trig
before insert or update or delete on order_info
DECLARE

    shipping_date char;

BEGIN

    shipping_date := to_char(sysdate, 'dy');

    if shipping_date in ('sat', 'sun') then

        raise_application_error(-20001, 'try on any
        weekdays');

    end if;

END;
```

Example: 2

Write a database trigger after update for each row giving the date and day on which the update has been performed on the table fleet_header.

```
CREATE OR REPLACE TRIGGER fleet_trig after update on fleet_header
for each row
```

```
DECLARE

    sydate char(3);
```

```
date1 date;
```

```
BEGIN
```

```
sydate:= to_char(sysdate, 'dy');
```

```
date:= sysdate;
```

```
dbms_output.put_line(' the day and date of updation is ' ||  
sydate || ' ' || to_char(date1));
```

```
END;
```

Example: 3

Write a database trigger before delete for each row not allowing deletion and give the appropriate message on the table route_detail.

```
CREATE OR REPLACE TRIGGER route_trig before delete on  
route_detail for each row
```

```
BEGIN
```

```
raise_application_error(-20003, 'deletion not allowed');
```

```
END;
```

Example: 4

Write a database trigger before insert/update/delete for each

statement not allowing any of these operations on the table route_header on Mondays, Tuesdays and Wednesdays.

```
CREATE OR REPLACE TRIGGER header_trig before insert or update or delete on route_header
```

```
DECLARE
```

```
    sydate char(3);
```

```
BEGIN
```

```
    sydate:= to_char(sydate, 'dy');
```

```
    if sydate in ('mon', 'tue', 'wed') then
```

```
        raise_application_error(-20005, 'no  
        insertion/updation/deletion allowed');
```

```
    end if;
```

```
END;
```



Example: 5

Write a database trigger before insert on the item table for each row. If the itemid is 1000 and is also the value is entered by the user while inserting then raise an error using the raise_application_error and display a corresponding message.

```
CREATE OR REPLACE TRIGGER trig before insert on item for each row
```


DECLARE

item_ident item.itemid%type;

BEGIN

select itemid into item_ident from item where qty=4543;

If item_ident = 1000 then

raise_application_error (-20001, 'enter some other number');

end if;

END;

Example: 6

This trigger illustrates that :old and :new

CREATE OR REPLACE TRIGGER TempDelete BEFORE DELETE ON temp_table FOR EACH ROW

DECLARE

v_TempRec temp_table%ROWTYPE;

BEGIN

/* This is not a legal assignment, since :old is not truly a record. */

v_TempRec := :old;

```
/* We can accomplish the same thing, however, by assigning  
the fields individually. */
```

```
v_TempRec.char_col := :old.char_col;
```

```
v_TempRec.num_col := :old.num_col;
```

```
END TempDelete;
```

Example: 7

Create a transparent audit system for a table `client_master`. The system must keep track of the records that are being deleted or modified and when they have been deleted or modified. This trigger is fired when an update or delete is fired on the table `employee`. It first checks for the operation being performed on the table. Then depending on the operation being performed, a variable is assigned the value 'update' or 'delete'. The previous values of the modified record of the table `client_master` are stored into variables. The contents of these variables are then inserted into the audit table `auditclient`.

Infotech Pvt. Ltd.

```
CREATE TRIGGER audit_trail AFTER UPDATE OR DELETE ON  
client_master FOR EACH ROW
```

```
DECLARE
```

```
/* the value in the oper variable will be used to insert a value  
for the operation field in the auditemployee table */
```

```
oper varchar2(8);
```

```
/* These variables hold the previous value of client_no, name  
and bal_due */
```

```
client_no number(4);
```

```
name varchar2(20);
```

```
bal_due number(2);
```

```
BEGIN
```

```
/* if the records are updated in client_master table then oper  
is set to 'update' */
```

```
if updating then
```

```
    oper: = 'update';
```

```
end if;
```

```
/* if the records are declared in client_master table then oper  
is set to 'delete' */
```

```
if deleting then
```

```
    oper: = 'delete';
```

```
end if;
```

```
/* store the previous values of client_no, name and bal_due  
in the variables. These variables can be used to insert data in  
auditclient table*/
```

```
client_no:= :old.client_no;
```

```
name := :old.name;
```

```
bal_due := :old:bal_due;
```

```
insert into auditclient values (client_no, name, bal_due, oper,
```

```
sysdate);
```

```
END;
```

Example: 8

For every row updated on product_master we need to check that qty_on_hand must not be less than 0. Thus we need to write a database trigger before an update is fired on qty_on_hand.

```
CREATE TRIGGER check_qty_on_hand BEFORE UPDATE OF  
qty_on_hand ON product_master FOR EACH ROW
```

```
DECLARE
```

```
/* A variable that hold the new value of qty_on_hand */
```

```
new_qty number(8);
```

```
BEGIN
```

```
/* Assigning the new qty_on_hand to a variable */
```

```
new_qty := :new.qty_on_hand;
```

```
/* If new qty_on_hand is less than 0 then it should abort the  
operation and display an error message */
```

```
If new_qty < 0 then
```

```
raise_application_error(-20001, 'Quantity on Hand  
cannot be less than 0');
```

```
end if;
```

```
END;
```

DELETING a TRIGGER:

A trigger can be deleted from the database by using the following command:

```
DROP TRIGGER <triggername>;
```

Example: DROP TRIGGER t_itemid;



Top:5 Creating Objects

Abstract Datatypes

Abstract datatypes are user defined datatypes that consist of one or more subtypes. These datatypes can be used to describe the user data more conveniently. For example an abstract datatype for addresses may consist of the following columns:

Street VARCHAR2(20)

City VARCHAR2(20)

State VARCHAR2(20)

Pincode VARCHAR2(6)

so during the creation of the employee table we can create a column that uses the abstract datatypes for addresses and thus constraints the street, city and pincode columns that are a part of that datatype. The result of the abstract datatype enforces a standard representation of data.

Object Tables

In oracle 8, it is possible to create table which are based only on an abstract datatype. Such tables are referred to as object tables and their contents are known as row objects.

Object Views

Object views allows to add object oriented concepts on existing

relational tables. In the oracle object relational database, object views allows to retrieve, update, insert and delete relational data as if they were stored as object types. You can also define views that have columns which are object datatypes, such as objects, and collections (nested tables and varrays).

A common object example

Syntax:

```
CREATE OR REPLACE TYPE [Schema.]Type_name AS OBJECT
(column1 datatype,.....);
```

Where type_name is the name of the abstract datatype, and schema is the owner. The attributes of the object type are listed first in the form.

Example:

```
SQL> CREATE TYPE ADDRESS_TY as object (Street VARCHAR2(20),
City VARCHAR2(20), State VARCHAR2(20), Pincode VARCHAR2(6));
```

The create type command creates the abstract datatype ADDRESS_TY. This abstract datatype can be used when creating column in a table. The table with such a column is a column object table. Now the table EMP can be created as follows:

```
SQL> Create table EMP (EMPNo number(4), Name VARCHAR2(20),
Address ADDRESS_TY, Fax VARCHAR2(10), Remarks
VARCHAR2(100));
```

table created.

When the user describes the EMP table, the output will be

```
SQL> DESC EMP;
```

Name	Null	Type
-----	-----	-----
EMPNO		NUMBER(3)
NAME		VARCHAR2(20)
ADDRESS		ADDRESS_TY
FAX		VARCHAR2(10)
REMARKS		VARCHAR2(100)

When we query from the data dictionary tables directly to check table structure then...

SQL> select Column_name, Data_type from **USER_TAB_COLUMNS**
where table_name='EMP';

Column_Name	Data_Type
-----	-----
EMPNO	NUMBER
NAME	VARCHAR2
ADDRESS	ADDRESS_TY
FAX	VARCHAR2
REMARKS	VARCHAR2

In the above case we find that the datatype of address is the abstract

datatype ADDRESS_TY.

To get details on the ADDRESS_TY. We will have to query the data dictionary table **USER_TYPE_ATTRS**.

SQL> select Attr_name, Length, Attr_Type_Name from **USER_TYPE_ATTRS** Where type_name='ADDRESS_TY';

ATTR_NAME	LENGTH	ATTR_TYPE_NAME
-----	-----	-----
STREET	20	VARCHAR2
CITY	20	VARCHAR2
STATE	20	VARCHAR2
PINCODE	6	VARCHAR2

Dropping Abstract Data type:

syntax:

DROP TYPE type_name;

example:

DROP TYPE address_ty;

DML on abstract data type columns:

To insert rows into the emp table we cannot use the ordinary syntax. Oracle solves this problem by creating a method or function known as constructor method.

To insert record in emp table:

```
SQL> insert into EMP values (1, 'SMITH', ADDRESS_TY ('tagore road',
'rajkot', 'gujarat', '360002'), '123456', 'Good');
```

1 row created.

The constructor method is automatically created by Oracle.

To update record in emp table:

```
SQL> update EMP P set P..ADDRESS.CITY='gondal' where empno= 1;
```

1 row updated.

Querying the values from abstract datatypes:

```
SQL> select * from EMP;
```

EMPNO

NAME

ADDRESS(STREET,CITY,STATE,PINCODE)

FAX

REMARKS

1, SMITH, ADDRESS_TY ('tagore road', 'gondal', 'gujarat', '360002'),

'123456', 'Good'

To query particular column (attribute) from the abstract datatype

```
SQL> select Empno, P.Address.City from EMP P;
```

<u>EMPNO</u>	<u>ADDRESS.CITY</u>
1	gondal

To query from the abstract datatype, we will use alias for the table. We use alias because if we refer the attribute as address.city then oracle will misinterpret as table.columnname. So we use alias from specifying tablename.columnname.attributename

In a table an object can be stored as --> Row --> Column.

There are two types of objects:

ROW object:

A table is referred to as row object table, if every row in the table is a object and does not contain any other columns.

COLUMN object:

A table is referred to as column object table, if one of the column in the table is stored as an object. Till now we discussed column objects.

ROW object tables:

To create a row object table the syntax is:

```
CREATE TABLE table_name OF object_type;
```

Example:

```
SQL> CREATE TYPE EMP_TY AS object (empno NUMBER(3), Name  
VARCHAR2(20), address ADDRESS_TY);
```

type created.

To create a table of this type the syntax is:

```
SQL> create table EMP_OBJ_TAB of EMP_TY;
```

table created.

Now EMP_OBJ_TAB is a row object in which every row belongs to an object of type EMP_TY.

inserting into row object table:

```
SQL> insert into EMP_OBJ_TAB values (EMP_TY (1, 'JAMES',  
ADDRESS_TY('tagore road', 'rajkot', 'gujarat', '360002')));
```

selecting from Row object table:

In order to see this information we have to make use of the VALUE operator. The VALUE operator returns the object, rather than a list of the attributes.

```
SQL> select value(M) from EMP_OBJ_TAB M;
```


VALUE(M) (EMPNO,NAME,ADDRESS(STREET,CITY,STATE,PINCODE))

EMP_TY(1, 'JAMES', ADDRESS_TY('tagore
road','rajkot','gujarat','360002'))



Atmiya
— Infotech Pvt. Ltd.

Top:8 PL/SQL Tables

PL/SQL tables are similar to arrays in C. Sentacitcally, they are treated like arrays. However, they are implimented differently. In order to declare a PL/SQL table, we first need to define the table type, and then you declare a variable of these type, as the following declarative section illustates,

```
DECLARE
```

```
/* Define the table type. variables of these type can hold the character strings with a max of 10 characters each. */
```

```
TYPE t_charactertable IS TABLE OF VARCHAR2(10)
```

```
INDEX BY BINARY_INTEGER;
```

```
/* Declare a variable of these type. this is what actually allocates the storage */
```

```
V_characters t_charactertable;
```

The general syntax for defining a table type is

```
TYPE tabletype IS TABLE OF type INDEX BY BINARY_INTEGER;
```

Where tabletype is the name of the new type being defined, and type is a predefined scalar type, or a reference to a scalar type via % type.

In the previous example tablename is **t_character** and type is **varchar2(10)**. The following declarative section illustrates several different PL/SQL table types and variable declarations.

DECLARE

TYPE t_NameTable IS TABLE OF students.first_name%TYPE

INDEX BY BINARY_INTEGER;

TYPE t_datetable IS TABLE OF DATE INDEX BY BINARY_INTEGER;

v_name t_nametable;

v_dates t_datetable;

Note:

PL/SQL version 2 requires the **INDEX BY BINARY_INTEGER** clause as part of the table definition. This clause is not necessary for a version 8 table.

Once the type and variables are declared, we can refer to an individual element in a PL/SQL table by using the syntax:

tablename(index)

Where tablename is the name of table and index is either a variable if type BINARY_INTEGER or a variable or expression that can be converted to BINARY_INTEGER. Given the declaration for the different table types, we could continue the PL/SQL block with

BEGIN

v_names(1) := 'Scott';

```
v_Dates(-4) := SYSDATE - 1; /* SYSDATE -1 evaluates the time 24  
hours ago*/
```

```
END;
```

Note that a table reference, like a record variable reference, is an lvalue since it points to storage that has been allocated by the PL/SQL engine.



Top:6 Nested Tables

A nested table is, as its name implies, a table within a table. In this case, it is a table that is represented as a column within a another table. You can have multiple rows in the nested table for each row in the main table. For example, we can have all the details of the employee details in a dept as a column called emp_det..

Create or replace type EMP_TY as object (No NUMBER(3), name VARCHAR2(25), Sal NUMBER(10,2));

The EMP_TY datatype contains a record for each employee - empno, name and salary. To use this datatype as the basis for a nested table, you need to create a new abstract datatype:

Create Type EMP_NT as table of EMP_TY;

The **as table of** clause of this create type command tell oracle that you will be using this type as the basis for a nested table. The name of the type, EMP_TY has a plurazed root to indicate that it stores multiple rows and has the suffix 'NT' to indicate that it will be a nested table.

Now create the dept table as follows:

create table dept (deptno NUMBER(2), dname VARCHAR2(20), EMP_DET EMP_NT) Nested table EMP_DET store as EMP_NT_TAB;

When creating a table that includes a nested table, you must specify the name of the table that will be used to store the nested table's data. That is, the data for the nested table is not stored "in-line" with the rest of the table's data. Instead, it is stored apart from the main table. Thus, the data in the emp_det column will be stored in one

table, and the data in the name column will be stored in a separate table. Oracle will maintain pointers between tables. In this example, the "out-of-line" data for the nested table is stored in a table named EMP_NT_TAB:

nested table emp_det store as EMP_NT_TAB;

inserting records in Nested tables:

You can insert records into a nested table by using the constructor methods for its datatype. For the EMP_DET column, the datatype is EMP_NT; thus, you will use the EMP_NT constructor method. The EMP_NT type, in turn, uses the EMP_TY datatype. As shown in the following example, inserting a record into a dept table requires you to use both the EMP_NT and EMP_TY constructor methods. In the example, three EMP_DET are listed for the dept named accts:

```
insert into DEPT values (10, 'Accts', EMP_NT(
EMP_TY(1, 'SMITH', 2000),
EMP_TY(2, 'JOHN', 2500),
EMP_TY(3, 'PHILIPS', 3000)));
```

Deleting records from the nested table:

```
delete from the (select emp_det from dept where deptno= 10) where
empno= 1;
```

If you do not already know the datatype structure of the table, you will need to query the data dictionary before you can query the table. First, query USER_TAB_COLUMNS to see the definitions of the

columns:

```
select column_name, data_type from USER_TAB_COLUMNS where
table_name= 'DEPT';
```

<u>COLUMN_NAME</u>	<u>DATA_TYPE</u>
DEPTNO	NUMBER
DNAME	VARCHAR2
EMP_DET	EMP_NT

To see the datatype used as the basis for the nested table, query USER_COL_TYPES:

```
select col_type, elem_type_owner, elem_type_name, upper_bound,
length from USER_COL_TYPES where type_name= 'EMP_NT';
```

The output from USER_COL_TYPES shows the EMP_NT nested table is based on the EMP_TY abstract datatype. You can query USER_TYPE_ATTRS to see the attributes of EMP_TY:

```
select attr_name, length, attr_type_name from USER_TYPE_ATTRS
where TYPE_NAME= 'EMP_TY';
```

<u>ATTR_NAME</u>	<u>LENGTH</u>	<u>ATTR_TYPE_NAME</u>
NO	25	NUMBER
NAME	25	VARCHAR2
SAL	12	NUMBER

Querying nested tables:

Nested tables support a great variety of queries. However, you need to consider the nature of the table during your queries. A nested table is column within a table. To support queries of the columns and rows of a nested table, Oracle provides a new keywords, **THE**.

```
select NT.sal from THE(select EMP_DET from DEPT where
dname= 'Accts') NT;
```

In the example of the use of the **THE function**, the purpose of the queries and DML operations was to manipulate the data within the nested table. If you are trying to deal with the main table, then you need to take a slightly different approach in your queries.

For example, what if you need to perform an insert as select involving only the nested table portion of the DEPT table? That is, you will be inserting a new record into DEPT, but the nested table values will be based on values already entered for another record.

To solve this problem, oracle introduced two keywords: **cast** and **multiset**. The cast keyword allows you to "cast" the result of a query as a nested table. The multiset keyword allows the cast query to contain multiple records. You use cast and multiset together, as shown in the following example. Here's the inserted dept record, corrected to use the cast and multiset keywords:

```
insert into dept values (12, 'EDP', Infotech Pvt. Ltd.
```

```
cast(multiset(select * from THE(select EMP_DET from DEPT where
dname= 'Accts')) as EMP_NT));
```

Dropping the nested table:

to drop the nested table...

DROP the type EMP_TY;

DROP the type EMP_NT;

DROP the table DEPT;



Top:7 Varrays

A varying array allows you to repeating attributes of a record in a single row.

Creating a varying array

you can create a varying array based on either an abstract datatype or one of oracle's standard datatypes such as NUMBER. For example, you could create a varying array for storing the marks of the student called MARKS_VA.

Create or replace type MARKS_VA as varray(5) of number(3);

When this create type command is executed, a type named MARKS_VA is created. The as varray(5) clause tells oracle that a varying array is being created, and it will hold a maximum of five entries per record..

Now that you have created the varying array MARKS_VA, you can use that as part of the creation of either a table or an abstract datatype. To create a table with this varray..

Create table STUDENT (studno NUMBER(2) PRIMARY KEY, name VARCHAR2(25), marks MARKS_VA);

Atmiqa
Infotech Pvt. Ltd.

Describing the varying array:

The student table will contain one record for each STUDENT, even if that STUDENT has multiple marks. The multiple marks will be stored

in the marks column, using the MARKS_VA varying array. If you describe the student table, you will see that oracle internally stores varying array data using the RAW datatype:

```
SQL> desc STUDENT
```

<u>Name</u>	<u>Null?</u>	<u>Type</u>
StudNo	NOTNULL	NUMBER(2)
NAME	NOTNULL	VARCHAR2(25)
		ROW(200)

You can use the USER_TAB_COLUMNS data dictionary view to see information about the structure of the marks column:

```
select column_Name, Data_type from USER_TAB_COLUMNS where
table_name= 'STUDENT';
```

<u>COLUMN_NAME</u>	<u>DATA_TYPE</u>
STUDNO	NUMBER
NAME	VARCHAR2
MARKS	MARKS_VA

Atmiya
Infotech Pvt. Ltd.

from the USER_TAB_COLUMNS output, you can see that the marks column uses the marks_va varying array as its datatype. What kind of datatype is marks_va? you can query the USER_TYPES data dictionary view to see what kind of datatype MARKS_VA is:

```
select typecode, attribute from USER_TYPES where
type_name='MARKS_VA';
```

TYPECODE	ATTRIBUTE
COLLECTION	0

Inserting record into the varying array:

When a datatype is created, the database automatically creates a methods called a constructor method of the datatype. You need to use the constructor method when inserting records into columns that use as abstract datatype. Since a varying array is an abstract datatype, you will need to use constructor methods in order to insert records into tables that use varying arrays. Furthermore, since the varying array is itself an abstract datatype, you may need to nest calls to multiple constructor methods in order to insert a record into a table that uses a varying array.

The columns of the STUDENT table are studno, name and marks, the last of which is a varying array using the MARKS_VA datatype. The following command will insert a single record into the STUDENT table. In this example, the record will have a single name and studno column value and three marks values.

```
insert into STUDENT values (1, 'SMITH', MARKS_VA(78,33,56));
```

Selecting record from varying array:

When you insert records into a varying array, you need to make sure that the number of entries you insert into the varying array does not exceed its maximum. The maximum number of entries in the varying

array is specified when the varying array is created and can be queried from USER_COL_TYPES. You can also query the varying array directly to determine its maximum number of entries per row, called its LIMIT, and the current number of entries per row, called is COUNT.

You cannot, however, query the varying array directly via a select command. In order to retrieve the COUNT, LIMIT and data from a varying array, you need to use PL/SQL. To query the data from a varying array, you can use a set of nested cursor FOR loop, as shown in the example:

set serveroutput on

DECLARE

cursor STUDENT_CURSOR is select * from STUDENT;

STUDENT_rec STUDENT_CURSOR%rowtype;

BEGIN

FOR STUDENT_rec IN STUDENT_CURSOR

LOOP

dbms_output.put_line('Stud Name:' ||
STUDENT_rec.name);

FOR I IN 1..STUDENT_rec.marks.count

loop

dbms_output.put_line(STUDENT_rec.marks(i));

end loop;

END LOOP;

end;

/

The output of this PL/SQL script is

Stud Name: SMITH

78

33

56

PL/SQL procedure successfully completed.

VARRAY V/S NESTED TABLE

When storing data that is related to as table, you can choose one of three methods: a varying array, a nested table, or a separate table. If you have a limited number of rows, a varying array may be appropriate. As the number of rows increases, however, you may begin to encounter performance problems during the access of the varying array. The performance problems may be caused by characteristic of varying array; they cannot be indexed. Nested table, although they do a better job than varying arrays of supporting multiple columns, cannot be indexed either. Relational tables, however, can indexed. As a result, the performance of a collector may

worsen as it grows in size.

Varying array are further burdened by the difficult involved in querying data from them (via PL/SQL blocks instead of SQL extension). If varying arrays are not a good solution for a large set of related records, then should you use a nested table or a separate relational table? it depends. Nested tables and relational tables serve different purpose, so your answer depends on what you are trying to do with the data. The key differences are as follows:

Nested tables are abstract datatypes, crested via the create type command. Therefore, they can have methods associated with them. If you plan to attach methods to the data, then you should use nested tables instead of relational tables. Alternatively, you could consider using object views with methods.

Relational tables are easily related to other tables. If the data is possibly related to many other tables, then it may be best not to nest the data within one table. Storing it in its own table will give you the greatest flexibility in managing the data relations.



Top:1 Initialization Parameter

The initialization parameter file is used to establish specific database features each time an Oracle8 instance is started. By changing initialization parameter values, you can specify features such as:

- The amount of memory the database uses
- Whether to archive full online redo log files
- Which control files currently exist for the database

Where is the Initialization Parameter File Located?

The initialization parameter file is located in INIT%ORACLE_SID%.ORA file located in the \ORAWIN95\DATABASE directory. The computer that starts the instance must have access to the appropriate initialization parameter file. The Starter database in Personal Oracle8 uses the initialization parameter file located in \ORAWIN95\DATABASE.,

Initialization Parameter File: Definition

An initialization parameter file is an ASCII text file containing a list of parameters. Every database instance has a corresponding initialization parameter file and ORACLE_SID parameter. To allow initialization parameters to be unique to a particular database, each database normally has its own initialization parameter file.

The Sample Initialization Parameter File: Definition

The sample initialization parameter file is the initialization parameter file (INITORCL.ORA) used by the Starter Database in \ORAWIN95\DATABASE. Use this file as a model for creating a new Personal Oracle7 database.

Initialization Parameters to Check When Creating a New Database

Check the initialization parameters described in this section carefully if you decide to create a new database; they cannot be modified after database creation.

DB_NAME

Specifies the name of the database to be created. The database name is a string of eight characters or less. You cannot change the name of a database once it has been created.

CHARACTER_SET

Specifies the database NLS character set to use. This parameter can be set only when you create the database.

CONTROL_FILES

Designates the names and locations of all control files to be created and maintained. By default, Personal Oracle8 installs a single control file for the Starter Database, CTL1ORCL.ORA, in \ORAWIN95\DATABASE.

If you create your database with the CREATE DATABASE command, Personal Oracle8 creates the control file, CTL1sid.ORA, where sid is the SID of that database. To reduce the risk of losing the control file due to disk drive failure, use at least two control files, each located on a separate storage device. The size of Personal Oracle8 control files varies according to the complexity of your database structure. The

maximum size of a Personal Oracle8 control file is 2500 database blocks.

To edit the sample initialization parameter file


You edit the initialization parameter file so that you can customize Personal Oracle8 database functions.

- Open the sample initialization parameter file in any ASCII text editor.

The sample initialization parameter file contains alternative values for the initialization parameters. These values and the annotations are preceded by comment signs (#), which prevent them from being processed.

- To activate a particular parameter, remove the preceding # sign.
- De-activate a particular parameter by adding a # sign

Example:

- DB_BLOCK_BUFFERS is an initialization parameter that can be used to create small, medium, or large System Global Areas (SGAs), respectively. By default, the parameter to create a small SGA is activated: 

db_block_buffers = 200___# SMALL

db_block_buffers = 550___# MEDIUM

db_block_buffers = 3200___# LARGE

- To create a medium-sized SGA, deactivate the small parameter definition and activate the medium:

```
# db_block_buffers = 200__# SMALL
```

```
db_block_buffers = 550____# MEDIUM
```

```
# db_block_buffers = 3200__# LARGE
```



Top:2 Oracle files

There are three major sets of files on disk that compose a database.

- Database files
- Control files
- Redo logs

The most important of these are the database files where the actual data resides. The control files and the redo logs support the functioning of the architecture itself.

All three sets of files must be present, open, and available to Oracle for any data on the database to be useable. Without these files, you cannot access the database.

System and User Processes

For the database files to be useable, you must have the Oracle system processes and one or more user processes running on the machine. The Oracle system processes, also known as Oracle background processes, provide functions for the user processes—functions that would otherwise be done by the user processes themselves. There are many background processes that you can initiate, but as a minimum, only the PMON, SMON, DBWR, and LGWR must be up and running for the database to be useable. Other background processes support optional additions to the way the database runs.

In addition to the Oracle background processes, there is one user process per connection to the database in its simplest setup. The user must make a connection to the database before he can access any of the objects. If one user logs into Oracle using SQL*Plus, another user chooses Oracle Forms, and yet another user employs the Excel spreadsheet, then you have three user processes against the database—one for each connection.

Memory: Oracle uses the memory (either real or virtual) of the system to run the user processes and the system software itself and to cache data objects. There are

two major memory areas used by Oracle: memory that is shared and used by all processes running against the database and memory that is local to each individual user process.

System Memory: Oracle database-wide system memory is known as the SGA, the system global area or shared global area. The data and control structures in the SGA are shareable, and all the Oracle background processes and user processes can use them. The combination of the SGA and the Oracle background processes is known as an **Oracle instance**, a term that you'll encounter often with Oracle. Although there is typically one instance for each database, it is common to find many instances (running on different processors or even on different machines) all running against the same set of database files.

User Process Memory: For each connection to the database, Oracle allocates a PGA (process global area or program global area) in the machine's memory. Oracle also allocates a PGA for the background processes. This memory area contains data and control information for one process and is not shareable between processes.

Network Software and SQL*Net

A simple configuration for an Oracle database has the database files, memory structures, and Oracle background and user processes all running on the same machine without any networking involved. However, much more common is the configuration that implements the database on a server machine and the Oracle tools on a different machine (such as a PC with Microsoft Windows). For this type of client/server configuration, the machines are connected with some non-Oracle networking software that enables the two machines to communicate. Also, you might want two databases running on different machines to talk to each other—perhaps you're accessing tables from both databases in the same transaction or even in the same SQL statements. Again, the two machines need some non-Oracle networking software to communicate.

Whatever type of networking software and protocols you use to connect the machines (such as TCP/IP) for either the client/server or server-server setup mentioned previously, you must have the Oracle SQL*Net product to enable Oracle to interface with the networking protocol. SQL*Net supports most of the major networking protocols for both PC LANs (such as IPX/SPX) and the largest mainframes (such as SNA). Essentially, SQL*Net provides the software layer between Oracle and the networking software, providing seamless communication between an Oracle client machine (running SQL*Plus) and the database server or from one database server to another.

You must install the SQL*Net software on both machines on top of the underlying networking software for both sides to talk to each other. SQL*Net software options enable a client machine supporting one networking protocol to communicate with another supporting a different protocol.

Figure shows the role of SQL*Net in a client/server environment with two server database machines.



Oracle Files

In this part, I discuss the different types of files that Oracle uses on the hard disk drive of any machine.

Database Files

The database files hold the actual data and are typically the largest in size (from a few megabytes to many gigabytes). The other files (control files and redo logs) support the rest of the architecture. Depending on their sizes, the tables (and other objects) for all the user accounts can obviously go in one database file—but

that's not an ideal situation because it does not make the database structure very flexible for controlling access to storage for different Oracle users, putting the database on different disk drives, or backing up and restoring just part of the database.

You must have at least one database file (adequate for a small or testing database), but usually, you have many more than one. In terms of accessing and using the data in the tables and other objects, the number (or location) of the files is immaterial.

The database files are fixed in size and never grow bigger than the size at which they were created.

Control Files

Any database must have at least one control file, although you typically have more than one to guard against loss. The control file records the name of the database, the date and time it was created, the location of the database and redo logs, and the synchronization information to ensure that all three sets of files are always in step. Every time you add a new database or redo log file to the database, the information is recorded in the control files.

Redo Logs

Any database must have at least two redo logs. These are the journals for the database; the redo logs record all changes to the user objects or system objects. If any type of failure occurs, such as loss of one or more database files, you can use the changes recorded in the redo logs to bring the database to a consistent state without losing any committed transactions. In the case of non-data loss failure, such as a machine crash, Oracle can apply the information in the redo logs automatically without intervention from the database administrator (DBA). The SMON background process automatically reapplies the committed changes in the redo logs to the database files.

Like the other files used by Oracle, the redo log files are fixed in size and never grow dynamically from the size at which they were created.

Online Redo Logs

The online redo logs are the two or more redo log files that are always in use while the Oracle instance is up and running. Changes you make are recorded to each of the redo logs in turn. When one is full, the other is written to; when that becomes full, the first is overwritten, and the cycle continues.

Offline/Archived Redo Logs

The offline or archived redo logs are exact copies of the online redo logs that have been filled; it is optional whether you ask Oracle to create these. Oracle only creates them when the database is running in ARCHIVELOG mode. If the database is running in ARCHIVELOG mode, the ARCH background process wakes up and copies the online redo log to the offline destination (typically another disk drive) once it becomes full. While this copying is in progress, Oracle uses the other online redo log. If you have a complete set of offline redo logs since the database was last backed up, you have a complete record of changes that have been made. You could then use this record to reapply the changes to the backup copy of the database files if one or more online database files are lost.

Other Supporting Files

When you start an Oracle instance (in other words, when the Oracle background processes are initiated and the memory structures allocated), the instance parameter file determines the sizes and modes of the database. This parameter file is known as the INIT.ORA file (the actual name of the file has the Oracle instance identifier appended to the filename). This is an ordinary text file containing parameters for which you can override the default settings. The DBA is responsible for creating and modifying the contents of this parameter file. On some Oracle platforms, a SGAPAD file is also created, which contains the starting memory address of the Oracle SGA.

Top:3 Processes

We have defined a database as being "a bunch of programs that manipulate datafiles." it's now time to discuss the programs; we prefer to call them processes since every time a program starts against the database, it communicate with oracle via a process. Later in this chapter we talk about support processes required to run the oracle database. There are two types of oracle processes you should know about: user and server.

User (client) processes:

User processes work on your behalf, requesting information from the server processes. Example of user processes are oracle forms. These are common tools any user of the data within the database uses to communicate with the database.

SERVER processes:

Server processes take requests from user processes and communicate with the database. Through this communication, user processes work with the data in the database.

A good way to think of the client/server process is to imagine yourself in a restaurant. You, the customer, communicate to the waiter who take your order. That person then communicates the request to the kitchen. The kitchen staff's job is to prepare the food, let the waiter know when it is ready, and stock inventory. The waiter then delivers the meal back to you. In this analogy, the waiter represents the client process, and the kitchen staff represents the server processes.

DATABASE SUPPORT PROCESSES

As we started before, server processes take requests from the user (client) processes; they communicate with the database on behalf of user processes. Let's take a look at a special set of server processes that help the database operate.

Database Writer (DBWR):

The database writer is mandatory process that writes changed data blocks back to the database files. It is one of the only two processes that are allowed to write to the datafiles that make up your oracle database. On certain operating system, oracle allows you to have multiple database writers. This is done for performance reasons.

Checkpoint (CKPT):

Checkpoint is an optional process. When users are working with an oracle database, they make requests to look at data. That data is read from the database files and put into an area of memory where users can look at it. Some of these users eventually make changes to the data that must be recorded back onto the original datafiles. Earlier in the chapter, we talked about redo logs and how they record all transactions. When the redo logs switch, a checkpoint occurs. When this switch happens, oracle goes into memory and writes any dirty data blocks information back to disk. In addition, it notifies the control file of the redo log switch. These tasks are normally performed by the log writer (lgwr) discussed in the next section. For performance reasons, the DBA can make changes to the database to enable the checkpoint process. This process's sole job is to take the checkpoint responsibility away from the log writer.

Log Writer (LGWR):

The log writer is a mandatory process that writes redo entries to the redo logs. Remember, the redo logs are a copy of every transaction that occurs in the database. This is done so that oracle is able to recover from various types of failure. In addition, since a copy every transaction is written in the redo log, Oracle does not have to spend its resources constantly writing data changes back to the datafiles immediately. This results in improved performance. The log writer is the only process that writes to the redo logs. It is also the only process in an oracle database that reads the redo logs.

System Monitor (SMON):

System monitor is a mandatory process that performs any recovery that is needed at startup. In the parallel server mode it can also perform recovery for a failed database on another computer. Remember, the two databases share the same datafiles.

Process Monitor (PMON):

Process monitor is a mandatory process that performs recovery for a failed user of the database. It assumes the identity of the failed user, releasing all the database resources that user was holding, and it rolls back the aborted transaction.



Archiver (ARCH):

Archiver is an optional process. As we discussed earlier in the "Redo logs" section, the redo logs are written to in a sequential manner. When a log fills up, there is a log switch to the next available redo log. When you are running the database in ARCHIVELOG mode, the

database goes out and makes a copy of the redo log. This is done so that when the database switches back to this redo log, there is a copy of the contents of this file for recovery purpose. This is the job of the Archiver process. Similar to a copy machine, it makes a copy of the file.

Lock (LCKn):

Lock is an optional process. When you are running the oracle database in the parallel server mode, you will see multiple lck processes. In parallel server mode these locks help the database communicate.

Recoverer (RECO):

You only see this optional process when the database is running the oracle-distributed option. The distributed transaction is one where two or more locations of the data must be kept in synch. For example, you might have one copy of data in Boston and another copy of the data in Mexico City. Let's say that while updating the data, the phone line to Mexico goes down due to a severe rainstorm, and a mudslide washes the phone line away. It is the job of the reco process to resolve transaction that may have completed in Boston but not in Mexico City. These transaction are referred to as in-doubt until they are resolved by this reco process.

Atmiya

Infotech Pvt. Ltd.

Dispatcher (Dnnn):

Dispatcher are optional background processes, present only when a multithreaded server configuration is used. At least one dispatcher process is created for every communication protocol (i.e. TCP/IP, SNA) in use (D000,&ldots;.,Dnnn). Each dispatcher process is responsible

for routing requests from connected user processes to available shared server processes and returning the responses back to the appropriate user processes.

Memory Structure

Now let us talk about how the client and server processes communicate to each other and themselves through memory structures. Just as the name implies, this is an area of memory set aside where processes can talk to themselves or to other processes.

Oracle uses two types of memory structures: the system global area, or SGA (think of it as an old - fashioned telephone line or the conference calling option on your phone) and program global area, or PGA (think of this as intercom system).

SYSTEM GLOBAL AREA

SGA is a place in memory where the Oracle database stores pertinent information about itself. It does this in memory, since memory is the quickest and most efficient way to allow processes to communicate. This memory structure is then accessible to all the user processes and server processes. Figure shows this clearly&ldots;

Since the SGA is the mechanism by which the various client and server processes communicate, it is important that you understand its various components. The Oracle server SGA is broken into the following key components.

Data Buffer Cache:

The data buffer cache is where oracle stores the most recently used blocks of database data. In other words, this is your data cache. When you put information into database, it is stored in data blocks. The data buffer cache is an area of memory in which oracles places these data blocks so that a user process can look at them. Before any user process can look at a piece of data, the data must first reside in the data cache. There is a physical limit on the size of the data buffer cache. Thus, as oracle fills it up, it leaves the hottest blocks in the cache and moves out the cold blocks. It does this via the least recently used (LRU) algorithm.

An important point to clarify: if a client process needs information that is not in the cache, the database goes out to the physical disk drive, reads the needed data blocks, then places them in the data buffer cache. It does this so that all other client and server processes get the benefit of the physical disk read.

Dictionary Cache (Row Cache):

A dictionary cache contains rows out of the data dictionary. The data dictionary contains all the information oracle needs to manage itself, such as what users have access to the oracle database, what database objects they own, and where those objects are located.

Redo log buffer:

Remember that another common name for the online redo logs is the transaction log. So, before any transaction can be recorded into the redo log, it must first reside in the redo log buffer. This is an area of

memory set aside for this event. Then the database periodically flushes this buffer to the online redo logs.

Shared SQL pool:

Think of the shared SQL pool as your program cache. This is where all your programs are stored. Programs within an oracle database are based on a standard language called SQL. This cache contains all the parsed SQL statements that are ready to run.

To summarize, the SGA is the great communicator. It is the place in memory where information is placed so that client and server processes can access it. It is broken up into major areas: the data cache, the redo log cache, the dictionary cache, and the shared SQL cache. We call SQL cache the sqlarea; these two terms can be used synonymously.

PROGRAM GLOBAL AREA

PGA is an area of memory that is used by a single oracle process. The program global area is not shared; It contains data and control information for a single process. It contains information such as process session variables and internal arrays. Like an intercom system in your home, the various parts of the process can communicate to each other but not to the outside world.

Top:4 Tablespace

For management, security, and performance reasons, the database is logically divided into one or more tablespaces that each comprise one or more database files. A database file is always associated with only one tablespace.

A tablespace is a logical division of a database comprising one or more physical database files.

Every Oracle database has a tablespace named SYSTEM that has the very first file of the database allocated to it. The SYSTEM tablespace is the default location of all objects when a database is first created. The simplest database setup is one database file in the SYSTEM tablespace (simple, but not recommended).

Typically, you create many tablespaces to partition the different parts of the database. For example, you might have one tablespace for tables, another to hold indexes, and so on, and each of these tablespaces would have one or more database files associated to them.

When you create objects that use storage in the database (such as tables), you should specify the tablespace location of the object as part of the CREATE statement for the object. Only system tables should occupy storage in the SYSTEM tablespace. The system tables are tables such as tab\$, col\$, ind\$, fet\$, and other internal tables.

Objects such as synonyms and views do not take up storage within the database other than the storage in the data dictionary table for their definitions, along with the definitions for all other types of objects.

Tablespaces can be added, dropped, taken offline and online, and associated with additional database files. By adding another file to a tablespace, you increase the size of the tablespace and therefore the database itself.

You cannot drop the SYSTEM tablespace; this would destroy the database because the system tables are there. You also cannot take the SYSTEM tablespace offline.

The create tablespace command allows one or more files to be assigned immediately to the tablespace. It also specifies a default space for any tables created without an explicit storage clause mentioned in the create table statement. This is the basic format for the create tablespace command:

```
CREATE TABLESPACE talbot datafile '/db01/oracle/GBT/talbot.dbf' size 1000k
```

```
default storage (initial 1M next 1M minextents 1 maxextents 100  
pctincrease 0)
```

```
permanent;
```

Note: The permanent keyword in the create tablespace command tells oracle that you will be storing permanent objects (such as tables) in the tablespace. If the tablespace is only used for temporary segments, then you can specify the temporary keyword instead. The default value is permanent.

The initial file assigned to this tablespace is included in the command, as well as its size in the bytes, not blocks. The number of bytes is an integer and can be followed by a K (to multiply by 1024- about a thousand) or an M (to be multiply by 1048576- about a million). Default storage sets up the storage that a table will get if storage is

not specified in the create table statement. Here, the initial default extent is 1M bytes (not blocks) and the next (incremental) extent is 1M bytes.

Minextents allows you to set aside additional extents beyond the first at the time a table is created. These additional extents will not necessarily be contiguous (physically adjacent) with the initial extent, or with each other, but the space will at least be reserved.

Maxextents is the limit of additional extents allowed. You can specify maxextents unlimited, in which case there is no limit to the number of extents allowed for the table or index.

pctincrease is a growth factor for extents. When set to a non-zero value, each incremental extent will be the specified percentage larger than the one before it. This has the effect of reducing the number of extents, and noncontiguous space, used by a table that grows large. However, it causes the space allocated to the table to grow exponentially. If the data volume in the table grows at a constant rate, you should set pctincrease to 0.

The default values for storage are operating-system-specific. The minimum and maximum values for each of these options are available in the alphabetical reference under create table and "storage". These options may be changed with the alter tablespace command. The create table command for the LEDGER table looks like this:

```
CREATE TABLE Ledger (ActionDate Date, Action varchar2(8), Item
varchar2(30), Quantity Number, QuantityType varchar2(10), Rate
Number, Amount number(9,2), Person varchar(25)) tablespace
TALBOT;
```

In this form, the Ledger table will inherit the default storage definitions of the TALBOT tablespace. To override these default, the storage clause is used in the create table command:

```
CREATE TABLE Ledger (ActionDate Date, Action varchar2(8), Item
```

varchar2(30), Quantity Number, QuantityType varchar2(10), Rate Number, Amount number(9,2), Person varchar(25))

tablespace TALBOT

storage (intial 512K next 512K minextents 2 maxextents 50
pctincrease 0) ;

If you use temporary tables, you can create a tablespace dedicated to their storage needs. use the create temporary tablespace command to support this special type of table.

Example:

To create new tablespace named Trans_tab

CREATE TABLESPACE Trans_tab

DATAFILE 'data1' SIZE 50M

DEFAULT STORAGE (

INITIAL 50K

NEXT 50K

MINEXTENTS 2

MAXEXTENTS 50

PCTINCREASE 0)

OFFLINE;

To add datafile of tablespace named Trans_tab

```
ALTER TABLESPACE Trans_tab ADD DATAFILE 'data2' SIZE 1M;
```

To Enlarge Datafile size

```
ALTER DATABASE DATAFILE 'data2' RESIZE 100M;
```

TYPES OF TABLESPACE:

SYSTEM tablespace:

The system tablespace is a required part of every Oracle database. This is where oracle stores all the information it needs to manage itself, such as names of tablespaces and what datafiles each tablespace contains.

TEMP tablespace:

The temp tablespace is where oracle stores all its temporary tables. This is the database's whiteboard or scratch paper. Just as you sometimes need a place to jot down some numbers so you can add them up, Oracle also has a need for some periodic disk space. In the case of a very active database, you might have more than one temp tablespace; for example, TEMP01, TEMP02 and TEMP03.

TOOLS tablespace:

The tools tablespace is where you store the database objects needed to support tools that you use with your database, such as oracle reports, with its own set of tables. Like any oracle application, oracle reports needs to store tables in the database. Most DBAs place the

tables needed to support tools in this tablespace.

USERS tablespace:

The users tablespace holds users' personal information. For example, when you are learning how to use oracle, you might want to create some database object. This is where the DBA will typically let you place your database object. Another example is you might keep different database for your individual clients.

DATA and INDEX tablespace:

In some installation, you see tablespace names such as DATA01, DATA02 ... which represent different places to hold data. In other sites, you might see DATA-1, INDEX01 etc. Think of a database index as the index in a book: to find a particular reference in the book, you look in the index for its location, rather than reading the whole book from page one. Indexes are a special database object that enable oracle to quickly find data stored within a table.

In oracle, looking at every row in a database is called a full table scan. Using an index search is called an index scan. Many other shops name their tablespace after the application data they hold. For example, in a hospital, the tablespace names might be lab_system or research.

ROLLBACK tablespace:

All oracle database need a location to store undo information. This tablespace, which holds your rollback segments, is typically called rollback or rbs. One of the primary reasons you use a database management system such as oracle is for its ability to recover from incomplete or aborted transaction as part of the core functionality.

Top:5 Oracle Blocks

Oracle "formats" the database files into a number of Oracle blocks when they're first created—making it easier for the RDBMS software to manage the files and easier to read data into the memory areas.

These blocks are usually 1 KB (the default for PC-DOS systems), 2 KB (the default for most UNIX machines and VAX VMS), 4 KB (the default for IBM mainframes), or larger. For a 50MB database file, there would be 25,600 Oracle blocks assuming a block size of 2 KB (50 MB/2 KB).

The block size should be a multiple of the operating system block size. Regardless of the block size, not all of the block is available for holding data; Oracle takes up some space to manage the contents of the block. This block header has a minimum size, but it can grow.

These Oracle blocks are the smallest unit of storage. Increasing the Oracle block size can improve performance, but you should do this only when the database is first created.

When you first create a database, it uses some of the blocks within the first file, and the rest of the blocks are free. In the data dictionary, Oracle maintains a list of the free blocks for each data file in each tablespace.

Each Oracle block is numbered sequentially for each database file starting at 1. Two blocks can have the same block address if they are in different database files.

Do not modify the Oracle block size once you've created the database.

ROWID

The ROWID is a unique database-wide physical address for every row on every table. Once assigned (when the row is first inserted into the database), it never changes until the row is deleted or the table is dropped.

The ROWID consists of the following three components, the combination of which uniquely identifies the physical storage location of the row.

- Oracle database file number, which contains the block with the row
- Oracle block address, which contains the row
- The row within the block (because each block can hold many rows)

The ROWID is used internally in indexes as a quick means of retrieving rows with a particular key value. Application developers also use it in SQL statements as a quick way to access a row once they know the ROWID.

Free Space and Automatic Compaction

When a database file is first created or added to a tablespace, all the blocks within that file are empty blocks that have never been used. As time goes by, the blocks within a database file are used by a segment (table), or they remain free blocks. Oracle tracks the file's free blocks in a list in the data dictionary. As you create and drop tables, the free space becomes fragmented, with free space in different parts of the database file. When the free blocks are scattered in this way, Oracle has no way to automatically bring the free storage together.

When two fragments of free space are physically next to each other in the database file, the two smaller fragments can be compacted together into one larger fragment, which is recorded in the free space

list. This compacting reduces the overhead when Oracle actually needs the free space (when a table wants to allocate another extent of a certain size, for example). The SMON background process performs this automatic compaction.



Top:6 Import, Export

Export and import empower the DBA and application developers to make dependable and quick copies of Oracle data. Export (invoked using the command exp80) makes a copy of data and data structures in an operating system file. Import (invoked using the command imp80) reads files created by export and places data and data structures in Oracle database files. These two handy utilities are used primarily for the following reasons:

- As part of backup and recovery procedures
- For moving data between different instances of Oracle. You may export data from your production database and use import to move all or part of the data in the export file into your development database.
- To move all or part of a user's data from one tablespace to another. Suppose userA has data residing in two tablespaces, tablespaceA and tablespaceB. You could move all of the data out of tablespaceB and place it in tablespaceA using export and import. There must be enough space in tablespaceA to accommodate the data being moved from tablespaceB. Import itself will not always add additional disk space to tablespaceA if it is not already there.
- When the need arises to rebuild an existing database, export and import are the only way to preserve the current database data before it is recreated.

Similarities between Export and Import

Export and import behave in the same way, and learning one of the two will put you more than 80 percent of the way to mastering both products. These two tools are similar in the following ways:

- Both can be run interactively or can read run-time parameters from a file.
- Both accept keywords (parameters started with keyword_value=) or positional parameters (those that mean something based on their order on the command line).
- Both work with oracle read only copies of data and data structures.
- Both are used to move data among different oracle accounts and hardware platforms.

Difference between Export and Import

Even though export and import are similar, there are some differences. Some parameters are used only with export, others only with import. For example, the fromuser and touser parameters are only used with import. Likewise, the compress parameter is only coded when using export.

- Import may report on a wide assortment of oracle errors, since it is creating and loading data into oracle database file.
- Export is sensitive to the amount of free space on a disk drive to which the export file is being written.

Even though these differences exist, export and import methods and modes of operation are the same.

Methods Of Operation

The methods we are about to discuss apply to export and import. Learning and experimenting with different methods is part of your job as a DBA. In this section, we will discuss:

- Invoking interactive export with no parameters
- Invoking interactive import with no parameters

When running export and import interactively, Oracle presents you with a list of questions. With export, the answers you give to those questions affect what is written to the export file. With import, these answers affect what data is retrieved from the export file. When export and import are parameter driven, you instruct oracle what you want written to or read from the export file based on values supplied with these parameters.

Interactive Export: Invoking with No Parameters

The next listing shows an example of the dialog between export and the user when export is invoked without any parameters.

```
D:\ORANT\BIN> exp80
```

```
Export: Release 8.0.3.0.1 - Production on Tue Nov 19 11:32:25 1998
```

```
Copyright © Oracle Corporation 1979, 1994. All rights reserved.
```

Username: scott/tiger

Connected to: oracle8 server release 8.0.3.0.1 - Production release

With the distributed, heterogeneous, replication, objects and parallel query options

PL/SQL release 3.0.3.1 - Production

Enter array fetch buffer size: 4096 > 102400

Export file: EXPDAT.DMP >

(2) U (sers), or (3) T(ables): 2(U) > 3

Export table data (yes/no): yes > Y

Compress extents (yes/no): yes > Y

Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character set

About to export specified tables via conventional path &ldots;

Table (T) or Partition (T:P) to be exported: (return to quit) > emp

..exporting table emp 14 rows exported

Table to be exported: (return to quit) >

Export terminated successfully without warnings.

You are asked to supply the information in the meaning and response column of table 1 before oracle commences the export.

<u>Prompt received from oracle</u>	<u>Meaning and response</u>
Username	The user name and password of the person running export.
Enter array fetch buffer size	The size of the chunk of memory to use as a work area enters values between 10240 and 10M.
Export file	The name of the export file. Default to expdat.dmp but can be changed.
(2) U(sers) or (3) T(ables)	Oracle wants to know which method you wish to run. You will be asked for names of one or more users if you choose 2 or the names of one or more of your own tables if you answer 3.
Export table data (yes/no)	Instructions on what to write to the export file. Oracle always writes SQL statements necessary to create exported objects to the export file. Answering yes to this prompt tells oracle to export the data in the objects as well.
Compress extents (yes/no)	Oracle wants to know if the create table statements written to the export file should include an initial space request capable of holding all the existing table data.
Table(T) or Partition(T:P) to be exported	The name of the table or partition name of a partitioned table to be exported.

Table: 1 Dialog when exp is invoked with no parameters

Interactive Import: Invoking with No Parameters

To give you a flavor of Oracle7 and Oracle8 import, the next listing illustrates an Oracle7 dialog and figure 2 shows invoking import with Oracle8 without any parameters.

```
D:\ORANT\BIN> imp80
```

```
Export: Release 7.1.4.0.0 - Production on Tue Nov 19 11:32:25 1998
```

```
Copyright © Oracle Corporation 1979, 1994. All rights reserved.
```

```
Username: scott/tiger
```

```
Connected to: oracle7 server release 7.1.4.0.0 - Production release
```

```
With the distributed, heterogeneous, replication, objects and parallel  
query options
```

```
PL/SQL release 3.0.3.1 - Production
```

```
Import file: EXPDAT.DMP >
```

```
Enter insert buffer size (minimum is 4096) 30720 >
```

```
Export file created by EXPORT:V07.01.04
```

```
List contents of import file only (yes/no): no>
```

```
Ignore create error due to object existence (yes/no): yes> Yes
```

Import grants (yes/no): yes> Yes

Import table data (yes/no): yes> Yes

Import entire export file (yes/no): yes> Yes

.importing scott's objects into scott

..importing table "emp" 14 rows imported

Import terminated successfully without warnings.

You are asked to supply the information in the meaning and response column of table 2 before oracle commences the import. If you answer no to the ignore create errors due to object existence prompt, then Oracle will not bring the table data in for tables that exist. Nearly all the time, you will answer yes to this prompt.

<u>Prompt received from oracle</u>	<u>Meaning and response</u>
Username	The user name and password of the person running import.
Import file	The name of the file you want import to read. Default to expdat.dmp but can be changed.
Enter insert buffer size (minimum is 4096)	The size of the chunk of memory to use as a work area enters values between 10240 and 10M.

Export file	The name of the export file. Default to expdat.dmp but can be changed.
List contents of import file only (yes/no)	Oracle will list the SQL statements written to the import file if you answer yes. If you answer no, import will bring the data and data definitions into the database.
Ignore create error due to object existence (yes/no)	Oracle wants to know what it should do when it encounters an object in the import file that already exists. If you answer yes, oracle ignores the fact that an object exists and brings in its data anyway. Answering no causes oracle to report an error and then move on to the next object when it encounters an object that already exists.
Import grants (yes/no)	Oracle wants to know whether to run the grant statements written to the import file after an object is imported.
Import table data (yes/no)	Oracle wants to know if it should bring in the table data (yes) or just run the SQL statements to create objects (no).
Import entire export file (yes/no)	Oracle wants to know if the complete file or only specified portions should be imported. If you answer yes, the import starts at once. If you answer no, Oracle will ask questions about what you wish to import.

Table: 2 Dialog when imp is invoked with no parameters



Atmiya
Infotech Pvt. Ltd.

Top:7 SQL * Loader

Oracle Loader reads files and places the data in the Oracle database based on the instructions it receives from a control file. The control file tells Oracle loader where to place data, and it describes the kinds of data being loaded into Oracle. It can filter records (i.e. not load records that do not conform), load data into multiple tables at the same time, and generate a unique key or manipulate data before placing it in an Oracle table.

Moving data out of your existing system into Oracle is a two-step process. First, you create a text file copy of your existing data using your current software, then you load the data from the text file into Oracle using Oracle Loader.

Sometimes Oracle Loader is called SQL *Loader; we use the two product names synonymously. Over the past few years, oracle has taken the "SQL*" prefix off most of its products and replaced it with the company name. Since you may use oracle loader to move data into one or more tables, throughout this chapter we will also use the words "table" and "tables" interchangeably.

RUNNING ORACLE LOADER - ORACLE8

By the end of this section, you will know how to invoke Oracle Loader and the most common parameters supplied when oracle loader is invoked.

To invoke Oracle Loader, enter the command `sqlldr80`. If you do not

include any parameter, you are given online help.

There is a long list of parameters; however, most sessions will be started with commands similar to `sqlldr80 username control=cfile.ctl`. In the following section, we will discuss `userid` and `control` plus a few more keywords from the previous listing which, if used, influence how oracle loader runs. Afterwards, we will present a few examples and show the command lines to accomplish the desired results. You can provide keywords and values on the command line, in any order.

USERID

`userid` must be the username and password for an account that owns the table being loaded or that has access to someone else's table for loading. If you omit the password, Oracle will prompt you for it as the session begins. Along with the `control` parameter, this is one of the two required inputs to oracle loader.

Normally, rather than include the keyword `userid` on the command line, you include an oracle user name and let oracle loader prompt for the password. Thus the command `sqlldr80 username` is the same as `sqlldr80 userid=username`; in both cases, you are prompted for the control filename, then the account password.

CONTROL

`Control` names a file that maps the format of the input datafile to the oracle table. The format of the control file discussed in the "Oracle Loader Control File" section later in this chapter. If you do not include the `control` keyword when calling oracle loader, you are prompted, as in the following;

Sqlload scott

Control = cfile

Password:

Sql*loader: Release 8.0.3.1 - Production on Sat Mar 11 13:21:54
2000

Copyright © oracle corporation 1994, 1996. All rights reserved.

PARALLEL

Running Oracle loader in parallel can speed up the time oracle loader takes to complete and, in situations where there are large amounts of input data, shrink runtimes dramatically. Invoking Oracle Loader with parallel=true runs multiple sessions, loading data simultaneously into the same table. When using this option, the target tables must have no indexes.

The parallel sessions data is merged by oracle in a number of temporary tables, then inserted as a single unit of data. This parameter defaults to false; a parallel session is started by coding parallel=true as loader is invoked.



DIRECT

When using a direct load, data is assembled in memory in the same format as oracle data block, and the data block is copied directly into data blocks in the target datafile. This parameter defaults to false; to run a direct load, code direct=true. The direct load runs faster than conventional loads, especially when accompanied by parallel=true. If

you choose `direct=true` for an oracle loader session, and for some reason the load aborts, any indexes on the target table will be left in direct load state and will have to be dropped and recreated.

SKIP

This parameter defaults to 0. If you code a positive integer value, Oracle Loader skips over the specified number of rows and starts loading with the record immediately after the specified number. This may prove useful in large loads. For example, you might browse the log file for a load that was supposed to move 1,000,000 rows into oracle and find that the table has run out of space and received only 275,000 rows. Rather than redo the load from scratch, for the next session you could include the parameter `skip=275000`.

LOAD

This parameter defaults to all. If you code a positive integer value, Oracle will load that exact number of rows, then quit. You may want to use this if you want a subset of a very large amount of data moved into a development or test database for a system on its way to production.

Atmiya

Infotech Pvt. Ltd.

LOG and BAD

These two parameters are not normally mentioned on the command line. They inherit their filenames from the name of the control file used for the session. The command `sqlldr80 control=cfile.ctl` would log the session to `cfile.log` and write records that contain bad data into the file `cfile.bad`.

DISCARD

Sometimes you place one or more conditions on the input data; in this case, records that do not pass the condition(s) are discarded. If you include this parameter followed by a filename, these discarded records are written to the specified file.

Let us put it all together with examples,

Example #1

Pretend you want to invoke an Oracle Loader session using the parameters and parameter values from the following table. Say it's a large load, and you want to run multiple load sessions at the same time.

<u>Component</u>	<u>Value</u>
UserName	scott
Password	tiger
Control File	cfile.ctl
Load	All records
Parallel	yes

Direct	No
--------	----

The command to accomplish the load as described in the previous table would be `sqlldr80 scott/tiger control=cfile paralled=true`. Note that there is no filename extension on the file `cfile`, so Oracle loader assumes the control filename is `cfile.ctl`. By excluding the parameters `load` and `direct`, they assume their defaults.

Example #2

This time we want to load an additional 1,000 records into a table. Records number 1 to 499 were loaded in a previous session. To speed things up, we wish to use the direct load path with parallel sessions.

<u>Component</u>	<u>Value</u>
------------------	--------------

UserName	scott
----------	-------

Password	tiger
----------	-------

Control File	cfile.ctl
--------------	-----------

Skip	500
------	-----

Direct	Yes
--------	-----

Load	1000
------	------

Parallel	yes
----------	-----

The command to accomplish the load described in the previous table would be `sqlldr80 scott/tiger control=cfile.crl parallel=true direct=true skip=500 load=1000`. Note the filename extension on the file `cfile` since it is not the oracle loader `.ctl` default.

Example #3

For this example, suppose you wanted to load records number 501 to 520 using the direct path load mechanism. This also illustrates how oracle loader prompts for missing components of a parameter (i.e. oracle loader expects a user name and password after the userid parameter, but we only supply the user name).

<u>Component</u>	<u>Value</u>
UserName	scott
Control File	cfile.ctl
Skip	500
Direct	No
Load	20

Notice in this example that we will not supply a password or the name of the control file when invoking Oracle Loader. The command would be `sqlldr80 scott direct=true load=20 skip=500`. Oracle will prompt for the missing parameters, as shown in the following listing:

Control= cfile

Password:

SQL* Loader: Release 8.0.3.1 - Production on Thu Dec 29 18:08:43 2001

Copyright © Oracle corporation 1994, 1996. All rights reserved.

Notice that we enter cfile for the control filename, and oracle loader assumes its extension will be .ctl.

ORACLE LOADER CONTROL FILE

We will now move on to building the control file. The control file sets up the environment for a loader session: it tells loader where to find the input datafile, which Oracle table the data should be loaded into, what, if any, restriction to place on which data is loaded, and how to match the input data to the columns in the target table. When you're just getting started with Oracle Loader, the control file is the area that can cause the most problems. If the control file has errors, the Oracle Loader session stops immediately. Let's look at the four main parts of an Oracle Loader Control file, as shown below:

-- Notice comment operator

load data -- part 1

infile 'MyData.dat' -- part 2

into table Stud -- part 3

(First_name position(01:14) char, -- part 4
surname position(15:28) char,
classn position(29:36) char,
d_date position(37:42) date 'YYMMDD')

We will now discuss the four parts shown in above example, focusing on the format and the instructions each part gives to oracle loader.

Part:1 Load Data

The keywords load data start most oracle loader control files, regardless of the contents of the rest of the control file. They serve as a starting point for the rest of the control file, and nothing else. Think of these two keywords as the title page of a book.



Part:2 Infile

This line names the input file. Notice in above example how the input filename is enclosed in single quotes. Through the quotes are not mandatory here, they are required in some situations. For example, in UNIX, let's say the input file description line is infile \$HOME/Mydata.dat. The dollar sign causes the errors to be raised.

Part:3 Into Table

This line instructs Oracle Loader where to place the data as it is loaded into Oracle. There are four modifiers to the into table portion of the control file:

1. Insert is the default and expects the table to be empty when the load begins.
2. Append adds new rows to the table's existing contents.
3. Replace deletes the rows in the table and loads the new rows.
4. Truncate behaves the same as replace.

Normally, you will not code the insert qualifier with oracle loader, since it is the default. The most common error you may encounter is when you try to load data into a table that contains rows, and you have not included append, replace, or truncate on the into table line. If this happens, Oracle Loader returns the following error:

SQL* Loader: Release 8.0.3.1 - Production on Thu Dec 19 22:17:50 1999

Infotech Pvt. Ltd.

Copyright © Oracle corporation 1994, 1996. All rights reserved.

SQL * Loader - 601: For INSERT option, table must be empty. Error on MyData.

Part:4 Column and Field Specifications

This section of the control file matches characters in the input file to the database columns of the target table. There are four parts to each

line in this specification: the column name in the target table, the keyword position, the start and end character positions, and the data type of those characters in the input file.

Loading date fields into oracle deserves special mention. Oracle dates default to the format DD-MON-YY. If the data in the input file is not in this format, you must tell Oracle how the dates appear in that file.

ORACLE LOADER OUTPUTS

As oracle loader runs, it writes a number of files that are used to figure out how successful the load was. By default, Oracle Loader writes a log file and based on the success or failure of the load and the parameters used when it is invoked, may write a bad and a discard file. Unless specified otherwise, these two extra files have the same name as the control file with the extensions .bad and .dsc respectively.

Log File - shows complete statistic of loader, if incomplete load then gives rejected row explanation, and also load statistics like&ldots;

Table Authors:

2903 Rows successfully loaded.

3 Rows not loaded due to data errors.

0 Rows not loaded because all WHEN clauses were failed.

0 Rows not loaded because all fields were null.

You should verify that all the records in the input file were read by summing the numbers in the load statistics section of the log file. The result should equal the number of lines in the input file.

Bad File - This file is only written when one or more rows from the input file are rejected.

Discard File - The discard file is not created unless the discard= parameter is used when invoking oracle loader.



Top:8 Instance Architecture

INTRODUCTION

When someone refers to the Oracle database, they are most likely referring to the entire Oracle database management system(DBMS). But as an Oracle professionals, you must recognize the difference between the database and the Instance - a distinction often confusing to non-Oracle administrators. In this chapter you explore the structure and configuration of the Oracle instance, and continue your exploration of the internals of the oracle Relational Database Management System (RDBMS) in the next chapter by looking in-depth at the oracle database. (To avoid confusion, the term RDBMS is used to describe the entire data Management server consisting of the Oracle database and instance).The creation of The instance is automatic and behind the scenes. The details of how and when this happens are also discussed.

DEFINING THE INSTANCE

To provide the degree of service, flexibility and performance that Oracle clients expect, much of the work done by the database is handled by a complex set of memory structures and operating system processes called the instance. Every Oracle database has an instance associated with it, and unless oracle Parallel Server option is implemented, a database is mounted by only one instance. The organization of the instance allows the RDBMS to service many types of transactions from multiple users simultaneously, while at the same time providing first class performance, fault tolerance, data integrity

and security.

The instance structure is loosely styled after UNIX's implementation of the multitasking operating system. Discrete processes perform specialized tasks within the RDBMS that work together to accomplish the goals of the instance. Each process has a separate memory block that it uses to store private variables, address stacks and other runtime information. The processes use a common shared memory area in which to do their work—a section of memory that can be written to and read from at the same time by many different programs and processes. This memory block is called the system global area (SGA).

NOTE

Because the SGA resides in a memory segment, it is also often referred to as the Shared Global Area.

You might think of the background processes as the hands of the database, handling its components directly; and you might think of the SGA as the brain, indirectly coordinating the hands in their information and storage retrieval as necessary. The SGA takes part in all information and server processing that occurs in the database.

NOTE

Single Oracle configuration, such as Personal Oracle Lite; do not use multiple process to perform database functions. Instead, all database functions are contained within one Oracle process. For this reason, single user it also known as single process Oracle.

Atmiya

Infotech Pvt. Ltd.

CREATING THE INSTANCE

Opening an Oracle database involves three steps:

1. Creating the Oracle instance (nomount stage).

2. Mounting the database by the instance (mount stage).

3. Opening the database (open stage).

The Oracle instance is created during the nomount stage of database startup. When the database passes through the nomount phase, the init.ora file is read, the background processes are started, and the SGA is initialized. The init.ora file defines the configuration of the instance, including such things as the size of the memory structures and the number and type of background processes started. The instance name is set according to the value of the ORACLE_SID environment variable and does not have to be the same as the database name being opened (but for convenience, it usually is). The next stage the database passes through is called the mount stage. The value of the control file parameter of the init.ora file determines the database the instance mounts. In the mount stage, the control file is read and accessible, and queries and modifications to the data stored within the control file can be performed. The final stage of the database is when it is opened. In this stage the database files whose names are stored in the control file are locked for exclusive use by the instance, and the database is made accessible to normal users. Open is the normal operating state of the database. Until a database is open, only the DBA is capable of accessing the database, and only through the Server Manager utilities.

In order to change the operating state of the database, you must be connected to the database as internal, or with SYSDBA privileges. When you go from a shutdown state to an open state you can step through each operating state explicitly, but when you shut down the database you can only go from the current operating state to a complete shutdown. For example, you can issue the STARTUP NOMOUNT command instance the Server Manager utility. This will put your database into the nomount stage. Next, you can issue ALTER DATABASE MOUNT or ALTER DATABASE OPEN to step through the operating stages. At any operating state, if you issue a SHUTDOWN command you will completely shutdown the database. For example,

you can't go from an open state to a mount state.

An instance that does not have a database mounted is referred to as idle-it uses memory but does not do any work. An instance can only attach to one database, and unless Parallel Server is being used, a database only has one instance assigned to it. The instance is the brain of the data management system-it does all the work while the database stores all the data.

THE COMPONENTS OF THE ORACLE INSTANCE

Figure - 1 is a visual representation of the Oracle instance. Explanations of the different components follow.



FIGURE-1 The Oracle Instance is a complex interaction of memory and

back ground processes.

Many parameters and techniques exist to help you configure the instance to best support your applications and requirements. Configuring the instance objects for peak performance is, in most cases, a trial and error procedure-you can start with likely parameters values, but only time and monitoring give you the best possible mix of all settings and variables.

Configuring instance parameters involves changing the necessary init.ora parameter and bouncing (stopping and starting) the database. There are numerous init.ora parameters, and many of these are undocumented. Although you should not change or add unfamiliar initialization parameter, you can reference the internal x\$ksppi table to view all the possible initialization parameters for a database. The ksp pinm and ksp pdesc columns give you the parameter name and a brief description of the parameter, respectively.

NOTE

Manipulating initialization file parameters without a clear understanding of the possible consequences is dangerous! There are many parameters that exist for pure diagnostic reasons, which can leave your database instance an unsynchronized or corrupted state. Undocumented parameters are named with a leading underscore. Do not add or change keys or values instance the init.ora file unless you are confident instance what you are doing!

For the most part, instance configuration is primarily concerned with the objects instance the SGA, and you find most of your database configuration and tuning time spent with these structures. However, there are issues and configuration options with the background process that also need to be addressed, and you explore those parts of the instance as well.

THE SYSTEM GLOBLE AREA(SGA)

The SGA is the primary component of the instance. It holds all the memory structures necessary for data manipulation, SQL statement parsing, and redo caching. The SGA is shared, which means that multiple processes can access and modify the data contained within it at the same time. All database operations use structures contained instance the SGA at one point or another. As mentioned instance the previous section, the SGA is when the instance is created, during the nomount stage of the database, and is deallocated when the instance is shut down.

The SGA consists of the following:

- Shared pool
- Database buffer cache
- Redo log buffer
- Multithreaded server(MTS) structures

These are explained instance the following sections.

THE SHARED POOL

The shared pool (see figure 2) contains the library cache, the dictionary cache and server control structures (such as database character set). The library cache stores the text, parsed format, and execution plan of SQL statements that have been submitted to the RDBMS, as well as the headers of PL/SQL packages and procedures that have been executed. The dictionary cache stores data dictionary rows that have been used to parse SQL statements.



FIGURE-2 The shared pool caches information used when parsing and executing SQL statements.

The Oracle server uses the library cache to improve the performance of SQL statements. When a SQL statement is submitted, the server first checks the library cache to see if an identical statement has already been submitted and cached. If it has, Oracle uses the stored parse tree and execution path for the statement, rather than rebuilding these structures from scratch. Although this might not affect the Performance of ad hoc queries, applications using stored code can gain significant performance improvements by utilizing this feature.

NOTE

For a SQL statement to use a previously cached version, it must be identical in all respects to the cached version, including punctuation and letter case-upper versus lower. Oracle identifies the statements by applying a hashing algorithm to the text of the statement-the hash value generated must be identical for both the current and cached statements in order for the cached version to be used.

The library cache contains both shared and private SQL areas. The shared SQL area contains the parse tree and execution path for SQL statements, whereas the private SQL area contains session specific

information, such as bind variables, environment and session parameters, runtime stacks and buffers, and so on. A private SQL area is created for each transaction initiated, and it is deallocated after the cursor corresponding to that private area is closed. The number of private SQL areas a user session can have open at one time is limited by the value of the `OPEN_CURSORS` `init.ora` parameter. Using these two structures, The Oracle server can reuse the information common across all executions of an SQL statement, while session specific information to the execution can be retrieved from the private SQL area. It is important to note however, that the session-specific information contained within the User Global Area (UGA) of a user's Process Global Area (PGA), including the private SQL areas, is held instance the SGA only with a Multithreaded Server (MTS) instance. Otherwise, it is held instance the dedicated server.

NOTE

An application that does not close cursors as they are used continues to allocate more and more memory for the application, instance part because of the private SQL areas allocated for each open cursor.

The private SQL area of the library cache is further divided into persistent and runtime areas. Persistent areas contain information that is valid and applicable through multiple executions of the SQL statement, whereas the runtime area contains data that is used only while the SQL statement is being executed.

The dictionary cache holds data dictionary information used by the RDBMS engine to parse SQL statements. Information such as segment information, security and access privileges, and available free storage space is held instance this area.

The size of shared pool is determined by the `init.ora` parameter `SHARED_POOL_SIZE`. This value is specified in bytes. You must set this value high enough to ensure that enough space is available to load and store PL/SQL blocks and SQL statements. The shared pool becomes fragmented over time from the loading and unloading of data

objects, and errors can occur if there is not enough contiguous free space in the pool to load an object. You can solve this problem in the short term by issuing the SQL command `ALTER SYSTEM FLUSH SHARED_POOL`, but if you are regularly encountering shared pool errors during database operation, you have to increase the shared pool size.

THE DATABASE BUFFER CACHE

The operation of the database buffer cache is one of the biggest factors affecting overall database performance. The buffer cache is made up of memory blocks the same size as the Oracle blocks. All data manipulated by Oracle is first loaded into the buffer cache before being used. Any data updates are performed on the blocks in memory. For this reason, it is obviously very important to size the buffer cache correctly. Memory access is hundreds of times faster than disk access, and in an OLTP environment, most of your data operations should take place completely in memory, using database blocks already loaded into the cache.

The Oracle RDBMS swaps data out of the buffer cache according to a Least Recently used (LRU) list. The LRU list keeps track of what data blocks are accessed and how often. When a block is accessed or retrieved into the buffer cache, it is placed on the Most Recently Used (MRU) end of the list. When the Oracle server needs more space in the buffer cache to read a data block from area disk, it accesses the LRU list to decide which blocks to swap out. Those blocks at the far end of the MRU side are removed first. This way, blocks that are frequently accessed are kept in memory.

NOTE

The exception to the LRU loading rule is that data accessed through a full table scan is automatically placed at the bottom of the LRU list. This behavior can be overridden by specifying the table as `CACHE`.

Buffer blocks that have been modified are called dirty and are placed

on the dirty list. The dirty list keeps track of all data modifications made to the cache that have not been flushed to disk. When Oracle receives a request to change data, the data change is made to the blocks in the buffer cache and written to the redo log, and then the block is put on the dirty list. Subsequent access to this data reads the new value from the changed data in the buffer cache.

The Oracle server used deferred, multiblock writes to lessen the impact of disk I/O on database performance. This means that an update to a piece of data does not immediately update the data in the data files. The RDBMS waits to flush changed data to the data files until a predetermined number of blocks have been changed, space needs to be reclaimed from the cache to load new data, a checkpoint occurs, or DBWR times out. When DBWR is signaled to perform a buffer cache write, it moves a group of blocks to the data files.

The key to configuring the buffer cache is to ensure that the correct amount of memory is allocated for optimal caching of data. This doesn't necessarily mean allocating all possible memory resources to the buffer cache; however, as in most computer application, there is a point of diminishing returns with increased memory allocation. There is a point of diminishing returns in adding memory to obtain an increasingly better cache hit ratio. The memory you are allocating to the buffer cache could be better used in other places, such as other oracle memory structures.

Infotech Pvt. Ltd.

Two initialization parameters determine the size of the buffer cache-DB_BLOCK_SIZE and DB_BLOCK_BUFFERS. The DB_BLOCK_SIZE parameter is used during database creation to set the size of the Oracle block, which is explained in chapter 7, "Exploring the Oracle Environment." The DB_BLOCK_BUFFERS parameter determines the number of blocks to allocate to the buffer cache. Multiplying DB_BLOCK_SIZE * DB_BLOCK_BUFFERS gives you the total amount of memory (in bytes) of the buffer cache.

THE REDO LOG BUFFER

The redo log buffer is used to store redo information in memory before it is flushed to the online redo log files. It is a circular buffer, which means that it fills from top to bottom and then returns to the beginning of the buffer. As the redo log buffer fills, its contents are written to the online redo log files.

The redo log buffer is sized by means of the LOG_BUFFER initialization parameter. The value is specified in bytes and determines how much space is reserved in memory to cache redo log entries. If this value is set too low, processes contend with each other and the Log Writer (LGWR) (explained later in this chapter) process reading and writing to the buffer, possibly causing performance problems. This is, however, a rarity in all but the most active of databases and can be monitored using the V\$SYSSTAT view. Query V\$SYSSTAT\$ for the value field with the field name equal to redo log space requests. This indicates the time user processes spent waiting for the redo log buffer.

To enforce the sequential nature of the redo log writes, the Oracle server controls access to the buffer using a latch. A latch is a lock by an Oracle process on a memory structure—similar in concept to a file or row lock. A process must hold the redo allocation latch to be capable of writing to the redo log buffer. While one process holds the allocation latch, no other process can write to the redo log buffer using the allocation latch.

Infotech Pvt. Ltd.

The Oracle server limits the amount of redo that can be written at one time using the value of the initialization parameter LOG_SMALL_ENTRY_MAX_SIZE. This parameter is specified in bytes, and the default value varies depending on OS and hardware. For server with multiple CPUs, the Oracle server does not allow redo entries needing space greater than the value of the LOG_SMALL_ENTRY_MAX_SIZE parameter to be written using the redo allocation latch. Instead, processes must hold a redo copy latch. The number of redo copy latches available is equal to the LOG_SIMULTANEOUS_COPIES initialization parameter. The default for LOG_SIMULTANEOUS_COPIES is the number of CPUs in the system. Using redo copy latches, multiple processes can simultaneously write

to the redo log buffer.

You can monitor the redo allocation and copy latches using the V\$LATCH dynamic performance view. (See Chapter 18, "Tuning Memory," for more information on tuning the redo latches.)

THE ORACLE BACKGROUND PROCESS

Within any given second, an Oracle database can be processing many rows of information, handling hundreds of simultaneous user requests, and performing complex data manipulations, all while providing the highest level of performance and data integrity. To accomplish these tasks, the Oracle database divides the grunt work between a number of programs, each of which operates in large part independently of one another and has a specific role to play. These programs are referred to as the Oracle background processes, and are the key to effectively handling the many operational stresses placed upon the database. A complete understanding of the background processes and the tasks they perform helps you analyze performance problems, pinpoint bottlenecks, and diagnose trouble spots in your database.

NOTE

On NT servers, the background processes are implemented as multiple threads to the Oracle Service. This allows the Oracle process to use shared memory address space more efficiently and results in less context changes by the NT OS to handle Oracle operations.

The Oracle background processes are as follows:

- SMON and PMON
- DBWR
- LGWR
- Dnnn

- ARCH
- CKPT RECO SNPn
- LCKn
- Pnnn
- Snnn

Also of interest are the user and server processes, which handle user transactions against the database, and the Parallel Query (Pnnn) processes, which performs parallel query operations for the database. Although these are not classified as Oracle background processes, it is important to understand the role they play in the Oracle environment. A discussion of each of these processes follows.

SMON AND PMON

For one reason or another, connections into the Oracle database might crash, hang or otherwise abnormally terminate. End users might shut down their client machine without logging out of the database application, or a network or system failure unrelated to the database might cause an automated database job to fail. Oracle server must be capable of transparently resolving the problems resulting from these kinds of failures.

Together SMON and PMON are the background processes responsible for automatically resolving database system problems. PMON, the Process Monitor, performs automatic cleanup of terminated or failed processes, including clearing the orphaned sessions left from an abnormally terminated process, rolling back uncommitted transactions, releasing the locks held by disconnected processes, and freeing SGA resources held by failed processes. It also monitors the server and dispatcher processes, automatically restarting them if they fail.

SMON, the System Monitor, plays a smaller but nonetheless very important role. Upon database startup, SMON is the process that performs automatic instance recovery. If the last database shutdown was not clean, SMON automatically rolls forward the operations that were in progress, and rolls back the uncommitted transactions. SMON is also the process that manages certain database segments, reclaiming temporary segment space no longer in use, and automatically combining contiguous of free space in data files.

NOTE

SMON only combines free space in tablespaces where the default storage parameter-used when creating tablespaces or tables-`pctincrease` is not 0. Set the `pctincrease` to at least 1 if you want SMON to automatically handle this operation.

SMON and PMON are two of the required background processes. The database does not start if either of these two processes fail on startup.

DBWR

DBWR, or the mandatory Database Writer process, is responsible for writing the dirty blocks from the database buffer cache to the data files. Rather than write out each block as it is modified, DBWR waits until certain are met, and instance batch reads dirty list and flushes all the blocks found in it to the data files. This provides a high level of performance and minimizes the extent to which the database is I/O bound.

DBWR flushes the dirty blocks when

- A checkpoint occurs,
- The dirty list reaches a specified length, determined by half of the value for the `init.ora` parameter `DB_BLOCK_WRITE_BATCH`,

- The number of used buffers reaches the value of the init.ora parameter DB_BLOCK_MAX_SCAN, or
- A DBWR timeout occurs (about every three seconds).

Configuring the DBWR background process is fairly straightforward, and the default values for settings are in many cases sufficient for a small- or medium-sized database. Larger, more active or specialized databases often have special needs, however, that force the manual configuration of some of the DBWR parameters.

In most installations, there is one DBWR process to handle all the write activity of the database. You can, however, start more than one DBWR process if you find DBWR is incapable of keeping up with the demands of the database. The init.ora parameter DB_WRITERS, which defaults to 1, sets the number DBWR processes that are created at startup. In most cases, the decision to use more than one DBWR process is only made when the OS on which Oracle server is being run does not support a synchronous I/O. If this is the case, multiple DBWR processes should be created. Some suggest that you should use as many DBWR processes as physical disks used to store data files on; other suggest setting the number to be equal to the number of data files in the database. Experiment with adding and subtracting DBWR processes until your best performer is reached.

The database buffer cache also uses latches to control access to the memory structure. The LRU latch controls the replacement of buffers in the buffer cache. In

Very active servers with multiple CPUs, there might be contention for these latches. If this happens, set the parameter DB_BLOCK_LRU_LATCHES to a number equal to the number of latches to create for the buffer cache. This number cannot be greater than twice the number of CPUs and is automatically set to the number of CPUs in the system.

Another init.ora parameter that affects DBWR behavior is

DB_BLOCK_CHECKPOINT_BATCH. This parameter sets the maximum number of blocks DBWR writes with each checkpoint (see the section below on the checkpoint process, CKPT, for more information). By increasing this number, you can decrease the number of times DBWR must flush the buffer cache. Increasing this number too much, however, might cause an unacceptable delay when DBWR finally does flush the buffer.

A third parameter to keep in mind is **DB_BLOCK_CHECKSUM**. This is a Boolean parameter that, when enabled, causes each database block to be written with a checksum value attached. When the block is subsequently read, the checksum is computed and compared with that stored in the database. If the values are different, an error is raised. This is a valuable parameter when troubleshooting data corruption problems, but should not be enabled all the time because of the performance hit taken from calculating and storing the checksum for each I/O operation.

LGWR

LGWR, or Log Writer, is the fourth and final mandatory background processes. Log Writer is the process that writes redo log entries from the buffer in the SGA to the online redo log files. LGWR performs this write when commit occurs, the inactivity timeout for LGWR is reached, the redo log buffer becomes one-third full, or DBWR completes a flush of the data buffer blocks at a checkpoints. LGWR also handles multiple user commits simultaneously, if one or more users issue a commit before LGWR has completed flushing the buffer on behalf of another user's commit.

It is important to note that Oracle does not regard a transaction as being complete until LGWR has flushed the redo information from the redo buffer to the online redo logs. It is LGWR's successful writing of the redo log entries into the online redo logs, and not the changing of the data files, which returns a success code to the server process.

The LGWR process is rarely a source of performance problems for the

database. In addition, few options are available for custom configuration of LGWR. Most of the configuration necessary involves the redo log buffer and memory structures supporting that buffer, rather than the LGWR process itself.

The exception to this, however, deals with the secondary task of the LGWR process, performing the operations necessary to conduct a database checkpoint. The LGWR performs this task unless the CKPT process is activated. A checkpoint causes process and I/O time to be spent by both LGWR and DBWR. More frequent checkpoint decrease the recovery time necessary if database failure occurs, as well as reducing the work necessary to perform each individual checkpoint. You must weigh both of these factors when deciding on the correct checkpoint interval. Several parameters govern the occurrence of database checkpoints.

LOG_CHECKPOINT_INTERVAL and LOG_CHECKPOINT_TIMEOUT are two parameters that can change the checkpoint interval-that is, the time or situation necessary for a database checkpoint to be triggered. LOG_CHECKPOINT_INTERVAL, when set, causes a checkpoint to be triggered when a number of OS blocks (not Oracle blocks) is written to redo. LOG_CHECKPOINT_TIMEOUT, when set, causes a checkpoint to occur after interval (in second) specified for the parameter.

These parameter should be used with care. If LOG_CHECKPOINT_INTERVAL is used, it should be set to that the number of OS blocks that trigger a checkpoint are relative to the size of the redo log group. Remember that when a redo log group fills, a checkpoint is triggered. Be careful that you do not set a LOG_CHECKPOINT_INTERVAL value that causes more checkpoints to occur than necessary or forces checkpoints when they are not needed. For example, consider a redo log group of 3 MB, and a LOG_CHECKPOINT_INTERVAL set to 2.5 MB. When 2.5 MB are written to the redo logs, the LOG_CHECKPOINT_INTERVAL value causes a checkpoint to occur. In addition, when the redo log group fills (after only 0.5 MB have been written), another checkpoint occurs. In essence, two checkpoints will occur right after each other.

You can also control the frequency of checkpoints by sizing your redo log groups accordingly. If you size your logs so that a log switch occurs every hour, you only have one checkpoint an hour from redo log group switches. However, if your groups are sized so that checkpoints occur every five minutes, you waste a lot of process and I/O time performing the related checkpoints.

A final parameter that is of use is the Boolean `LOG_CHECKPOINTS_TO_ALERT`. This places a stamp in the `alert.log` file for the database whenever a checkpoint occurs and is valuable when trying to pinpoint the exact checkpoint interval.

DISPATCHER PROCESS (Dnnn)

As mentioned previously, server processes can be either dedicated to a user process or shared among user processes. Using shared servers requires configuring the Multithreaded Server (MTS), as discussed in Chapter 37, "Installing and Configuring the OAS". When using shared server processes, at least one dispatcher process must be present, and more can be present depending on the needs of the environment. The dispatcher process passes user requests to the SGA request queue and returns the server responses back to the correct user process.

The number of dispatcher processes is controlled using a number of `init.ora` parameters. The `MTS_DISPATCHERS` parameter specifies the protocol the dispatcher uses as well as the number of dispatchers to start that use protocol. Multiple protocol groups can be configured using multiple `MTS_DISPATCHERS` lines. A typical `MTS_DISPATCHER` line might look like this:

```
MTS_DISPATCHERS = "tcp, 4"
```

```
MTS_DISPATCHERS = "spx, 2"
```

Multiple protocol groups can also be configured within the same

MTS_DISPATCHERS parameter, like so

```
MTS_DISPATCHERS = ("tcp, 4", "spx, 2")
```

The MTS_MAX_DISPATCHERS parameter controls the maximum number of dispatcher processes allowed for the RDBMS. (See Chapter 34, "The Advanced Security Option," for more details on configuring the MTS services.)

ARCH

The archiver process is responsible for copying full online redo logs to the archived redo log files. This only occurs when the database is operating in ARCHIVELOG mode. Archivelog mode is required for point-in-time recovery. It also permits "hot" backups. While the archiver is copying the redo log, no other processes can write to the log. This is important to keep in mind, because of the circular nature of the redo logs. If the database needs to switch redo logs but the archiver is still copying the next log in the sequence, all database activity halts until archiver finishes. Also note that if ARCH is for some reason unable to finish copying the log, it wait until the error stopping it from finishing the write is resolved.

It is important to note that the ARCHIVE_LOG_START parameter in the init.ora file must be set to TRUE for ARCH to automatically start when a database opens. Placing the database in archivelog mode and don't automatically start the ARCH process, the database hangs when all online redo logs fill, waiting for you to manually archive the online logs.

CKPT

CKPT, the checkpoint process, is an optional background process that performs the checkpoint tasks that LGWR would normally perform—namely updating the data file and control file headers with the current version information. Enable this process to reduce the amount of work on LGWR when there are frequent checkpoints occurring, frequent log

switches, or many data files in the database.

Setting the `CHECKPOINT_PROCESS` parameter to `TRUE` enables the CKPT process. All other parameters related to checkpoints that are described also hold true when the CKPT process is running.

CAUTION

With Oracle 8.x, the `CHECKPOINT_PROCESS` parameter is obsolete because it is already integrated into the RDBMS with a setting of `TRUE`. If you include it in Oracle 8.x `init.ora` file, your instance will fail to start.

RECO

RECO, the recovery process, is responsible for recovering failed transactions in distributed database systems. It is automatically started when the database is configured for distributed transactions (that is, when the `DISTRIBUTED_TRANSACTIONS` `init.ora` parameter is set to a value greater than zero.) The RECO process operates with little or no DBA intervention when an in-doubt transaction occurs in a distributed system. The RECO process attempts to connect to the remote database and resolves the in-doubt transaction when a database connection is successful. (See Chapter 40, "Distributed Database Management," for more information on RECO and the two phase commit.)

SNPs

SNPs, the snapshot process, handle the automatic refreshing of database snapshots and runs the database procedures scheduled through the `DBMS_JOB` package. The `init.ora` parameter `JOB_QUEUE_PROCESS` sets how many snapshot processes are started, and `JOB_QUEUE_INTERVAL` determines how long (in seconds) the snapshot processes sleep before waking to process any pending jobs or transactions.

LCKn

In a parallel server environment, multiple instances mount one database. The lock process is responsible for managing and coordinating the locks held by the individual instances. Each instance in parallel server installation has 1-10 lock processes assigned, and each instance must have the same number. This process has no purpose in a non-parallel server environment. See Chapter 39, "Parallel Server Management," for more information on the Lock background process.

Pnnn

Parallel query processes are named Pnnn. The Oracle server starts and stops query processes depending on database activity and your configuration of the parallel query option. These processes are involved in parallel index creations, table creations, and queries. These are always as many processes started as specified in the `PARALLEL_MIN_SERVERS` parameter; and there are never more than as specified by `PARALLEL_MAX_SERVERS`.

For more information on configuring the parallel query processes, see Chapter 38, "parallel Query Management."

USER AND SERVER PROCESSES (Snnn)

Applications and utilities access the RDBMS through a use process. The user process connects to a server process, which can be dedicated to one user process or shared (with MTS) among many. The server process parses and executes SQL statements that are submitted to it and returns the result sets back to the user process. It is also the process that reads data blocks from the data files into the database buffer cache.

Each user process is allocated a section of memory referred to as the Process Global Area (PGA). The contents of the PGA differ depending on what type of connection is made tom the database. When a user

process connects to the database via a dedicated server process, user session data, stack space, and cursor state information is stored in the PGA. The user session data consists of security and resource usage information; the stack space contains local variables specific to the user session; and the cursor state area contains runtime information for the cursor, including rows returned and return codes. If, however, the user process connects through a shared server process, the session and cursor state information is stored within the SGA. Although this does not increase the memory requirements for the database as a whole, it does require a larger SGA to hold the extra session information.

ANATOMY OF A TRANSACTION

To gain a better understanding of how all the preceding components of the instance interact, look at a typical transaction as it moves through the instance structures.

A transaction begins when a user session connects to a server session using an SQL*Net driver. This connection can be dedicated connection with its own server process or a shared connection handled through a dispatcher process. The server session hashes the SQL statement passed to it and compares that hash number with the hash numbers of statements already saved in the shared SQL area. If an exact duplicate of the statement is found in the shared pool, the parsed form of the statement and the execution plan that are already stored are used. If a match is not found in the shared pool, the server session parses the statement.

Next, the server session checks to see whether the data blocks necessary to complete the transaction are already stored in the database buffer cache. If the blocks are not in the cache, the server session reads the necessary blocks from the data files and copies them into the cache. If the transaction is a query, the server session returns the result of the query to the user session (performing the data block read and copy as many times as necessary to return all data).

For a transaction that modifies data, there are more steps involved. For this example, assume the transaction is an update. After the necessary data blocks are read into the buffer cache, the blocks in memory are modified. Modifying cached blocks marks them as dirty, and they are placed on the dirty list. Redo information is also generated, and is stored in the redo log cache.

The transaction continues until one of several things happens. If the transaction is relatively short lived (for example, an update to one row of sales data), it finishes and the user commits, which signals LGWR to flush the redo log buffer to the online redo log files. If the transaction is fairly long and complex, any of the following can happen:

- The redo generated causes the redo log buffer to become one-third full. This triggers a redo log buffer flush by LGWR.
- The number of blocks placed on the dirty list reaches a threshold length. This triggers DBWR to flush all the dirty blocks in the database buffer cache to the data files, which in turn also causes LGWR to flush the redo log buffer cache to disk.
- A database checkpoint occurs. This triggers a database buffer cache flush, as well as a redo log cache flush.
- The number of available free buffers in the buffer cache drops below the threshold value. This also causes a database buffer cache flush.
- An unrecoverable database error occurs. This forces the transaction to be terminated and rolled back and an error reported back to the server session.

While the transaction is processing with redo being generated to the redo cache and flushed, the online redo logs gradually fill. When the current log fills, LGWR begins writing to the next log group, while ARCH copies the redo log to disk or tape. Because the transaction

never records as successful until all redo log information is written from the redo log buffer to the online redo logs, both LGWR and ARCH must be capable of completing their respective tasks without error.

MONITORING THE INSTANCE

For the majority of the time, the SGA and Oracle background processes operate without administrator intervention. However, there are times when problems must be diagnosed and fixed. There are several methods available to the DBA to monitor and track the behavior of the instance and its associate structures.

USING THE TRACE FILES

The best place to find information about instance problems is in the trace files of the processes themselves. These trace files are written to the location defined in the `USER_DUMP_DEST` or `BACKGROUND_DUMP_DEST`, depending on the specific process and the error encountered. When a background processes is terminated or abnormally aborts an operation, it usually produces a trace file containing the error message(s) causing the failure, dumps of the current process stacks, currently executing cursors, and any other information pertinent to the problem. Although some of this information is useful to you as a DBA, it is more important to collect and forward these trace files to Oracle worldwide customer support consultants who might be able to help you diagnose your problems. They have tools available to pinpoint exactly where the problem occurs. Background process failures also usually write an entry into the `alert.log` file for the database or to their own separate trace files located in the directory specified by the `init.ora` parameter `background_dump_dest`.

TRACKING THROUGH THE OPERATING SYSTEM

Background processes can also be tracked through the OS using system commands. In a UNIX environment, each background process is a separate task and can therefore be tracked separately. It is often very valuable to look at OS memory and CPU utilization of processes (using such tools as sar, ps, vmstat, and top) to identify performance problems or runaway queries. Sometimes the only way to resolve a hung or broken server or user processes is by terminating them at the OS level. Use caution, however, when attempting to modify or terminate any other Oracle background process. Most background processes will crash the entire database if abnormally terminated.

In an NT server environment, tracking the background processes is a little trickier. This is because the entire Oracle instance is implemented on the NT OS as a single background process called a service. The individual background processes are implemented as threads belonging to the service. Although there are plenty of utilities available on NT to track and monitor the behavior of processes, thread administration tools are fairly uncommon. One solution is to use the Performance Monitor utility that ships with the NT OS to monitor, among other things, the memory consumption and context switches of all the threads belonging to the service. By converting the SPID column from the following query from decimal to hexadecimal, you can match the NT thread ID with the background process from the Oracle side.

```
SELECT spid, name FROM V$process, V$bgprocess WHERE addr=
paddr;
```

USING THE V\$ TABLES TO MONITOR INSTANCE STRUCTURES

Numerous dynamic performance views are available to the DBA to display instance information. These views are invaluable when attempting to discover the current state of the database instance and troubleshoot problems related to the instance.

MONITORING DATABASE CONNECTIONS

Both user and background processes that are connected to the instance can be monitored using the V\$ views. The V\$process view displays information about all processes that are connected to the database, including background process and user processes. V\$bgprocess contains a list of all possible background processes, with an addition column, PADDR, which contains the hexadecimal address of running background processes (or 00 for those that are not running).

The columns of interest to you from the V\$process table are as shown in Table 1

Table 1 V\$process Table Columns

<u>Column</u>	<u>Usage</u>
ADDR	Oracle address of the process
PID	Oracle process ID
SPID	OS system process ID
USERNAME	OS process owner
SERIAL#	Oracle process serial #

TERMINAL	OS terminal identifier
PROGRAM	OS program connection
BACKGROUND	1 for background process, NULL for process

The columns interest to you from the V\$bgprocess table are as shown in Table 2.

Table 2 V\$process Table Columns

<u>Column</u>	<u>Usage</u>
PADDR	Oracle process address (same as ADDR column of V\$process)
NAME	Name of the background process
DESCRIPTION	Description of the background process
ERROR	Error state code (0 for no error)

You can display the address and names of all running background processes by joining the V\$process and V\$bgprocess table, as in the following query:

```
SELECT spid, name
FROM V$process, V$bgprocess
WHERE padr(+) = addr;
```

Information about user sessions that are connected to the database

are stored in the V\$session view. The V\$session view contains many fields, and a great deal of valuable information can be accessed from this view.

The columns of interest from the V\$session view are as shown in Table 3:

TABLE 3 V\$session COLUMN

<u>Column</u>	<u>usage</u>
SID	Session identifier
SERIAL#	Session serial #
PADDR	Address of parent session
USER#	Oracle user identifier (from the SYS.USER\$ table)
USERNAME	Oracle username
COMMAND	Current command in progress for this session. For number to command translations, see the sys.audit_actions table.
STATUS	Status of the session (ACTIVE, INACTIVE, KILLED)
SERVER	Type of server connection the session has (DEDICATED, SHARED, PSEDUO or NONE)
OSUSER	OS Username and connection has been made from
PROGRAM	OS Program making the connection into the database

TERMINAL	Type of terminal the connection is made from
TYPE	Type of session (BACKGROUND or USER) SQL_HASH_VALUE and SQL_ADDRESS. Used to uniquely identify the currently executing SQL statement

The following query displays important information on connected processes. It also demonstrates the manner in which the process views relate to each other:

```
col bgproc format a6 heading 'BGProc'
```

```
col action format a10 heading 'DB Action'
```

```
col program format a10
```

```
col username format a8
```

```
col terminal format a10
```

```
SELECT
```

```
b.name bgproc, p.said, s.sid, p.serial#, s.osuser,  
s.username, s.terminal,
```

```
DECODE (a.name, 'UNKNOWN', '&ldots;&ldots;..', a.name) action
```

```
FROM
```

```
V$process p, V$session s, V$bgprosess b,
```

```
Sys.audit_actions a
```


WHERE

```
p.addr = s.paddr (+) AND b.paddr (+) = s.paddr AND  
a.action = NVL(s.action, 0)
```

ORDER BY Sid;



By querying the V\$access view, you can display information on what database objects users are currently accessing. This is useful when trying to figure out what a third-party application or undocumented procedure is doing and can also be useful to resolve security problems. By using a DBA account to run an application or procedure that is giving you security problems, you can determine the exact objects to which security should be granted.

Finally, the V\$mts view contains tracking information for shared server processes. This view contains columns for maximum connections, server started, servers terminated, and servers highwater.

MONITORING THE SHARED SQL AREA

Often it is useful to be able to look into the RDBMS engine and see what SQL statements are being executed. This V\$sqlarea view contains information on SQL statements in the shared SQL area, including the text of SQL statements executed, the number of users accessing the statements, disk blocks and memory blocks accessed while executing the statement, and other information.

NOTE

The disk_reads and buffer_gets columns that are found in V\$sqlarea

track the number of blocks that are read from the buffer cache. These two columns are quick and easy ways to find queries that are utilizing large amounts of database resources.

The V\$open_cursor view is also useful to investigate cursors that have not yet been closed. The following query displays all open cursors for a given user's SID:



```
SELECT b.piece, a.sql_text  
FROM V$open_cursor a, V$sqltext b  
WHERE  
  
a.sid = & SID and  
  
a.address = b.address and  
  
a.hash_value = b.hash_value  
  
ORDER BY  
  
b.address, b.hash_value, b.piece asc ;
```

The V\$sqltext view can also be used to determine what SQL statements are passed to the database engine. Unlike V\$sqlarea, which only stores the first 80 characters of the SQL statements, this view holds the entire SQL statement. The V\$sqltext_with_newlines view is identical to V\$sqltext, that the newline characters in the SQL statements have been left in place.

NOTE

The SQL statements stored in V\$sqltext are split into pieces. To retrieve the entire statement, you have to retrieve all the parts of the

SQL statement and order by the PIECE column.

MONITORING THE SGA

There are two V\$ views available that provide information about the operation of the SGA. The V\$sga view displays the size (in bytes) of each major component of the SGA, including the redo log cache, the database buffer cache, and the shared pool. The V\$sgastat contains much more interesting information. Within this view you find the specific size for each individual memory structure contained in the SGA, including the memory set aside for stack space and PL/SQL variables and stacks. You can also query this view to find the amount of free memory available in the SGA:

```
SELECT bytes FROM V$sgastat WHERE name = 'free memory';
```

MONITORING THE LIBRARY AND DICTIONARY CACHE

Two views exist that contain information regarding the library and data dictionary cache. V\$llibrarycache contains library cache performance information for each type of object in the library cache. The V\$rowcache view contains performance information for the data dictionary cache.

MONITORING THE PARALLEL QUERY PROCESS

The V\$pq_sysstat and V\$pq_tqstat views contain information on the parallel server processes and their behavior. Query V\$pq_sysstat to display current runtime information on parallel query servers such as the number of query servers busy and idle and dynamic server creation and termination statistics. The V\$pq_tqstat view contains

information on queries that have previously run that used parallel query servers.

MONITORING THE ARCHIVER PROCESSES

Archiver activity is stored in the V\$archive view. You can retrieve information on the archived logs written by ARCH from this view.

MONITORING THE MULTITHREADED SERVER PROCESSES

The V\$mmts, V\$dispatcher, and V\$shared_server views contain information on the status of the MTS processes and memory structures. V\$mmts contains tracking information on the shared server processes such as the number of servers started, terminated, and the highwater value for running servers. V\$dispatcher contains information on the dispatcher processes running. From this view you can query the name, supported protocol, number of bytes processed, number of messages processed, current status, and other runtime information relating to the dispatcher processes. The V\$shared_server view provides the same type of information for the shared server processes running.