UNIT -1
DBMS OVERVIEW

WHAT IS A DATABASE?

Data is a row material. Anything can be a data, for example a name of person, name of city, a number etc. When a data is meaningful, it is called information. The database is an organized collection of related information, e.g. records of ***EMPLOYEES*** details in any company, telephone directory etc.

In simple word, a database is a collection of information that is organized so that it can easily be accessed, managed, and updated. In one view, databases can be classified according to types of content: bibliographic, full-text, numeric, and images.

DATABASE MANAGEMENT SYSTEM (DBMS):

Database Management System (DBMS) is a system which helps us in managing database. DBMS allows inserting, updating, deleting and processing of data.

DBMS is used to build and manage the database, i.e. add, change, delete and sort information in the database, to keep database up-to-date. DBMS also helps to get back the desired information in the required format from the database. For example from the database of school it can find out the complete name, address, phone number and GRNO of any student. DBMS is also used to perform calculation based on the data stored in it. For example one can calculate salary of all or selected employees in company’s database.

FoxPro, FoxBASE, dBASE III PLUS etc. are examples of DBMS.

Features of DBMS

1. **Data Security** – DBMS is capable of protecting the data stored inside the database. Examples are database passwords, schema management etc.
2. **Data Integrity** – DBMS ensures the data integrity by maintaining the transcriptional and user level access. It eliminates the unwanted duplicity.
3. **Data Access** – DBMS provides an efficient way to access and manage the data called SQL (Structured Query Language). All modern databases support SQL.
4. **Data Audit** – DBMS should allow to audit and manage the data stored inside the database.

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS):

RDBMS is a DBMS which follows maximum rules out of the twelve rules given by Dr. E. F. Codd in 1970. In practice there is no RDBMS which satisfies all the twelve rules. In RDBMS it will store the data in the form of related tables. As per our requirement we can extract data from the database. The most important feature of RDBMS is that a single database can be spread across several tables.

Today, popular commercial RDBMS for large databases include Oracle, Microsoft SQL Server and Sybase SQL Server etc. The most commonly used free RDBMS are MySQL, PostgreSQL.

Characteristics RDBMS

- RDMS Supports relational data structure.
- RDBMS has Data Manipulation Language at least as powerful as the relational algebra.
- Data is stored in a set of tables.

- Tables are joined by relational links.
- RDBMS Reduces Duplication of data in database (Normalization).
- Allows greater flexibility and efficiency.
- in RDBMS Each table must have a unique references for each record called Primary key.
- Replicating these into other tables creates the Foreign Key.
- These foreign keys form the Relationship that link the tables together.
- Each RDBMS table consists of database table rows. Each database table row consists of one or more database table fields.
- In RDBMS data need only be updated once as it would only have been entered once.
- RDBMS eliminates the problems in using Flat file databases

Advantages of RDBMS

- Highly secured
- Supports data independence
- Avoids data redundancy
- Uses OOPs concept.
- Multiple users can access which is not possible in DBMS.
- Avoids data redundancy problems using NORMAL FORMS
- It performs all DML operations which is not possible with ordinary DBMS (it performs only INSERT n RETRIEVE)
- Supports data Independence.
- Finally it supports DDBMS concepts too and so on.
- Avoid redundant data.
- Avoid Typographical errors.

Dr. E.F.Codd’s twelve rules:

Dr.E.F.Codd is an IBM researcher who first develops the rules for RDBMS in 1970. Thereafter in 1985 he published a list of twelve rules which are being followed by the whole world for RDBMS.

1. **The Information Rule –** All data should be presented in table format.
2. **Guaranteed Access Rule –** The users who have created the data will be allowed to access that data.
3. **Systematic Treatment of Null Values –** A field should be allowed to remain empty. Remember that a blank string, zero and null value are different.
4. **Dynamic On-Line Catalog Based on the Relational Model –** A relational database must provide access to the structure of the table with same tool we are using for entering the data.
5. **Comprehensive Data Sublanguage Rule –** The database must support at least one clearly defined language, like SQL.
6. **View Updating Rule –** The data can be represented in a various logical combination called views. Each view should support a full range of data providing update and delete.
7. **High-Level Insert, Update, and Delete –** Data can be retrieved from RDBMS using multiple tables. On the other hand we can update or delete data of more than one table at a time.

8. **Physical Data Independence –** The user must be isolated from the physical data which is stored in the database. The changes made to hardware will not affect the data.
9. **Logical Data Independence –** How data is viewed should not be changed when the logical structure (table's structure) of the database changes. This rule is particularly difficult to satisfy
10. **Integrity Independence –** The database language like SQL should support integrity constraints like primary key, foreign key etc.
11. **Distribution independence –** The user should be totally unaware of whether or not the database is distributed.
12. **Non subversion rule –** there should be no way to modify the database structure other than your database language. Even your administrator will not be allowed to change your data.

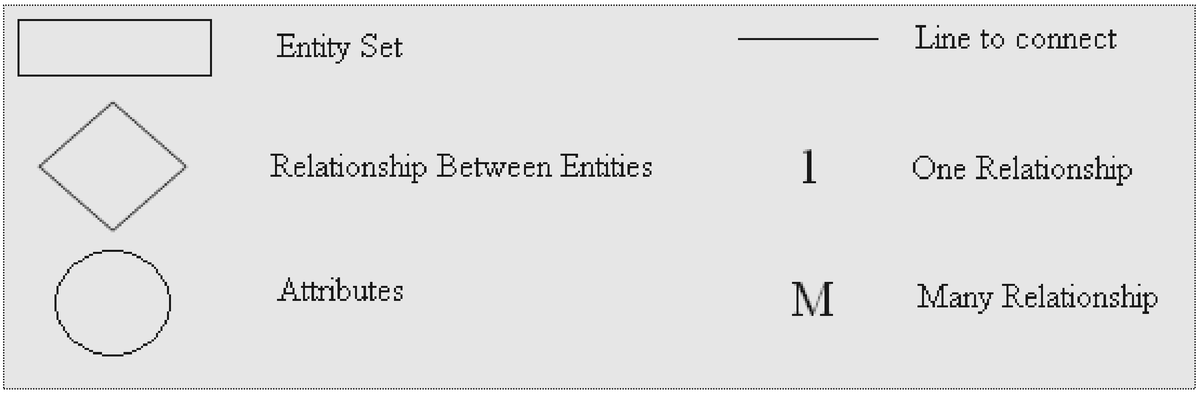
Oracle satisfies most of the above rules so we can say that Oracle is an RDBMS.

E-R DIAGRAM:

Entity Relationship Diagram has been created by system analyst or system developer to perform and calculate how the relationship has been developed among various tables and fields of the system data. It is a pictorial representation of the whole system just like the flowcharts. It has various symbols which represents the system.

In E-R diagram all the thing are in an easy format with very much less symbols, so that even a person who is not know anything about the system can also understand the whole system.

The symbol which are used in the E-R Diagram are



Above symbols are used to develop E-R Diagram, here the entity set is the name of table while the attributes are name of the fields. For example suppose we have a table named employee having the fields emp_no, name, salary, birthdate, dept_no and we have another table named department having dept_no and dept_name fields.

If we want to create relationship among above tables then we can set the relationship like employee works in a department. E-R Diagram of above table is as under

RELATIONSHIP

When we have more than one table and these tables are connected with each other in some way, this connection between the tables is called Relationship.

Types of Relationship

1) One to One (1:1):

In one to one relationship each records of one table is connected with the one record of second table. The relationship between student and their G.R. number is the example of one to one relationship.

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

In one to many relationships one record of table is connected with many record of another table. For example one book has many authors. If we reverse the one to many relationships it is called many to one relationship. In many to one relationship more than one records of one table is connected with one record of another table. For example many authors have one book.

3) Many to Many (M:M):

When many records of one table is connected with many records of another table, it is called many to many relationship. For example many students have many teachers.

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anamolies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. Normalization is used for mainly two purpose, Eliminating redundant (useless) data. Ensuring data dependencies make sense i.e data is logically stored

.

Problem without Normalization

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updating and Deletion Anomalies are very frequent if Database is not normalized. To understand these anomalies let us take an example of Student table.

S_id	S_Name	S_Address	Subject_opted
401	Adam	Noida	Bio
402	Alex	Panipat	Maths
403	Stuart	Jammu	Maths
404	Adam	Noida	Physics

- **Updation Anamoly** : To update address of a student who occurs twice or more than twice in a table, we will have to update S_Address column in all the rows, else data will become inconsistent.
- **Insertion Anamoly :** Suppose for a new admission, we have a Student id(S_id), name and address of a student but if student has not opted for any subjects yet then we have to insert NULL there, leading to Insertion Anamoly.
- **Deletion Anamoly :** If (S_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

Normalization Rule

Normalization rule are divided into following normal form.

- First Normal Form
- Second Normal Form
- Third Normal Form
- BCNF

First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

Student Table :

Student	Age	Subject
Adam	15	Biology, Maths
Alex	14	Maths
Stuart	17	Maths

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

Student Table following 1NF will be :

Student	Age	Subject
Adam	15	Biology
Adam	15	Maths
Alex	14	Maths
Stuart	17	Maths

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails Second normal form.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

New Student Table following 2NF will be :

Student	Age
Adam	15
Alex	14
Stuart	17

In Student Table the candidate key will be Student column, because all other column i.e. Age is dependent on it.

New Subject Table introduced for 2NF will be :

Student	Subject
Adam	Biology
Adam	Maths
Alex	Maths
Stuart	Maths

In Subject Table the candidate key will be {Student, Subject} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

Third Normal Form (3NF)

Third Normal form applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this transitive functional dependency should be removed from the table and also the table must be in Second Normal form. For example, consider a table with following fields.

Student_Detail Table :

Student_id	**Student_name**	**DOB**	**Street**	**city**	**State**	**Zip**

In this table Student_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called transitive dependency. Hence to apply 3NF, we need to move the street, city and state to new table, with Zip as primary key.

New Student_Detail Table:

Student_id	Student_name	DOB	Zip

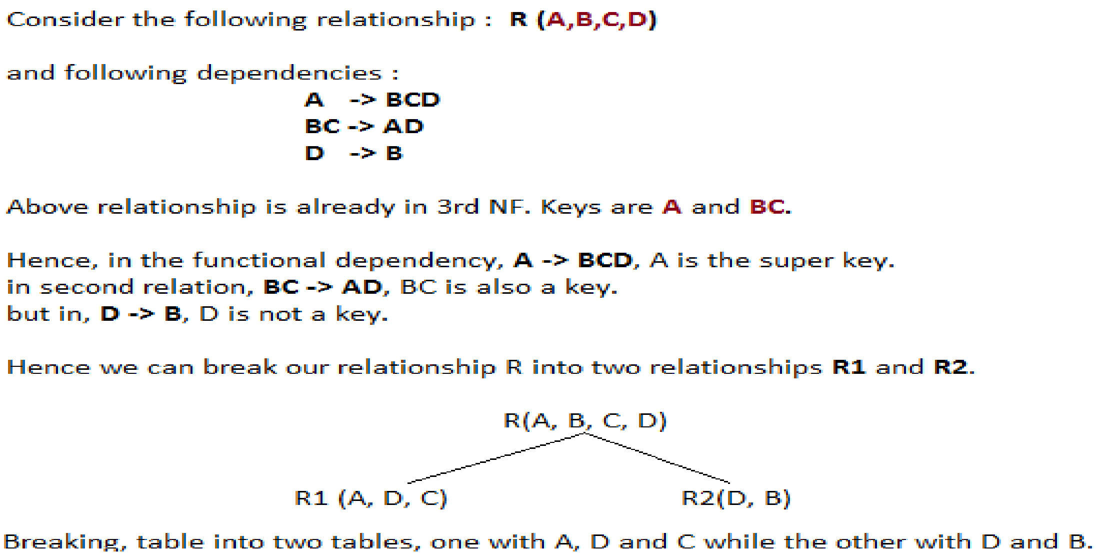
Address Table:

Zip	Street	city	state

The advantage of removing transitive dependency is, Amount of data duplication is reduced. Data integrity achieved.

Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:
R must be in 3rd Normal Form
and, for each functional dependency (X -> Y), X should be a super Key.



Consider the following relationship : **R (A,B,C,D)**

and following dependencies :

A -> BCD
BC -> AD
D -> B

Above relationship is already in 3rd NF. Keys are **A** and **BC**.

Hence, in the functional dependency, **A -> BCD**, A is the super key.
in second relation, **BC -> AD**, BC is also a key.
but in, **D -> B**, D is not a key.

Hence we can break our relationship R into two relationships **R1** and **R2**.

R(A, B, C, D)
R1 (A, D, C)
R2(D, B)

Breaking, table into two tables, one with A, D and C while the other with D and B.

DIFFERENT BETWEEN DBMS AND RDBMS:

DBMS	RDBMS
In DBMS relationship between two tables or files are maintained programmatically.	In RDBMS relationship between two tables or files can be specified at the time of table creation.
DBMS does not supports client/Server Architecture	Most of the RDBMS supports client/Server Architecture
DBMS does not supports distributed database	Most of RDBMS supports distributed database
In DBMS, there is no security of data	In RDBMS, there are multiple level s of security 1. Logging in at O/S level 2. Command Level 3. Object Level.
Each table is given an extension in DBMS	Many tables are grouped in one database in RDBMS
The speed of operation is low.	The speed of operation is fast.
Hardware and software requirement are less.	Hardware and software requirement are more.
Facilities given are less.	Facilities given are more.
Generally uses DOS platform.	It can be used on any platform.
It uses the concept of file.	It uses the concept of table and records.
Examples are Dbase, FoxBASE etc.	Examples are Oracle, Ingress etc

UNIT 1

2. SQL, SQL*PLUS

Introduction to Oracle

In the June 1970 Dr. E. F. Codd has published a paper entitled "A Relational Model of Data for Large Shared Data Banks." That model was sponsored by IBM. This model has been accepted by the whole world as s model for RDBMS. The language which supports this model was Structured English Query Language (SEQUEL). Afterward the word English has been removed from the language, so it becomes **SQL (Structured Query Language).**

In 1979 a company called Relational Software Incorporation has released the first RDBMS. In 1985 the company was taken over by Oracle Corporation. **Oracle Corporation is a company that produces the most widely used, Server based, Multi user RDBMS named Oracle.**

Introduction to SQL

Structure Query Language (SQL) is a language that provides an interface to relational database systems. **SQL was developed by IBM in the 1970's** for use in System R, and is a de facto standard, as well as an ISO and ANSI standard. SQL is often pronounced SQLUEL.

In Common usages SQL also encompasses DML **(Data Manipulation Language)**, for INSERTs, UPDATEs, DELETEs and DDL (**Data Definition Language**), used for creating and modifying tables and other databases structure.

The development of SQL is governed by standards. The American National Standards Institute (ANSI) is an organization that approves certain standards in many different industries. SQL has been deemed the standard language in relational database communication, originally approved in 1986 based on IBM's implementation. In 1987, **the International Standards Organization** (ISO) accepted the ANSI SQL standard as the international standard. The standard was revised again in 1992 and was called SQL - 92. The newest standard is now called SQL-99, it is also referred to as SQL-3. SQL-3 support objects extension and are partially implemented in Oracle8 and oracle 9.

SQL has been a common language for communication with the oracle 9i server from any tool or application. Oracle SQL contains many extensions. When an SQL statement is entered, it is stored in a part of memory called the SQL buffer and remains there until a new SQL statement is entered.

SQL *PLUS is an Oracle tool that recognizes and submits SQL statements to the Oracle 9i server for execution. It contains its own command language.

Features of SQL

- SQL can be used by a person with little or no programming experience.
- It is non procedural language.
- It reduces the amount and time required for creating and maintaining the system.
- It is English like language.

Features of SQL * PLUS

- SQL *PLUS accepted ad hoc entry of statements
- It accepts SQL input from files
- It provides a line editor for modifying SQL statements
- It controls environmental settings

- It formats query results into basic reports
- It accessed local and remote databases.

COMPONENTS OF SQL :

There are mainly four components of SQL.

- **DDL (Data Definition Language):** It is a set of SQL commands to create, modify and delete database structure but not data. These commands are normally used by system administrator and not by the general user.
  - ***Ex. Create table, Alter Table, Drop Table , Trancate , Comment, Grant, Revoke***
- **DML (Data Manipulation Language):** It is the area of SQL that allows changes in the data of the table.
  - ***Ex. Insert, Delete, Update Call , Explain Plan, Lock .***
- **DCL (Data Control Language):** This is the component of SQL which controls access to the data and the database. On various occasions DML and DCL are grouped together.
  - ***Ex. Commit, SavePoint , Rollback, Grant, Revoke, Set Transaction.***
- **DQL (Data Query Language):** This is the component of SQL which allows getting data from the database and impose ordering on it. It covers the heart of the SQL.
  - **Ex. Select**

Rules Of SQL

- Any SQL statement must start with a verb like select, create, alter etc.
- Each verb has been followed by adjective like from, where, having ect.
- There must be white space between verb and adjective.
- A comma is useful to separate the field names.
- A semicolon is used to complete the SQL statement.
- We can split our statement into multiple lines.
- Variable name length must be less than or equal to 30 characters.
- Character values must be enclosed within single quotes.
- For comments we can use /* and */

Oracle Data types

When we install Oracle on our computer, it will create one database for each user. Like other database management systems, we are not required to create database, but instead of that we have to create tables in our database, which is the fundamental thing in Oracle. Table is the basic part of our database where our required data will be stored. We can store following type of data in table.

- **CHAR (n):** This data type is useful to store character string values of fixed length. In brackets we have to specify the size of the field. The maximum size available is 256 characters. Remember that if you have specified the size as 20 characters and if you write only 5 characters in that field of particular record, then remaining 15 characters will be white spaces instead of null.
- **VARCHAR (n) /VARCHAR2 (n):** This kind of data type is useful to store variable length of alphanumeric data. The maximum limit of n will be up to 4000 characters. The main difference between VARCHAR and CHAR is, if your data is less than the size allocated then white spaces will not be generated, but it will be blank.

- **DATE:** This data type is useful to represent date and time. The standard format is DD-MON-YY. So 21-1-2009 will be written as 21-JAN-09. To enter dates other than standard format we can use appropriate functions. The default date for the field is first day of current month. Valid range of date field is from January 1st 4712 BC to December 31st 4712 AD.

- **NUMBER (p, s):** The number data type is useful to store numeric value having fixed or floating point number. The precision p determines the maximum length of the data, whereas the scale s determines the number of places after decimal point. For example if you write number (7, 2), then it will store 5 numbers before the decimal and 2 numbers after decimal.

- **LONG: This** data type is useful to store variable length of character string up to 2 GB. Remember that you can define only one field as long in a table.

- **RAW / LONG RAW:** This data type is useful to store binary type of data such as digitized pictures or images. RAW data type can have maximum of length 256 bytes, while the LONG RAW can contain up to 2 GB.

Introduactio to SQL *PLUS

You can use the SQL*Plus program in conjunction with the SQL database language and its procedural language extension, PL/SQL. The SQL database language allows you to store and retrieve data in Oracle. PL/SQL allows you to link several SQL commands through procedural logic.

SQL*Plus enables you to execute SQL commands and PL/SQL blocks, and to perform many additional tasks as well. Through SQL*Plus, you can

- enter, edit, store, retrieve, and run SQL commands and PL/SQL blocks
- format, perform calculations on, store, print and create web output of query results
- list column definitions for any table
- access and copy data between SQL databases
- send messages to and accept responses from an end user
- perform database administration

SQL*Plus Command

Command	Description
@ ("at" sign)	Runs the SQL*PLus statements in the specified script. The script can be called from the local file system or from a web server.
@@ (double "at" sign)	Runs a script. This command is similar to the @ ("at" sign) command. It is useful for running nested scripts because it looks for the specified script in the same path as the script from which it was called.
/ (slash)	Executes the SQL command or PL/SQL block.
ACCEPT	Reads a line of input and stores it in a given user variable.
APPEND	Adds specified text to the end of the current line in the buffer.

Command	Description
ARCHIVE LOG	Starts or stops the automatic archiving of online redo log files, manually (explicitly) archives specified redo log files, or displays information about redo log files.
ATTRIBUTE	Specifies display characteristics for a given attribute of an Object Type column, and lists the current display characteristics for a single attribute or all attributes.
BREAK	Specifies where and how formatting will change in a report, or lists the current break definition.
BTITLE	Places and formats a specified title at the bottom of each report page, or lists the current BTITLE definition.
CHANGE	Changes text on the current line in the buffer.
CLEAR	Resets or erases the current clause or setting for the specified option, such as BREAKS or COLUMNS.
COLUMN	Specifies display characteristics for a given column, or lists the current display characteristics for a single column or for all columns.
COMPUTE	Calculates and prints summary lines, using various standard computations, on subsets of selected rows, or lists all COMPUTE definitions.
CONNECT	Connects a given user to Oracle.
COPY	Copies results from a query to a table in a local or remote database.
DEFINE	Specifies a user variable and assigns it a CHAR value, or lists the value and variable type of a single variable or all variables.
DEL	Deletes one or more lines of the buffer.
DESCRIBE	Lists the column definitions for the specified table, view, or synonym or the specifications for the specified function or procedure.
DISCONNECT	Commits pending changes to the database and logs the current user off Oracle, but does not exit SQL*Plus.
EDIT	Invokes a host operating system text editor on the contents of the specified file or on the contents of the buffer.
EXECUTE	Executes a single PL/SQL statement.
EXIT	Terminates SQL*Plus and returns control to the operating system.
GET	Loads a host operating system file into the SQL buffer.
HELP	Accesses the SQL*Plus help system.
HOST	Executes a host operating system command without leaving SQL*Plus.
INPUT	Adds one or more new lines after the current line in the buffer.
LIST	Lists one or more lines of the SQL buffer.
PASSWORD	Allows a password to be changed without echoing the password on an input device.
PAUSE	Displays the specified text, then waits for the user to press [Return].
PRINT	Displays the current value of a bind variable.
PROMPT	Sends the specified message to the user's screen.
QUIT	Terminates SQL*Plus and returns control to the operating system. QUIT is identical to EXIT.
RECOVER	Performs media recovery on one or more tablespaces, one or more datafiles, or the entire database.

Command	Description
REMARK	Begins a comment in a script.
REPFOOTER	Places and formats a specified report footer at the bottom of each report, or lists the current REPFOOTER definition.
REPHEADER	Places and formats a specified report header at the top of each report, or lists the current REPHEADER definition.
RUN	Lists and executes the SQL command or PL/SQL block currently stored in the SQL buffer.
SAVE	Saves the contents of the SQL buffer in a host operating system file (a script).
SET	Sets a system variable to alter the SQL*Plus environment for your current session.
SHOW	Shows the value of a SQL*Plus system variable or the current SQL*Plus environment.
SHUTDOWN	Shuts down a currently running Oracle instance.
SPOOL	Stores query results in an operating system file and, optionally, sends the file to a printer.
START	Executes the contents of the specified script. The script can only be called from a url.
STARTUP	Starts an Oracle instance and optionally mounts and opens a database.
STORE	Saves attributes of the current SQL*Plus environment in a host operating system file (a script).
TIMING	Records timing data for an elapsed period of time, lists the current timer's title and timing data, or lists the number of active timers.
TTITLE	Places and formats a specified title at the top of each report page, or lists the current TTITLE definition.
UNDEFINE	Deletes one or more user variables that you defined either explicitly (with the DEFINE command) or implicitly (with an argument to the START command).
VARIABLE	Declares a bind variable that can be referenced in PL/SQL.
WHENEVER OSERROR	Performs the specified action Exits SQL*Plus if an operating system command generates an error.
WHENEVER SQLERROR	Performs the specified action Exits SQL*Plus if a SQL command or PL/SQL block generates an error.

SQL Operators Overview

An operator manipulates individual data items and returns a result. The data items are called operands or arguments. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL. There are two general classes of operators: unary and binary. Oracle Lite SQL also supports set operators.

1. Unary Operators

A unary operator uses only one operand. A unary operator typically appears with its operand in the following format:

Ex. operator operand

2. Binary Operators

A binary operator uses two operands. A binary operator appears with its operands in the following format:

Ex. operand1 operator operand2

3. Set Operators

Set operators combine sets of rows returned by queries, instead of individual data items. All set operators have equal precedence. Oracle Lite supports the following set operators:

- **UNION**
- **UNION ALL**
- **INTERSECT**
- **MINUS**

The following lists the levels of precedence among the Oracle Lite SQL operators from high to low. Operators listed on the same line have the same level of precedence: Table Levels of Precedence of the Oracle Lite SQL Operators

Precedence Level	SQL Operator
1	Unary + - arithmetic operators, PRIOR operator
2	* / arithmetic operators
3	Binary + - arithmetic operators, || character operators
4	All comparison operators
5	NOT logical operator
6	AND logical operator
7	OR logical operator

Other Operators

Other operators with special formats accept more than two operands. If an operator receives a null operator, the result is always NULL. The only operator that does not follow this rule is CONCAT.

Arithmetic Operators

Arithmetic operators manipulate numeric operands. The - operator is also used in date arithmetic. Table Arithmetic Operators

Operator	Description	Example
+ (unary)	Makes operand positive	SELECT +3 FROM DUAL;
- (unary)	Negates operand	SELECT -4 FROM DUAL;
/	Division (numbers and dates)	SELECT SAL / 10 FROM EMP;
*	Multiplication	SELECT SAL * 5 FROM EMP;
+	Addition (numbers and dates)	SELECT SAL + 200 FROM EMP;
-	Subtraction (numbers and dates)	SELECT SAL - 100 FROM EMP;

Character Operators

Character operators are used in expressions to manipulate character strings.
Table Character Operators

Operator	Description	Example
||	Concatenates character strings	SELECT 'The Name of the employee is: ' || ENAME FROM EMP;

Concatenating Character Strings

You can concatenate character strings with the following results: Concatenating two character strings results in another character string. Oracle preserves trailing blanks in character strings by concatenation, regardless of the strings' datatypes. Oracle provides the CONCAT character function as an alternative to the vertical bar operator. For example:

SQL>SELECT CONCAT (CONCAT (ENAME, ' is a '),job) FROM EMP WHERE SAL > 2000;

This returns:

```
CONCAT(CONCAT(ENAME
------------------------
KING     is a PRESIDENT
BLAKE    is a MANAGER
CLARK    is a MANAGER
JONES    is a MANAGER
FORD     is a ANALYST
SCOTT    is a ANALYST
6 rows selected.
```

Oracle treats zero-length character strings as nulls. When you concatenate a zero-length character string with another operand the result is always the other operand. A null value can only result from the concatenation of two null strings.

Comparison Operators

Comparison operators are used in conditions that compare one expression with another. The result of a comparison can be TRUE, FALSE, or UNKNOWN.

Operator	Description	Example
=	Equality test.	SELECT ENAME "Employee" FROM EMP WHERE SAL = 1500;
!=, ^=, <>	Inequality test.	SELECT ENAME FROM EMP WHERE SAL ^= 5000;
>	Greater than test.	SELECT ENAME "Employee", JOB "Title" FROM EMP WHERE SAL > 3000;
<	Less than test.	SELECT * FROM PRICE WHERE MINPRICE < 30;
>=	Greater than or equal to test.	SELECT * FROM PRICE WHERE MINPRICE >= 20;
<=	Less than or equal to test.	SELECT ENAME FROM EMP WHERE SAL <= 1500;

IN	"Equivalent to any member of" test. Equivalent to "= ANY".	SELECT * FROM EMP WHERE ENAME IN ('SMITH', 'WARD');
ANY/ SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, or >=. Evaluates to FALSE if the query returns no rows.	SELECT * FROM DEPT WHERE LOC = SOME ('NEW YORK','DALLAS');
Operator	**Description**	**Example**
NOT IN	Equivalent to "!= ANY". Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM DEPT WHERE LOC NOT IN ('NEW YORK', 'DALLAS');
ALL	Compares a value with every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, or >=. Evaluates to TRUE if the query returns no rows.	SELECT * FROM emp WHERE sal >= ALL (1400, 3000);
[NOT] BETWEEN x and y	[Not] greater than or equal to x and less than or equal to y.	SELECT ENAME, JOB FROM EMP WHERE SAL BETWEEN 3000 AND 5000;
EXISTS	TRUE if a sub-query returns at least one row.	SELECT * FROM EMP WHERE EXISTS (SELECT ENAME FROM EMP WHERE MGR IS NULL);
IS [NOT] NULL	Tests for nulls. This is the only operator that should be used to test for nulls.	SELECT * FROM EMP WHERE COMM IS NOT NULL AND SAL > 1500;

Logical Operators

Logical operators manipulate the results of conditions.

Operator	Description	Example

NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	SELECT * FROM EMP WHERE NOT (job IS NULL)
		SELECT * FROM EMP WHERE NOT (sal BETWEEN 1000 AND 2000)

AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise returns UNKNOWN.	SELECT * FROM EMP WHERE job='CLERK' AND deptno=10
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise, returns UNKNOWN.	SELECT * FROM emp WHERE job='CLERK' OR deptno=10

SQL V/S SQL *PLUS

SQL	SQL *PLUS
SQL is a language for communication with the Oralce Server to access data	SQL * PLUS recognizes SQL Statements and sends them to the server
SQL based on American National Standards Institute (ANSI) standard Data.	SQL * PLUS is the Oralce property interface for executing SQL statement
SQL manipulates data and table definitions in the Database	SQL * PLUS does not allow manipulation of values in the database
SQL is entered into the SQL buffer on one or more line	SQL * PLUS is entered one line at a time, not stored in the SQL buffer.
SQL does not have a continuation character	SQL * PLUS uses a dash (-) as a continuation character if the commend is longer than one line
it cannot be abbreviated	It can be abbreviated
SQL uses a termination character to execute commands immediately	SQL * PLUS does not require termination characters ; executes commands immediately
SQL uses function to perform some formatting.	SQL * PLUS uses commands to format data

UNIT -2

MANAGING TABLES AND DATA

CREATE TABLE COMMAND

This command is useful to create a new table. Here with this command you have to specify field name, data type and size of the field. Each of the fields should be separated by comma and the SQL statement will be terminated with a semicolon.

RULES FOR CREATING TABLE:

- A name can have maximum 30 characters.
- Alphabet and numbers are only allowable characters.
- A name must begin with alphabet.
- A special character underscore is allowed.
- SQL keywords are not allowed as name.

Syntax	Example
CREATE TABLE <table name> **(<fieldname 1> datatype (size),** **<fieldname 2> datatype (size),** **....** **....** **<fieldname N> datatype(size));**	**CREATE TABLE stud** **(Grno number(4),** **Name varchar2(20),** **Std number(2),** **Bdate date,** **Age number(3));**

INSERT COMMAND: This command is useful to insert records in the table. For inserting the record in a table you have to specify the name of the columns and the values for the each column of your present record. Remember that when you want to insert numeric data then there is no question, but in case of string and date data type of data at the time of inserting you have to give this data in single quotation mark. Second thing is you have to enter date type of data in a default format.

Syntax

```
SQL> INSERT INTO <table name>
       (<Fieldname1>,< Fieldname2>.....< Fieldname N>)
        VALUES
      (<value1>,<value2>……<valueN>);
```

Example

```
SQL>INSERT INTO stud(Grno,Name,Std,Bdate,Age)VALUES(1,’ABC’,12,’01-jan-80’,27);
```

VIEWING DATA OF THE TABLE : Once the data has been entered in the table the next logical operation is to view the data. For viewing the data SQL has given us the SELECT statement.

```
SQL>SELECT <field names separated by a  comma> FROM <table name>;
SQL>SELECT * FROM stud;
```

WHERE CLAUSE : In some situation we need to filter our data at that time we have to specify the condition with select statement by using WHERE clause. For example

```
SQL>SELECT * FROM stud WHERE age<20;
SQL>SELECT Grno, Std, Bdate WHERE Name=’ABC’;
```

SELECTING UNIQUE ROWS FROM THE TABLE : In some situation you require that you want to eliminate the duplicate data from the table while selecting rows. At that time you have to use DISTINCT clause with SELECT command.

```
SQL>SELECT DISTINCT dept FROM employee;
```

SORTING DATA OF TABLE: Oracle allows data from a table to be viewed in a sorted order. We can retrieve the sorted data either in ascending order or in descending order. For that we have to give order by clause in SELECT statement. Remember that order by gives us ascending order output by default, but if we want descending order output then the word DESC will be written at the end.

```
SQL>SELECT * FROM stud ORDER BY std;  Displays output in ascending order
SQL>SELECT * FROM stud ORDER BY std DESC;  Displays output in descending order
```

DESC COMMAND :

This command is useful to display the structure of the table.

```
SQL>DESC <table name>;
SQL>DESC stud;
```

DELETE COMMAND :

Delete command is useful for deleting record from table. With the help of DELETE command you can delete all the records or by using WHERE clause you can delete specific records of table. Remember that in oracle there is no command to recall the deleted data.

```
SQL>DELETE FROM <table name>   WHERE   <condition>;

SQL>DELETE FROM stud;
SQL>DELET FROM stud WHERE city='RAJKOT';
```

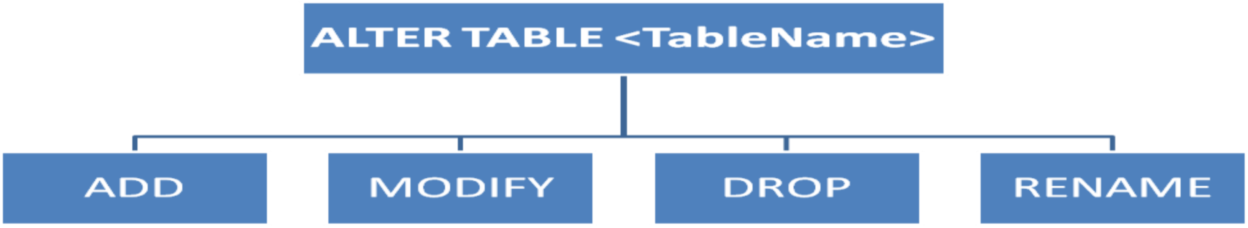
UPDATE COMMAND:

Update command is useful to change or modify the data in one or more than one fields of a particular table. The update command will update the existing rows of a table with new values and with the help of SET clause we can set the new value.

```
SQL>UPDATE <table name> SET <column name>=<value> WHERE <condition>;

SQL>UPDATE stud SET city='RAJKOT' WHERE Std=12;
SQL>UPDATE marks SET total=maths+science+eng;
```

MODIFYING THE STRUCTURE OF THE TABLE

When you want to modify the structure of your table you have three options.



1. Add a new column and Constraint

```
SQL>ALTER TABLE <table name> ADD (fieldname  datatype (size));
SQL>ALTER TABLE stud ADD (pincode varchar2(7));
SQL>ALTER TABLE stud ADD primary key (grno);
```

2. Modify the size of the column and datatype and constraint

```
SQL>ALTER TABLE <table name> MODIFY (field_name  datatype (size));
SQL>ALTER TABLE stud MODIFY  (pincode varchar2(10));
SQL> ALTER TABLE stud MODIFY (pincode number(6));
```

3. Delete the column and constraint

```
SQL>ALTER TABLE <table name> DROP  COLUMN <column name>;
SQL>ALTER TABLE stud DROP COLUMN    Pincode;
SQL>ALTER TABLE stud DROP constraint stud_grno;
```

4. To Rename column name

```
SQL>ALTER TABLE <TABLE NAME> RENAME COLUMN <OLD NAME> TO <NEW NAME>
SQL> ALTE TABLE STUD RENAME COLUMN STUDENTNAME TO S_NAME
```

5. To RENAME TABLE NAME

```
SQL>ALTER TABLE <TABLE NAME> RENAME TO <NEW NAME>
SQL>ALTER TABLE STUD RENAME TO STUDENT
```

RENAME THE TABLE : Oracle gives you the facility to change the name of the table created by you.

```
SQL>RENAME <table name> TO <new table name>;
SQL>RENAME stud TO stud_master;
```

DESTROYING THE TABLE: In some situation we may need to completely destroy the whole table and its content all together. For that we have the command DROP TABLE.

```
SQL>DROP TABLE <table name>;
SQL>DROP TABLE stud;
```

TRUNCATING THE TABLE : Truncating the table means to delete all the records of a table and make it empty. TRUNCATE command is similar to delete command, but they have some difference.

When you give the DELETE command it will delete the records one by one. While when you give the TRUNCATE command it will drop the whole table and then recreate the structure, so this command is faster then delete command.

```
SQL>TRUCATE TABLE <table name>;
SQL>TRUNCATE TABLE stud;
```

CREATING A TABLE FROM A TABLE : In some situations we may need to create a table from an existing table, for that the CREATE command will be used with SELECT command.

```
SQL>CREATE TABLE <table name>(column1, column2, column3 ...)
     AS
    SELECT  (column1, column2, column3 ...)FROM <table name>;

SQL>CREATE TABLE stud1(no,stud_name,std)
    AS
    SELECT(grno,name,std) FROM stud;
```

Remember that when you create a table from a table the data of the columns will also be copied.

INSERT DATA INTO TABLE FROM ANOTHER TABLE

In oracle when we use INSERT INTO command along with the SELECT command then we can insert the records from one table to another table.

SQL>INSERT INTO<table name>SELECT <col1>, <col2>... FROM <table name>;
SQL>INSERT INTO stud1 SELECT Grno, name, std from stud;

Remember that all the columns of new table must be satisfied with the old table, because there is no query which will allow us to insert a particular column's data in a particular column.

DATA CONSTRAINTS :

In some situation we need that the data should be checked before it is entered into the table. So in that situation, while creating the table we have to set some rules which are called as data constraints.

Oracle permits us data constraints to be attached to table with the help of SQL statements. We can set constraints in two ways
1. At the time of creating the table (with the use of create table command) and
2. While modifying the table (Alter table command).

Once a constraint attached to the table column, any insert or update operation automatically being checked before the actual change or data storage.

Primary key constraint

A primary key is a constraint which is useful to uniquely identify each row of a table. None of the field which is a part of primary key can contain null value. Remember that whenever you set any column as primary key then the data in that column must be unique.

Features of Primary Key:

1. Its main purpose is uniqueness of record.
2. It will not allow **null** values.
3. It will not be allow duplicate value
4. It is not compulsory but it is recommended.
5. It cannot be used on long and long raw data type.
6. Unique index will be created automatically on primary key column.
7. When we define primary key on more than one column then it is called as composite key.
8. We can create primary key on maximum 16 columns.

We can set primary key at column level or at table level.

Column Level	Table Level
SQL>CREATE TABLE stud (grno number(4) primary key, Name varchar2(30), City varchar2(30));	**SQL>CREATE TABLE stud (grno number(3), Name varchar2(30) Cityvarchar2(30), Primary key (grno));**

Foreign key constraints:

Foreign key represents relationship between the tables. A foreign key is a column whose values are derived from primary key table. The table in which the foreign key has been defined is known as foreign table or detail table, while the primary key table is known as primary table or master table. Foreign key can be defined by using the word **'REFERENCES'** in foreign table.

Features

1. The foreign key column must have a primary key column in master table.
2. Child may have duplicates if specified.
3. Master table cannot be update or delete if the child exists.
4. Record inserted must have a corresponding value in the master table.
5. Parent record can be delete provide no child record exist.

Example

SQL>CREATE TABLE marks (Grno number (4) REFERENCES stud, m1 number (3), m2 number (3), m3 number (3), total number(3));

Foreign key constraint with ON DELETE CASCADE

When you set the foreign key with ON DELETE CASCADE clause then if you delete a record from master table then all the corresponding records in the foreign table will be automatically deleted.

Example

Column Level	Table level
SQL>CREATE TABLE marks (Grno number (4) REFERENCES stud ON DELETE CASCADE, m1 number(3), m2 number (3), m3 number (3), total number(3));	**SQL>CREATE TABLE marks (Grno number (4),, m1 number(3), m2 number (3), m3 number (3), total number(3), FOREIGN KEY (grno) REFERENCES stud(grno) ON DELETE CASCADE);**

Foreign key constraint with ON DELETE SET NULL

When you define a foreign key with ON DELETE SET NULL then when you delete one record from the master table then it will not delete the corresponding records in the foreign table, but it will make the value in the foreign table for the corresponding record as null.

Example

Column Level	Table level
SQL>CREATE TABLE marks (grno number (4) REFERENCES stud REFERENCES stud ON DELETE SET NULL , m1 number(3), m2 number (3), m3 number (3), total number(3));	**SQL>CREATE TABLE marks (grno number (4),, m1 number(3), m2 number (3), m3 number (3), total number(3), FOREIGN KEY (grno) REFERENCES stud(grno) ON DELETE SET NULL);**

Unique key constraint:

Unique column constraint permits multiple entries of null values in a column. These null values are clubbed at the top of the records. The main difference between primary key and unique key is primary key does not allow blank values while the unique key will allow the blank values.

Features

1. It will not allow duplicate values.
2. Record will be indexed automatically.
3. Unique key can have 16 possibilities of combination to become composite key.
4. Unique key is not allowed on long or long row data type.

Example

SQL>CREATE TABLE stud(grno number(4) UNIQUE, Name varchar2(30), City varchar2(30));

Null value constraint

Sometimes in our table there are some records in which some fields does not have any value then it is called as null value. With help of null value constraint we can restrict the user from leaving any value as null.

Example :

SQL>CREATE TABLE stud(grno number(4) Primary key, Name varchar2(30) NOT NULL, City varchar2(30));

Check constraint

This constraint is useful when you want to check each and every value that is entered by the user. Remember that when you apply check constraint, any value beyond the given limit will not be accepted in any circumstances.

Example

SQL> CREATE TABLE marks(grno number(4),maths number(3)check maths>=0 and maths<=100,English number(3) check English>=0 and English<=100);

SQL>CREATE TABLE client (client_no varchar2 (6) check client_no LIKE 'C%', client_name varchar2 (30), city varchar2 (30));

Default value constraint

At the time of creating a table we can define the default value constraint for a column. Whenever a new row (record) will be inserted in a table, oracle will automatically assign a value to that column while inserting the row. We can define default value constraint on numeric, character or date type of fields.

Example

```
SQL>CREATE TABLE stud (grno number (4) primary key, name varchar2 (25),
        City varchar2 (20) DEFAULT  'Rajkot');
SQL>INSERT INTO stud (grno, name) VALUES (1, 'ABC');
SQL>INSERT INTO stud(grno,name, city) VALUES (1,'ABC', 'Ahmadabad');
```

ARITHMETIC OPERATORS

Following arithmetic operators are available in oracle.

(1) + For addition (2) - For subtraction (3) * For multiplication
(4) % For division (5) ** for exponentiation

The above operators are useful with any numerical fields for performing mathematical calculation. For example if we want to calculate annual salary of an employee from their monthly salary, we can give the command like
SQL> SELECT salary, salary * 12 FROM emp;

RENAMING THE COLUMN IN OUTPUT

In above example the title for annual salary is salary*12, which does not look good. Now if we you want to change the title than give the command likes
SQL> SELECT salary, salary * 12 FROM "Annual Salary" emp;

RANGE SEARCHING USING BETWEEN OPERATOR :

When we want to select some data within the range of particular values, the between operator will be useful. Remember following points while using the between operator.

1. First value must be less than the second value.
2. Output will be inclusive of both the values.

Example: ***SQL>SELECT name, std FROM stud WHERE std BETWEEN 8 AND 12;***

PATTERN MATCHING USING LIKE

When you want to select data from character data type with some specific conditions the LIKE operator will be useful. When using the like operator we have to use following two characters

- **%** for any number of characters and
- **_ (underscore)** for single character

Examples

1. Display the name of customers whose name is starting with character ch.
   SQL>SELECT name FROM client_master WHERE name LIKE 'ch%';
2. List the clients name whose second letter of name is either a or s.
   SQL> SELECT name FROM client_master WHERE name LIKE '_a%' OR name LIKE '_s%';
3. List the clients for the persons name starting with IV and name must be 4 character long.
   SQL>SELECT name FROM client_master WHERE name LIKE 'IV_ _';

IN AND NOT IN OPERATOR :

When you use equal operator it will check only one condition, if you want more than one condition to be checked simultaneously then you have to use OR operator. Instead of using or operator we can use IN clause to get the desired output.

Example: -Display list of student either from Rajkot city or Ahmadabad or Baroda or Surat or Bhavnagar.
SQL>SELECT * FROM stud WHERE city IN ('Rajkot', 'Ahmadabad', 'Baroda', 'Surat', 'Bhavnagar');

If you use NOT clause with IN than the output will become negative.

SQL>SELECT * FROM stud WHERE city NOT IN ('Rajkot', 'Ahmadabad', 'Baroda', 'Surat','Bhavnagar');

THE ORACLE SPECIAL TABLE DUAL:

DUAL is a table Owner by SYS. SYS owns the Data Dictionary and DUAL is part of the Data Dictionary. Dual is a small oracle worktable, which consists of only one row, one column, and contains the value X in that column. Besides arithmetic calculation, it also support date retrieval and it's formatting.

To facilitate such calculation via a SELECT , oracle provides a DUMMY table call DUAL, against which SELECT statement that are required to manipulate numeric literals can be fired and appropriate output obtained.

The structure of the dual table if viewed is as follow

SQL> DESC DUAL

```
NAME                Null?   TYPE
--------------------------------------------------
Dummy                       Varchar2(1)
```

If the dual table is required for records the output is as follow

OUTPUT

```
D
---
X
```

Example:

SQL > Select 2*2 FROM dual

SPECIAL DATE FORMATS USING TO CHAR FUNCTION:

Some time you need that date value is required to be displayed in a special format. For example instead of displaying 03-jan-81 you would like to display like 3rd January, 1981.

To do the above things oracle provides you a list of special attributes which are used with TO_CHAR function.

Use of 'TH' in TO CHAR function: When you need the date to be displayed in the format like 2nd, 3rd, 4th etc. at that time this will be useful. For example

SQL> SELECT no, name, TO_CHAR(dob,'DDTH-MON-YY') FROM stud;

```
     NO   NAME      TO_CHAR (DOB
  ----------  ----------    -----------
      1    ABC        01ST-JAN-80
      2    XYZ        22ND-MAY-82
      3    MNO        05TH-APR-81
```

Use of 'SP' in TO CHAR function – it will display date in character format as under

SQL> SELECT no, name, TO_CHAR(dob,'DDSP-MON-YY') FROM stud;

```
     NO    NAME      TO_CHAR(DOB,'DDSP-M
----------    ----------    -------------------
      1    ABC        ONE-JAN-80
      2    XYZ        TWENTY-TWO-MAY-82
      3    MNO        FIVE-APR-81
```

Use of 'SPTH"' in TO CHAR function: When you use SPTH it will convert the date into character format and also add the 'TH' behind it. For example

SQL> SELECT no, name, TO_CHAR(dob,'DDSPTH-MON-YY') FROM stud;

```
      NO    NAME      TO_CHAR(DOB,'DDSPTH-M
----------     ----------     ---------------------
       1      ABC        FIRST-JAN-80
       2      XYZ        TWENTY-SECOND-MAY-82
       3      MNO        FIFTH-APR-81
```

EXPRESSIONS

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.

SQL>SELECT amount FROM checks;

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

SQL>SELECT NAME, ADDRESS, PHONE FROM ADDRESSBOOK;

GROUPING OF DATA IN A TABLE :

The GROUP BY clause is used to break down the results of a query based on a column or columns. Once the rows are subdivided into groups, the aggregate functions described earlier can be applied to these groups. Note the following rules about using the GROUP BY clause:

1. All columns in a SELECT statement that are not in the GROUP BY clause must be part of an aggregate function.
2. The WHERE clause can be used to filter rows from the result before the grouping functions are applied.
3. The GROUP BY clause also specifies the sort order; this can be overridden with an ORDER BY clause.
4. Column aliases cannot be used in the GROUP BY clause.
5. GROUP BY clause does not support the use of column alias, but the actual names.
6. GROUP BY clause can only be used with aggregate functions like SUM, AVG, COUNT, MAX, and MIN. If it is used with single row functions, Oracle throws and exception as "**ORA-00979: not a GROUP BY expression**".
7. Aggregate functions cannot be used in a GROUP BY clause. Oracle will return the "**ORA-00934: group function not allowed**" here error message.

Syntax:

SELECT expression1, expression2, ... expression_n, aggregate_function (aggregate_expression) FROM tables [WHERE conditions]
GROUP BY expression1, expression2, ... expression_n;

Parameters or Arguments

expression1, expression2, ... expression_n

The expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.

aggregate_function

It can be a function such as SUM, COUNT, MIN, MAX, or AVG functions.

aggregate_expression

HAVING CLAUSE:

The HAVING clause is analogous to the WHERE clause, except that the HAVING clause applies to aggregate functions instead of individual columns or single-row function results. A query with a HAVING clause still returns aggregate values, but those aggregated summary rows are filtered from the query output based on the conditions in the HAVING clause.

Example

1. list the no. of student in each standard in which there are greater than or equal to 2.

SQL>SELECT std,COUNT(name)FROM stud GROUP BY std having COUNT(name) >=2;

2. list the number of student in standard greater than 10 in which there are greater than or equal to

SQL>SELECT std, COUNT (name) FROM stud WHERE std>10 GROUP BY std HAVING COUNT(name) >=2 ;

ROLLUP OPERATOR

ROLLUP enables a SELECT statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates a grand total. ROLLUP is a simple extension to the GROUP BY clause, so its syntax is extremely easy to use. The ROLLUP extension is highly efficient, adding minimal overhead to a query.

The action of ROLLUP is straight forward: it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. Finally, it creates a grand total.

SQL> SELECT std, COUNT (name) FROM stud GROUP BY ROLLUP(std);

```
     STD    COUNT(NAME)
----------  -----------
      10        1
      11        3
      12        3
                7
```

SQL> select job, deptno, count(job) from emp group by Rollup(deptno,job);

```
JOB                       DEPTNO       COUNT(JOB)
---------                 ----------   ----------
CLERK                     10            1
MANAGER                   10            1
PRESIDENT                 10            1
                          10            3
ANALYST                   20            2
CLERK                     20            2
MANAGER                   20            1
                          20            5
CLERK                     30            1
MANAGER                   30            1
SALESMAN                  30            4
                          30            6
                                       14
13 rows selected.
```

CUBE OPERATOR : The CUBE operator takes the ROLLUP operator a step further and provides rollups of aggregate functions in both directions across the fields that are to be aggregated.

Example :

```
SQL> select job, deptno, count(job) from emp group by cube (deptno,job);
JOB                DEPTNO  COUNT(JOB)
---------          ----------  ----------
CLERK                  10       1
MANAGER                10       1
PRESIDENT              10       1
                       10       3
ANALYST                20       2
CLERK                  20       2
MANAGER                20       1
                       20       5
CLERK                  30       1
MANAGER                 30      1
SALESMAN               30       4
                       30       6
ANALYST                         2
CLERK                           4
MANAGER                         3
PRESIDENT             1
SALESMAN                        4
                     14
18 rows selected.
```

SUB-QUERIES:

A Sub-query or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

A sub-query is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Sub-queries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that sub-queries must follow:

- Sub-queries must be enclosed within parentheses.
- A sub-query can have only one column in the SELECT clause, unless multiple columns are in the main query for the sub-query to compare its selected columns.
- An ORDER BY cannot be used in a sub-query, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a sub-query.
- Sub-queries that return more than one row can only be used with multiple value operators, such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A sub-query cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a sub-query; however, the BETWEEN operator can be used within the sub-query.

Types of Subqueries

Single Row Sub Query

A single-row sub-query is used when the outer query's results are based on a single, unknown value. Although this query type is formally called "single-row," the name implies that the query returns multiple columns-but only one row of results. However, a single-row sub-query can return only one row of results consisting of only one column to the outer query.

In the below SELECT query, inner SQL returns only one row i.e. the minimum salary for the company. It in turn uses this value to compare salary of all the employees and displays only those, whose salary is equal to minimum salary.

SQL> SELECT **first_name, salary, department_id FROM employees WHERE salary = (SELECT MIN (salary) FROM employees);**

A HAVING clause is used when the group results of a query need to be restricted based on some condition. If a sub-query's result must be compared with a group function, you must nest the inner query in the outer query's HAVING clause.

SQL > SELECT department_id, MIN (salary) FROM employees GROUP BY department_id HAVING MIN (salary) < (SELECT AVG (salary) FROM employees)

Multiple Row Sub Query

Multiple-row sub-queries are nested queries that can return more than one row of results to the parent query. Multiple-row sub-queries are used most commonly in WHERE and HAVING clauses. Since it returns multiple rows, it must be handled by set comparison operators (IN, ALL, ANY).While IN operator holds the same meaning as discussed in earlier chapter, ANY operator compares a specified value to each value returned by the sub query while ALL compares a value to every value returned by a sub query.

Below query shows the error when single row sub query returns multiple rows.

```
SQL > SELECT first_name, department_id FROM employees WHERE department_id
= (SELECT department_id  FROM employees WHERE LOCATION_ID = 100)
department_id = (select
                *
ERROR at line 4:
ORA-01427: single-row sub-query returns more than one row
```

Usage of Multiple Row operators

- [> ALL] More than the highest value returned by the sub-query
- [< ALL] Less than the lowest value returned by the sub-query
- [< ANY] Less than the highest value returned by the sub-query
- [> ANY] More than the lowest value returned by the sub-query
- [= ANY] Equal to any value returned by the sub-query (same as IN)

Above SQL can be rewritten using IN operator like below.

SQL > SELECT first_name, department_id FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE LOCATION_ID = 100)

Correlated Sub Query

As opposed to a regular sub-query, where the outer query depends on values provided by the inner query, a correlated sub-query is one where the inner query depends on values provided by the outer query. This means that in a correlated sub-query, the inner query is executed repeatedly, once for each row that might be selected by the outer query.

Correlated sub-queries can produce result tables that answer complex management questions.

Consider the below SELECT query. Unlike the sub-queries previously considered, the sub-query in this SELECT statement cannot be resolved independently of the main query. Notice that the outer query specifies that rows are selected from the employee table with an alias name of e1. The inner query compares the employee department number column (DepartmentNumber) of the employee table with alias e2 to the same column for the alias table name e1.

```
SQL> SELECT EMPLOYEE_ID, salary, department_id FROM   employees E WHERE
salary > (SELECT AVG(salary)  FROM   EMP T  WHERE E.department_id =
T.department_id)
```

Multiple Column Sub Query

A multiple-column sub query returns more than one column to the outer query and can be listed in the outer queries FROM, WHERE, or HAVING clause. For example, the below query shows the employee's historical details for the ones whose current salary is in range of 1000 and 2000 and working in department 10 or 20.

```
SQL > SELECT first_name, job_id, salary FROM emp_history WHERE (salary,
department_id) in (SELECT salary, department_id  FROM employees  WHERE
salary BETWEEN 1000 and 2000   AND department_id BETWEEN 10 and 20)
ORDER BY first_name;
```

When a multiple-column sub query is used in the outer query's FROM clause, it creates a temporary table that can be referenced by other clauses of the outer query. This temporary table is more formally called an inline view. The sub-query's results are treated like any other table in the FROM clause. If the temporary table contains grouped data, the grouped subsets are treated as separate rows of data in a table. Consider the FROM clause in the below query. The inline view formed by the sub-query is the data source for the main query.

```
SQL> SELECT *   FROM (SELECT salary, department_id FROM employees WHERE
salary BETWEEN 1000 and 2000);
```

CONCATENATING OPERATOR (||)

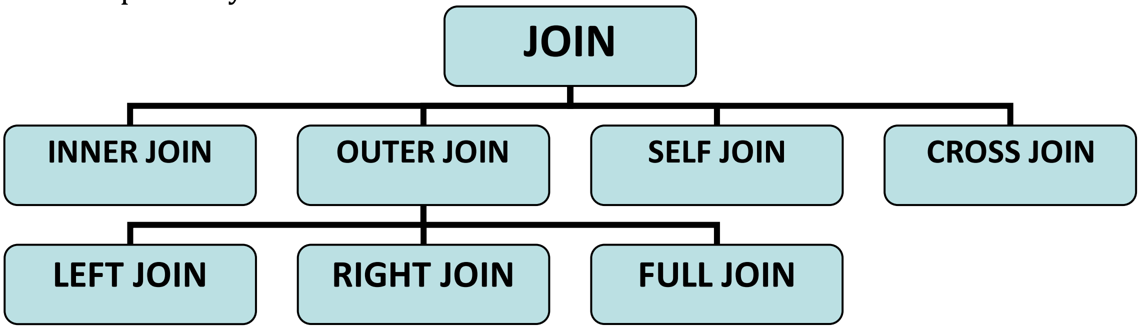
When we want to develop English like sentence at that time concatenating operator will be useful. For example

SQL>SELECT name || ' is staying in ' || city FROM stud;

Output will be like --- XYZ is staying in Rajkot.

JOINS :

In oracle sometime it is necessary to work with multiple tables just like they are single table. Then a single SQL statement can manipulate data from all the tables. Joins are used to achieve this. Tables are joined on columns that have same data type. Generally this columns are primary key and foreign key of master table and transaction table respectively.



Inner Join

Inner joins are also known as **Equi Joins**, this is the most common joins in SQL*Plus. This tipe of joins can be used in a situation where selecting only those rows that have a value in common in the column specified is required. In short inner join returns all the rows from both the tables where there is a match.

Example :

SQL> SELECT stud.grno, stud.name, marks.total FROM stud, marks WHERE stud.grno = marks.grno;

OR

SQL> SELECT s.grno, s.name, m.total FROM stud s, marks m WHERE s.grno = m.grno;

```
   GRNO  NAME        TOTAL
----------  ----------   ----------
       1     A           100
       2     B           125
       3     C           105
       4     D            95
       5     E           150
```

Outer Join:

Outer joins are similar to inner joins, but it will give us more flexibility when selecting the data from the related tables. This type of join will be used in a special situation like when you need to select rows from the table on the left, it will select all the records from the left table and then it will not check for the record on the right side. Or if we say Right outer join then it will select all the records from the right side table regardless of the record available on the left side.

Left outer join

ANSI STYLE

SQL>SELECT a.name, b.total FROM stud a LEFT JOIN marks b on a.grno=b.grno;

Theta style

SQL>SELECT a.name, b.total FROM stud a, marks b WHERE a.grno=b.grno(+);

Right outer join

ANSI STYLE

SQL>SELECT a.name,b.total FROM stud a RIGHT JOIN marks b on a.grno=b.grno;

Theta style

SQL>SELECT a.name, b.total FROM stud a, marks b WHERE a.grno(+)=b.grno;

CROSS JOIN

A cross join gives us the output as Cartesian product of both the tables. This means that in this type o f join it will combine each and every row of left table with each and every row of right table. This type of join is useful when you need to get all the possible combination of all the rows and columns from both the tables. These kinds of joins are usually not preferred because it may take long time to run and a huge result will be produced. For Example

SQL>SELECT a.name, b.total FROM stud a marks b;

SELF JOIN :

In some situation you need to join a table to itself, at that time the same table will be worked as left table as well as right table. To join a table to itself you need two copies of the same table in the memory. Remember that here you have to open both the copies of table using alias. Suppose you have a table named EMP and in that table you have following data.

EMPNO	EMPNAME	MNGRNO
1	A	
2	B	
3	C	1
4	D	1
5	E	2
6	F	2

Now you want to retrieve the employee name and respective manager name.

```
SQL>SELECT a.empname  "EMP NAME",  b.empname "MNGR NAME"  FROM
      emp a, emp b WHERE a.mngrno=b.empno;
```

OUTPUT

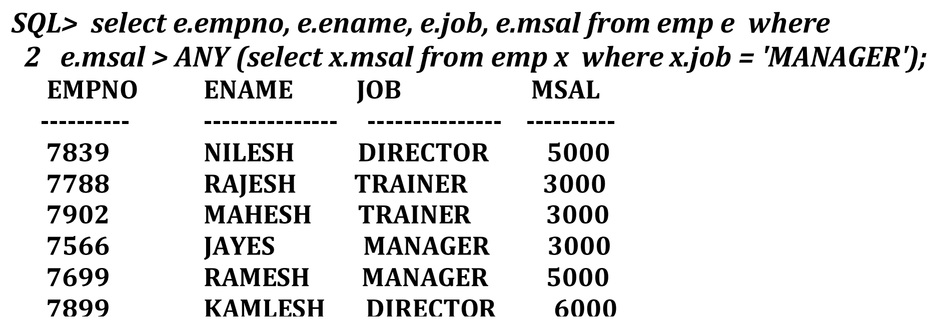
```
EMP NAME   MNGR NAME
----------   ----------
C            A
D            A
E            B
F            B
```

ANY AND ALL OPERATORS :

SQL allows you to combine standard comparison operators (<, >, =, and so on) with sub queries returning any number of rows. You can do that by specifying ANY or ALL between the comparison operator and the sub query. Suppose you have following table and data.

```
SQL> select * from emp;
  EMPNO            ENAME      JOB            MSAL
 ----------        --------------  --------------  ---------
   7839            NILESH     DIRECTOR       5000
   7788            RAJESH     TRAINER        3000
   7902            MAHESH     TRAINER        3000
   7566            JAYES      MANAGER        3000
   7698            BABU       MANAGER        2500
   7699            RAMESH     MANAGER        5000
   7899            KAMLESH    DIRECTOR       6000
```

Now you want to find out the employees who have more salary than at Least one manager. To do this work you can use ANY operator as under.



```
SQL>  select e.empno, e.ename, e.job, e.msal from emp e  where
  2   e.msal > ANY (select x.msal from emp x  where x.job = 'MANAGER');
   EMPNO       ENAME       JOB             MSAL
  ----------   ------------  ------------  ----------
   7839        NILESH       DIRECTOR       5000
   7788        RAJESH       TRAINER        3000
   7902        MAHESH       TRAINER        3000
   7566        JAYES         MANAGER       3000
   7699        RAMESH       MANAGER        5000
   7899        KAMLESH      DIRECTOR        6000
```

anager. To do this work you can use ALL operators as under.

```
SQL>  select e.empno, e.ename, e.job, e.msal from emp e  where
  2   e.msal > ALL (select x.msal from emp x  where x.job = 'MANAGER');
   EMPNO     ENAME        JOB            MSAL
   ----------    ---------------   --------------     ----------
    7699       KAMLESH     DIRECTOR       6000
```

EXISTS OPERATOR

EXISTS returns true in WHERE clause if the sub query that follows it returns at least one row. The select clause in the sub query can be any field name or asterisk. Suppose you have table named job with following data and above table named emp.

SQL> select * from job;

```
JOB
---------------
DIRECTOR
TRAINER
MANAGER
ACCOUNTANT
CLERK
```

The following query list from the job table all the records that have matching job in the emp table. Thus the result of this query is the list of all the job that employees of company have.

SQL> SELECT job.job FROM job WHERE EXISTS (SELECT * from emp where emp.job=job.job)

```
JOB
---------------
DIRECTOR
TRAINER
MANAGER
```

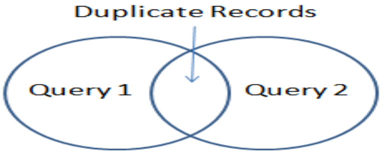
SET OPERATORS (UNION, INTERSECT AND MINUS):

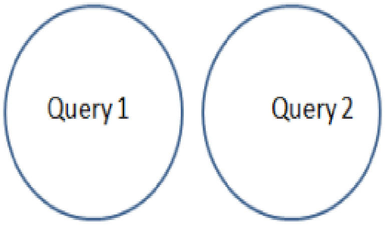
You can use the SQL set operators UNION, MINUS, and INTERSECT to combine the results of two independent query blocks into a single result. These SQL operators correspond with the union, minus, and intersect operators you know from mathematics. The UNION, MINUS, and INTERSECT operators cannot be applied to any arbitrary set of two queries. The intermediate (separate) results of queries Q1 and Q2 must be “compatible” in order to use them as arguments to a set operator. In this context, compatibility means the following:

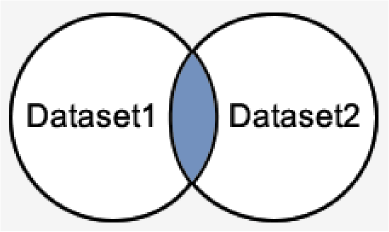
- Q1 and Q2 must select the same number of column expressions.
- The data types of those column expressions must match.
- Some other rules and guidelines for SQL set operators are the following:
- The result table inherits the column names (or aliases) from Q1.
- Q1 cannot contain an ORDER BY clause.
- If you specify an ORDER BY clause at the end of the query, it doesn't refer to Q2, but rather to the total result of the set operator.

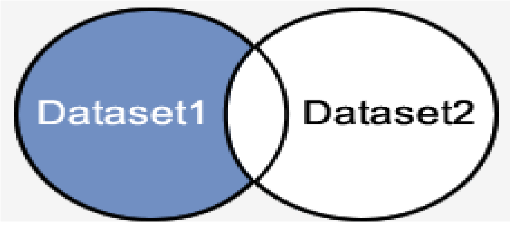
Example: Suppose you have following tables

Operator	Description	Example

UNION	Returns all distinct rows selected by either query.
	
	
		SELECT * FROM
		(SELECT ENAME FROM EMP WHERE JOB = 'CLERK'
		UNION
		SELECT ENAME FROM EMP WHERE JOB = 'ANALYST');

UNION ALL	Returns all rows selected by either query, including all duplicates.
	
	
		SELECT * FROM
		(SELECT SAL FROM EMP WHERE JOB = 'CLERK'
		UNION ALL
		SELECT SAL FROM EMP WHERE JOB = 'ANALYST');

INTERSECT	Returns all distinct rows selected by both queries.
	
	
		SELECT * FROM orders_list1
		INTERSECT
		SELECT * FROM orders_list2

MINUS	Returns all distinct rows selected by the first query but not the second.
	
	
		SELECT * FROM (SELECT SAL FROM EMP WHERE JOB = 'PRESIDENT'
		MINUS
		SELECT SAL FROM EMP WHERE JOB = 'MANAGER');

FUNCTIONS

AGGREGATE FUNCTIONS: Aggregate functions are those types of functions which are useful while preparing the queries with GROUP BY clause. Generally these functions are useful on a group of data of a table.

(1) AVG() : This function is useful to get Average of the given numeric field of a table.
Syntax: **AVG(<field name>)**
Example: SELECT AVG(total) FROM marks;
Explanation: The above function will return the average of field total.

(2) MIN() : This function is useful to get Minimum out of the given numeric field of a table.
Syntax: **MIN(<field name>)**
Example: SELECT MIN(total) FROM marks;
Explanation: The above function will return the Minimum total out of field total.

(3) MAX() : This function is useful to get Maximum out of the given numeric field of a table.
Syntax: **MAX(<field name>)**
Example: SELECT MAX(total) FROM marks;
Explanation: The above function will return the Maximum total out of field total.

(4) SUM() : This function is useful to get sum of the given numeric field of a table.
Syntax: **SUM(<field name>)**
Example: SELECT SUM(total) FROM marks;
Explanation: The above function will return the sum of field total.

(5) COUNT(): This function is useful to count number of records available in particular field of a table.
Syntax: **COUNT (<field name>)**
Example: SELECT COUNT (total) FROM marks;
Explanation: The above function will return the number of records in the total fields.
If we give SELECT COUNT (*) FROM marks; then it will return total no. of records in the table.

NUMERIC FUNCTIONS (Arithmetic Functions):

Many of the uses you have for the data you retrieve involve mathematics. At that time these functions are useful.
(1)ABS () : The ABS function returns the absolute value of the number you point to.
Syntax: **ABS(n)**
Example: SELECT ABS(-5) FROM DUAL;
Explanation: The output of above function is 5.
ABS changes all the negative numbers to positive and leaves positive numbers alone.

(2) POWER() : To raise one number to the power of another, use POWER. In this function the first argument is raised to the power of the second:
Syntax: **POWER(m,n)**
Example: SELECT POWER(3,2) FROM DUAL;
Explanation: The output of above function is 9.

(3) ROUND() : This function is useful to make rounding the given fractional number up to given number of points. If number of decimal places is omitted it will rounded to the decimal place 0.

Syntax: **ROUND(m,n)**
Example: SELECT ROUND(15.19,1) FROM DUAL;
Explanation: The output of above function is 15.2.

(4) SQRT() : The function SQRT returns the square root of an argument. Because the square root of a negative number is undefined, you cannot use SQRT on negative numbers

Syntax: **SQRT(n)**
Example: SELECT SQRT (4) FROM DUAL;
Explanation: The output of above function is 2.

(5) EXP() :EXP enables you to raise e (e is a mathematical constant used in various formulas) to a power. Value of e=2.71828183

Syntax: **EXP(n)**
Example: SELECT EXP (5) FROM DUAL;
Explanation: The output of above function is 148.413159.

(6) EXTRACT(): This function is useful to Extract a part of date /month/year from given value of date.

Syntax: **EXTRACT(<year/month/day. FROM <date>)**
Example: SELECT EXTRACT (year from sysdate) FROM DUAL;
Explanation: The output of above function is 2008.

(7) GREATEST() : This function is useful to find the maximum value out of given values.

Syntax: **GREATEST (expr1, expr2……exprn)**
Example: SELECT GREATEST (5, 10, 5) FROM DUAL;
Explanation: The output of above function is 15.

If you use character type of data as argument then this function will return the highest value according to ASCII value.

Example: SELECT GREATEST ('A', 'C' ,'B') FROM DUAL;
Explanation: The output of above function is C.

(8) LEAST() : This function is useful to find the minimum value out of given values.

Syntax: **LEAST (expr1, expr2……exprn)**
Example: SELECT LEAST (5, 10, 5) FROM DUAL;
Explanation: The output of above function is 5.

If you use character type of data as argument then this function will return the lowest value according to ASCII value.

Example: SELECT LEAST ('A', 'C' ,'B') FROM DUAL;
Explanation: The output of above function is A.

(9) MOD (m,n) : This function is useful to find remainder after the division of m by n.

Syntax: **MOD (m,n)**
Example: SELECT MOD (10,3) FROM DUAL;
Explanation: The output of above function is 1.

(10) TRUNC (m,n) : This function is useful to truncate decimal number m up to n places.
Syntax: **TRUNC(m,n)**
Example: SELECT TRUNC (10.6582,2) FROM DUAL;
Explanation: The output of above function is 10.65.

(11) FLOOR() : This function is useful to find nearest integer value which is less than or equal to given no.
Syntax: **FLOOR (m,n)**
Example: SELECT FLOOR (10.6) FROM DUAL;
Explanation: The output of above function is 10.

(12) CEIL() : This function is useful to find nearest integer value which is greater than or equal to given no.
Syntax: **CEIL (n)**
Example: SELECT CEIL (10.6) FROM DUAL;
Explanation: The output of above function is 11.
(13) log() :

The Oracle LOG() function is used to return the logarithm, base B of N. The base can be any positive value other than 0 or 1 and the number can be any positive value. The function takes any numeric or nonnumeric data type (can be implicitly converted to a numeric data type) as an argument.

If any argument is BINARY_FLOAT or BINARY_DOUBLE, then the function returns BINARY_DOUBLE. Otherwise, the function returns NUMBER.

Syntax: LOG(B, N);

Name	**Description**
B	Indicates the base of N.
N	A number.

Example: SELECT LOG(10,1000) FROM dual;
Explanation: The output of above function is 3

(14) log10()

The LOG10 function computes the logarithm base 10 of an expression.

Syntax : LOG10(expression)

Expression

The value of expression must be greater than zero. When the value is equal to or less than 0 (zero), LOG10 returns an NA value.

Return Value

DECIMAL

Examples: SELECT LOG10 (1000) FROM DUAL

This example uses the LOG10 function to calculate the base 10 logarithm of 1,000.

O/P

3.00

(15) sign () :

This function is used to return the sign of a given number. This function takes as an argument any numeric data type and any nonnumeric data can also comes in the argument but that can be implicitly converted to number and returns number.

The function returns:

- 1 when the value of the argument is positive
- -1 when the value of the argument is negative
- 0 when the value of the argument is 0

For binary floating-point numbers (BINARY_FLOAT and BINARY_DOUBLE), this function returns the sign bit of the number. The sign bit is:

- -1 if n<0
- +1 if n>=0 or n=NaN

Syntax : **SIGN(n)**

Here ,

n is A number whose sign is to be retrieved.

Example: SELECT SIGN(-145), SIGN(0), SIGN(145) FROM dual;

```
O/P
SIGN(-145)   SIGN(0)  SIGN(145)
     -1          0          1
```

(16) SIN()

The function takes any numeric or nonnumeric data type (can be implicitly converted to a numeric data type) as an argument. If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise, the function returns the same numeric data type as the argument.

Syntax : **sin(n)**

Here, n is A number whose SIN value is to be retrieved.

Example: SELECT **SIN(1)** FROM **dual;**

```
O/P
SIN(1)
     .841470985
```

(17) COS()

The COS() function is used to get the cosine of a number where the number is given in radians. The function takes any numeric or nonnumeric data type (can be implicitly converted to a numeric data type) as an argument. If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise, the function returns the same numeric data type as the argument

Syntax : COS(N)

Here,

n is a number whose cosine value is to be retrieved.

Example: SELECT COS(180 * 3.14159265359/180) "Cosine 180 degrees" FROM DUAL;

O/P

```
Cosine 180 degrees
-------------------------
          -1
```

(18) TAN() Function

This function returns the tangent of n (an angle expressed in radians). The function takes any numeric or nonnumeric data type (can be implicitly converted to a numeric data type) as an argument.

If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise, the function returns the same numeric data type as the argument

Syntax: **TAN(n)**

Here

n is a number

Example: SELECT TAN(2.465)FROM dual;

O/P

```
TAN(2.465)
----------
-.80304132
```

(19) SINH() Function

This SINH() function returns the hyperbolic sine of n (an angle expressed in radians). The function takes any numeric or nonnumeric data type (can be implicitly converted to a numeric data type) as an argument.

If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise, the function returns the same numeric data type as the argument.

Syntax : SINH(n)

Here,

n is A number whose sine value is to be retrieved.

Example: SELECT SINH(.6) "Hyperbolic sine of .6" FROM dual;

O/P

```
Hyperbolic sine of .6
 -------------------------
          .636653582
```

(20)Decode() Function

DECODE is a function in Oracle and is used to provide if-then-else type of logic to SQL. It is not available in MySQL or SQL Server.

Syntax

SELECT DECODE ("column_name", "search_value_1", "result_1", ["search_value_n", "result_n"], {"default_result"});

"search_value" is the value to search for, and "result" is the value that is displayed.

[ ] means that the "search_value_n", "result_n" pair can occur zero, one, or more times.

Example Table Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
San Francisco	300	Jan-08-1999
Boston	700	Jan-08-1999

To display 'LA' for 'Los Angeles', 'SF' for 'San Francisco', 'SD' for 'San Diego', and 'Others' for all other cities, we use the following SQL,

Example

SELECT DECODE (Store_Name, 'Los Angeles', 'LA', 'San Francisco', 'SF', 'San Diego', 'SD', 'Others') "Area", Sales, Txn_Date FROM Store_Information;

"Area" is the name given to the column that the DECODE function operates on.

Result:

Area	Sales	Txn_Date
LA	1500	Jan-05-1999

SD	250	Jan-07-1999
SF	300	Jan-08-1999
Others	700	Jan-08-1999

To achieve what DECODE does SQL Server, we would use the CASE function.

STRING FUNCTIONS (Character Functions)

Many implementations of SQL provide functions to manipulate characters and strings of characters. This section covers the most common character functions.

(1) CHR () : CHR returns the character equivalent of the number it uses as an argument.
Syntax: **CHR (n)**
Example: SELECT CHR (65) FROM DUAL;
Explanation: The output of above function is A.
(2) LOWER () : This function is useful to convert the given string in the lower case..
Syntax: **LOWER (char/string)**
Example: SELECT LOWER (‘PGDCa’) FROM DUAL;
Explanation: The output of above function is pgdca.
(3) UPPER () : This function is useful to convert the given string in the lower case..
Syntax: **UPPER (char/string)**
Example: SELECT UPPER (‘pgdca’) FROM DUAL;
Explanation: The output of above function is PGDCA.

(4) SUBSTR () : This three-argument function enables you to take a piece out of a target string. The first argument is the target string. The second argument is the position of the first character to be output. The third argument is the number of characters to show.
Syntax: **SUBSTR(string, start position, no of charscters)**
Example: SELECT SUBSTR (‘pgdca’,2,2) FROM DUAL;
Explanation: The output of above function is gd.

(5) ASCII () : This function is useful to get the ASCII code of given character. If we give string as parameter then it will return the ASCII code of the first character of the string.
Syntax: **ASCII (character, string)**
Example: SELECT ASCII (‘ABC’) FROM DUAL;
Explanation: The output of above function is 65.
(6) INSTR (): This function is useful to find the instance of the second string character in the first string. This function will return on which position the second string is there in the first string.
Syntax: **INSTR(string1, string2, [start position, nth occurrence])**
Example: SELECT INSTR (‘PGDCA’,’G’) FROM DUAL;
Explanation: The output of above function is 2.

(7)TRANSLATE (): This function is useful to replace a sequence of character in a string with another set of string.
Syntax: **TRANSLATE (string1, string tor eplace, replacement string)**

Example: SELECT TRANSLATE ('PGDCA','G','X') FROM DUAL;
Explanation: The output of above function is PXDCA.

(8) LENGTH (): This function is useful to get the number of characters in the given string.
Syntax: LENGTH (string)
Example: SELECT LENGTH ('PGDCA') FROM DUAL;
Explanation: The output of above function is 5.

(9) LTRIM () : This function is useful to remove the set of characters from the left side from given string.
Syntax: **LTRIM (string, set)**
Example: SELECT LTRIM ('PPPPGPDCA','P') FROM DUAL;
Explanation: The output of above function is GPDCA.

(10) RTRIM () : This function is useful to remove the set of characters from the right side from given string.
Syntax: RTRIM (string, set)
Example: SELECT RTRIM ('PGDCAAA','A') FROM DUAL;
Explanation: The output of above function is PGDC.

(11) TRIM () : This function is useful to remove the set of characters from left side, right side or from both the side from given string.
Syntax: TRIM(<leading>|<trailing>|<both> <trim character> from <string>)

Example: SELECT TRIM ('   INDIA   ') FROM DUAL;
Explanation: The output of above function is INDIA.

Example: SELECT TRIM (leading 'X' from 'XXXINDIA') FROM DUAL;
Explanation: The output of above function is INDIA.

Example: SELECT TRIM (trailing 'X' from 'INDIAXXX') FROM DUAL;
Explanation: The output of above function is INDIA.

Example: SELECT TRIM (both 'X' from 'XXXXINDIAXXX') FROM DUAL;
Explanation: The output of above function is INDIA.

(12) LPAD (): This function is useful to pad given string from the left side with given character for a number of times specified by n. Remember that this function will pad the given string by given character from the left side until the total length of string will become n.
Syntax: **LPAD(string, n character, string to be padded)**
Example: SELECT LPAD ('PGDCA',10,'*') FROM DUAL;
Explanation: The output of above function is *****PGDCA.

(13)RPAD(): This function is useful to pad given string from the right side with given character for a number of times specified by n. Remember that this function will pad the given string by given character from the right side until the total length of string will become n.
Syntax: **RPAD(string, n character, string to be padded)**

Example: SELECT RPAD ('PGDCA',10,'*') FROM DUAL;
Explanation: The output of above function is PGDCA*****.

(14) CONCAT () function

This function returns the result (a string) of concatenating two string values. This function is equivalent to the concatenation operator (||).

Syntax :

CONCAT(char1, char2)

Name	Description	Data Types
char1, char2	A string value to concatenate to the other values.	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB

Return Value Type :

CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB

If there are two different data types in concatenations Oracle Database returns the data type that results in a lossless conversion.

Therefore, if one of the arguments is a LOB, then the returned value is a LOB. If one of the arguments is a national data type, then the returned value is a national data type. Here are some examples:

- CONCAT(CLOB, NCLOB) returns NCLOB
- CONCAT(NCLOB, NCHAR) returns NCLOB
- CONCAT(NCLOB, CHAR) returns NCLOB
- CONCAT(NCHAR, CLOB) returns NCLOB

Example: SELECT CONCAT ('JJKCC', ' RAJKOT') AS "College" from dual;
O/P

```
College
-------------
JJKCC RAJKOT
```

(15) INITCAP () Function

This function sets the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Syntax : INITCAP(string1)

Name	Description	Data Types
string1	A string whose first character in each word will be converted to uppercase and the rest characters will be converted to lowercase.	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB

Return Type: CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB
Example: SELECT INITCAP('ivan bayross') "Emp. Name" FROM DUAL;
O/P

```
Emp. Name
-----------
Ivan Bayross
```

(16) REPLACE () Function

This function is used to return char with every occurrence of search_string replaced with replacement_string. If replacement_string is omitted or null, then all occurrences of search_string are removed. When search_string is null, then char is returned.

Syntax :

REPLACE(char, search_string [, replacement_string])

Name	Description	Data Type
Char	A string	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB
search_string	A string, to be searched.	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB
replacement_string	A string, to be replaced.	CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB

Return Value : CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB

Example: SELECT REPLACE('MAN and MAT','M','F') "New String" FROM DUAL;

```
New String
-----------
FAN and FAT
```

(17)SOUNDEX() function

This function returns a character string containing the phonetic representation of char. This function lets you compare words that are spelled differently, but sound alike in English.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

Syntax: SOUNDEX (char)

Name	Description	Data Types
char	A string whose SOUNDEX string is to be retrieved.	CHAR, VARCHAR2, NCHAR, or NVARCHAR2

Return Value : CHAR, VARCHAR2, NCHAR, or NVARCHAR2

Example:

SELECT last_name, first_name FROM employees WHERE SOUNDEX(last_name) = SOUNDEX('SMYTHE') ORDER BY last_name, first_name;

O/P

```
AST_NAME  FIRST_NAME
---------- ----------
Smith     Lindsey
Smith     William
```

Conversion Functions

(1) TO_NUMBER () : This function is useful to convert the given string into numbers.

Syntax: **TO_NUMBER(char)**

Example: SELECT TO_NUMBER (‘100’) FROM DUAL;

Explanation: The output of above function is 100.

(2) TO_CHAR () –Numeric conversion : This function is useful to convert the given number into string with given format

Syntax: **TO_CAHR(number, format)**

Example: SELECT TO_CHAR (58452, ‘$99,999’) FROM DUAL;

Explanation: The output of above function is 100.

DATE FUNCTION

(1) ADD_MONTHS () : This function is useful to return date by adding number of months in given date.
Syntax: **ADD_MONTHS(date, no. of months)**
Example: SELECT ADD_MONTHS (sysdate, 3) FROM DUAL;
Explanation: The output of above function is 28-FEB-09.

(2) LAST_DAY () : This function is useful to get last date of the month for the given date.
Syntax: **LAST_DAY(date)**
Example: SELECT LAST_DAY (sysdate) FROM DUAL;
Explanation: The output of above function is 30-NOV-08.

(3) MONTHS_BETWEEN (): This function is useful to find number of months between given two dates. Remember that dae1 must be greater than date2 otherwise the output will be in minus. This function will also return the number of months in fraction if the date part of the dates is not equal.
Syntax: **MONTHS_BETWEEN (date1, date2)**
Example: SELECT MONTHS_BETWEEN ('1-feb-09','1-jan-09') FROM DUAL;
Explanation: The output of above function is 1.

(4) NEXT_DAY () : This function is useful to find the date on which the given day will come after the given date.
Syntax: **NEXT_DAY(date,day)**
Example: SELECT NEXT_DAY (sysdate,'Saturday') FROM DUAL;
Explanation: The output of above function is 29-NOV-08.

(5) TO_CHAR () –Date conversion : This function is useful to convert the given date into string with given format
Syntax: **TO_CAHR(date, format)**
Example: SELECT TO_CHAR (sysdate, 'Month DD, YYYY') FROM DUAL;
Explanation: The output of above function is November 28, 2008.

(6) TO_DATE () : This function is useful to convert the given string into date with given format
Syntax: **TO_DATE(string, format)**
Example: SELECT TO_DATE ('28-NOV-08',' DD-MON-YY') FROM DUAL;
Explanation: The output of above function is 28-NOV-08.

(7)SYSDATE() Function

The SYSDATE function is used to get the current date and time set for the operating system on which the database resides.
Syntax: **SYSDATE**

Note: The function requires no parameters.

Example:

```
        SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW"  FROM DUAL;
NOW
-------------------
05-08-2017 15:39:45
```

(8) ROUND() Function

This function is used to get the date rounded to the unit specified by the format model. It operates according to the rules of the Gregorian calendar.

Syntax : ROUND(date [, fmt])

Name	**Description**
date	The specified date.
fmt (Optional)	The unit of measure to apply for rounding. If the parameter is not present, then a date is rounded to the nearest day.

Return Value Type :

The value returned is always of data type DATE. Date format models for the ROUND function :

Format Model	**Rounding Unit**
CC SCC	One greater than the first two digits of a four-digit year
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

Example:

```
        SELECT ROUND(TO_DATE ('16-SEP-2015'),'MONTH') "New Month",
ROUND(TO_DATE ('16-SEP-2015'),'YEAR') "New Year"  3) FROM DUAL;
```

O/P

```
          New Month      New Year
        ----------- ---------------------
        01-OCT-2015 01-JAN-2016
```

(9) TRUNC() Function

This function is used to get the date with the time portion of the day truncated to a specific unit of measure. It operates according to the rules of the Gregorian calendar.

Syntax: TRUNC(date [, fmt])

Name	**Description**
date	The date to truncate.
fmt (Optional)	The unit of measure for truncating. If fmt is not present, then the date is truncated to the nearest day.

Return Value Type :

The value returned is always of datatype DATE Date format models for the TRUNC function :

Format Model	Rounding Unit
CC SCC	One greater than the first two digits of a four-digit year
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

Example:

```
    SELECT TRUNC(TO_DATE('02-MAR-17','DD-MON-YY'), 'YEAR') "New Year" FROM DUAL;
```

O/P

```
        New Year
        ---------
        01-JAN-17
```

(10) systimestamp() Function

The SYSTIMESTAMP function is used to get the system date, including fractional seconds and time zone, of the system on which the database resides.

Syntax: SYSTIMESTAMP

Return Type :

The return type is TIMESTAMP WITH TIME ZONE.

Example: SELECT SYSTIMESTAMP FROM DUAL;

O/P

```
SYSTIMESTAMP
---------------------------------------------
01-MAY-15 03.45.45.761000 PM +05:30
```

GENERAL FUNCTION

(1) COALESCE() function

This function returns the first non-null expression in the list. If all expressions evaluate to null, then the COALESCE function will return null.

Syntax: COALESCE(expr1, expr2, ... expr_n)

Parameters or Arguments

expr1, expr2, ... expr_n to test for non-null values.

Example

```
SELECT COALESCE( address1, address2, address3 ) result FROM suppliers;
```

The above COALESCE function is equivalent to the following IF-THEN-ELSE statement:

```
IF address1 is not null THEN
        result := address1;
ELSIF address2 is not null THEN
        result := address2;
ELSIF address3 is not null THEN
        result := address3;
ELSE
        result := null;
END IF;
```

The COALESCE function will compare each value, one by one.

(2) CASE WHEN() Function

The CASE statement has the functionality of an IF-THEN-ELSE statement. Starting in Oracle 9i, you can use the CASE statement within a SQL statement.

Syntax

```
CASE [ expression ]
        WHEN condition_1 THEN result_1
        WHEN condition_2 THEN result_2
         ...
         WHEN condition_n THEN result_n
        ELSE result
END
```

Parameters or Arguments

expression

Optional. It is the value that you are comparing to the list of conditions. (ie: condition_1, condition_2, ... condition_n)

condition_1, condition_2, ... condition_n

The conditions that must all be the same datatype. The conditions are evaluated in the order listed. Once a condition is found to be true, the CASE statement will return the result and not evaluate the conditions any further.

result_1, result_2, ... result_n

Results that must all be the same datatype. This is the value returned once a condition is found to be true.

Note

- If no condition is found to be true, then the CASE statement will return the value in the ELSE clause.
- If the ELSE clause is omitted and no condition is found to be true, then the CASE statement will return NULL.
- You can have up to 255 comparisons in a CASE statement. Each WHEN ... THEN clause is considered 2 comparisons.

The CASE statement can be used in Oracle/PLSQL.

You could use the CASE statement in a SQL statement as follows: (includes the expression clause)

Example

```
SELECT table_name,
CASE owner
        WHEN 'SYS' THEN 'The owner is SYS'
        WHEN 'SYSTEM' THEN 'The owner is SYSTEM'
        ELSE 'The owner is another value'
END
FROM all_tables;
```

This is the column or expression that the aggregate_function will be used on.

tables

The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

WHERE conditions

Optional. The conditions that must be met for the records to be selected.

EXAMPLE

SELECT product, SUM(sale) AS "Total sales" FROM order_details GROUP BY product;

UNIT -2

DATA CONTROL AND TRANSACTION CONTROL COMMAND

DATA CONTROL COMMAND: Data Control Language is used for the control of data. That is a user can access any data based on the privileges given to him.

- DATA CONTROL LANGUAGE is known as DCL.
- DCL Statement is used for securing the database.
- DCL Statement control access to database.
- As data is important part of whole database system we must take proper steps to check that no invalid user access the data and invalidate the information created by us. To kept such a kind of watch we must have to execute certain DCL statement.
- Two main DCL statements are Grant and Revoke.

User privileges may be defined as the permission by which users can perform certain tasks or run some commands on the Oracle database. A user privilege may be to run a SQL statement or access object of another user. Roles may be defined as the group of privileges or other roles that are granted to or evoked from users by administrators. An administrator may grant or evoke one or more Roles from a user.

CREATE USER

Use the CREATE USER statement to create and configure a database **user**, or an account through which you can log in to the database and establish the means by which Oracle permits access by the user.

You must have CREATE USER system privilege. When you create a user with the CREATE USER statement, the user's privilege domain is empty. To log on to Oracle, a user must have CREATE SESSION system privilege. Therefore, after creating a user, you should grant the user at least the CREATE SESSION privilege.

```
CREATE USER user IDENTIFIED
{ BY password | EXTERNALLY | GLOBALLY AS 'external_name' }
[{ DEFAULT TABLESPACE tablespace
 | TEMPORARY TABLESPACE tablespace
 | QUOTA { integer [ K | M ] | UNLIMITED } ON tablespace
   [QUOTA { integer [ K | M ] | UNLIMITED } ON tablespace]...
 | PROFILE profile
 | PASSWORD EXPIRE
 | ACCOUNT { LOCK | UNLOCK }
}
 [ DEFAULT TABLESPACE tablespace
 | TEMPORARY TABLESPACE tablespace
 | QUOTA { integer [ K | M ] | UNLIMITED } ON tablespace
   [QUOTA { integer [ K | M ] | UNLIMITED } ON tablespace]...
 | PROFILE profile
 | PASSWORD EXPIRE
 | ACCOUNT { LOCK | UNLOCK }
 ]... ] ;
```

user

Specify the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Schema Object Naming Rules". Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

IDENTIFIED Clause

The IDENTIFIED clause lets you indicate how Oracle authenticates the user.

BY password

The BY *password* clause lets you creates a **local user** and indicates that the user must specify *password* to log on. Passwords can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

Passwords must follow the rules described in the section "Schema Object Naming Rules", unless you are using Oracle's password complexity verification routine. That routine requires a more complex combination of characters than the normal naming rules permit. You implement this routine with the UTLPWDMG.

EXTERNALLY Clause

Specify EXTERNALLY to create an **external user.** Such a user must be authenticated by an external service (such as an operating system or a third-party service). In this case, Oracle to relies on the login authentication of the operating system to ensure that a specific operating system user has access to a specific database user.

GLOBALLY Clause

The GLOBALLY clause lets you create a **global user**. Such a user must be authenticated by the enterprise directory service. The '*external_name*' string can take one of two forms:

- The X.509 name at the enterprise directory service that identifies this user. It should be of the form 'CN=*username,other_attributes*', where *other attributes* is the rest of the user's distinguished name (DN) in the directory.
- A null string (' ') indicating that the enterprise directory service will map authenticated global users to the appropriate database schema with the appropriate roles.

DEFAULT TABLESPACE Clause

Specify the default tablespace for objects that the user creates. If you omit this clause, objects default to the SYSTEM tablespace.

Restriction on Default Temporary Tablespaces

You cannot specify a locally managed tablespace (including an undo tablespace) or a dictionary-managed temporary tablespace as a user's default tablespace.

TEMPORARY TABLESPACE Clause

Specify the tablespace for the user's temporary segments. If you omit this clause, temporary segments default to the SYSTEM tablespace.

Restrictions on a User's Temporary Tablespace

- The tablespace must be a temporary tablespace and must have a standard block size.
- The tablespace cannot be an undo tablespace or a tablespace with automatic segment-space management.

QUOTA Clause

Use the QUOTA clause to allow the user to allocate up to *integer* bytes of space in the tablespace. Use K or M to specify the quota in kilobytes or megabytes. This quota is the maximum space in the tablespace the user can allocate.

A CREATE USER statement can have multiple QUOTA clauses for multiple tablespaces.

UNLIMITED lets the user allocate space in the tablespace without bound.

PROFILE Clause

Specify the profile you want to assign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user.

QUOTA Clause

Use the QUOTA clause to allow the user to allocate up to *integer* bytes of space in the tablespace. Use K or M to specify the quota in kilobytes or megabytes. This quota is the maximum space in the tablespace the user can allocate.

A CREATE USER statement can have multiple QUOTA clauses for multiple table spaces.

UNLIMITED lets the user allocate space in the table space without bound.

PROFILE Clause

Specify the profile you want to assign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user.

Examples

Creating a Database User: Example

If you create a new user with PASSWORD EXPIRE, the user's password must be changed before attempting to log in to the database. You can create the user sidney by issuing the following statement:

```
CREATE USER JJKCC
 IDENTIFIED BY out_standing1
 DEFAULT TABLESPACE example
 QUOTA 10M ON example
 TEMPORARY TABLESPACE temp
 QUOTA 5M ON system
  PROFILE app_user
  PASSWORD EXPIRE;
```

CREATE ROLE

Use the CREATE ROLE statement to create a role, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the GRANT statement.

If you create a role that is NOT IDENTIFIED or is IDENTIFIED EXTERNALLY or BY password, then Oracle Database grants you the role with ADMINOPTION. However, if you create a role IDENTIFIED GLOBALLY, then the database does not grant you the role.

Syntax:

```
CREATE ROLE role
  [NOT IDENTIFIED| IDENTIFIED { BY password | USING [ schema. ] package|
EXTERNALLY| GLOBALLY }] ;
```

role

Specify the name of the role to be created. Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters. The maximum number of user-defined roles that can be enabled for a single user at one time is 148.

Some roles are defined by SQL scripts provided on your distribution media.

NOT IDENTIFIED Clause

Specify NOT IDENTIFIED to indicate that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED Clause

Use the IDENTIFIED clause to indicate that a user must be authorized by the specified method before the role is enabled with the SET ROLE statement.

BY password The BY password clause lets you create a local role and indicates that the user must specify the password to the database when enabling the role. The password can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

USING package The USING package clause lets you create an application role, which is a role that can be enabled only by applications using an authorized package. If you do not specify schema, then the database assumes the package is in your own schema.

EXTERNALLY Specify EXTERNALLY to create an external role. An external user must be authorized by an external service, such as an operating system or third-party service, before enabling the role.

Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.

GLOBALLY Specify GLOBALLY to create a global role. A global user must be authorized to use the role by the enterprise directory service before the role is enabled at login.

If you omit both the NOT IDENTIFIED clause and the IDENTIFIED clause, then the role defaults to NOT IDENTIFIED.

Creating a Role: Example The following statement creates the role dw_manager:

CREATE ROLE dw_manager;

Users who are subsequently granted the dw_manager role will inherit all of the privileges that have been granted to this role.

You can add a layer of security to roles by specifying a password, as in the following example:

CREATE ROLE dw_manager   IDENTIFIED BY warehouse;

Users who are subsequently granted the dw_manager role must specify the password warehouse to enable the role with the SET ROLE statement.

The following statement creates global role warehouse_user:

CREATE ROLE warehouse_user IDENTIFIED GLOBALLY;

The following statement creates the same role as an external role:

CREATE ROLE warehouse_user IDENTIFIED EXTERNALLY;

GRANT Statement

Grant privilege (Rights which are to be allocated) is used when we want our database to share with other users, with certain type of right granted to him. Consider that if we want our end user to have only access privilege to our database, we can grant it by executing command. Grant privilege is assigned not only on table object, but also views, synonyms, indexes, sequences, etc.

Syntax:

GRANT privileges ON object TO user;

privileges

The privileges to assign.

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
INDEX	Ability to create an index on the table with the create index statement.
ALL	All privileges on table.

object

The name of the database object that you are granting privileges for. In the case of granting privileges on a table, this would be the table name.

user

The name of the user that will be granted these privileges.

SQL> grant select on emp to endusers;

here emp is the table_name whose access right is being allocated to the user who logged on as endusers.

SQL>grant insert,select,delete on emp to operators;

here user who logged on as operators are granted access, insertion and deletion right on the database.

SQL>grant insert (empno, ename, job) on emp to endusers;

In some case, we require to hide the information to particular users, this can be achieved by grant as in the above command we want to hide the detail of employee salary to end users, so by executing above command we can hide the information regarding empsalary to the end users

REVOKE Statement

Revoke privilege (Rights which are to be de-allocated) is used when we want our database to stop sharing the information with other users, with certain type of right revoked to him. Consider that if we want our operators to have only access privilege to our database, we can revoke it by executing command. Revoke privilege is assigned not only on table object, but also views, synonyms, indexes, sequences ,etc.

Syntax

REVOKE privileges ON object FROM user;

privileges

The privileges to assign.

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
INDEX	Ability to create an index on the table with the create index statement.
ALL	All privileges on table.

object

The name of the database object that you are granting privileges for. In the case of granting privileges on a table, this would be the table name.

user

The name of the user that will be granted these privileges.

example

SQL> revoke insert, delete on emp from operators;

What is transaction?

A transaction is logical piece of work consisting of one or more SQL statements. A transaction is started whenever data is read or written and they are ended by a COMMIT or ROLLBACK. DDL statements always perform a commit first this is called an implicit commit this is because the user did not issue the commit.

Oracle uses transaction locking and multi-version concurrency control using undo records to ensure serializability in transactions, this stops any user conflicts while ensuring database consistency.

Transaction Properties

Database transactions should exhibit attributes described by the ACID properties:

1. **Atomicity** :- A transaction either happens completely, or none of it happens
2. **Consistency** :- A transaction takes the database from consistent state to the next
3. **Isolation** : - The effects of a transaction may not be visible to other transaction until the transaction has committed.
4. **Durability** : - Once the transaction is committed, it is permanent all changes are written to the redo log first then the data files.

Starting and Ending a Transaction

A transaction has a beginning and an end. A transaction starts when one of the following events take place –

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place

1. A COMMIT or a ROLLBACK statement is issued.
2. A DDL statement, such as CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
3. A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
4. User disconnects from the database.
5. User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
6. SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
7. A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

```
COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

```
ROLLBACK;
```

Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as −

SET AUTOCOMMIT ON;

You can turn-off the auto commit mode using the following command −

SET AUTOCOMMIT OFF;

TRANSACATION CONTROL COMMAND :

SET TRANSACTION STATEMENT

This Oracle tutorial explains how to use the Oracle SET TRANSACTION statement with syntax and examples. the SET TRANSACTION statement is used to set a transaction as read-only, set a transaction as read/write, set a transaction's isolation level, assign a name to a transaction, or assign a rollback segment to a transaction.

Syntax

```
SET TRANSACTION [ READ ONLY | READ WRITE ]
          [ ISOLATION LEVEL [ SERIALIZE | READ COMMITED ]
          [ USE ROLLBACK SEGMENT 'segment_name' ]
          [ NAME 'transaction_name' ];
```

Parameters or Arguments

READ ONLY

Optional. If specified, it sets the transaction as a read-only transaction.

READ WRITE

Optional. If specified, it sets the transaction as a read/write transaction.

ISOLATION LEVEL

Optional. If specified, it has two options:

ISOLATION LEVEL SERIALIZE

If a transaction attempts to update a resource that has been updated by another transaction and uncommitted, the transaction will fail.

ISOLATION LEVEL READ COMMITTED

If a transaction requires row locks held by another transaction, the transaction will wait until the row locks are released.

USE ROLLBACK SEGMENT

Optional. If specified, it assigns the transaction to a rollback segment identified by 'segment_name' which is the segment name enclosed in quotes.

NAME

Assigns a name to the transaction identified by 'transaction_name' which is enclosed in quotes.

EXAMPLE

Let's look at an example that shows how to use the SET TRANSACTION statement in Oracle.

READ ONLY

SET TRANSACTION READ ONLY NAME 'RO_example';

READ WRITE

SET TRANSACTION READ WRITE NAME 'RW_example';

COMMIT STATEMENT

This Oracle tutorial explains how to use the Oracle COMMIT statement with syntax and examples. In Oracle, the COMMIT statement commits all changes for the current transaction. Once a commit is issued, other users will be able to see your changes.

SYNTAX

COMMIT [WORK] [COMMENT clause] [WRITE clause] [FORCE clause];

Parameters or Arguments

WORK

Optional. It was added by Oracle to be SQL-compliant. Issuing the COMMIT with or without the WORK parameter will result in the same outcome.

COMMENT clause

Optional. It is used to specify a comment to be associated with the current transaction. The comment that can be up to 255 bytes of text enclosed in single quotes. It is stored in the system view called DBA_2PC_PENDING along with the transaction ID if there is a problem.

WRITE clause

Optional. It is used to specify the priority that the redo information for the committed transaction is to be written to the redo log. With this clause, you have two parameters to specify:

WAIT or NOWAIT (WAIT is the default if omitted)

WAIT - means that the commit returns to the client only after the redo information is persistent in the redo log.

NOWAIT - means that the commit returns to the client right away regardless of the status of the redo log.

IMMEDIATE or BATCH (IMMEDIATE is the default if omitted)

IMMEDIATE

forces a disk I/O causing the log writer to write the redo information to the redo log.

BATCH

forces a "group commit" and buffers the redo log to be written with other transactions.

FORCE clause

Optional. It is used to force the commit of a transaction that may be corrupt or in doubt. With this clause, you can specify the FORCE in 3 ways:

- FORCE 'string', [integer] or FORCE CORRUPT_XID 'string' or FORCE CORRUPT_XID_ALL
- FORCE 'string', [integer] - allows you to commit a corrupt or in doubt transaction in a distributed database system by specifying the transaction ID in single quotes as string. You can find the transaction ID in the system view called DBA_2PC_PENDING. You can specify integer to assign the transaction a system change number if you do not wish to commit the transaction using the current system change number.
- FORCE CORRUPT_XID 'string' - allows you to commit a corrupt or in doubt transaction by specifying the transaction ID in single quotes as string. You can find the transaction ID in the system view called V$CORRUPT_XID_LIST.
- FORCE CORRUPT_XID_ALL - allows you to commit all corrupted transactions.

EXAMPLE

COMMIT WORK WRITE WAIT IMMEDIATE;

In this example, the WORK keyword is implied and the omission of the WRITE clause would default to WRITE WAIT IMMEDIATE so the first 2 COMMIT statements are equivalent. Comment Let's look at an example of a COMMIT that shows how to use the COMMENT clause: For example, you can write the COMMIT with a comment in two ways:

COMMIT COMMENT 'This is the comment for the transaction';
OR
COMMIT WORK COMMENT 'This is the comment for the transaction';

Since the WORK keyword is always implied, both of these COMMIT examples are equivalent. The COMMIT would store the comment enclosed in quotes along with the transaction ID in the DBA_2PC_PENDING system view, if the transaction was in error or in doubt.

Force

Finally, look at an example of a COMMIT that shows how to use the FORCE clause. For example, you can write the COMMIT of an in-doubt transaction in two ways:

COMMIT FORCE '22.14.67';
OR
COMMIT WORK FORCE '22.14.67';

Since the WORK keyword is always implied, both of these COMMIT examples would force the commit of the corrupted or in doubt transaction identified by the transaction ID '22.14.67'.

ROLLBACK STATEMENT

This Oracle tutorial explains how to use the Oracle ROLLBACK statement with syntax and examples. In Oracle, the ROLLBACK statement is used to undo the work performed by the current transaction or a transaction that is in doubt.

SYNTAX

The syntax for the ROLLBACK statement is:

ROLLBACK [WORK] [TO [SAVEPOINT] savepoint_name | FORCE 'string'];

Parameters or Arguments

WORK

Optional. It was added by Oracle to be SQL-compliant. Issuing the ROLLBACK with or without the WORK parameter will result in the same outcome.

TO SAVEPOINT savepoint_name

Optional. The ROLLBACK statement undoes all changes for the current session up to the savepoint specified bysavepoint_name. If this clause is omitted, then all changes are undone.

FORCE 'string'

Optional. It is used to force the rollback of a transaction that may be corrupt or in doubt. With this clause, you specify the transaction ID in single quotes as string. You can find the transaction ID in the system view called DBA_2PC_PENDING.

NOTE

You must have DBA privileges to access the system views - DBA_2PC_PENDING and V$CORRUPT_XID_LIST. You can not rollback a transaction that is in doubt to a savepoint.

EXAMPLE

ROLLBACK;

This ROLLBACK example would perform the same as the following:

ROLLBACK WORK;

In this example, the WORK keyword is implied so the first 2 ROLLBACK statements are equivalent. These examples would rollback the current transaction.

Savepoint

Let's look at an example of a ROLLBACK that shows how to use the rollback to a specific savepoint. For example, you can write the ROLLBACK to a savepoint in two ways:

ROLLBACK TO SAVEPOINT savepoint1;

OR

ROLLBACK WORK TO SAVEPOINT savepoint1;

Since the WORK keyword is always implied, both of these ROLLBACK examples would rollback the current transaction to the savepoint called savepoint1. Force Finally, look at an example of a ROLLBACK that shows how to force the rollback of a transaction that is in doubt. For example, you can write the ROLLBACK of an in-doubt transaction in two ways:

ROLLBACK FORCE '22.14.67';

OR

ROLLBACK WORK FORCE '22.14.67';

Since the WORK keyword is always implied, both of these ROLLBACK examples would force the rollback of the corrupted or in doubt transaction identified by the transaction ID '22.14.67'.

SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

Syntax

SAVEPOINT SAVEPOINT_NAME;

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as follows:

ROLLBACK TO SAVEPOINT_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

Example:

Consider the CUSTOMERS table having the following records:

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

Now, here is the series of operations:

```
SQL> SAVEPOINT SP1;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.

SQL> SAVEPOINT SP2;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.

SQL> SAVEPOINT SP3;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
SQL> ROLLBACK TO SP2;

Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2:

```
SQL> SELECT * FROM CUSTOMERS;
| ID | NAME     | AGE | ADDRESS   | SALARY   |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |

6 rows selected.
```

The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for RELEASE SAVEPOINT is as follows:

RELEASE SAVEPOINT SAVEPOINT_NAME;

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

UNIT 3

OTHER ORACLE DATABASE OBJECT

VIEW

The result of a query is always a table, or more precisely, a derived table. Compared with "real" tables in the database, the result of a query is volatile, but nevertheless, the result is a table.

The only thing that is missing for the query result is a name. Essentially, a view is nothing more than a query with a given name. A more precise definition is as follows:

DEFINITION: A view is a virtual table with the result of a stored query as its "contents," which are derived each time you access the view.

A view can be considered as a stored query or a virtual table. It is a customized represented of data from one or more table. **Only the query (Definition of View)** is stored in the Oracle Data Dictionary, the actual data is not copied anywhere. So creating view does not take any storage space, other than the space in the data dictionary.

A view can also hide query complexity. There may be multiple table joins in the query but the user sees only the view. The view can have different column names than the base table.

View can be created to limit the data accessible to other users. All operation performed on the view affect the underlying base table. When a query is issued against a view most of the time oracle merges the query with the query the defines the view then executes the resulting query as if the query was issued directly against the base tables. This helps to use the indexes if these are any defined on the tables.

The maximum number of column that can be defined In a view is 1000, just like a table. When new records are inserted/ deleted / updated in the base table, the changes are automatically reflected in the view and refresh method is not required as view does not have its own table. When the view is referred using query, then base tables is referred and data is fetched from the base table(s) depending upon the view definition. View once defined can be used in joins and sub query also

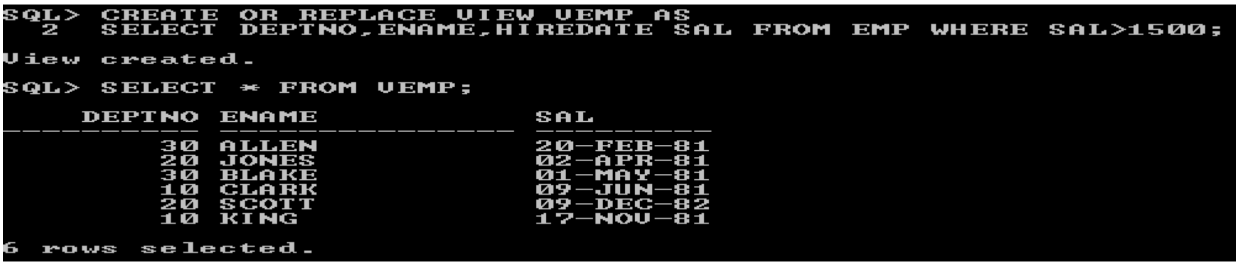
Types of view

1. Read only View
2. Updateable view

Read only view is used to just display the data. User can't update the whereas updateable view allows user to modify the data

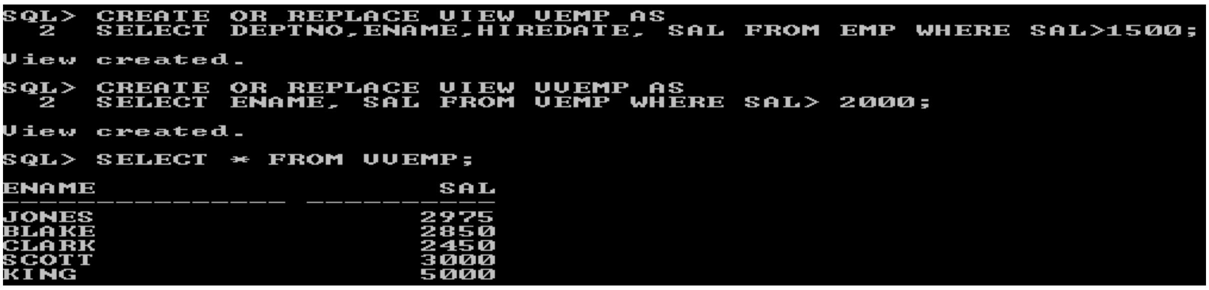
Syntax

CREATE [OR REPLACE] [FORCE] VIEW < VIEW NAME> AS
SELECT …. . . . . FROM . . . . . WHERE . . . . . . GROUP BY . . . . . . HAVING. . . . .



```
SQL> CREATE OR REPLACE VIEW VEMP AS
  2  SELECT DEPTNO,ENAME,HIREDATE SAL FROM EMP WHERE SAL>1500;

View created.

SQL> SELECT * FROM VEMP;

    DEPTNO ENAME                SAL
---------- -------------------- ---------
        30 ALLEN                20-FEB-81
        20 JONES                02-APR-81
        30 BLAKE                01-MAY-81
        10 CLARK                09-JUN-81
        20 SCOTT                09-DEC-82
        10 KING                 17-NOV-81

6 rows selected.
```

The query cannot have a FOR UPDATE clause. ORDER BY clause was not permitted in version prior or oracle 8i, in oracle 8i, the query can have an ORDER BY clause. REPLACE option is used to overwrite existing view. We may also create a view on the top another view.



```
SQL> CREATE OR REPLACE VIEW VEMP AS
  2  SELECT DEPTNO,ENAME,HIREDATE, SAL FROM EMP WHERE SAL>1500;

View created.

SQL> CREATE OR REPLACE VIEW VVEMP AS
  2  SELECT ENAME, SAL FROM VEMP WHERE SAL> 2000;

View created.

SQL> SELECT * FROM VVEMP;

ENAME                  SAL
---------------- ----------
JONES                 2975
BLAKE                 2850
CLARK                 2450
SCOTT                 3000
KING                  5000
```

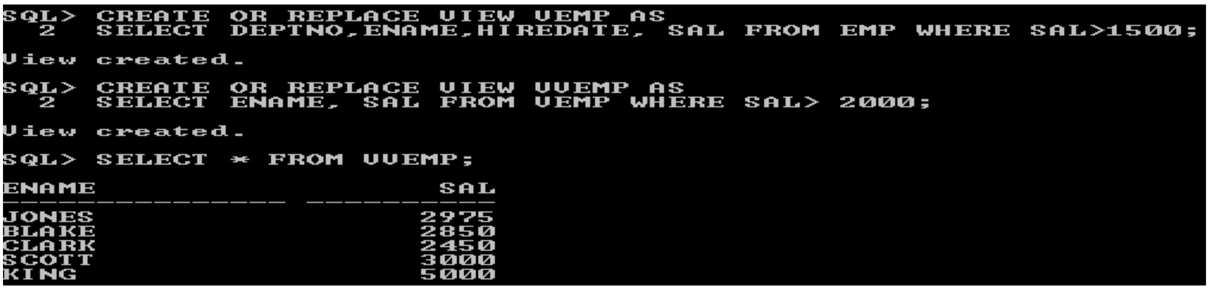
You can update, insert and delete rows through a view with restrictions. If the view is joining more than one table at a time. For updating or inserting into a view, all the columns that are parts of a constraint should be in the view definition.

UPDATEABLE VIEW

Views can also be used for data manipulation. Views on which data manipulation can be done are called updateable views. When an updateable view name is given in an Insert, Update, or Delete SQL statement, modification to data in the view will be immediately passed to the underlying table.

View defined from single table.

- For INSERT, Primary key and all NOT NULL columns must be including in the view.
- User can UPDATE/ DELETE records with the help of view even if the PRIMARY KEY column are excluded from the view definition
- The column that can be update in a view can be queried from the data dictionary USER_UPDATABLE_COLUMNS



```
SQL> CREATE OR REPLACE VIEW VEMP AS
  2  SELECT DEPTNO,ENAME,HIREDATE, SAL FROM EMP WHERE SAL>1500;

View created.

SQL> CREATE OR REPLACE VIEW VVEMP AS
  2  SELECT ENAME, SAL FROM VEMP WHERE SAL> 2000;

View created.

SQL> SELECT * FROM VVEMP;

ENAME                  SAL
---------------- ----------
JONES                 2975
BLAKE                 2850
CLARK                 2450
SCOTT                 3000
KING                  5000
```

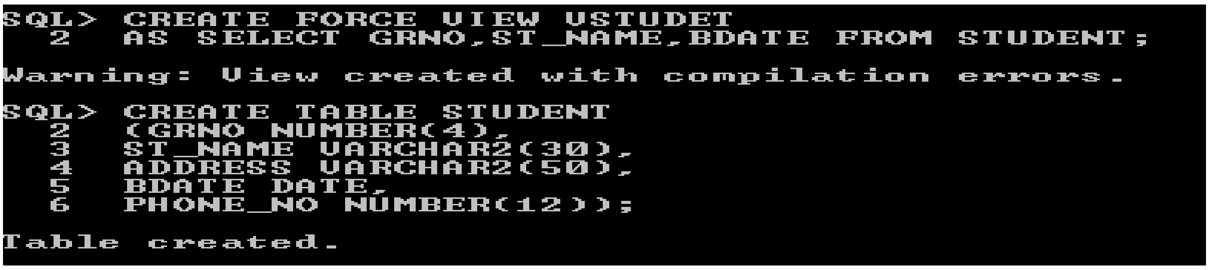
FORCE VIEW

A view can be created even if the defining query of the view cannot be executed, as long as the CREATE VIEW command has no syntax errors. We call such a view a **view with errors**. For example, if a view refers to a non-existent table or an invalid column of an existing table, or if the owner of the view does not have the required privileges, then the view can still be created and entered into the data dictionary.

You can only create a view with errors by using the FORCE option of the CREATE VIEW command:

CREATE FORCE VIEW AS ...;

When a view is created with errors, Oracle returns a message and leaves the status of the view as INVALID. If conditions later change so that the query of an invalid view can be executed, then the view can be recompiled and become valid. Oracle dynamically compiles the invalid view if you attempt to use it.

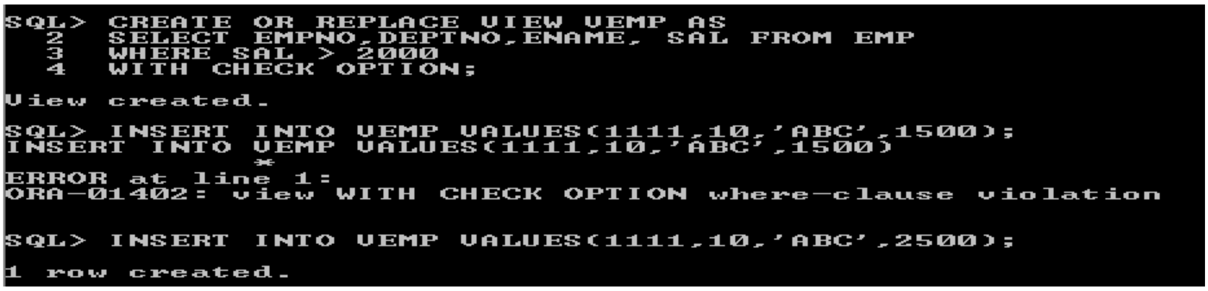


```
SQL> CREATE FORCE VIEW VSTUDET
  2  AS SELECT GRNO,ST_NAME,BDATE FROM STUDENT;

Warning: View created with compilation errors.

SQL> CREATE TABLE STUDENT
  2  (GRNO NUMBER(4),
  3  ST_NAME VARCHAR2(30),
  4  ADDRESS VARCHAR2(50),
  5  BDATE DATE,
  6  PHONE_NO NUMBER(12));

Table created.
```

You can create views with an optional WITH clause. WITH READ ONLY specifies that the view cannot be update or delete and that new rows cannot be inserted WITH CHECK OPTION specifies that inserts and updates done through the view should satisfy the WHERE clause of the view. WITH CHECK OPTION creates a constraint with constraint type “V”. if you do not provide a name the constraint is created with a SYS_name.

Example

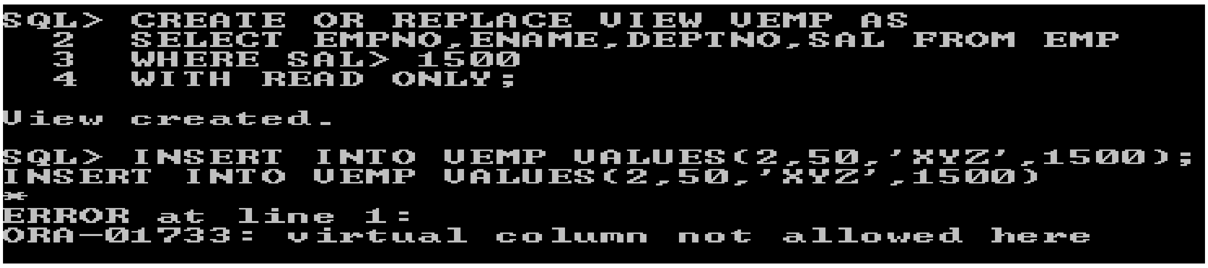
Following example shows that record cannot be inserted into table through view unless if satisfies VIEW CONDITION



```
SQL> CREATE OR REPLACE VIEW VEMP AS
  2  SELECT EMPNO,DEPTNO,ENAME, SAL FROM EMP
  3  WHERE SAL > 2000
  4  WITH CHECK OPTION;

View created.

SQL> INSERT INTO VEMP VALUES(1111,10,'ABC',1500);
INSERT INTO VEMP VALUES(1111,10,'ABC',1500)
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation


SQL> INSERT INTO VEMP VALUES(1111,10,'ABC',2500);

1 row created.
```

Example

Following example shows that view created using **WITH READ ONLY** option cannot be used to insert new row.



```
SQL> CREATE OR REPLACE VIEW VEMP AS
  2  SELECT EMPNO,ENAME,DEPTNO,SAL FROM EMP
  3  WHERE SAL> 1500
  4  WITH READ ONLY;

View created.

SQL> INSERT INTO VEMP VALUES(2,50,'XYZ',1500);
INSERT INTO VEMP VALUES(2,50,'XYZ',1500)
*
ERROR at line 1:
ORA-01733: virtual column not allowed here
```

DROPPING VIEW

The view definition is dropped from the directory and the privileges and grants on the view are also dropped. Other view and stored programs that refer to the dropped view become invalid.

SQL> DROP VIEW <VIEW NAME>;

SQL> DROP VIEW VEMP;

SEQUENCE:

In some situation when you need that oracle will automatically generate a sequential number and it will store that generated number into a particular field, then you have to use SEQUENCEs. The maximum limit of generated sequence is the number of 38 digits.

SYNTAX:

```
SQL>  CREATE SEQUENCE <sequence name>
          INCREMENT BY <value>
           START WITH <value>
          MAXVALUE <value>/NOMAXVALUE
          MINVALUE <value>/NOMINVALUE
           CYCLE / NOCYCLE
          CACHE <value> / NOCACHE
          ORDER / NOORDER;
```

EXPLANATION:

- **INCREMENTE BY**- it specifies the interval between two values of the sequence, it can be positive or negative, but cannot be zero. If you not use this clause then default value is 1.
- **START WITH** – specifies the first number to be started with by the sequence
- **MAXVALUE** - specifies the maximum value of the sequence.
- **NOMAXVALUE** – if you don't want to specify maximum value then you can define. this clause. It will give you the maximum value of 1037.
- **MINVALUE** – specifies the minimum value of sequence.
- **NOMINVALUE** – if you don't want to specify minimum value then you can define. this clause. It will give you the minimum value of 10-37.
- **CYCLE** – specifies that the sequence continue to generate numbers after reaching to its maximum value.
- **NOCYCLE** –the sequence will not generate more numbers after reaching its maximum value.
- **CACHE** – it will specify the value of how many numbers oracle will prepare for the next value in the memory.
- **NOCACHE** – does not prepare value for cache.
- **ORDER** – this will guarantee you that the sequence numbers are generated in the order.
- **NOORDER** - this will not guarantee you that sequence numbers are generated in order.

EXAMPLE:

```
SQL> CREATE SEQUENCE seq1 INCREMENT BY 1 START WITH 1 MAXVALUE 999
     NOCYCLE NOCACHE NOORDER;
```

HOW TO USE SEQUENCE:

When you need to use a particular sequence you can use two constant i.e. NEXTVAL for getting next value of the sequence and CURRVAL for getting current value of the sequqnce.

EXAMPLE:

```
SQL>INSERT INTO stud(grno, name, std)VALUES(seq1.NEXTVAL, 'ABC',12);
```

Above command will generate next value of the sequence, so when you add records like this it will automatically gives you new number each and every time.

The following command will display the current value of the sequence.

```
SQL> SELECT seq1.CURRVAL FROM DUAL;
```

DELETE THE SEQUENCE

```
SQL>DROP SEQUENCE <sequence name>;
SQL>DROP SEQUENCE seq1;
```

SYNONYMS:

Synonym is a name assigned to a table, view or sequence that may thereafter be used to refer to it.

If you have access to another users table, you may create a SYNONYM for it and refer to it by the synonym alone, without entering the user's name as a qualifier.

SYNTAX:

SQL> **CREATE** [**OR REPLACE**] [**PUBLIC**] **SYNONYM** [schema.]synonym **FOR** [schema.]object [@ dblink] ;

PUBLIC

Specify PUBLIC to create a public synonym. Public synonyms are accessible to all users. However each user must have appropriate privileges on the underlying object in order to use the synonym.

When resolving references to an object, Oracle Database uses a public synonym only if the object is not prefaced by a schema and is not followed by a database link.

If you omit this clause, then the synonym is private and is accessible only within its schema. A private synonym name must be unique in its schema.

Notes on Public Synonyms The following notes apply to public synonyms:

If you create a public synonym and it subsequently has dependent tables or dependent valid user-defined object types, then you cannot create another database object of the same name as the synonym in the same schema as the dependent objects.
Take care not to create a public synonym with the same name as an existing schema. If you do so, then all PL/SQL units that use that name will be invalidated.

schema

Specify the schema to contain the synonym. If you omit schema, then Oracle Database creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified PUBLIC.

synonym

Specify the name of the synonym to be created.

FOR Clause

Specify the object for which the synonym is created. The schema object for which you are creating the synonym can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- User-defined object type
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restriction on the FOR Clause The schema object cannot be contained in a package.

schema Specify the schema in which the object resides. If you do not qualify object with schema, then the database assumes that the schema object is in your own schema.
If you are creating a synonym for a procedure or function on a remote database, then you must specify schema in this CREATE statement. Alternatively, you can create a local

public synonym on the database where the object resides. However, the database link must then be included in all subsequent calls to the procedure or function.

dblink

You can specify a complete or partial database link to create a synonym for a schema object on a remote database where the object is located. If you specify dblink and omit schema, then the synonym refers to an object in the schema specified by the database link. Oracle recommends that you specify the schema containing the object in the remote database. If you omit dblink, then Oracle Database assumes the object is located on the local database. Restriction on Database Links You cannot specify dblink for a Java class synonym.

Examples To define the synonym offices for the table locations in the schema hr, issue the following statement:

SQL> CREATE SYNONYM offices FOR hr.locations;

To create a PUBLIC synonym for the employees table in the schema hr on the remote database, you could issue the following statement:

SQL>CREATE PUBLIC SYNONYM emp_table FOR hr.employees@remote.us.oracle.com;

A synonym may have the same name as the underlying object, provided the underlying object is contained in another schema.

Database Links

Use the **CREATE DATABASE LINK** statement to create a database link. A **database link** is a schema object in one database that enables you to access objects on another database. The other database need not be an Oracle Database system. However, to access non-Oracle systems you must use Oracle Heterogeneous Services.

After you have created a database link, you can use it in SQL statements to refer to tables and views on the other database by appending ***@dblink*** to the table or view name. You can query a table or view on the other database with the **SELECT** statement. You can also access remote tables and views using any **INSERT**, **UPDATE**, **DELETE**, or **LOCK TABLE** statement.

To create a private database link, you must have the **CREATE DATABASE LINK** system privilege. To create a public database link, you must have the **CREATE PUBLIC DATABASE LINK** system privilege. Also, you must have the **CREATE SESSION** system privilege on the remote Oracle database. Oracle Net must be installed on both the local and remote Oracle databases.

Syntax

CREATE [**SHARED**] [**PUBLIC**] **DATABASE LINK** dblink
[**CONNECT TO** { **CURRENT_USER** | user **IDENTIFIED BY** password [dblink_authentication] }
| dblink_authentication]...
[**USING** connect_string] ;

Parameters

PUBLIC

Specify **PUBLIC** to create a public database link visible to all users. If you omit this clause, then the database link is private and is available only to you.

The data accessible on the remote database depends on the identity the database link uses when connecting to the remote database:

- If you specify **CONNECT TO** ***user*** **IDENTIFIED BY** ***password***, then the database link connects with the specified user and password.
- If you specify **CONNECT TO CURRENT_USER**, then the database link connects with the user in effect based on the scope in which the link is used.
- If you omit both of those clauses, then the database link connects to the remote database as the locally connected user.

SHARED

- Specify **SHARED** to create a database link that can be shared by multiple sessions using a single network connection from the source database to the target database. In a shared server configuration, shared database links can keep the number of connections into the remote database from becoming too large. Shared links are typically also public database links. However, a shared private database link can be useful when many clients access the same local schema, and therefore use the same private database link.
- In a shared database link, multiple sessions in the source database share the same connection to the target database. Once a session is established on the target database, that session is disassociated from the connection, to make the connection available to another session on the source database. To prevent an unauthorized session from attempting to connect through the database link, when you specify **SHARED** you must also specify the***dblink_authentication*** clause for the users authorized to use the database link.

dblink

- Specify the complete or partial name of the database link. If you specify only the database name, then Oracle Database implicitly appends the database domain of the local database.
- Use only ASCII characters for ***dblink***. Multibyte characters are not supported. The database link name is case insensitive and is stored in uppercase ASCII characters. If you specify the database name as a quoted identifier, then the quotation marks are silently ignored.
- If the value of the **GLOBAL_NAMES** initialization parameter is **TRUE**, then the database link must have the same name as the database to which it connects. If the value of **GLOBAL_NAMES** is **FALSE**, and if you have changed the global name of the database, then you can specify the global name.
- The maximum number of database links that can be open in one session or one instance of an Oracle RAC configuration depends on the value of the**OPEN_LINKS** and **OPEN_LINKS_PER_INSTANCE** initialization parameters.
- **Restriction on Creating Database Links** You cannot create a database link in another user's schema, and you cannot qualify ***dblink*** with the name of a schema. Periods are permitted in names of database links, so Oracle Database interprets the entire name, such as **ralph.linktosales**, as the name of a database link in your schema rather than as a database link named **linktosales** in the schema **ralph**.)

CONNECT TO Clause

- The **CONNECT TO** clause lets you specify the user and credentials, if any, to be used to connect to the remote database.

CURRENT_USER Clause

- Specify **CURRENT_USER** to create a **current user database link**. The current user must be a global user with a valid account on the remote database.

- If the database link is used directly rather than from within a stored object, then the current user is the same as the connected user.
- When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, **CURRENT_USER** is the name of the user that owns the stored object, and not the name of the user that called the object. For example, if the database link appears inside procedure **scott.p** (created by**scott**), and user **jane** calls procedure **scott.p**, then the current user is **scott**.
- However, if the stored object is an invoker-rights function, procedure, or package, then the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure **scott.p** (an invoker-rights procedure created by **scott**), and user Jane calls procedure **scott.p**, then **CURRENT_USER** is **jane** and the procedure executes with Jane's privileges.

user* IDENTIFIED BY *password

- Specify the user name and password used to connect to the remote database using a **fixed user database link**. If you omit this clause, then the database link uses the user name and password of each user who is connected to the database. This is called a **connected user database link**.

dblink_authentication

- You can specify this clause only if you are creating a shared database link—that is, you have specified the **SHARED** clause. Specify the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user.

USING '*connect string*'

- Specify the service name of a remote database. If you specify only the database name, then Oracle Database implicitly appends the database domain to the connect string to create a complete service name. Therefore, if the database domain of the remote database is different from that of the current database, then you must specify the complete service name.

Examples

The examples that follow assume two databases, one with the database name local and the other with the database name remote. The examples use the Oracle Database domain. Your database domain will be different.

Defining a Public Database Link: Example The following statement defines a shared public database link named remote that refers to the database specified by the service name remote:

```
CREATE PUBLIC DATABASE LINK remote    USING 'remote';
```

This database link allows user hr on the local database to update a table on the remote database (assuming hr has appropriate privileges):

```
UPDATE employees@remote
  SET salary=salary*1.1
  WHERE last_name = 'Baer';
```

INDEX

WHAT ARE INDEXES?

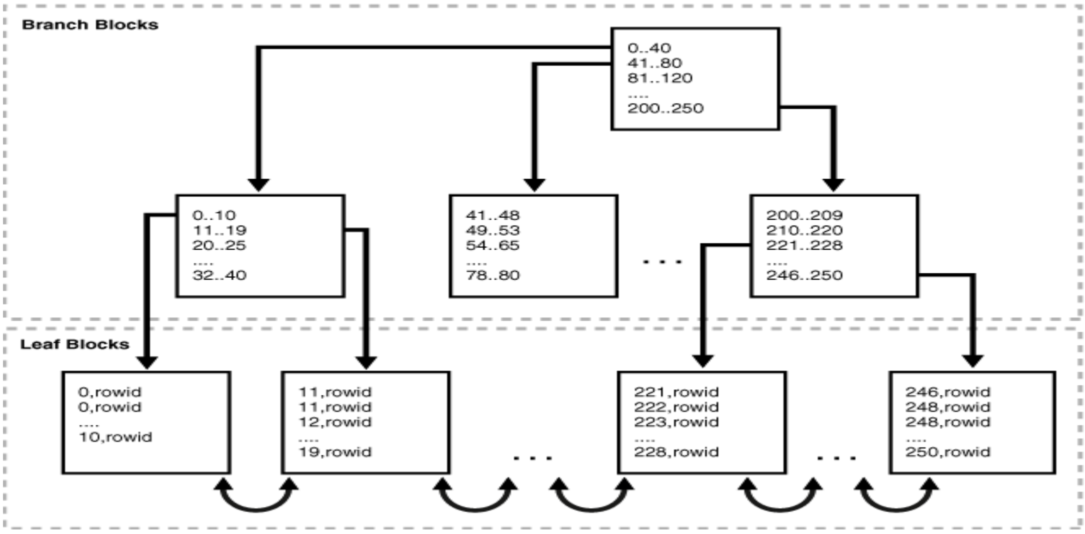
Data can be retrieved from a database using two methods. The first method, often called the Sequential Access Method, requires SQL to go through each record looking for a match. This search method is inefficient, but it is the only way for SQL to locate the correct record. Think back to the days when libraries had massive card catalog filing systems.

B-Tree Index

B-trees, short for balanced trees, are the most common type of database index. A B-tree index is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

Figure illustrates the structure of a B-tree index. The example shows an index on the department_id column, which is a foreign key column in the employees table.

Internal Structure of a B-tree Index



Branch Blocks and Leaf Blocks

A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks. In Figure, the root branch block has an entry 0-40, which points to the leftmost block in the next branch level. This branch block contains entries such as 0-10 and 11-19. Each of these entries points to a leaf block that contains key values that fall in the range.

A B-tree index is balanced because all leaf blocks automatically stay at the same depth. Thus, retrieval of any record from anywhere in the index takes approximately the same amount of time. The height of the index is the number of blocks required to go from the root block to a leaf block. The branch level is the height minus 1. In Figure, the index has a height of 3 and a branch level of 2.

Branch blocks store the minimum key prefix needed to make a branching decision between two keys. This technique enables the database to fit as much data as possible on each branch block. The branch blocks contain a pointer to the child block containing the key. The number of keys and pointers is limited by the block size.

The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. Each entry is sorted by (key, rowid). Within a leaf block, a key and rowid is linked to its left and right sibling entries. The leaf blocks themselves are also doubly linked. In Figure the leftmost leaf block (0-10) is linked to the second leaf block (11-19)

Indexes in columns with character data are based on the binary values of the characters in the database character set.

Bitmap Indexes

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows.

Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion. Situations that may call for a bitmap index include:

- The indexed columns have low **cardinality**, that is, the number of distinct values is small compared to the number of table rows.
- The indexed table is either read-only or not subject to significant modification by DML statements.

For a data warehouse example, the **sh.customers** table has a **cust_gender** column with only two possible values: **M** and **F**. Suppose that queries for the number of customers of a particular gender are common. In this case, the **customers.cust_gender** column would be a candidate for a bitmap index.

Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a B-tree index although it uses a different internal representation.

If the indexed column in a single row is updated, then the database **locks** the index key entry (for example, **M** or **F**) and not the individual bit mapped to the updated row. Because a key points to many rows, DML on indexed data typically locks all of these rows. For this reason, bitmap indexes are not appropriate for many **OLTP** applications.

Function-Based Indexes

You can create indexes on functions and expressions that involve one or more columns in the table being indexed. A **function-based index** computes the value of a function or expression involving one or more columns and stores it in the index. A function-based index can be either a B-tree or a bitmap index.

The function used for building the index can be an arithmetic expression or an expression that contains a SQL function, user-defined PL/SQL function, package function

Uses of Function-Based Indexes

Function-based indexes are efficient for evaluating statements that contain functions in their WHERE clauses. The database only uses the function-based index when the function is included in a query. When the database processes INSERT and UPDATE statements, however, it must still evaluate the function to process the statement. For example, suppose you create the following function-based index:

Example

```
CREATE INDEX emp_total_sal_idx
  ON employees (12 * salary * commission_pct, salary, commission_pct);
```

Application Domain Indexes

An **application domain index** is a customized index specific to an application. Oracle Database provides **extensible indexing** to do the following:

- Accommodate indexes on customized, complex data types such as documents, spatial data, images, and video clips
- Make use of specialized indexing techniques

You can encapsulate application-specific index management routines as an **indextype** schema object and define a domain index on table columns or attributes of an object type. Extensible indexing can efficiently process application-specific **operators**.

The application software, called the **cartridge**, controls the structure and content of a domain index. The database interacts with the application to build, maintain, and search the domain index. The index structure itself can be stored in the database as an index-organized table or externally as a file.

Dropping An Index:

```
SQL>DROP INDEX <index name>;
SQL>DROP INDEX idx1;
```

CLUSTER

A cluster is composed of one or more tables. The cluster includes a cluster index, which stores all the values for the corresponding cluster key. Each value in the cluster index points to a data block that contains only rows with the same value for the cluster key.

A **table cluster** is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single **data block** can contain rows from multiple tables. For example, a block can store rows from both the **employees** and **departments** tables rather than from only a single table.

The **cluster key** is the column or columns that the clustered tables have in common. For example, the **employees** and **departments** tables share the **department_id** column. You specify the cluster key when creating the table cluster and when creating every table added to the table cluster.

The **cluster key value** is the value of the cluster key columns for a particular set of rows. All data that contains the same cluster key value, such as **department_id=20**, is physically stored together. Each cluster key value is stored only once in the cluster and the cluster index, no matter how many rows of different tables contain the value.

For an analogy, suppose an HR manager has two book cases: one with boxes of employees folders and the other with boxes of departments folders. Users often ask for the folders for all employees in a particular department. To make retrieval easier, the manager rearranges all the boxes in a single book case. She divides the boxes by department ID. Thus, all folders for employees in department 20 and the folder for department 20 itselfs are in one box; the folders for employees in department 100 and the folder for department 100 are in a different box, and so on.

You can consider clustering tables when they are primarily queried (but not modified) and records from the tables are frequently queried together or joined. Because table clusters store related rows of different tables in the same data blocks, properly used table clusters offer the following benefits over non-clustered tables:

- Disk I/O is reduced for **joins** of clustered tables.
- Access time improves for joins of clustered tables.
- Less storage is required to store related table and index data because the cluster key value is not stored repeatedly for each row.

Typically, clustering tables is not appropriate in the following situations:

- The tables are frequently updated.
- The tables frequently require a **full table scan**.
- The tables require truncating.

Indexed Clusters

An **indexed cluster** is a table cluster that uses an index to locate data. The **cluster index** is a B-tree index on the cluster key. A cluster index must be created before any rows can be inserted into clustered tables.

Assume that you create the cluster employees_departments_cluster with the cluster key department_id, as shown in Example. Because the HASHKEYS clause is not specified, this cluster is an indexed cluster. Afterward, you create an index named idx_emp_dept_cluster on this cluster key.

SQL> CREATE CLUSTER employees_departments_cluster (department_id NUMBER(4)) SIZE 512;

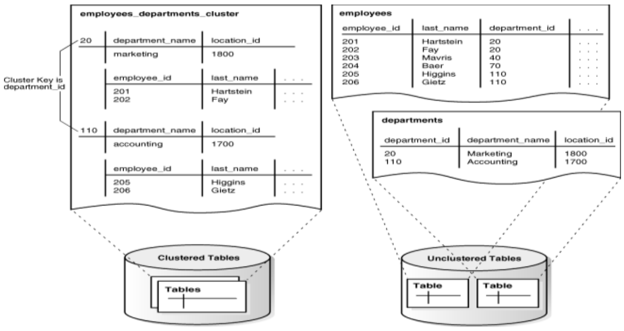
SQL> CREATE INDEX idx_emp_dept_cluster ON CLUSTER employees_departments_cluster;

You then create the employees and departments tables in the cluster, specifying the department_id column as the cluster key, as follows (the ellipses mark the place where the column specification goes):

```
CREATE TABLE employees ( ... )
     CLUSTER employees_departments_cluster (department_id);
CREATE TABLE departments ( ... )
     CLUSTER employees_departments_cluster (department_id);
```

Finally, you add rows to the employees and departments tables. The database physically stores all rows for each department from the employees and departments tables in the same data blocks. The database stores the rows in a heap and locates them with the index.

Figure shows the employees_departments_cluster table cluster, which contains employees and departments. The database stores rows for employees in department 20 together, department 110 together, and so on. If the tables are not clustered, then the database does not ensure that the related rows are stored together.



Clustered Table Data

The B-tree cluster index associates the cluster key value with the database block address (DBA) of the block containing the data. For example, the index entry for key 20

shows the address of the block that contains data for employees in department 20:
20, AADAAAA9d

The cluster index is separately managed, just like an index on a non-clustered table, and can exist in a separate table space from the table cluster.

SNAPSHOT

Oracle uses ***snapshots***, sometimes referred to as materialized views, to meet the requirements of delivering data to non-master sites in a replicated environment and caching "expensive" queries from a data warehouse

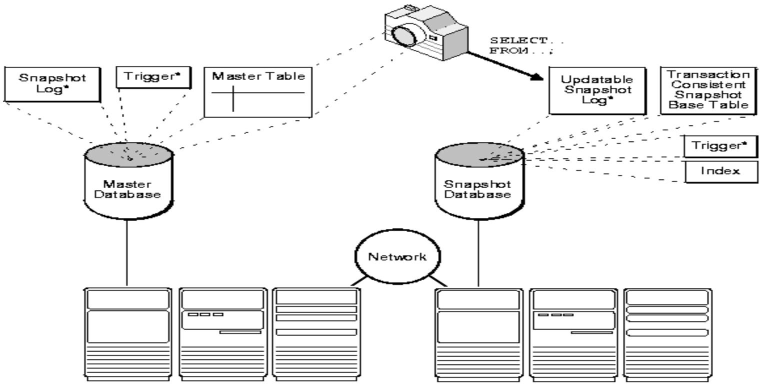
WHAT IS A SNAPSHOT?

A snapshot is a replica of a target master table from a single point-in-time. Whereas in multimaster replication tables are continuously being updated by other master sites, snapshots are updated by one or more master tables via individual batch updates, known as a *refresh*, from a single master site.

Snapshots also have the option of containing a WHERE clause so that snapshot sites can contain custom data sets, which can be very helpful for regional offices or sales forces that don't require the complete corporate data set.

Oracle offers a variety of snapshots to meet the needs of many different replication (and non-replication) situations; each of these snapshots will be discussed in detail in following sections. You might use a snapshot to achieve one or more of the following:

- Ease Network Loads
- Mass Deployment
- Data Subsetting
- Disconnected Computing



Syntax

```
CREATE SNAPSHOT [schema.]snapshot
   [ [PCTFREE  integer] [PCTUSED  integer]
     [INITRANS integer] [MAXTRANS integer]
     [TABLESPACE tablespace]
     [STORAGE storage_clause]
   [ USING INDEX [  PCTFREE integer | TABLESPACE tablespace
                | INITTRANS integer | MAXTRANS integer
                | STORAGE storage_clause ] ...
   | [CLUSTER cluster (column [, column]...)] ]
   [ REFRESH [FAST | COMPLETE | FORCE] [START WITH date] [NEXT date]]
   AS subquery
```

schema is the schema to contain the snapshot. If you omit schema, Oracle creates the snapshot in your schema.

Snapshot

is the name of the snapshot to be created. Oracle chooses names for the table, views, and index used to maintain the snapshot by prefixing the snapshot name. To limit these names to 30 bytes and allow them to contain the entire snapshot name, Oracle Corporation recommends that you limit your snapshot names to 23 bytes.

PCTFREE , PCTUSED , INITRANS, MAXTRANS

Establishes values for these parameters for the internal table Oracle uses to maintain the snapshot\'s data.

TABLESPACE

Specifies the table space in which the snapshot is to be created. If you omit this option, Oracle creates the snapshot in the default table space of the owner of the snapshot\'s schema.

STORAGE

Establishes storage characteristics for the table Oracle uses to maintain the snapshot\'s data.

USING INDEX

specifies the storage characteristics for the index on a simple snapshot.If the USING INDEX clause not specified, the index is create with the same tablespace and storage parameters as the snapshot.

CLUSTER

Creates the snapshot as part of the specified cluster. Since a clustered snapshot uses the cluster\'s space allocation, do not use the PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameters, the TABLESPACE option, or the STORAGE clause in conjunction with the CLUSTER option.

REFRESH

Specifies how and when Oracle automatically refreshes the snapshot:

FAST

Specifies a fast refresh, or a refresh using only the updated data stored in the snapshot log associated with the master table.

COMPLETE

Specifies a complete refresh, or a refresh that re- executes the snapshot\'s query.

FORCE

Specifies a fast refresh if one is possible or complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time.If you omit the FAST, COMPLETE, and FORCE options, Oracle **uses FORCE by default.**

START WITH

Specifies a date expression for the first automatic refresh time.

NEXT

Specifies a date expression for calculating the interval between automatic refreshes. Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression when you create the snapshot. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the snapshot only once.If you omit both the START WITH and NEXT values or if you omit the REFRESH clause entirely, Oracle does not automatically refresh the snapshot.

AS subquery

specifies the snapshot query. When you create the snapshot, Oracle executes this query and places the results in the snapshot. The select list can contain up to 253 expressions. A snapshot query is subject to the same restrictions as a view query.

UNIT 3

CONCURRENCY CONTROL USING LOCK

Data Concurrency and Consistency

In a single-user database, the user can modify data in the database without concern for other users modifying the same data at the same time. However, in a multiuser database, the statements within multiple simultaneous transactions can update the same data. Transactions executing at the same time need to produce meaningful and consistent results. Therefore, control of data concurrency and data consistency is vital in a multiuser database.

- Data concurrency means that many users can access data at the same time.
- Data consistency means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users.

To describe consistent transaction behavior when transactions execute at the same time, database researchers have defined a transaction isolation model called serializability. The serializable mode of transaction behavior tries to ensure that transactions execute in such a way that they appear to be executed one at a time, or serially, rather than concurrently.

While this degree of isolation between transactions is generally desirable, running many applications in this mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insert into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Preventable Phenomena and Transaction Isolation Levels

The four levels of transaction isolation with differing degrees of impact on transaction processing throughput. These isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions. The three preventable phenomena are:

- Dirty reads: A transaction reads data that has been written by another transaction that has not been committed yet
- Nonrepeatable (fuzzy) reads: A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.
- Phantom reads: A transaction re-executes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

There are four levels of isolation

isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Not possible	Not possible	Not possible	Not possible

Oracle offers the read committed and serializable isolation levels, as well as a read-only mode that is not part of SQL92. Read committed is the default.

Locking Mechanisms

Multiuser databases use some form of data locking to solve the problems associated with data concurrency, consistency, and integrity. Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource. Resources include two general types of objects:

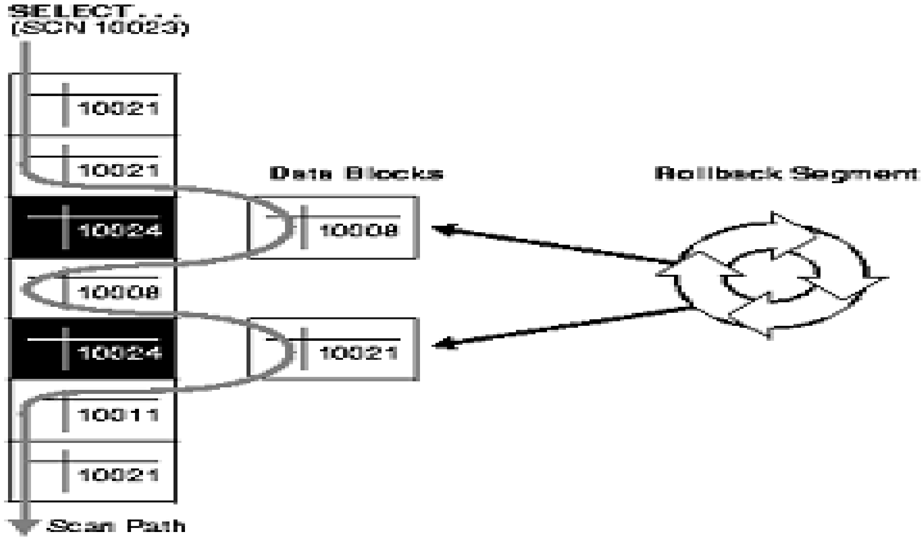
- User objects, such as tables and rows (structures and data)
- System objects not visible to users, such as shared data structures in the memory and data• dictionary rows.

Multiversion Concurrency Control

Oracle automatically provides read consistency to a query so that all the data that the query sees comes from a single point in time (statement-level read consistency). Oracle can also provide read consistency to all of the queries in a transaction (transaction-level read consistency).

Oracle uses the information maintained in its rollback segments to provide these consistent views. The rollback segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Figure shows how Oracle provides statement level read consistency using data in rollback segments.

Transactions and Read Consistency



As a query enters the execution stage, the current system change number (SCN) is determined. In Figure this system change number is 10023. As data blocks are read on behalf of the query, only blocks written with the observed SCN are used. Blocks with changed data (more recent SCNs) are reconstructed from data in the rollback segments, and the reconstructed data is returned for the query. Therefore, each query returns all committed data with respect to the SCN recorded at the time that query execution began. Changes of other transactions that occur during a query's execution are not observed, guaranteeing that consistent data is returned for each query.

Statement-Level Read Consistency

Oracle always enforces statement-level read consistency. This guarantees that all the data returned by a single query comes from a single point in time--the time that the query began. Therefore, a query never sees dirty data nor any of the changes made by transactions that commit during query execution. As query execution proceeds, only data committed before the query began is visible to the query. The query does not see changes committed after statement execution begins.

A consistent result set is provided for every query, guaranteeing data consistency, with no action on the user's part. The SQL statements SELECT, INSERT with a subquery, UPDATE, and DELETE all query data, either explicitly or implicitly, and all return consistent data. Each of these statements uses a query to determine which data it will affect (SELECT, INSERT, UPDATE, or DELETE, respectively).

A SELECT statement is an explicit query and can have nested queries or a join operation. An INSERT statement can use nested queries. UPDATE and DELETE statements can use WHERE clauses or subqueries to affect only some rows in a table rather than all rows.

Queries used in INSERT, UPDATE, and DELETE statements are guaranteed a consistent set of results. However, they do not see the changes made by the DML statement itself. In other words, the query in these operations sees data as it existed before the operation began to make changes.

Transaction-Level Read Consistency

Oracle also offers the option of enforcing transaction-level read consistency. When a transaction executes in serializable mode, all data accesses reflect the state of the database as of the time the transaction began. This means that the data seen by all queries within the same transaction is consistent with respect to a single point in time, except that queries made by a serializable transaction do see changes made by the transaction itself. Transaction-level read consistency produces repeatable reads and does not expose a query to phantoms

Oracle Isolation Levels.

Oracle provides these transaction isolation levels.

Isolation Level	**Description**
Read committed	This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query never reads dirty (uncommitted) data. Because Oracle does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that executes a given query twice can experience both non-repeatable read and phantoms.
Serializable	This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle query never reads dirty (uncommitted) data. Because Oracle does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that executes a given query twice can experience both non-repeatable read and phantoms.
Read-only	Read-only transactions see only those changes that were committed at the time the transaction began and do not allow INSERT, UPDATE, and DELETE statements.

Set the Isolation Level

Application designers, application developers, and database administrators can choose appropriate isolation levels for different transactions, depending on the application and workload. You can set the isolation level of a transaction by using one of these statements at the beginning of a transaction:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SET TRANSACTION ISOLATION LEVEL READ ONLY;

To save the networking and processing cost of beginning each transaction with a SET TRANSACTION statement, you can use the ALTER SESSION statement to set the transaction isolation level for all subsequent transactions:

ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;

ALTER SESSION SET ISOLATION_LEVEL READ COMMITTED;

Read Committed Isolation

The default isolation level for Oracle is read committed. This degree of isolation is appropriate for environments where few transactions are likely to conflict. Oracle causes each query to execute with respect to its own materialized view time, thereby permitting nonrepeatable reads and phantoms for multiple executions of a query, but providing higher potential throughput. Read committed isolation is the appropriate level of isolation for environments where few transactions are likely to conflict.

Serializable Isolation

Serializable isolation is suitable for environments:

- With large databases and short transactions that update only a few rows
- Where the chance that two concurrent transactions will modify the same rows is relatively low
- Where relatively long-running transactions are primarily read-only

Serializable isolation permits concurrent transactions to make only those database changes they could have made if the transactions had been scheduled to execute one after another. Specifically, Oracle permits a serializable transaction to modify a data row only if it can determine that prior changes to the row were made by transactions that had committed when the serializable transaction began.

To make this determination efficiently, Oracle uses control information stored in the data block that indicates which rows in the block contain committed and uncommitted changes. In a sense, the block contains a recent history of transactions that affected each row in the block. The amount of history that is retained is controlled by the INITRANS parameter of CREATE TABLE and ALTER TABLE.

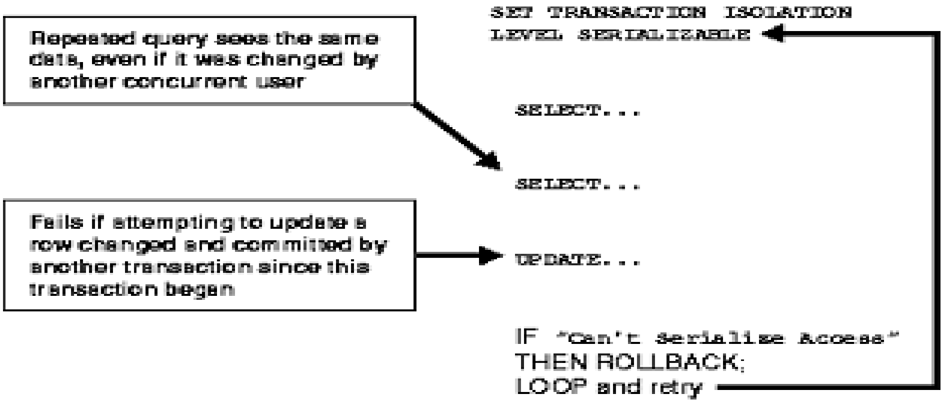
Under some circumstances, Oracle can have insufficient history information to determine whether a row has been updated by a "too recent" transaction. This can occur when many transactions concurrently modify the same data block, or do so in a very short period. You can avoid this situation by setting higher values of INITRANS for tables that will experience many transactions updating the same blocks. Doing so enables Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block.

Oracle generates an error when a serializable transaction tries to update or delete data modified by a transaction that commits after the serializable transaction began:

ORA-08177: Cannot serialize access for this transaction

When a serializable transaction fails with the "Cannot serialize access" error, the application can take any of several actions:

Commit the work executed to that point Execute additional (but different) statements (perhaps after rolling back to a savepoint established earlier in the transaction) Roll back the entire transaction Figure shows an example of an application that rolls back and retries the transaction after it fails with the "Cannot serialize access" error:


Repeated query sees the same data, even if it was changed by another concurrent user
Fails if attempting to update a row changed and committed by another transaction since this transaction began
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SELECT...
SELECT...
UPDATE...
IF "Can't Serialize Access" THEN ROLLBACK; LOOP and retry

How Oracle Locks Data

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource--either user objects such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.

In all cases, Oracle automatically obtains necessary locks when executing SQL statements, so users need not be concerned with such details. Oracle automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency yet also provide fail-safe data integrity. Oracle also allows the user to lock data manually.

Transactions and Data Concurrency

Oracle provides data concurrency and integrity between transactions using its locking mechanisms. Because the locking mechanisms of Oracle are tied closely to transaction control, application designers need only define transactions properly, and Oracle automatically manages locking.

Keep in mind that Oracle locking is fully automatic and requires no user action. Implicit locking occurs for all SQL statements so that database users never need to lock any resource explicitly. Oracle's default locking mechanisms lock data at the lowest level of restrictiveness to guarantee data integrity while allowing the highest degree of data concurrency.

Modes of Locking

Oracle uses two modes of locking in a multiuser database:

- Exclusive lock mode prevents the associates resource from being shared. This lock mode is obtained to modify data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
- Share lock mode allows the associated resource to be shared, depending on the operations involved. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

Lock Duration

All locks acquired by statements within a transaction are held for the duration of the transaction, preventing destructive interference including dirty reads, lost updates, and destructive DDL operations from concurrent transactions. The changes made by the SQL statements of one transaction become visible only to other transactions that start *after* the first transaction is committed.

Oracle releases all locks acquired by the statements within a transaction when you either commit or roll back the transaction. Oracle also releases locks acquired after a savepoint when rolling back to the savepoint. However, only transactions not waiting for the previously locked resources can acquire locks on the now available resources. Waiting transactions will continue to wait until after the original transaction commits or rolls back completely.

Data Lock Conversion V/s Lock Escalation

A transaction holds exclusive row locks for all rows inserted, updated, or deleted within the transaction. Because row locks are acquired at the highest degree of restrictiveness, no lock conversion is required or performed.

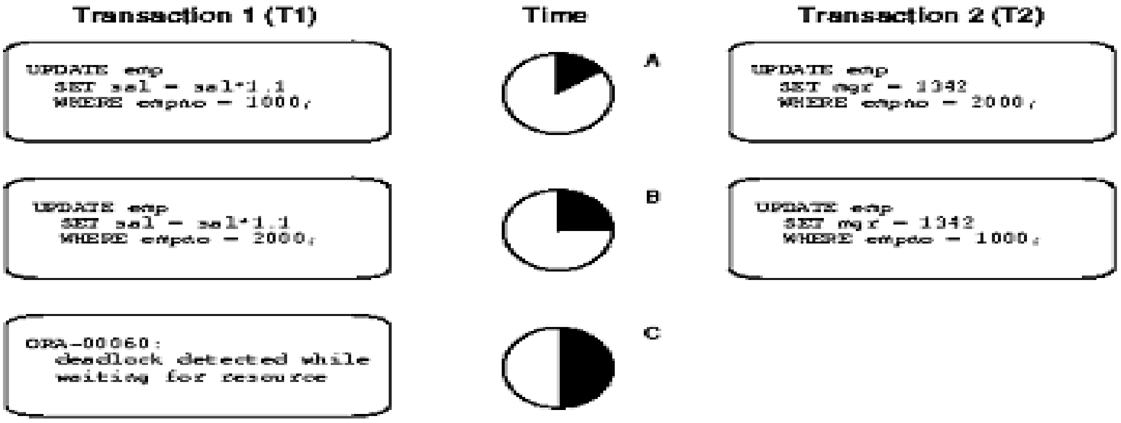
Oracle automatically converts a table lock of lower restrictiveness to one of higher restrictiveness as appropriate. For example, assume that a transaction uses a SELECT statement with the FOR UPDATE clause to lock rows of a table. As a result, it acquires the exclusive row locks and a row share table locks for the table. If the transaction later updates one or more of the locked rows, the row share table lock is automatically converted to a row exclusive table lock.

Lock escalation occurs when numerous locks are held at one level of granularity (for example, rows) and a database raises the locks to a higher level of granularity (for example, table). For example, if a single user locks many rows in a table, some databases automatically escalate the user's row locks to a single table. The number of locks is reduced, but the restrictiveness of what is being locked is increased.

Oracle never escalates locks. Lock escalation greatly increases the likelihood of deadlocks. Imagine the situation where the system is trying to escalate locks on behalf of transaction T1 but cannot because of the locks held by transaction T2. A deadlock is created if transaction T2 also requires lock escalation of the same data before it can proceed.

Deadlocks

Deadlock is a situation in which two or more competing are waiting for the other to finish. it can occur when two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work. Figure is a hypothetical illustrates two transactions in a deadlock.



```
UPDATE emp
  SET sal = sal*1.1
  WHERE empno = 1000;
```

```
UPDATE emp
  SET mgr = 1342
  WHERE empno = 2000;
```

```
UPDATE emp
  SET sal = sal*1.1
  WHERE empno = 2000;
```

```
UPDATE emp
  SET mgr = 1342
  WHERE empno = 1000;
```

```
ORA-00060:
  deadlock detected while
  waiting for resource
```

In Figure, no problem exists at time point A, as each transaction has a row lock on the row it attempts to update. Each transaction proceeds without being terminated. However, each tries next to update the row currently held by the other transaction. Therefore, a deadlock results at time point B, because neither transaction can obtain the resource it needs to proceed or terminate. It is a deadlock because no matter how long each transaction waits, the conflicting locks are held.

Deadlock Detection

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging to the transaction that detects the deadlock. Usually, the signaled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Deadlocks most often occur when transactions explicitly override the default locking of Oracle. Because Oracle itself does no lock escalation and does not use read locks for queries, but does use row-level locking (rather than page-level locking), deadlocks occur infrequently in Oracle.

Avoid Deadlocks

Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order, either through implicit or explicit locks. For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table. If such rules are properly designed and then followed in all applications, deadlocks are very unlikely to occur.

When you know you will require a sequence of locks for one transaction, consider acquiring the most exclusive (least compatible) lock first.

Types of Locks

Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users. Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource. The lock is released automatically when some event occurs so that the transaction no longer requires the resource.

Throughout its operation, Oracle automatically acquires different types of locks at different levels of restrictiveness depending on the resource being locked and the operation being performed.

Oracle locks fall into one of three general categories.

Lock	**Description**
DML locks **(data locks)**	DML locks protect data. For example, table locks lock entire tables, row locks lock selected rows.
DDL locks **(dictionary locks)**	DDL locks protect the structure of schema objects--for example, the definitions of tables and views.
Internal locks and latche	I nternal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.

The following sections discuss DML locks, DDL locks, and internal locks.

DML Locks

The purpose of a DML (data) lock is to guarantee the integrity of data being accessed concurrently by multiple users. DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations. For example, Oracle DML locks guarantee that a specific row in a table can be updated by only one transaction at a time and that a table cannot be dropped if an uncommitted transaction contains an insert into the table.

DML operations can acquire data locks at two different levels: for specific rows and for entire tables.

Row Locks (TX)

The only DML locks Oracle acquires automatically are row-level locks. There is no limit to the number of row locks held by a statement or transaction, and Oracle does not escalate locks from the row level to a coarser granularity. Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.

The combination of multiversion concurrency control and row-level locking means that users contend for data only when accessing the same rows, specifically:

- Readers of data do not wait for writers of the same data rows.
- Writers of data do not wait for readers of the same data rows unless SELECT ... FOR UPDATE is used, which specifically requests a lock for the reader.
- Writers only wait for other writers if they attempt to update the same rows at the same time

A transaction acquires an exclusive DML lock for each individual row modified by one of the following statements: INSERT, UPDATE, DELETE, and SELECT with the FOR UPDATE clause.

A modified row is **always** locked exclusively so that other users cannot modify the row until the transaction holding the lock is committed or rolled back. However, if the transaction dies due to instance failure, block-level recovery makes a row available before the entire transaction is recovered. Row locks are always acquired automatically by Oracle as a result of the statements listed previously.

If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

Table Locks (TM)

A transaction acquires a table lock when a table is modified in the following DML statements: INSERT, UPDATE, DELETE, SELECT with the FOR UPDATE clause, and LOCK TABLE. These DML operations require table locks for two purposes: to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction. Any table lock prevents the acquisition of an exclusive DDL lock on the same table and thereby prevents DDL operations that require such locks. For example, a table cannot be altered or dropped if an uncommitted transaction holds a table lock for it.

A table lock can be held in any of several modes: row share (RS), row exclusive (RX) share (S), share row exclusive (SRX), and exclusive (X). The restrictiveness of a table lock' mode determines the modes in which other table locks on the same table can be obtained an held.

Table shows the table lock modes that statements acquire and operations that those locks permit and prohibit.

Table Summary of Table Locks

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
SELECT...FROM table...	**none**	**Y**	**Y**	**Y**	**Y**	**Y**
INSERT INTO table ...	**RX**	**Y**	**Y**	**N**	**N**	**N**
UPDATE table ...	**RX**	**Y***	**Y***	**N**	**N**	**N**
DELETE FROM table ...	**RX**	**Y***	**Y***	**N**	**N**	**N**
SELECT ... FROM table FOR UPDATE OF ...	**RS**	**Y***	**Y***	**Y***	**Y***	**Y***
LOCK TABLE table IN ROW SHARE MODE	**RS**	**Y**	**Y**	**Y**	**Y**	**N**
LOCK TABLE table IN ROW EXCLUSIVE MODE	**RX**	**Y**	**Y**	**N**	**N**	**N**
LOCK TABLE table IN SHARE MODE	**S**	**Y**	**N**	**N**	**N**	**N**
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	**SRX**	**Y**	**N**	**N**	**N**	**N**
LOCK TABLE table IN EXCLUSIVE MODE	**X**	**N**	**N**	**N**	**N**	**N**

RS: row share, **RX**: row exclusive, **S**: share, **SRX**: share row exclusive, **X**: exclusive
***Yes,** if no conflicting row locks are held by another transaction. Otherwise, waits occur.

Row Share Table Locks (RS)

A row share table lock (also sometimes called a **subshare table lock, SS**) indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. A row share table lock is automatically acquired for a *table* when one of the following SQL statements is executed:

SELECT ... FROM *table* ... FOR UPDATE OF ... ;

LOCK TABLE *table* IN ROW SHARE MODE;

A row share table lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.

<u>Permitted Operations</u>: A row share table lock held by a transaction allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, other transactions can obtain simultaneous row share, row exclusive, share, and share row exclusive table locks for the same table.

<u>Prohibited Operations:</u> A row share table lock held by a transaction prevents other transactions from exclusive write access to the same table using only the following statement:

LOCK TABLE *table* IN EXCLUSIVE MODE;

Row Exclusive Table Locks (RX)

A row exclusive table lock (also called a subexclusive table lock, SX) generally indicates that the transaction holding the lock has made one or more updates to rows in the table. A row exclusive table lock is acquired automatically for a table modified by the following types of statements:

INSERT INTO *table* ... ;
UPDATE *table* ... ;
DELETE FROM *table* ... ;
LOCK TABLE *table* IN ROW EXCLUSIVE MODE;

A row exclusive table lock is slightly more restrictive than a row share table lock.

<u>Permitted Operations:</u> A row exclusive table lock held by a transaction allows other transactio to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, ro exclusive table locks allow multiple transactions to obtain simultaneous row exclusive and ro share table locks for the same table.

<u>Prohibited Operations:</u> A row exclusive table lock held by a transaction prevents other transactions from manually locking the table for exclusive reading or writing. Therefore, other transactions cannot concurrently lock the table using the following statements:

LOCK TABLE *table* IN SHARE MODE;

LOCK TABLE *table* IN SHARE EXCLUSIVE MODE;

LOCK TABLE *table* IN EXCLUSIVE MODE;

Share Table Locks (S)

A share table lock is acquired automatically for the *table* specified in the following statement:

LOCK TABLE *table* IN SHARE MODE;

Permitted Operations: A share table lock held by a transaction allows other transactions only to query the table, to lock specific rows with SELECT ... FOR UPDATE, or to execute LOCK TABLE ... IN SHARE MODE statements successfully. No updates are allowed by other transactions. Multiple transactions can hold share table locks for the same table concurrently. In this case, no transaction can update the table (even if a transaction holds row locks as the result of a SELECT statement with the FOR UPDATE clause). Therefore, a transaction that has a share table lock can update the table only if no other transactions also have a share table lock on the same table.

Prohibited Operations: A share table lock held by a transaction prevents other transactions from modifying the same table and from executing the following statements:

LOCK TABLE *table* IN SHARE ROW EXCLUSIVE MODE;
LOCK TABLE *table* IN EXCLUSIVE MODE;
LOCK TABLE *table* IN ROW EXCLUSIVE MODE;

Share Row Exclusive Table Locks (SRX)

A share row exclusive table lock (also sometimes called a **share-subexclusive table lock, SSX**) is more restrictive than a share table lock. A share row exclusive table lock is acquired for a *table* as follows:

LOCK TABLE *table* IN SHARE ROW EXCLUSIVE MODE;

Permitted Operations: Only one transaction at a time can acquire a share row exclusive table lock on a given table. A share row exclusive table lock held by a transaction allows other transactions to query or lock specific rows using SELECT with the FOR UPDATE clause, but not to update the table.

Prohibited Operations: A share row exclusive table lock held by a transaction prevents other transactions from obtaining row exclusive table locks and modifying the same table. A share row exclusive table lock also prohibits other transactions from obtaining share, share row exclusive, and exclusive table locks, which prevents other transactions from executing the following statements:

LOCK TABLE *table* IN SHARE MODE;
LOCK TABLE *table* IN SHARE ROW EXCLUSIVE MODE;
LOCK TABLE *table* IN ROW EXCLUSIVE MODE;
LOCK TABLE *table* IN EXCLUSIVE MODE;

Exclusive Table Locks (X)

An exclusive table lock is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. An exclusive table lock is acquired for a *table* as follows:

LOCK TABLE *table* IN EXCLUSIVE MODE;

Permitted Operations: Only one transaction can obtain an exclusive table lock for a table. An exclusive table lock permits other transactions only to query the table.

Prohibited Operations: An exclusive table lock held by a transaction prohibits other transactions from performing any type of DML statement or placing any type of lock on the table.

DML Locks Automatically Acquired for DML Statements

The previous sections explained the different types of data locks, the modes in which they can be held, when they can be obtained, when they are obtained, and what they prohibit. The following sections summarize how Oracle automatically locks data on behalf of different DML operations.

Table summarizes the information in the following sections.

Locks Obtained By DML Statement

DML Statement	**Row Locks?**	**Mode of Table Lock**
SELECT ... FROM *table*		
INSERT INTO *table* ...	**X**	**RX**
UPDATE *table* ...	**X**	**RX**
DELETE FROM *table* ...	**X**	**RX**
SELECT ... FROM *table* ... FOR UPDATE OF ...	**X**	**RS**
LOCK TABLE *table* IN ...		
ROW SHARE MODE		**RS**
ROW EXCLUSIVE MODE		**RX**
SHARE MODE		**S**
SHARE EXCLUSIVE MODE		**SRX**
EXCLUSIVE MODE		**X**

X: exclusive, **RX**: row exclusive, **RS**: row share, **S**: share, **SRX**: share row exclusive

Default Locking for Queries

Queries are the SQL statements least likely to interfere with other SQL statements because they only read data. INSERT, UPDATE, and DELETE statements can have implicit queries as part of the statement. Queries include the following kinds of statements:

SELECT
INSERT ... SELECT ... ;
UPDATE ... ;
DELETE ... ;

They do **not** include the following statement:

SELECT ... FOR UPDATE OF ... ;

The following characteristics are true of all queries that do not use the FOR UPDATE clause:

- A query acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries lacking FOR UPDATE clauses do not acquire any data locks to block other operations, such queries are often referred to in Oracle as **nonblocking queries**.
- A query does not have to wait for any data locks to be released; it can always proceed. (Queries may have to wait for data locks in some very specific cases of pending distributed transactions.)

Default Locking for INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE

The locking characteristics of INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE statements are as follows:

- The transaction that contains a DML statement acquires exclusive row locks on the rows modified by the statement. Other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back.

- The transaction that contains a DML statement does not need to acquire row locks on any rows selected by a subquery or an implicit query, such as a query in a WHERE clause. A subquery or implicit query in a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement it is part of.
- A query in a transaction can see the changes made by previous DML statements in the same transaction, but cannot see the changes of other transactions begun after its own transaction.
- In addition to the necessary exclusive row locks, a transaction that contains a DML statement acquires at least a row exclusive table lock on the table that contains the affected rows. If the containing transaction already holds a share, share row exclusive, or exclusive table lock for that table, the row exclusive table lock is not acquired. If the containing transaction already holds a row share table lock, Oracle automatically converts this lock to a row exclusive table lock.

DDL Locks

A data dictionary lock (DDL) protects the definition of a schema object while that object is acted upon or referred to by an ongoing DDL operation. Recall that a DDL statement implicitly commits its transaction. For example, assume that a user creates a procedure. On behalf of the user's single-statement transaction, Oracle automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent objects referenced in the procedure from being altered or dropped before the procedure compilation is complete.

Oracle acquires a dictionary lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. Only individual schema objects that are modified or referenced are locked during DDL operations. The whole data dictionary is never locked.

DDL locks fall into three categories: exclusive DDL locks, share DDL locks, and breakable parse locks.

Exclusive DDL Locks

Most DDL operations, except for those listed in the next section, "Share DDL Locks", require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, a DROP TABLE operation is not allowed to drop a table while an ALTER TABLE operation is adding a column to it, and vice versa.

During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

DDL operations also acquire DML locks (data locks) on the schema object to be modified.

Share DDL Locks

Some DDL operations require share DDL locks for a resource to prevent destructive interference with conflicting DDL operations, but allow data concurrency for similar DDL operations. For example, when a CREATE PROCEDURE statement is executed, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and therefore acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table. No transaction can alter or drop a referenced table. As a result, a transaction that holds a share DDL lock is guaranteed that the definition of the referenced schema object will remain constant for the duration of the transaction.

A share DDL lock is acquired on a schema object for DDL statements that include the following statements: AUDIT, NOAUDIT, COMMENT, CREATE [OR REPLACE] VIEW/ PROCEDURE/PACKAGE/PACKAGE BODY/FUNCTION/ TRIGGER, CREATE SYNONYM, and CREATE TABLE (when the CLUSTER parameter is not included).

Breakable Parse Locks

A SQL statement (or PL/SQL program unit) in the shared pool holds a parse lock for each schema object it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. A parse lock does not disallow any DDL operation and can be broken to allow conflicting DDL operations, hence the name **breakable parse lock**.

A parse lock is acquired during the parse phase of SQL statement execution and held as long as the shared SQL area for that statement remains in the shared pool.

Duration of DDL Locks

The duration of a DDL lock depends on its type. Exclusive and share DDL locks last for the duration of DDL statement execution and automatic commit. A parse lock persists as long as the associated SQL statement remains in the shared pool.

DDL Locks and Clusters

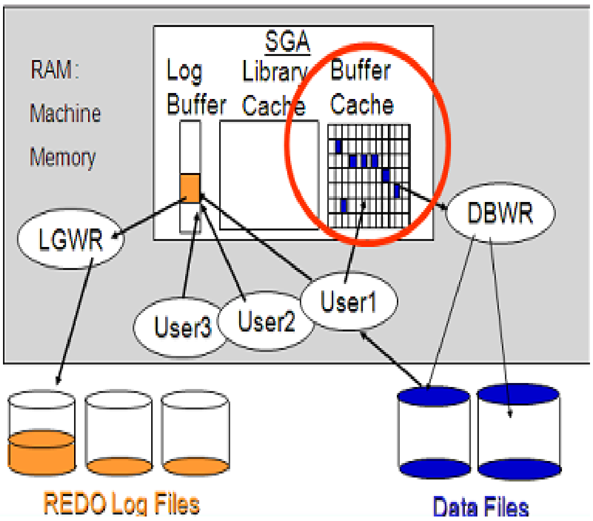
A DDL operation on a cluster acquires exclusive DDL locks on the cluster and on all tables and materialized views in the cluster. A DDL operation on a table or materialized view in a cluster acquires a share lock on the cluster, in addition to a share or exclusive DDL lock on the table or materialized view. The share DDL lock on the cluster prevents another operation from dropping the cluster while the first operation proceeds

Latches and Internal Locks

Latches and internal locks protect internal database and memory structures. Both are inaccessible to users, because users have no need to control over their occurrence or duration. The following section helps to interpret the Enterprise Manager or SQL*Plus LOCKS and LATCHES monitors.

Latches

Latches are simple, low-level serialization mechanisms to protect shared data structures in the system global area (SGA). For example, latches protect the list of users currently accessing the database and protect the data structures describing the blocks in the buffer cache. A server or background process acquires a latch for a very short time while manipulating or looking at one of these structures. The implementation of latches is operating system dependent, particularly in regard to whether and how long a process will wait for a latch.



Internal Locks

Internal locks are higher-level, more complex mechanisms than latches and serve a variety of purposes

Dictionary Cache Locks

These locks are of very short duration and are held on entries in dictionary caches while the entries are being modified or used. They guarantee that statements being parsed do not see inconsistent object definitions. Dictionary cache locks can be shared or exclusive. Shared locks are released when the parse is complete. Exclusive locks are released when the DDL operation is complete.

File and Log Management Locks

These locks protect various files. For example, one lock protects the control file so that only one process at a time can change it. Another lock coordinates the use and archiving of the redo log files. Datafiles are locked to ensure that multiple instances mount a database in shared mode or that one instance mounts it in exclusive mode. Because file and log locks indicate the status of files, these locks are necessarily held for a long time.

Tablespace and Rollback Segment Locks

These locks protect tablespaces and rollback segments. For example, all instances accessing a database must agree on whether a tablespace is online or offline. Rollback segments are locked so that only one instance can write to a segment.

Explicit (Manual) Data Locking

Oracle always performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can override the Oracle default locking mechanisms. Overriding the default locking is useful in situations such as these:

Applications require transaction-level read consistency or **repeatable reads**. In other words, queries in them must produce consistent data for the duration of the transaction, not reflecting changes by other transactions. You can achieve transaction-level read consistency by using explicit locking, read-only transactions, serializable transactions, or by overriding default locking.

Applications require that a transaction have exclusive access to a resource so that the transaction does not have to wait for other transactions to complete.

Oracle's automatic locking can be overridden at the transaction level or the session level. At the transaction level, transactions that include the following SQL statements override Oracle's default locking:

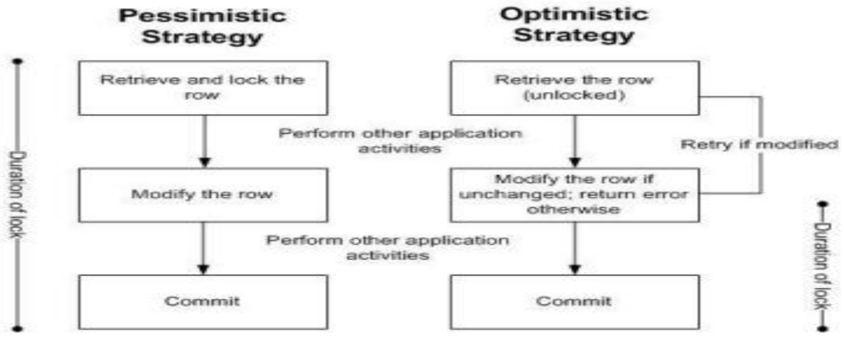
The SET TRANSACTION ISOLATION LEVEL statement

The LOCK TABLE statement (which locks either a table or, when used with views, the underlying base tables)

The SELECT ... FOR UPDATE statement

Locks acquired by these statements are released after the transaction commits or rolls back. At the session level, a session can set the required transaction isolation level with the ALTER SESSION statement.

Differences between Pessimistic and Optimistic Locking


Pessimistic Strategy
Optimistic Strategy
Retrieve and lock the row
Retrieve the row (unlocked)
Perform other application activities
Retry if modified
Modify the row
Modify the row if unchanged; return error otherwise
Perform other application activities
Commit
Commit
Duration of lock
Duration of lock

Locking prevents users from different transactions from causing data conflicts. When locking is acquired on a row, it prevents other transactions from changing that row until the transaction ends. There are two different approaches to transaction locking Pessimistic locking and optimistic Locking

Pessimistic Locking

Pessimistic was the default locking in Oracle. This means that the row is locked in advance once one of its attribute is changed. If anyone else attempts to acquire a lock of the same row during this process, he will be forced to wait until the first transaction has completed. This is achieved by using the familiar SELECT…FOR UPDATE syntax. This is the safest locking mode because two transactions will never make inconsistent change to the same row. However, this locking mode has disadvantages such that.

1. If a user selects a record for update, and then leaves for lunch without finishing or aborting the transaction. All other users that need to update that record are forced to wait until the user returns and completes the transaction, or until the DBA kills the offending transaction and releases the lock.
2. The Deadlock – Users A and B are both updating the database at the same time. User A locks a record and then attempt to acquire a lock held by user B – who is waiting to obtain a lock held by user A.

Pessimistic locking, which is the default, should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This releases all the locks that pessimistic locking had created.

An example about pessimistic locking based on well known *hr* schema, suppose user1 and user2 are two different users (two distinct transactions) using pessimistic locking, both of them try to change the same row of data as follows:

User1 calls Employee Salary is 1000 on a particular row, so user1 immediately acquire a lock on that row.

Now user2 calls Employee Salary is 2000 on the same row, user2 tries to acquire a lock on the row and receives.

Advantages of Pessimistic Locking

The following are the advantages of pessimistic locking:

- It prevents users and applications from editing data that is being or has been changed.
- Processes know immediately when a locking violation occurs, rather than after the transaction is complete.

Disadvantages of Pessimistic Locking

The following are the disadvantages of pessimistic locking:

- It is not fully supported by all databases.
- It consumes extra database resources.
- It requires Oracle AS TopLink to maintain an open transaction and database lock for the duration of the transaction, which can lead to database deadlocks.
- It decreases the concurrency of connection pooling when using the server session, which affects the overall scalability of your application.

Optimistic Locking

With Optimistic locking, a resource is not actually locked when it is first is accessed by a transaction. Instead the state of the resource at that time is saved. Other transactions are able to concurrently access the resource and the possibility of conflicting charges exists. At commit time, when the resource is about to be updated in persistent strong, the state of the resource is read from storage again and it is compared to the state that was saved when the resource was first accessed in the transaction. If two states differ- indicating a conflicting update- the transaction will be rolled back.

Oracle recommends using optimistic locking for web applications. Instead of locking a row as soon as it is changed, under optimistic locking, transaction waits until changed row is posted before attempting to acquire a lock. An exception is not thrown until the conflicting transactions attempt to post their changes to the database.

An example about optimistic locking, suppose user1 and user2 are two different users (two distinct transactions) using optimistic locking, both of them try to change the same row of data as follows:

1. User1 calls Employee Salary 1000 on a particular row; user1 does not immediately acquire a lock on that row.
2. User2 calls Employee Salary 2000 on the same row. User1 and User2 now have different entity cache for the same row.
3. User2 calls commit action, as part of the commit cycle the changed row is posted to the database. before the update can be executed, user2 acquires a lock on that row. The lock expires immediately, when the commit command is sent to the database
4. User1 now calls commit action, transaction tries to post the changed row to the database, right before posting it, it attempts to acquire a lock on that row. BC4J recognizes that the row has been changed by another user and that updating the row would overwrite another transaction's changes, so it throws an oracle.jbo.RowInconsistentException.

Advantages of optimistic locking:

- It prevents users and applications from editing stale data.
- It notifies users of any locking violation immediately, when updating the object.
- It does not require you to lock up the database resource.
- It prevents database deadlocks.

UNIT 4

INTRODUCTION TO PL/SQL

PL/SQL stands for "Procedural Language Extensions to SQL." PL/SQL extends SQL by adding programming structures and subroutines available in any high-level language. PL/SQL can also handle runtime errors. Options such as loops and IF...THEN statements give PL/SQL the power of third-generation programming languages. PL/SQL allows you to write interactive, user-friendly programs that can pass values into variables.

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.

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:

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:

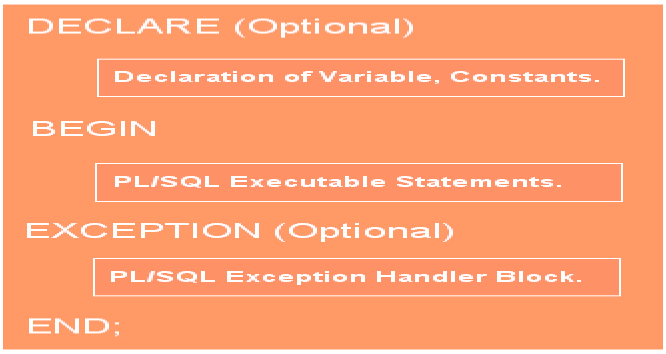
SYNTAX:

```
    -- This is a one-line comment.
/*  This is a
    multiple-line
    comment.*/
```

PL/SQL is a block-structured language. Each of the basic programming units you write to build your application is (or should be) a logical unit of work. The PL/SQL block allows you to reflect that logical structure in the physical design of your programs.

The block determines both the scope of identifiers (the area of code in which a reference to the identifier can be resolved) and the way in which exceptions are handled and propagated. A block may also contain nested sub-blocks of code, each with its own scope

PL/SQL is block structured language divided into three logical blocks. BEGIN block and END; keyword are compulsory, and other two block DECLARE and EXCEPTION are optional block. END; is not a block only keyword to end of PL/SQL program. PL/SQL block structure follows divide-and-conquer approach to solve the problem stepwise.



DECLARE (Optional)

Declaration of Variable, Constants.

BEGIN

PL/SQL Executable Statements.

EXCEPTION (Optional)

PL/SQL Exception Handler Block.

END;

DECLARE

Variables and constants are declared, initialized within this section.

- **Variables and Constants** : In this block, declare and initialize variables (and constants). You must have to declare variables and constants in declarative block before referencing them in procedural statement.
- **Declare Variables and Assigning values** : You can define variable name, data type of a variable and its size. Date type can be: CHAR, VARCHAR2, DATE, NUMBER, INT or any other.
- **Declare Constants and Assigning values :** Constants are declared same as variable but you have to add the CONSTANT keyword before defining data type. Once you define a constant value you can't change the value.
- designation CONSTANT VARCHAR2(30) := 'Web Developer';

BEGIN

BEGIN block is procedural statement block which will implement the actual programming logic. This section contains conditional statements (if...else), looking statements (for, while) and Branching Statements (goto) etc.

EXCEPTION

PL/SQL easily detects user defined or predefined error condition. PL/SQL is famous for handling errors in smart way by giving suitable user friendly messages. Errors can be rise due to wrong syntax, bad logical or not passing a validation rules.

You can also define exception in your declarative block and later you can execute it by RAISE statement.

END

This marks the end of a PL/SQL block.

NOTE:

1. BEGIN block and END; keyword are compulsory of any PL/SQL program.
2. DECLARE and EXCEPTION block are optional.

PL/SQL CHARACTER SET : The PL/SQL language is constructed from letters, digits, symbols, and whitespace, as defined in the following table:

Type	Characters
Letters	A-Z, a-z
Digits	0-9
Symbols	~!@#$%^&*( )_-+=|[ ]{ }:;"'< >,.?/ ^
Whitespace	space, tab, newline, carriage return

Characters are grouped together into four lexical units: identifiers, literals, delimiters, and comments

Identifiers: Identifiers are names for PL/SQL objects such as constants, variables, exceptions, procedures, cursors, and reserved words. Identifiers have the following characteristics:

- Can be up to 30 characters in length
- Cannot include whitespace (space, tab, carriage return)
- Must start with a letter
- Can include a dollar sign ($), an underscore (_), and a pound sign (#)
- Are not case-sensitive
- In addition, you must not use PL/SQL's reserved words as identifiers.

LITERALS: A literal is numeric value or a character string used to represent itself.

Numeric Literals:

These can be integer or float is being represented, then the integer part must be separated from the float part by a period.

For Example: 25, 6.34, 7g2, 25e-03, .1, 1., 1.e4, +17,-5

String Literals:

There are represented by one or more legal characters and must be enclosed within single quotes. The single quote character can be represented, by writing It twice in a string literal. This is definitely not the same as a double quote. For Example: 'Hello World', 'Don''t go without saving your work'

'That''s Entertainment!' ->That's Entertainment!

Character Literal:

These are string literals consisting of single characters.

For Example: '*' , 'A' ,'Y'

Logical (Boolean) Literal:

These are predetermined constants. The values that can be assigned to this data type are TRUE, FALSE, And NULL. Literals are specific values not represented by identifiers. For example, TRUE, 3.14159, 6.63E-34, `Moby Dick', and NULL are all literals of type Boolean, number, or string. There are no complex data type literals as they are internal representations. Unlike the rest of PL/SQL, literals are case-sensitive.

DATA TYPES IN PL/SQL

As you know about the variables and constants stores value in specific storage format. There are six built-in PL/SQL Data types

1. **Scalar data types** - Scalar data types haven't internal components.
2. **Composite data types** - Composite data types have internal components to manipulate data easily.
3. **Reference data types** - This data types works like a pointer to hold some value.
4. **LOB data types** - Stores large objects such as images, graphics, video.
5. **Unknown Column types** - Identify columns when not know type of column.

Scalar types

Scalar data type haven't internal components. It is like a linear data type. Scales data type divides into four different types character, numeric, Boolean or date/time type. Numeric Data types Following are numeric data types in PL/SQL:

<table>
<tr><th>Data types</th><th>Description, Storage(Maximum)</th></tr>
<tr><td><b>NUMBER(p,s)</b></td><td>NUMBER data type used to store numeric data. It's contain letters, numbers, and special characters. Storage Range : Precision range(p) : 1 to 38 and Scale range(s) : -84 to 127 NUMBER Subtypes : This sub type defines different types storage range.
<table>
<tr><th>Sub Data types</th><th>Maximum Precision</th><th>Description</th></tr>
<tr><td>INTEGER</td><td>38 digits</td><td rowspan="9">This data types are used to store fixed decimal points. You can use based on your requirements.</td></tr>
<tr><td>INT</td><td>38 digits</td></tr>
<tr><td>SMALLINT</td><td>38 digits</td></tr>
<tr><td>DEC</td><td>38 digits</td></tr>
<tr><td>DECIMAL</td><td>38 digits</td></tr>
<tr><td>NUMERIC</td><td>38 digits</td></tr>
<tr><td>REAL</td><td>63 binary digits</td></tr>
<tr><td>DOUBLE PRECISION</td><td>126 binary digits</td></tr>
<tr><td>FLOAT</td><td>126 binary digits</td></tr>
</table></td></tr>
<tr><td><b>BINARY_INTEGER</b></td><td><b>BINARY_INTEGER</b> data type store signed integer's value.
Note :
BINARY_INTEGER values require less storage space compare of NUMBER data type values. Storage Range : from -2147483647 to 2147483647
BINARY_INTEGER Subtypes : This sub type define constraint to store a value.
<table>
<tr><th>Sub Data types</th><th>Description</th></tr>
<tr><td>NATURAL</td><td rowspan="2">NATURAL/POSITIVE data type prevent to store negative value, allow only positive values.</td></tr>
<tr><td>POSITIVE</td></tr>
<tr><td>NATURALN</td><td rowspan="2">NATURALN/POSITIVEN data type prevent to assign a NULL value.</td></tr>
<tr><td>POSITIVEN</td></tr>
<tr><td>SIGNTYPE</td><td>SIGNTYPE allow only -1, 0, and 1 values.</td></tr>
</table></td></tr>
<tr><td><b>PLS_INTEGER</b></td><td>PLS_INTEGER data type used to store signed integers data.
Note :
PLS_INTEGER data type value require less storage space compare of NUMBER data type value. Storage Range : from -2147483647 to 2147483647
Performance : PLS_INTEGER data type give you better performance on your data. PLS_INTEGER perform arithmetic operation fast than NUMBER/BINARY_INTEGER data type.</td></tr>
</table>

Character Data types

Character Data types used to store alphabetic/alphanumeric character. Following are some character data types in PL/SQL,

<table>
<tr><th>Data types</th><th>Description</th><th>Storage (Maximum)</th></tr>
<tr><td><b>CHAR</b></td><td>CHAR data type used to store character data within predefined length.</td><td>32767 bytes</td></tr>
<tr><td><b>CHARACTER</b></td><td>CHARACTER data type same as CHAR type, is this another name of CHAR type.</td><td>32767 bytes</td></tr>
<tr><td><b>VARCHAR2</b></td><td>VARCHAR2 data type used to store variable strings data within predefined length. VARCHAR2 Subtypes : Following sub type defines same length value.
<table>
<tr><th>Sub Data types</th><th>Description</th></tr>
<tr><td>STRING</td><td rowspan="2">We can access this data type.</td></tr>
<tr><td>VARCHAR</td></tr>
</table></td><td>32767 bytes</td></tr>
<tr><td><b>NCHAR</b></td><td>NCHAR data type used to store national character data within predefined length.</td><td>32767 bytes</td></tr>
<tr><td><b>NVARCHAR2</b></td><td>NVARCHAR2 data type used to store Unicode string data within predefined length.</td><td>32767 bytes</td></tr>
<tr><td><b>RAW</b></td><td>The RAW data type used to store binary data such as images, graphics etc.</td><td>32767 bytes</td></tr>
<tr><td><b>LONG</b></td><td>LONG data type used to store variable string data within predefined length, This data type used for backward compatibility. Please use LONG data to the CLOB type.</td><td>32760 bytes</td></tr>
<tr><td><b>LONG RAW</b></td><td>LONG RAW data type same as LONG type used to store variable string data within predefined length, This data type used for backward compatibility.
Use LONG RAW data type for storing BLOB type data.</td><td>32760 bytes</td></tr>
<tr><td><b>ROWID</b></td><td colspan="2">The ROWID data type represents the actual storage address of a row. And table index identities as a logical rowid. This data type used to storing backward compatibility. We strongly recommended to use UROWID data type.</td></tr>
<tr><td><b>UROWID [(size)]</b></td><td>The UROWID data type identifies as universal rowid, same as ROWID data type. Use UROWID data type for developing newer applications.
Optional, You can also specify the size of UROWID column type.</td><td>4000 bytes</td></tr>
</table>

Boolean Data types

Boolean Data types stores logical values either TRUE or FALSE. Let's see Boolean data types in PL/SQL:

Data types	Description
Boolean	Boolean data type stores logical values. Boolean data types don't take any parameters. Boolean data type store either TRUE or FALSE. Also store NULL, Oracle treats NULL as an unassigned Boolean variable. You cannot fetch Boolean column value from another table.

Date/Time Data types

Date/time variable can holds value, we can say date/time data type. PL/SQL automatically converts character value in to default date format ('DD-MM-YY') value. Following are the Date/Time data types in PL/SQL:

<table>
<tr><th>Data Types</th><th>Description</th><th>Range</th></tr>
<tr><td><b>DATE</b></td><td>DATE data type stores valid date-time format with fixed length. Starting date from Jan 1, 4712 BC to Dec 31, 9999 AD.</td><td>Jan 1, 4712 BC to Dec 31, 9999 AD</td></tr>
<tr><td><b>TIMESTAMP</b></td><td colspan="2">TIMESTAMP data type stores valid date (year, month, day) with time (hour, minute, second).
<table>
<tr><th>Type</th><th>TIMESTAMP Type</th></tr>
<tr><td>1</td><td>Syntax : TIMESTAMP [(fractional_seconds_precision)]
Example : TIMESTAMP '2014-04-13 18:10:52.124'
fractional_seconds_precision optionally specifies the number of digits in the fractional part of the second precision. Range from 0 to 9. The default is 6.</td></tr>
<tr><td>2</td><td>Syntax : TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE
Example : TIMESTAMP '2014-04-13 18:10:52.124 +05:30' WITH TIME ZONE specify the UTC time zone. Following two values represent the same instant in UTC.
TIMESTAMP '1999-04-15 8:00:00 -8:00' (8.00 AM Pacific Standard Time) or
TIMESTAMP '1999-04-15 11:00:00 -5:00' (11:00 AM Eastern Standard Time) both are same.</td></tr>
<tr><td>3</td><td>Syntax : TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE
Example : COL_NAME TIMESTAMP(3) WITH LOCAL TIME ZONE;
WITH LOCAL TIME ZONE specifies when you insert values into the database column, value is stored with the time zone of the database.
The time-zone displacement is not stored in the column. When you retrieve value from Oracle database, returns it according to your UTC local time zone.</td></tr>
</table>
</td></tr>
</table>

LOB types

LOB data types used to store large objects such as image, video, graphics, text or audio. Maximum size is up to 4 Gigabytes. Following are the LOB data types in PL/SQL:

Data types	Description	Storage(Maximum)
BFILE	BFILE data type is used to store large binary objects into Operating System file. BFILE stores full file locator's path which are points to a stored binary object with in server. BFILE data type is read only, you can't modify them.	Size: up to 4GB (232 - 1 bytes) Directory name: 30 character File name: 255 characters
BLOB	BLOB data type is same as BFILE data type used to store unstructured binary object into Operating System file. BLOB type fully supported transactions are recoverable and replicated.	Size: 8 TB to 128 TB (4GB - 1) * DB_BLOCK_SIZE
CLOB	CLOB data type is used to store large blocks of character data into Database. Store single byte and multi byte character data. CLOB type is fully supported transactions, also that are recoverable and replicated.	Size: 8 TB to 128 TB (4GB - 1) * DB_BLOCK_SIZE

NCLOB	NCLOB data type to store large blocks of NCHAR data into Database. Store single byte and multi byte character data. NCLOB type fully supported transactions are recoverable and replicated. NCLOB data type is used to store large blocks of NCHAR data into Database. Store single byte and multi byte character data. NCLOB type fully supported transactions, also that are recoverable and replicated.	Size: 8 TB to 128 TB (4GB - 1) * DB_BLOCK_SIZE

Unknown Column types

PL/SQL this data type is used when column type is not know.

<table>
<tr><th>Data types</th><th>Description</th></tr>
<tr><td><b>%Type</b></td><td>This data type is used to store value unknown data type column in a table. Column is identified by %type data type.
<b>Eg. emp.eno%type</b>
emp name is table, eno is a unknown data type column and %Type is data type to hold the value.</td></tr>
<tr><td><b>%RowType</b></td><td>This data type is used to store values unknown data type in all columns in a table. All columns are identified by %RowType datatype.
<b>Eg. emp%rowtype</b>
emp name is table, all column type is %rowtype.</td></tr>
<tr><td><b>%RowID</b></td><td>RowID is data type. RowID is two type extended or restricted. Extended return 0 and restricted return 1 otherwise return the row number.
Function of Row ID:
<table>
<tr><th>Function RowID</th><th>Description</th></tr>
<tr><td><b>ROWID_Verify</b></td><td>Verify if the rowid can be extended.</td></tr>
<tr><td><b>ROWID_Type</b></td><td>0 = rowid, 1 = extended.</td></tr>
<tr><td><b>ROWID_Block_Number</b></td><td>Block number that contain the record return 1 extended.</td></tr>
<tr><td><b>ROWID_Object</b></td><td>Object number of the object that contain record.</td></tr>
<tr><td><b>ROWID_Relative_FNumber</b></td><td>Relative file number that contain record.</td></tr>
<tr><td><b>ROWID_Row_Number</b></td><td>Row number of the Record.</td></tr>
<tr><td><b>ROWID_To_Absolute_FNumber</b></td><td>Return the absolute file number.</td></tr>
<tr><td><b>ROWID_To_Extended</b></td><td>Convert the extended format.</td></tr>
<tr><td><b>ROWID_To_Restricted</b></td><td>Convert the restricted format.</td></tr>
</table>
</td></tr>
</table>

CONTROL STRUCTURE:

The flow of control statements can be classified into the following categories:

Conditional Statements: There are several varieties of IF-THEN-ELSE.

IF-THEN combination
 	
IF condition THEN
executable statement(s)
END IF;
	For example:
	
	IF caller_type = 'VIP' THEN
	generate_response('GOLD');
	END IF;

IF-THEN-ELSE combination
 	
IF condition THEN
TRUE
sequence_of_executable_statement(s)
ELSE
FALSE/NULL
sequence_of_executable_statement(s)
END IF;
	For example:
	
	IF caller_type = 'VIP' THEN
	generate_response('GOLD');
	ELSE
	generate_response('BRONZE');
	END IF;

IF-THEN-ELSIF combination
 	
IF condition-1 THEN
statements-1
ELSIF condition-N THEN
statements-N
[ELSE
ELSE statements]
END IF;
	For example:
	
	IF caller_type = 'VIP' THEN
	generate_response('GOLD');
	ELSIF priority_client THEN
	generate_response('SILVER');
	ELSE
	generate_response('BRONZE');
	END IF;

ITERATIVE(LOOP) STRUCTURE :

Iterative control structure perform one or more statement repeatedly, either a certain number of times or until a condition is met. There are three types of iterative structure

Simple Loop	Example

LOOP
executable_statement(s)
EXIT [WHEN condition];
END LOOP
	DECLARE
	i number:=0;
	BEGIN
	LOOP
	i:=i+2;
	EXIT WHEN i>10;
	END LOOP;
	dbms_output.put_line('Loop exited as the value of i has reached'||to_char(i));
	END;

The simple loop should contain an EXIT or EXIT WHEN unless you want it to execute infinitely. Use the simple loop when you want the body of the loop to execute at least once. For example:

FOR Loop

Syntax	Example

FOR loop_index(Variable) IN [REVERSE] lowest_number (start) ... highest_number(End)
LOOP
executable_statement(s)
END LOOP;
	BEGIN
	FOR counter IN 1 .. 4
	LOOP
	DBMS_OUTPUT.PUT(counter);
	END LOOP;
	DBMS_OUTPUT.NEW_LINE;
	FOR counter IN REVERSE 1 .. 4
	LOOP
	DBMS_OUTPUT.PUT(counter);
	END LOOP;
	DBMS_OUTPUT.NEW_LINE;END;
	End
	\

The PL/SQL runtime engine automatically declares the loop index a PLS_INTEGER variable; never declare a variable with that name yourself. The lowest_number and highest_number ranges can be variables, but are evaluated only once—on initial entry into the loop. The REVERSE keyword causes PL/SQL to start with the highest_number and decrement down to the lowest_number. For example, this code: yields the following output:

WHILE Loop

```
WHILE condition
LOOP
  executable_statement(s)
END LOOP;
```

Use the WHILE loop in cases where you may not want the loop body to execute even once: Example:

```
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;
```

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.

GOTO Statement

Certain PL/SQL control structures offer structured methods for processing executable statements in your program. You use an IF statement or a CASE statement to test a condition to determine which parts of your code to execute; you use a LOOP variation to execute a section of code more than once. In addition to these well-structured approaches to program control, PL/SQL offers the GOTO. The GOTO statement performs unconditional branching to another executable statement in the same execution section of a PL/SQL block. As with other constructs in the language, if you use GOTO appropriately and with care, your programs will be stronger for it. The general format for a GOTO statement is:

GOTO label_name;

where label_name is the name of a label identifying the target statement. This GOTO label is defined in the program as follows:

<<label_name>>

You must surround the label name with double enclosing angle brackets (<< >>). When PL/SQL encounters a GOTO statement, it immediately shifts control to the first executable statement following the label. Following is a complete code block containing both a GOTO and a label:

```
BEGIN
  GOTO second_output;
          DBMS_OUTPUT.PUT_LINE('This line will never execute.');
   <<second_output>>
  DBMS_OUTPUT.PUT_LINE('We are here!');
END;
```

Contrary to popular opinion (including mine), the GOTO statement can come in handy. There are cases where a GOTO statement can simplify the logic in your program. On the other hand, because PL/SQL provides so many different control constructs and modularization techniques, you can almost always find a better way to do something than with a GOTO.

Restrictions on the GOTO Statement

- At least one executable statement must follow a label.
- The target label must be in the samescope as the GOTO statement.
- The target label must be in the same part of the PL/SQL block as the GOTO.

These restrictions are described in detail in the following sections.

At least one executable statement must follow a label A label itself is not an executable statement (notice that it does not have a semicolon after the label brackets), so it cannot take the place of one. All of the uses of the **<<all_done>>** label in the following code are illegal because the labels are not followed by an executable statement:

**IF status_inout = 'COMPLETED'
THEN
<<all_done>> /* Illegal! */
ELSE
schedule_activity;
END IF;**
	**DECLARE
	CURSOR company_cur IS ...;
	BEGIN
	FOR company_rec IN company_cur
	LOOP
	apply_bonuses (company_rec.company_id);
	<<all_done>> /* Illegal! */
	END LOOP;
	END;**

CURSORS IN PL/SQL :

it needs to create an area of memory known as the context area; this will have the information needed to process the statement. This information includes the number of rows processed by the statement, a pointer to the parsed representation of the statement (parsing an SQL statement is the process whereby information is transferred to the server, at which point the SQL statement is evaluated as being valid). In a query, the active set refers to the rows that will be returned. A cursor is a handle, or pointer, to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. Two important features about the cursor are as follows:

1. Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
2. A cursor is named so that it can be referenced.

TYPES OF CURSORS

There are two types of cursors:

1. An **implicit cursor** is automatically declared by Oracle every time an SQL statement is executed. The user will not be aware of this happening and will not be able to control or process the information in an implicit cursor.
2. An **explicit cursor** is defined by the program for any query that returns more than one row of data. That means the programmer has declared the cursor within the PL/SQL code block. This declaration allows for the application to sequentially process each row of data as it is returned by the cursor.

IMPLICIT CURSOR : In order to better understand the capabilities of an explicit cursor, you first need to run through the process of an implicit cursor. The process is as follows:

- Any given PL/SQL block issues an implicit cursor whenever an SQL statement is executed, as long as explicit cursor does not exist for that SQL statement.
- A cursor is automatically associated with every DML (Data Manipulation) statement (UPDATE, DELETE, and INSERT).
- All UPDATE and DELETE statements have cursors that identify the set of rows that will be affected by the operation.
- An INSERT statement needs a place to receive the data that is to be inserted in the database; the implicit cursor fulfills this need.
- The most recently opened cursor is called the 'SQL%' cursor.
- The following table lists the implicit cursor attributes:

Attributes	Description
SQL%ISOPEN	Always FALSE because the cursor is opened implicitly and closed immediately after the statement is executed.
SQL%FOUND	NULL before the statement. TRUE if one or more rows were inserted, merged, updated, or deleted or if only one row was selected. FALSE if no row was selected, merged, updated, inserted, or deleted.
SQL%NOTFOUND	NULL before the statement. TRUE if no row was selected, merged, updated, inserted, or deleted. FALSE if one or more rows were inserted, merged, updated, or deleted.
SQL%ROWCOUNT	Number of rows affected by the cursor.

In the following UPDATE statement, which gives everyone in the company a 10% raise, PL/SQL creates an implicit cursor to identify the set of rows in the table which would be affected by the update:

SQL > UPDATE employee SET salary = salary * 1.1;

The following single row query calculates and returns the total salary for a department. Once again, PL/SQL creates an implicit cursor for this statement:

Select sum(sal) into department_total -- department_total →variable name
From emp Where deptno=10;

Implicit cursor attributes are referenced via the SQL cursor. For example:

```
BEGIN
        UPDATE activity SET last_accessed := SYSDATE WHERE UID = user_id;
        IF SQL%NOTFOUND THEN
                INSERT INTO activity_log (uid,last_accessed) VALUES (user_id,SYSDATE);
        END IF
END;
```

Drawbacks of Implicit Cursors

Even if your query returns only a single row, you might still decide to use an explicit cursor. The implicit cursor has following drawbacks:

- It is less efficient than an explicit cursor
- It is more vulnerable / weak to data errors.
- It gives you less programmatic control.

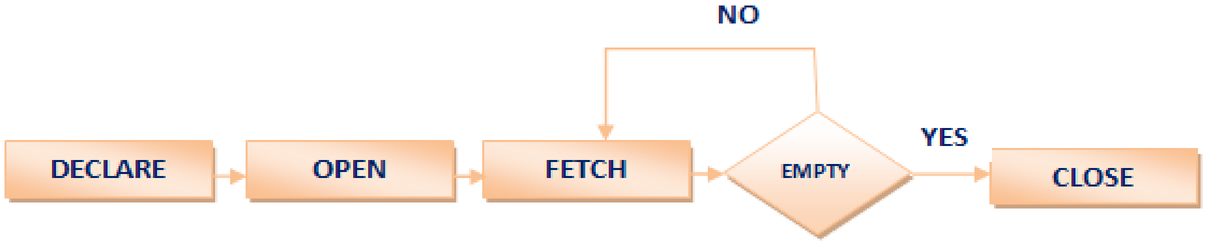
An implicit cursor executes as a SQL statement and Oracle's SQL statement and Oracle's SQL is ANSI – standard. ANSI dictates that a single-row query must not only fetch the first record, but must also perform a second fetch to determine if too many rows will returned by that query. Thus, an implicit query always performs a minimum of two fetches, while an explicit cursor only needs to perform a single fetch.

If an implicit SELECT statement returns more than one row, it raises the TOO_MANY_ROWS exception. When this happens, execution in the current block terminates and control is passed to the exception section. It may well be that today, with the current data; the query will only return a single row. If the nature of the data ever changes, however, you may find that the SELECT statement, which formerly identified a single row, now returns several. With the implicit query, you cannot easily handle these different possibilities. With an explicit query, your program will be protected against changes in data and will continue to fetch rows without raising exceptions.

Explicit Cursors

An explicit cursor is a SELECT statement that is explicitly defined in the declaration section of your code and, in the process assigned a name. There is no such thing as an explicit cursor for UPDATE, DELETE and INSERT statements.

With explicit cursors, you have complete control over how to access information in the database. You decide when to OPEN the cursor, when to FETCH records from the cursor, how many records to fetch and when to CLOSE the cursor. Information about the current state of your cursor is available through examination of the cursor attributes



Declaring PL/SQL Cursor

To use PL/SQL cursor, first you must declare it in the declaration section of PL/SQL block or in a package as follows:

```
CURSOR cursor_name [ ( [ parameter_1 [, parameter_2 ...] ) ]
    [ RETURN return_specification ]
IS sql_select_statements
[FOR UPDATE [OF [column_list]];
```

- First, you declare the name of the cursor cursor_name after the CURSOR keyword. The name of the cursor can have up to 30 characters in length and follows the naming rules of identifiers in PL/SQL. It is important to note that cursor's name is not a variable so you cannot use it as a variable such as assigning it to other cursor or using it in an expression. The parameter1, parameter2… are optional elements in the cursor declaration. These parameters allow you to pass arguments into the cursor. The RETURN return_specification is also an optional part.
- Second, you specify a valid SQL statement that returns a result set where the cursor points to.
- Third, you can indicate a list of columns that you want to update after the FOR UPDATE OF. This part is optional so you can omit it in the CURSOR declaration.

Here is an example of declaring a cursor:

```
CURSOR cur_chief
IS
SELECT first_name, last_name, department_name FROM
employees e INNER JOIN departments d ON d.manager_id = e.employee_id;
```

We retrieved data from employees and departments tables using the SELECT with the INNER JOIN clause and set the cur_chief cursor to this result set.

Opening a PL/SQL Cursor

After declaring a cursor, you can open it by using the following syntax:

```
OPEN cursor_name [ ( argument_1 [, argument_2 ...] ) ];
```

You have to specify the cursor's name cursor_name after the keyword OPEN. If the cursor was defined with a parameter list, you need to pass corresponding arguments to the cursor.

When you OPEN the cursor, PL/SQL executes the SELECT statement and identifies the active result set. Notice that the OPEN action does not actually retrieve records from the database. It happens in the

FETCH step. If the cursor was declared with the FOR UPDATE clause, PL/SQL locks all the records in the result set.

We can open the cur_chief cursor as follows:

```
OPEN cur_chief;
```

Fetching Records from PL/SQL Cursor

Once the cursor is open, you can fetch data from the cursor into a record that has the same structure as the cursor. Instead of fetching data into a record, you can also fetch data from the cursor to a list of variables.

The fetch action retrieves data and fills the record or the variable list. You can manipulate this data in memory. You can fetch the data until there is no record found in active result set.

The syntax of FETCH is as follows:

```
FETCH cursor_name INTO record or variables
```

You can test the cursor's attribute %FOUND or %NOTFOUND to check if the fetch against the cursor is succeeded. The cursor has more attributes that we will cover in the next section.

We can use PL/SQL LOOP statement together with the FETCHto loop through all records in active result set as follows:

```
LOOP
-- fetch information from cursor into record
        FETCH cur_chief INTO r_chief;
        EXIT WHEN cur_chief%NOTFOUND;
-- print department - chief
        DBMS_OUTPUT.PUT_LINE(r_chief.department_name || ' - ' ||
                r_chief.first_name || ',' ||
                r_chief.last_name);
END LOOP;
```

Closing PL/SQL Cursor

You should always close the cursor when it is no longer used. Otherwise, you will have a memory leak in your program, which is not expected.
To close a cursor, you use CLOSE statement as follows:

CLOSE cursor_name;

And here is an example of closing the cur_chief cursor:

CLOSE cur_chief;

Cursor Attributes

These are the main attributes of a PL/SQL cursor and their descriptions.

Attribute	Description
cursor_name%FOUND	returns **TRUE** if record was fetched successfully by cursor cursor_name
cursor_name%NOTFOUND	returns **TRUE** if record was not fetched successfully by cursor cursor_name
cursor_name%ROWCOUNT	returns the number of records fetched from the cursor cursor_name at the time we test %ROWCOUNT attribute
cursor_name%ISOPEN	returns **TRUE** if the cursor cursor_nameis open

In this tutorial, you've learned how to use PL/SQL Cursor to loop through a set of rows with all necessary steps that need to be done including DECLARE, OPEN, FETCH and CLOSE.

Exception Handling

An Exception is an error situation, which arises during program execution. When an error occurs exception is raised, normal execution is stopped and control transfers to exception-handling part. Exception handlers are routines written to handle the exception. The exceptions can be internally defined (system-defined or pre-defined) or User-defined exception.

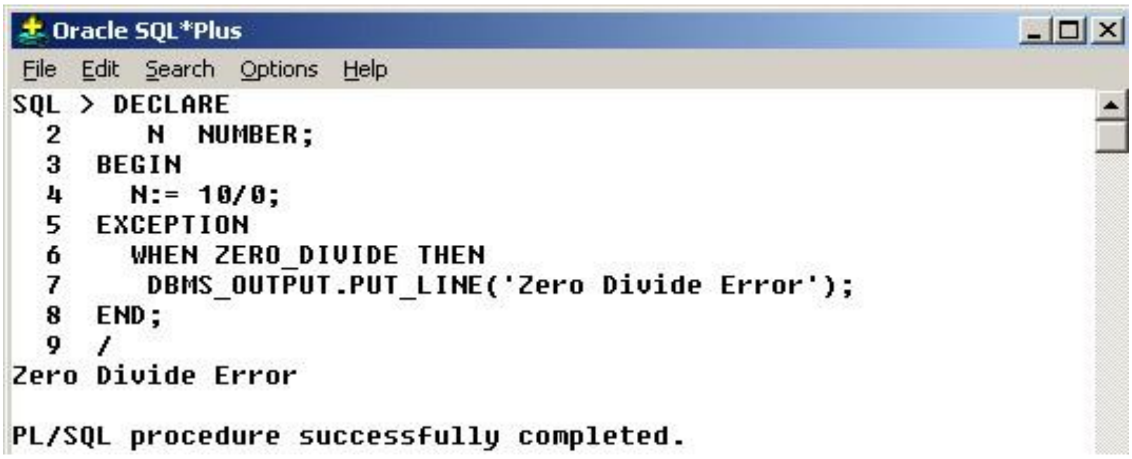
Predefined exception is raised automatically whenever there is a violation of Oracle coding rules. Predefined exceptions are those like ZERO_DIVIDE, which is raised automatically when we try to divide a number by zero. Other built-in exceptions are given below. You can handle unexpected Oracle errors using OTHERS handler. It can handle all raised exceptions that are not handled by any other handler. It must always be written as the last handler in exception block.

- **CURSOR_ALREADY_OPEN** – Raised when we try to open an already open cursor.
- **DUP_VAL_ON_INDEX** – When you try to insert a duplicate value into a unique column
- **INVALID_CURSOR** – It occurs when we try accessing an invalid cursor
- **INVALID_NUMBER** – On usage of something other than number in place of number value.
- **LOGIN_DENIED** – At the time when user login is denied
- **TOO_MANY_ROWS** – When a select query returns more than one row and the destination variable can take only single value.
- **VALUE_ERROR** – When an arithmetic, value conversion, truncation, or constraint error occurs.

Predefined exception handlers are declared globally in package STANDARD. Hence we need not have to define them rather just use them.

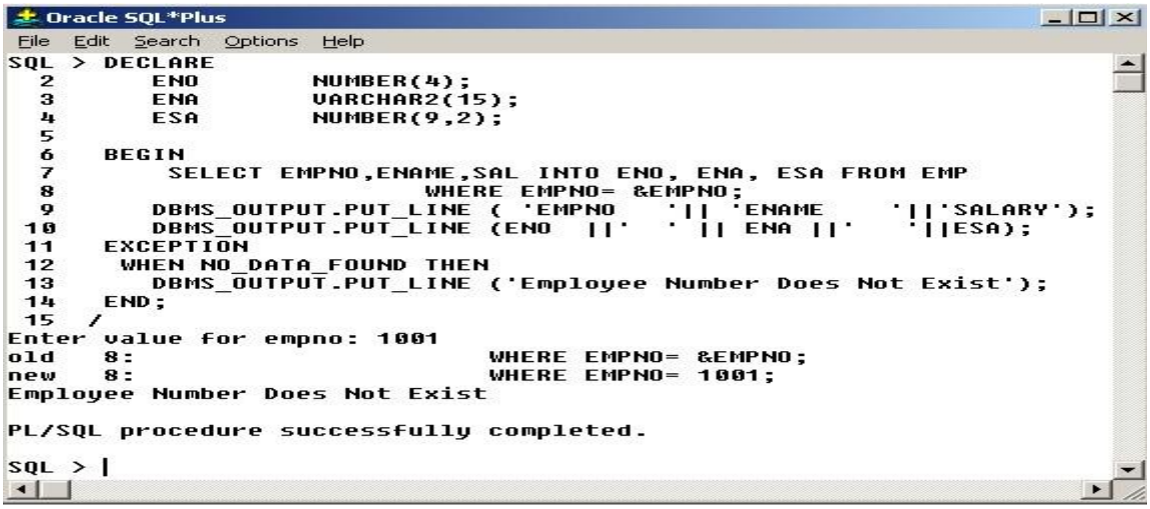
The biggest advantage of exception handling is it improves readability and reliability of the code. Errors from many statements of code can be handles with a single handler. Instead of checking for an error at every point we can just add an exception handler and if any exception is raised it is handled by that. For checking errors at a specific spot it is always better to have those statements in a separate begin – end block.

Examples 1: Following example gives the usage of **ZERO_DIVIDE** exception



```
SQL > DECLARE
       N  NUMBER;
  BEGIN
     N:= 10/0;
  EXCEPTION
     WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Zero Divide Error');
  END;
  /
Zero Divide Error

PL/SQL procedure successfully completed.
```

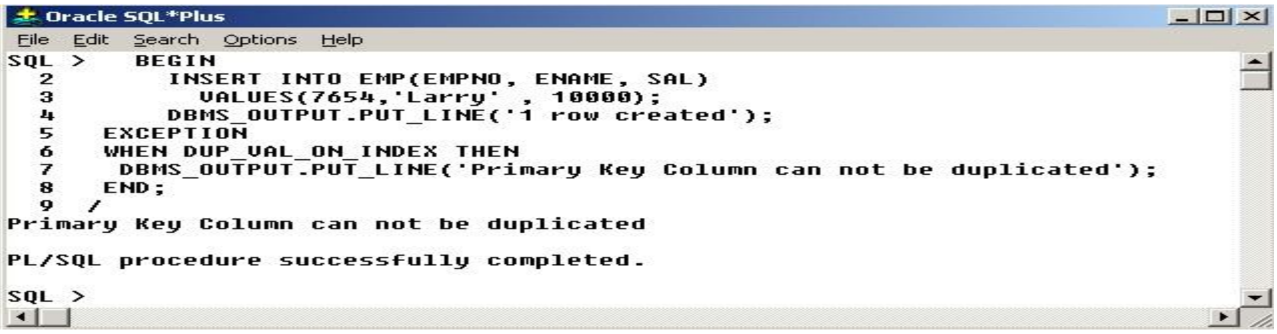
Example 2: I have explained the usage of **NO_DATA_FOUND** exception in the following



```
SQL > DECLARE
  2       ENO        NUMBER(4);
  3       ENA        VARCHAR2(15);
  4       ESA        NUMBER(9,2);
  5
  6    BEGIN
  7        SELECT EMPNO,ENAME,SAL INTO ENO, ENA, ESA FROM EMP
  8                        WHERE EMPNO= &EMPNO;
  9       DBMS_OUTPUT.PUT_LINE ( 'EMPNO   '|| 'ENAME    '||'SALARY');
 10       DBMS_OUTPUT.PUT_LINE (ENO  ||'  ' || ENA ||'   '||ESA);
 11    EXCEPTION
 12     WHEN NO_DATA_FOUND THEN
 13       DBMS_OUTPUT.PUT_LINE ('Employee Number Does Not Exist');
 14    END;
 15   /
Enter value for empno: 1001
old   8:                      WHERE EMPNO= &EMPNO;
new   8:                      WHERE EMPNO= 1001;
Employee Number Does Not Exist

PL/SQL procedure successfully completed.

SQL > |
```

The **DUP_VAL_ON_INDEX** is raised when a SQL statement tries to create a duplicate value in a column on which a primary key or unique constraints are defined.

Example 3: to demonstrate the exception **DUP_VAL_ON_INDEX.**



```
SQL >    BEGIN
  2        INSERT INTO EMP(EMPNO, ENAME, SAL)
  3          VALUES(7654,'Larry' , 10000);
  4       DBMS_OUTPUT.PUT_LINE('1 row created');
  5     EXCEPTION
  6     WHEN DUP_VAL_ON_INDEX THEN
  7      DBMS_OUTPUT.PUT_LINE('Primary Key Column can not be duplicated');
  8     END;
  9    /
Primary Key Column can not be duplicated

PL/SQL procedure successfully completed.

SQL >
```

More than one Exception can be written in a single handler as shown below.

EXCEPTION

When NO_DATA_FOUND or TOO_MANY_ROWS then

Statements;

END;

User-defined Exceptions

A User-defined exception has to be defined by the programmer. User-defined exceptions are declared in the declaration section with their type as exception. They must be raised explicitly using RAISE statement, unlike pre-defined exceptions that are raised implicitly. RAISE statement can also be used to raise internal exceptions.

Declaring Exception:

DECLARE

myexception EXCEPTION;

BEGIN

Raising Exception:

BEGIN

RAISE myexception;

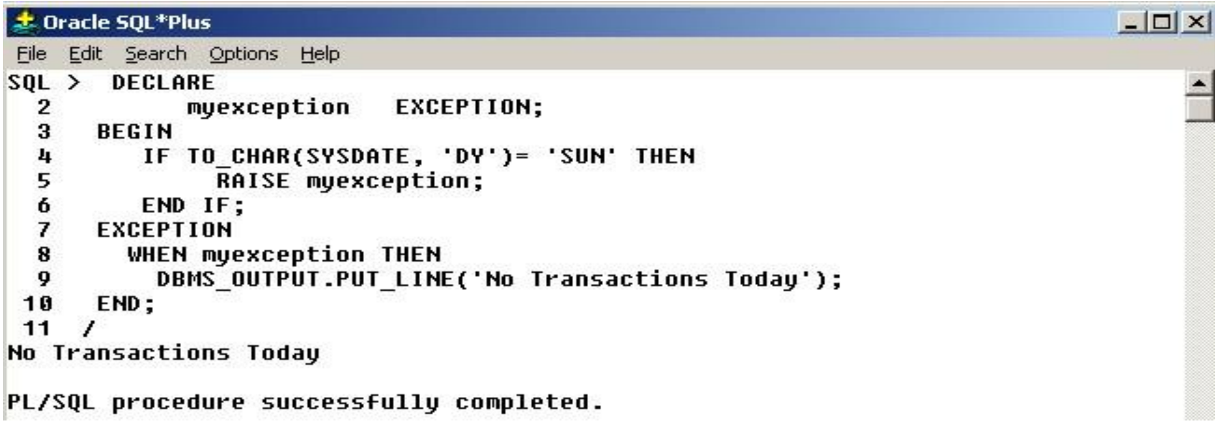
Handling Exception:

BEGIN
EXCEPTION
        WHEN myexception THEN
        Statements;
END;

Points To Ponder:

- An Exception cannot be declared twice in the same block.
- Exceptions declared in a block are considered as local to that block and global to its sub-blocks.
- An enclosing block cannot access Exceptions declared in its sub-block. Where as it possible for a sub-block to refer its enclosing Exceptions.

The following example explains the usage of User-defined Exception

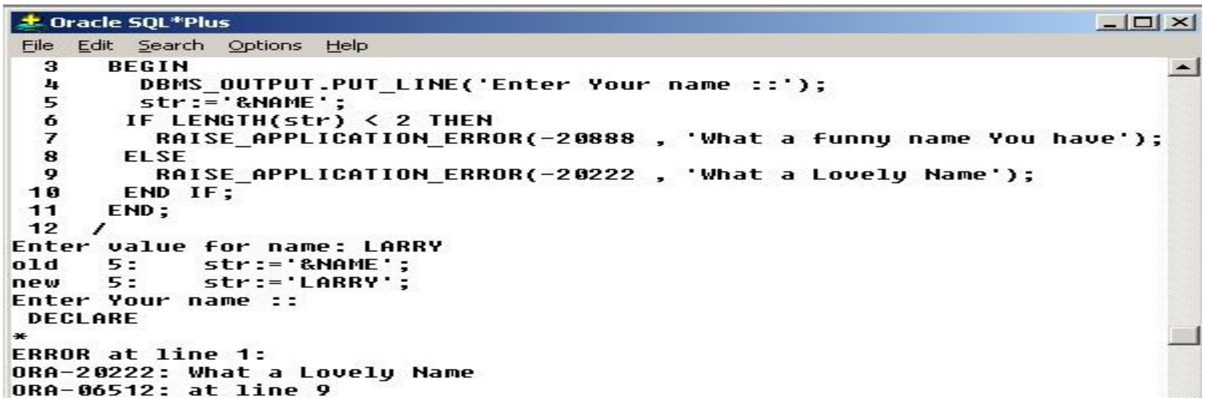


```
SQL >  DECLARE
  2         myexception   EXCEPTION;
  3   BEGIN
  4      IF TO_CHAR(SYSDATE, 'DY')= 'SUN' THEN
  5         RAISE myexception;
  6      END IF;
  7   EXCEPTION
  8     WHEN myexception THEN
  9       DBMS_OUTPUT.PUT_LINE('No Transactions Today');
 10   END;
 11  /
No Transactions Today

PL/SQL procedure successfully completed.
```

RAISE APPLICATION ERROR

To display your own error messages one can use the built-in **RAISE_APPLICATION_ERROR.** They display the error message in the same way as Oracle errors. You should use a negative number between **–20000 to –20999** for the error_number and the error message should not exceed 512 characters. The syntax to call raise_application_error is

RAISE_APPLICATION_ERROR (error_number, error_message, { TRUE | FALSE });



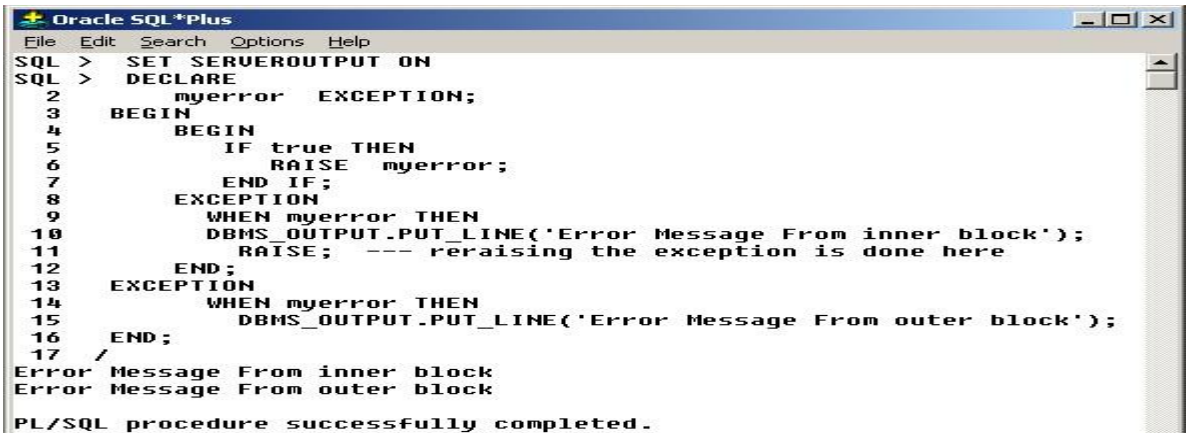
```
  3   BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Enter Your name ::');
  5     str:='&NAME';
  6    IF LENGTH(str) < 2 THEN
  7      RAISE_APPLICATION_ERROR(-20888 , 'What a funny name You have');
  8    ELSE
  9      RAISE_APPLICATION_ERROR(-20222 , 'What a Lovely Name');
 10    END IF;
 11   END;
 12  /
Enter value for name: LARRY
old   5:    str:='&NAME';
new   5:    str:='LARRY';
Enter Your name ::
 DECLARE
*
ERROR at line 1:
ORA-20222: What a Lovely Name
ORA-06512: at line 9
```

Propagation of Exceptions

When an exception is raised and corresponding handler is not found in the current block then it propagates to the enclosing blocks till a handler is found. If a handler is not found in its enclosing blocks also then it raises an unhandled exception error to the host environment. Exceptions cannot be propagated across remote procedure calls. i.e. a PL/SQL program cannot catch exceptions raised by remote subprograms

Reraising exceptions

When you want an exception to be handles in the current block as well in its enclosing block then you need to use RAISE statement without an exception name.



```
SQL >  SET SERVEROUTPUT ON
SQL >  DECLARE
           myerror  EXCEPTION;
       BEGIN
           BEGIN
              IF true THEN
                 RAISE  myerror;
              END IF;
           EXCEPTION
             WHEN myerror THEN
             DBMS_OUTPUT.PUT_LINE('Error Message From inner block');
               RAISE;  --- reraising the exception is done here
           END;
       EXCEPTION
             WHEN myerror THEN
               DBMS_OUTPUT.PUT_LINE('Error Message From outer block');
       END;
      /
Error Message From inner block
Error Message From outer block

PL/SQL procedure successfully completed.
```

IMPLICIT CURSOR	EXPLICIT CURSOR
Maintained internally By PL/SQL. Opened and closed automatically when the query is executed	Defined, Opened and closed explicitly in the program. The cursor has a name.
The cursor attributes are prefixed with SQL Ex. **SQL%FOUND**	The cursor attributes are prefixed with the cursor name Ex. C1%FOUND
The cursor attribute %ISOPEN is always FALUSE because the implicit cursor is closed immediately when the query completes.	The %ISOPEN attributes will have a valid value depending upon the status of the cursor
Only one row can be processed, the SELECT statement with the INTO clause is used.	Any number of rows can be processed. Iterative routines should be set up in the program and each row should be fetched explicitly (except for a cursor FOR loop)

UNIT 4

ADVANCED PL/SQL

PROCEDURES:

SQL procedures and functions behave very much like procedures and functions in other third – generation languages. They share many of the same properties. Collectively, procedures and functions are also known as subprograms. The syntax for the create or replace procedure statement is:

```
CREATE [OR REPLACE] PROCEDURE PROCEDURE_NAME  (argument [IN ||OUT||IN OUT ] ){IS/AS}
        <Variable> declarations;
        <costant> declarations;
BEGIN
        <pl/SQL SUBPROGRAM BODY>
EXCEPTION
        <Exception pl/SQL  Block>
END;
```

PROCEDURE HEADER

The portion of the procedure definition that comes before the IS keyword is called the procedure header. The header provides all the information a programmer needs to call that procedure, namely:

* The procedure name
* The parameter list, if any

A programmer does not need to know about the inside of the procedure in order to be able to call it properly from another program.

PROCEDURE BODY

The body of the procedure is the code required to implement the procedure consists of the declaration, execution, and exception sections of the function. Everything after the IS keyword in the procedure makes up that procedure's body. The procedure body of a procedure is a PL/SQL block with declarative, executable, and exception sections. The declarative section is located between the IS or AS keyword and the BEGIN keyword. The executable section is located between the BEGIN and EXCEPTION keywords. The exception section is located between the EXCEPTION and END keywords.

THE END LABEL

You can append the name of the procedure directly after the END keyword when you complete your procedure as shown below:

```
PROCEDURE DISPLAY_STORES (REGION IN VARCHAR2) IS
BEGIN
....
END DISPLAY_STORES;
```

This name serves as a label that explicitly links up the end of the program with its beginning. You should as a matter of habit use an END label. It is especially important to do so when you have a procedure that spans more than a single page, or is one in a series of procedures and function in a package body.

Once again, the declaration and exception sections are optional. If you have no exception handlers, you will leave off the EXCEPTION keyword and simple enter the END statement to terminate the procedure.

If you do not have any declarations, the BEGIN statement simply follows immediately after the IS keyword.

Calling a Procedure

A procedure is called as an executable PL/SQL statement. In other words, a call to a procedure must end with a semicolon (;) and be executed before and after other SQL or PL/SQL statements.

In order to change the code of a procedure, the procedure must be dropped and then re-created. Since this a common operation while the procedure is under development, the OR REPLACE keywords allow this to be done in one operation. If the procedure, use the DROP PROCEDURE command. If the procedure does not already exist, then it is simply created. If the procedure exists and the OR REPLACE keywords are not present, the CREATE statement will return an error. As with other CREATE statements, creating a procedure is a DDL operation, so an implicit COMMIT is done both before and after the procedure is created. Either the IS or AS keyword can be used – they are equivalent.

```
CREATE OR REPLACE PRCEDURE ADDNEWSTUDENT(T_FIRSTNAME STUDENTS.FIRST_NAME%TYPE, T_LASTNAME STUDENTS.LAST_NAME%TYPE, T_SCORE STUDENTS.SCORE%TYPE) AS
BEGIN
        INSERT INTO STUDENTS (ID,FIRST_NAME,LAST_NAME, SCOR)
        VALUES (STUDENT_SEQUENCE.NEXTVAL, T_FIRST_NAME, T_LAST_NAME,    T_SCOR)
END ADDNEWSTUDENT;

BEGIN
        ADDNEWSTUDENT ('ZELDA', 'ZUBIN', 'DISTINCTION');
END;
```

The AddNewStudent procedure is created first with CREATE OR REPLACE PROCEDURE statement. When a procedure is created, it is first compiled, and then stored in the database in compiled form. This compiled code can be rune later from another PL/SQL block.

- When the procedure is called, parameters can be passed. In the preceding example, the newstudent's first name, last name and score are passed to the procedure at run time. Inside the procedure, the parameter, T_FirstName will have the value 'Zelda', T_LastName will have the value 'Zubin' and T_Score will have the value 'Distinction' these literals will pass when it is called.
- A procedure call is a PL/SQL statement by itself. It is not called as part of an expression. When a procedure is called, control passes to the first executable statement inside the procedure. When the procedure finishes, control resumes at the statement following the procedure call.
- A procedure is a PL/SQL block, with a declarative section, executable section, and exception handling section. As in an anonymous block, only the executable section is required. AddNewStudent only has an executable section.

FUNCTION

A function is very similar to a procedure. Both take arguments, which can be of any mode. Both are different forms of PL/SQL blocks, with a declarative, executable, and exception section. Both can be stored in the database or declared within a block. However, a procedure call is a PL/SQL statement by itself, while a function call is called as part of an expression.

Function Syntax

The syntax for creating a stored function is very similar to the syntax for a procedure. It is

```
CREATE  [OR REPLACE] function function_name (argument [IN ||OUT||IN OUT ] )
 RETURN  < return_ datatype > {IS/AS}
        <Variable> declarations;
        <costant> declarations;
BEGIN
        <pl/SQL SUBPROGRAM BODY>
EXCEPTION
        <Exception pl/SQL  Block>
END;
```

Name

The name of the procedure comes directly after the keyword FUNCTION.

Parameters

An optional list of parameters that you define to both pass information into the procedure and send information out of the procedure, back to the calling program.

The RETURN data type

The return_datatype is the data type of the value returned by the function. This data type can be any data type supported by PL/SQL, including scalars like these:

- VARCHAR2 - BINARY_INTEGER - BOOLEAN
- NUMBER -

Calling a Function

A function is called as part of an executable PL/SQL statement, whenever an expression can be used. The following examples illustrate the different ways that the tot_sales function.

- Call tot_sales to assign a value to a local PL/SQL variable:
  sales_for_1995 := tot_sales (1504,'C');
- Use a call to tot_sales to set a default value for a variable:
  DECLARE
  SALES_FOR_1995 := NUMBER TOT_SALES(1504,'C');
- Use tot_sales directly in an expression
  IF TOT_SALES (275,'C') >1000 THEN

Function without parameters

If a function has no parameters, then the function call must be written without parameters, as the line below illustrates:

If tot _sales then …….

Notice that you cannot tell just by looking at the above line of code whether tot_sales is a function or a variable. You would, in fact, have to check the declaration section of your PL/SQL block if you really needed to know. And that's the whole point. A function returns a value, as does a variable just happens to execute some code to come up with that value, whereas a variable has that value as its very attribute, available for return.

Function Header

The portion of the function definition that comes before the IS keyword is called the function header. The header provides all the information a programmer needs to call that function, namely:

- The function name
- The parameter list, if any

- The RETURN data type

A programmer does not need to know about the inside of the function in order to be able to call properly from another program.

The header for the tot_sales function discussed above is:

```
FUNCTION TOT_SALES
        (COMPANY_ID IN COMPANY.COMPANY_ID%TYPE,
        STATUS IN ORDER.STATUS_CODE%TYPE) RETURN NUMBER
```

it consists of the module type, the name, a list of two parameters, and a RETURN data type of NUMBER. This means that the PL/SQL statement containing a call to tot_sales must be use a numeric value.

Function Body

The body of the function is the code required to implement the function. It consists of the declaration, execution, and exception sections of the function. Everything after the IS keyword in the function makes up that function's body.

Once again, the declaration and exception sections are optional. If you have no exception handlers, you will simply leave off the EXCPEPTION keyword and enter the END statement to terminate the function.

The RETURN Statement

A function must have at least one RETURN statement in its execution section of statements. It can have more than one 'Return', but only one of those statements is executed each time the function is called. The RETURN statement that is executed by the function determines the value that is returned by that function. When a RETURN statement is processed, the function terminates immediately and returns control to the calling PL/SQL block.

The RETURN clause in the header of the function is different from the RETURN statement in the execution section of the body of the function. While the RETURN clause indicates the data type of the return or result value of the function, the RETURN data type in the header, but then also include at least one RETURN statement in the function.

Modes

IN	The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter acts like a PL/SQL constant – it is considered read-only and cannot be changed. When the procedure finishes and control returns to the calling environment, the actual parameter is not changed.
OUT	Any value the actual parameter has when the procedure is called will be ignored. Inside the procedure, the formal parameter is considered write-only; it can only be assigned to and cannot be read from. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.
IN OUT	This mode is a combination of IN and OUT. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter can be read from and written to. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.

PACKAGES :

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('JJKCC');

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:

syntax :	Example:

CREATE OR REPLACE PACKAGE PACKAGENAME
AS
(PACKAGE SPECIFICATIONS)
END PACKAGENAME;
CREATE OR REPLACE PACKAGE BODY PACKAGENAME
AS
(PACKAGE BODY SPECIFICATIONS)
END PACKAGENAME;
	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.

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:

```
--Creating a package
CREATE OR REPLACE PACKAGE jack is
        procedure first (routes char);
        procedure second (route char);
END jack;

--Package body
CREATE OR REPLACE PACKAGE BODY jack AS
        -- Procedure first
        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
        PROCEDURE 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.

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.

Benefits 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 fired 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 a large applications.

Some differences are between procedure and trigger which have to be considered first. In Procedures its 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 manipulate 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 containts 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 trigger

A row triggers are 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 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 executes 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.

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.

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.] Trigger_Name
                 { BEFORE, AFTER }
Event or         {DELETE, INSERT, UPDATE [OF column ...] }
Statement        ON [schema.] Table_Name
                 [FOR EACH ROW

Restriction      [WHEN condition]]

                 DECLARE
                         variable declaration;
                         constant declaration;
                 BEGIN
Action                   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.
- **Trigger name** 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 triggers 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 these 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** 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;
```

Nested Table

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 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';
COLUMN_NAME       DATA_TYPE
DEPTNO            NUMBER
DNAME             VARCHAR2
EMP_DET           EMP_NT
```

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 data type or one of oracle's standard data types 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);
```

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
NAME NULL? TYPE
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:

```
SQL> select column_Name, Data_type from USER_TAB_COLUMNS where table_name = 'STUDENT';

COLUMN_NAME DATA_TYPE
STUDNO NUMBER
NAME VARCHAR2
MARKS MARKS_VA
```

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:

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 mayworsen 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.

IN	OUT	IN OUT
The default.	Must be specified.	Must be specified.
Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression; must be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

UNIT 5

ORACLE DATABASE STRUCTURE AND STORAGE

The RDBMS consists of the database (the information) and the instance (the embodiment of the system). The database contains the physical files that reside on the system and the logical pieces such as the database schema. These database files take various forms, as described in the following section. The instance is the method used to access the data and consists of processes and system memory.

THE DATABASE: The Oracle database has a logical layer and a physical layer. The physical layer consists of the files that reside on the disk; the components of the logical layer map the data to these physical components.

THE PHYSICAL LAYER: The physical layer of the database consists of three types of files:

- **One or more data files: -** Data files store the information contained in the database. You can have as few as one data file or as many as hundreds of data files. The information for a single table can span many data files or many tables can share a set of data files. Spreading table spaces over many data files can have a significant positive effect on performance. The number of data files that can be configured is limited by the Oracle parameter MAXDATAFILES.
- **Two or more redo log files: -** Redo log files hold information used for recovery in the event of a system failure. Redo log files, known as the redo log, store a log of all changes made to the database. This information is used in the event of a system failure to reapply changes that have been made and committed but that might not have been made to the data files. The redo log files must perform well and be protected against hardware failures (through software or hardware fault tolerance). If redo log information is lost, you cannot recover the system.
- **One or more control files**: - Control files contain information used to start an instance, such as the location of data files and redo log files; Oracle needs this information to start the database instance. Control files must be protected. Oracle provides a mechanism for storing multiple copies of control files.

THE LOGICAL LAYER: The logical layer of the database consists of the following elements:

- One or more tablespaces.
- The database schema, which consists of items such as tables, clusters, indexes, views, stored procedures, database triggers, sequences, and so on.

TABLE-SPACES AND DATA-FILES

The database is divided into one or more logical pieces known as Table spaces. A table space is used to logically group data together. For example, you can create one table space for accounting and a separate tablespace for purchasing. Segmenting groups into different table spaces simplifies the administration of these groups. Tablespaces consist of one or more data files. By using more than one datafile per Tablespace, you can spread data over many different disks to distribute the I/O load and improve performance. As part of the process of creating the database, Oracle automatically creates the SYSTEM table space for you. Although a small database can fit within the SYSTEM table space, it's recommended that you create a separate table space for user data. The SYSTEM table space is where the data dictionary is kept. The data dictionary contains information about tables, indexes, clusters, and so on.

DATA-BASE SCHEMA:

The database schema is a collection of logical-structure objects, known as schema objects that define how you see the database's data. These schema objects consist of structures such as tables, clusters, indexes, views, stored procedures, database triggers, and sequences.

- **Table: -** A table, which consists of a table name and rows and columns of data, is the basic logical storage unit in the Oracle database. Columns are defined by name and data type. A table is stored within a table space; often, many tables share a table space.
- **Cluster: -** A cluster is a set of tables physically stored together as one table that shares a common column. If data in two or more tables is frequently retrieved together based on data in the common column, using a clustered table can be quite efficient. Tables can be accessed separately even though they are part of a clustered table. Because of the structure of the cluster, related data requires much less I/O overhead if accessed simultaneously.
- **Index: -** An index is a structure created to help retrieve data more quickly and efficiently (just as the index in this book allows you to find a particular section more quickly). An index is declared on a column or set of columns. Access to the table based on the value of the indexed column(s) (as in a WHERE clause) will use the index to locate the table data.
- **View: -** A view is a window into one or more tables. A view does not store any data; it presents table data. A view can be queried, updated, and deleted as a table without restriction. Views are typically used to simplify the user's perception of data access by providing limited information from one table, or a set of information from several tables transparently. Views can also be used to prevent some data from being accessed by the user or to create a join from multiple tables.
- **Stored procedure: -** A stored procedure is a predefined SQL query that is stored in the data dictionary. Stored procedures are designed to allow more efficient queries. Using stored procedures, you can reduce the amount of information that must be passed to the RDBMS and thus reduce network traffic and improve performance.
- **Database trigger: -** A database trigger is a procedure that is run automatically when an event occurs. This procedure, which is defined by the administrator or developer, triggers, or is run whenever this event occurs. This procedure could be an insert, a deletion, or even a selection of data from a table.
- **Sequence: -** The Oracle sequence generator is used to automatically generate a unique sequence of numbers in cache. By using the sequence generator you can avoid the steps necessary to create this sequence on your own such as locking the record that has the last value of the sequence, generating a new value, and then unlocking the record.

SEGMENTS, EXTENTS, AND DATA BLOCKS :

To store data oracle uses space, which is being controlled by the use of logical structures. These structures consist of the following:

Data Blocks

A block is the smallest unit of storage in an Oracle database. The database block contains header information concerning the block itself as well as the data. They are physically stored on disk. Although the data block in most systems is 2KB (2,048 bytes), you can change this size for efficiency depending on your application or operating system.

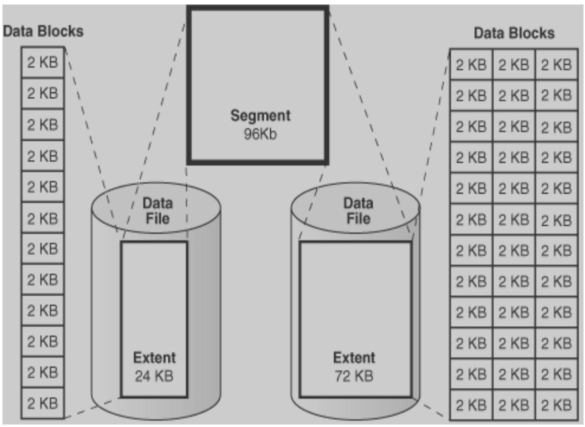
Extents

Extents consist of data blocks. Extents are the building blocks of segments; in turn, they consist of data blocks. An extent is used to minimize the amount of wasted (empty) storage. As more and more data is entered into table spaces in your database, the extents used to store that data can grow or shrink as necessary. In this manner, many table spaces can share the same storage space without pre-allocating the divisions between those table spaces. At table space-creation time, you can specify the minimum number of extents to allocate as well as the number of extents to add at a time when that allocation has been used. This arrangement gives you efficient control over the space used in your database.

Segments

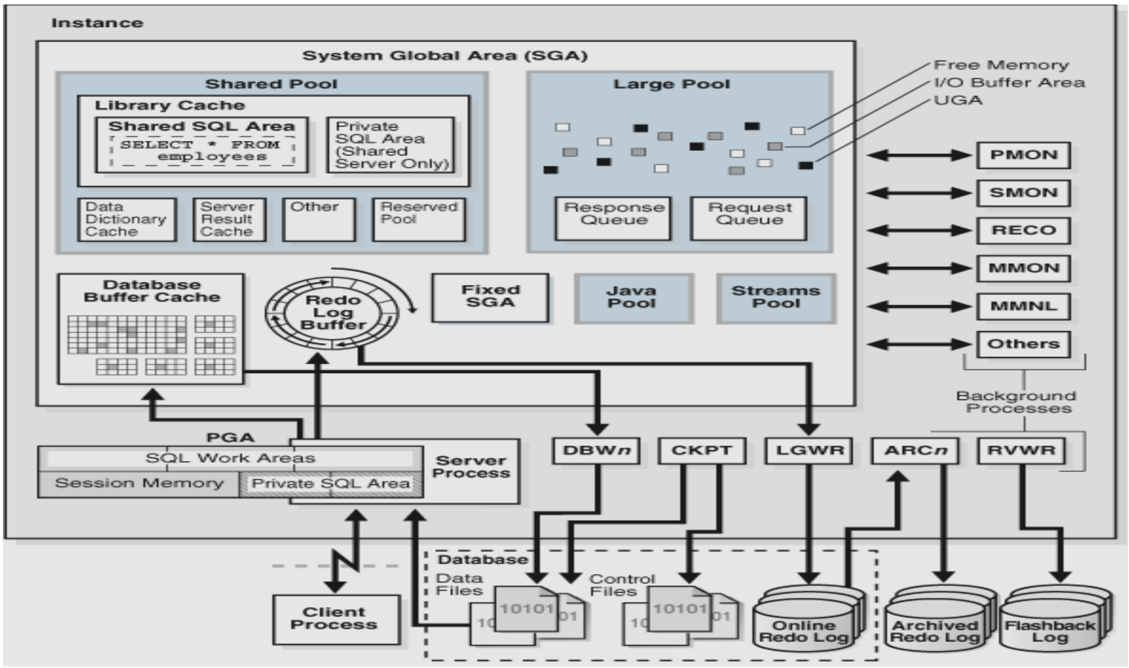
A segment is a set of extents used to store a particular type of data. An Oracle database can use four types of segments:

1. **Data segment : -** Stores user data within the database.
2. **Index segment: -** Stores indexes.
3. **Rollback segment: -** Stores rollback information used when data must be rolled back.
4. **Temporary segment: -** Created when a SQL statement needs a temporary work area; these segments are destroyed when the SQL statement is finished. These segments are used during various database operations, such as sorts.



Oracle Instance Architecture:

The Oracle instance consists of the Oracle processes and shared memory necessary to access information in the database. The instance is made up of the user processes, the Oracle background processes, and the shared memory used by these processes



The System Global Area (SGA):

The SGA is a shared memory region that Oracle uses to store data and control information for one Oracle instance. The SGA is allocated when the Oracle instance starts and de-allocated when the Oracle instance shuts down. Each Oracle instance that starts has its own SGA. The information in the SGA consists of the following elements, each of which has a fixed size and is created at instance startup:

The database buffer cache :- This stores the most recently used data blocks. These blocks can contain modified data that has not yet been written to disk (sometimes known as dirty blocks), blocks that have not been modified, or blocks that have been written to disk since modification (sometimes known as clean blocks). Because the buffer cache keeps blocks based on a most recently used algorithm, the most active buffers stay in memory to reduce I/O and improve performance.

The redo log bufferi: - This store redoes entries, or a log of changes made to the database. The redo log buffers are written to the redo log as quickly and efficiently as possible. Remember that the redo log is used for instance recovery in the event of a system failure.

The shared pool :- This is the area of the SGA that stores shared memory structures such as shared SQL areas in the library cache and internal information in the data dictionary. The shared pool is important because an insufficient amount of memory allocated to the shared pool can cause performance degradation. The shared pool consists of the library cache and the data-dictionary cache.

The Library Cache

The library cache is used to store shared SQL. Here the parse tree and the execution plan for every unique SQL statement are cached. If multiple applications issue the same SQL statement, the shared SQL area can be accessed by each to reduce the amount of memory needed and to reduce the processing time used for parsing and execution planning.

The Data-Dictionary Cache

The data dictionary contains a set of tables and views that Oracle uses as a reference to the database. Oracle stores information here about the logical and physical structure of the database. The data dictionary contains information such as the following:

1. User information, such as user privileges .
2. Integrity constraints defined for tables in the database .
3. Names and data types of all columns in database tables .
4. Information on space allocated and used for schema objects .

The data dictionary is frequently accessed by Oracle for the parsing of SQL statements.

Process:

Process refers to the mechanism of execution and can refer to a traditional process or a thread. The Oracle RDBMS uses two types of processes: user processes and Oracle processes (also known as background processes).

User Processes

User, or client, processes are the user's connections to the RDBMS system. The user process manipulates the user's input and communicates with the Oracle server process through the Oracle program interface. The user process is also used to display the information requested by the user and, if necessary, can process this information into a more useful form.

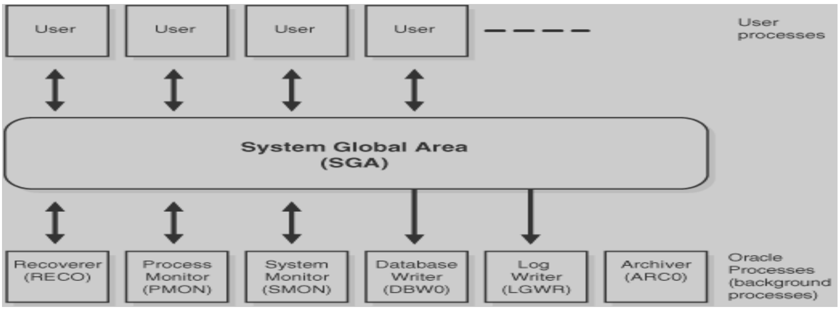
Oracle Processes

Oracle processes perform functions for users. Oracle processes can be split into two groups: server processes (which perform functions for the invoking process) and background processes (which perform functions on behalf of the entire RDBMS).

Server Processes (Shadow Processes)

Server processes, also known as shadow processes, communicate with the user and interact with Oracle to carry out the user's requests. For example, if the user process requests a piece of data not already in the SGA, the shadow process is responsible for reading the data blocks from the datafiles into the SGA. There can be a one-to-one correlation between user processes and shadow processes (as in a dedicated server configuration); although one shadow process can connect to multiple user processes (as in a multithreaded server configuration), doing so reduces the utilization of system resources.

Background Processes



Background processes are used to perform various tasks within the RDBMS system. These tasks vary from communicating with other Oracle instances and performing system maintenance and cleanup to writing dirty blocks to disk. Following are brief descriptions of the nine Oracle background processes:

DBWR (Database Writer)

DBWR is responsible for writing dirty data blocks from the database block buffers to disk. When a transaction changes data in a data block, that data block need not be immediately written to disk. Therefore, the DBWR can write this data to disk in a manner that is more efficient than writing when each transaction completes. The DBWR usually writes only when the database block buffers are needed for data to be read. Data is written in a least recently used fashion. For systems in which asynchronous I/O (AIO) is available, there should be only one DBWR process. For systems in which AIO is not available, performance can be greatly enhanced by adding more DBWR processes

LGWR (Log Writer)

The LGWR process is responsible for writing data from the log buffer to the redo log. The LGWR process is responsible for flushing to disk the contents of the redo log buffer located in the SGA. It does this when one of the following is true:

- Every three seconds
- Whenever a commit is issued by any transaction
- When the redo log buffer is one-third full or contains 1MB of buffered data

For these reasons, having an enormous (hundreds of megabytes) redo log buffer is not practical. Oracle will never be able to use it all. The logs are written to with sequential writes as compared to the scattered I/O DBWn must perform. Doing large batch writes like this is much more efficient than doing many scattered writes to various parts of a file. This is one of the main reasons for having a LGWR and redo logs in the first place. The efficiency in just writing out the changed bytes using sequential I/O outweighs the additional I/O incurred. Oracle could just write database blocks directly to disk when you commit, but that would entail a lot of scattered I/O of full blocks, and this would be significantly slower than letting LGWR write out the changes sequentially.

CKPT (Checkpoint)

The CKPT process is responsible for signaling the DBWR process to perform a checkpoint and to update all the data files and control files for the database to indicate the most recent checkpoint. A checkpoint is an event in which all modified database buffers are written to the data files by the DBWR. The CKPT process is optional. If the CKPT process is not present, the LGWR assumes these responsibilities. It simply assists with the check pointing process by updating the file headers of the data files. It used to be that CKPT was an optional process, but starting with version 8.0 of the database, it is always started, so if you do a ps on UNIX, you'll always see it there. The job of updating data files' headers with checkpoint information used to belong to the LGWR; however, as the number of files increased along with the size of a database over time, this additional task for LGWR became too much of a burden. If LGWR had to update dozens, or hundreds, or even thousands of files, there would be a good chance sessions waiting to commit these transactions would have to wait far too long. CKPT removes this responsibility from LGWR.

PMON (Process Monitor)

PMON is responsible for keeping track of database processes and cleaning up if a process prematurely dies (PMON cleans up the cache and frees resources that might still be allocated). PMON is also responsible for restarting any dispatcher processes that might have failed. For example, if your dedicated server "fails" or is killed for some reason, PMON is the process responsible for fixing (recovering or undoing work) and releasing your resources. PMON will initiate the rollback of uncommitted work, release locks, and free SGA resources allocated to the failed process. In addition to cleaning up after aborted connections, PMON is responsible for monitoring the other Oracle background processes and restarting them if necessary (and if possible). If a shared server or a dispatcher fails (crashes), PMON will step in and restart another one (after cleaning up for the failed process). PMON will watch all of the Oracle processes and either restart them or terminate the instance as appropriate.

SMON (System Monitor)

SMON performs instance recovery at instance startup. This includes cleaning temporary segments and recovering transactions that have died because of a system crash. The SMON also defragments the database by coalescing free extents within the database. The SMON background process performs all system monitoring functions on the Oracle database.

The SMON process performs a "warm start" each time that Oracle is re-started, ensuring that any in-flight transaction at the time of the last shutdown is recovered. For example, if Oracle crashed hard with a power failure, the SMON process is attached at startup time, and detects any uncompleted work, using the rollback segments to recover the transactions. In addition, SMON performs periodic cleanup of temporary segments that are no longer needed, and also perform table space operations, coalescing contiguous free extents into.

RECO (Recovery)

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.

ARCH(Archiver)

ARCH is responsible for copying the online redo log files to archival storage when they become full. ARCH is active only when the RDBMS is operated in ARCHIVELOG mode. When a system is not operated in ARCHIVELOG mode, it might not be possible to recover after a system failure. It is possible to run in NOARCHIVELOG mode under certain circumstances, but typically should operate in ARCHIVELOG mode.

These archived redo log files can then be used to perform media recovery. Whereas online redo log is used to "fix" the data files in the event of a power failure (when the instance is terminated), archived redo logs are used to "fix" data files in the event of a hard disk failure.

LCKn (Parallel Server Lock)

The lock process is responsible for managing and coordinating the locks held by the individual instances. Each instancein 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.

Each instance in a parallel server has its own set of background processes, which are identical to the background processes of a single server in exclusive mode. The DBWR, LGWR, PMON, and SMON processes are present for every instance; the optional processes, ARCH, CKPT, Dnnn and RECO, can be enabled by setting the appropriate initialization parameters.

Dnnn (Dispatcher)

When the Multithreaded Server option is used, at least one Dispatcher process is used for every communications protocol in use. The Dispatcher process is responsible for routing requests from the user processes to available shared server processes and back.

How Transactions Work

The term transaction is used to describe a logical group of work that can consist of one or many SQL statements and must end with a commit or a rollback. The following steps are executed to complete the transaction:

1. The application processes the user input and creates a connection to the server via SQL*Net.
2. The server picks up the connection request and creates a server process on behalf of the user.
3. The user executes a SQL statement or statements. In this example, the user changes the value of a row in a table.
4. The server process checks the shared pool to see whether there is a shared SQL area that has this identical SQL statement. If it finds an identical shared SQL area, the server process checks whether the user has access privileges to the data. If so, the server process uses the shared SQL area to process the request. If a shared SQL area is not found, a new shared SQL area is allocated, and the statement is parsed and executed.
5. The server process finds the data in the SGA (if it is present there) or reads the data from the data file into the SGA.

6. The server process modifies the data in the SGA. Remember that the server processes can read only from the datafiles. At some later time, the DBWR process writes the modified blocks to permanent storage.
7. The user executes either the COMMIT or ROLLBACK statement. A COMMIT will finalize the transaction, a ROLLBACK will undo the changes. If the transaction is being committed, the LGWR process immediately records the transaction in the redo log file.
8. If the transaction is successful, a completion code is returned across the network to the client process. If a failure has occurred, an error message is returned.
9. While transactions occur, the Oracle background processes do their jobs, keeping the system running smoothly. While this process occurs, hundreds of other users might be performing similar tasks. Oracle's job is to keep the system in a consistent state, to manage contention and locking, and to perform at the necessary rate.

This overview is intended to give you an understanding of the complexity and amount of interaction involved in the Oracle RDBMS. As you look in detail at the tuning of the server processes and applications later in this book, you can use this overview as a reference to the basics of how the Oracle RDBMS operates. Because of the differences in operating systems, minor variances in different environments will be discussed individually.

Initialization Parameter

The initialization parameter file is used to establish specific database features each time an Oracle 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.

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.

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 redoes 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.

TABLESPACE

Table spaces are used to store database objects that take up space on disks. These objects are called segments. Examples of segments include tables, indexes, and clusters, undo segments, and materialized views. Database objects that do not take up space on disks are stored in the data dictionary. Examples of these objects include triggers, functions, stored procedures, and database links.

Table spaces are also used to store segments in a logical manner. You could store every database segment in one table space. This would be similar to placing all of your workstation's files in one giant folder. Table spaces let you better manage your database segments. When segments are in multiple table spaces, you have more flexibility. For instance, you can:
Separate a schema's segments from other schema segments.

- Balance I/O load among different physical disk devices.
- Separate the data dictionary; undo segments, temporary segments, and application segments from interfering with each other.

- Make certain segments READ ONLY by placing them in a READ ONLY table space. This protects from unwanted changes to data.
- Backup certain segments by backing up just the table space that contains those segments.
- Limit the amount of data a schema can store with table space quotas.

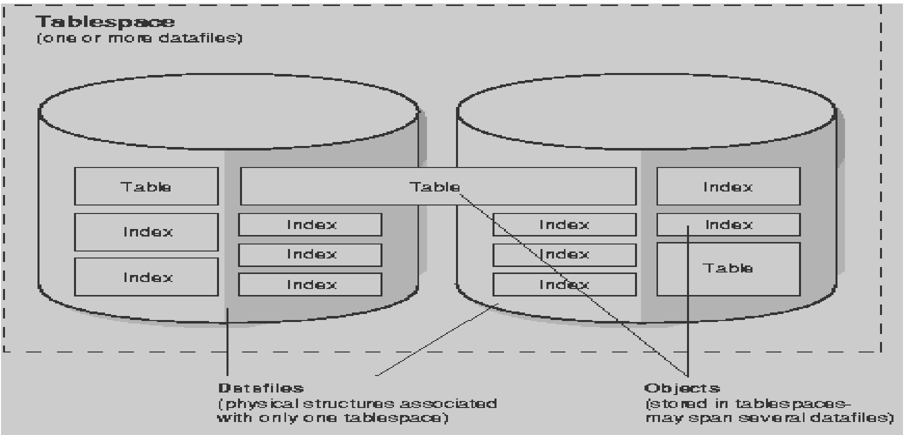


Table space is a logical storage unit in database. It a bridge between the datafiles and oracle instance.

Hierarchy of table space structure.

- Table space can belonging to only database at a time
- Table space can have more than one data files

Creating tablespace Tablespace clauses

```
CREATE [UNDO] TABLESPACE tablespace
[DATAFILE datafile_tempfile_spec [, datafile_tempfile_spec]... ]
[{ MINIMUM EXTENT integer [ K | M ]
 | BLOCKSIZE integer [K]
 | logging_clause
 | FORCE LOGGING
 | DEFAULT [data_segment_compression] storage_clause
 | { ONLINE | OFFLINE }
 | { PERMANENT | TEMPORARY }
 | extent_management_clause
 | segment_management_clause
 }
 [ MINIMUM EXTENT integer [ K | M ]
 | BLOCKSIZE integer [K]
 | logging_clause
 | FORCE LOGGING
 | DEFAULT [data_segment_compression] storage_clause
 | { ONLINE | OFFLINE }
 | { PERMANENT | TEMPORARY }
 | extent_management_clause
 | segment_management_clause
 ]... ] ;
```

- **Blocksize** – set nonstandard block size for the tablespace. You must set db_cache_size and at least one db_nk_cache_size.
- **Logging (d) nologging** – all tables and indexes within the tablespace have all changes written to redo.
- **Default** – specify default storage for all objects created in the tablespace creation. (DMT only) (Initial, next, pctincrease, minextends, maxenteds) Online offline (d) – tablespace availability
- **Permanent temporary** – tablespace can be used to hold permanent object or temporary object (sorting)
- **Autoextend –** this enable automatic extension of datafile
- **Next** – this increment of next disk space to be allocated auto when more extents are required.
- **Maxsize** - maximum disk space allowed for auto extension of datafile.
- **Unlimited** – unlimited disk space allowed for auto extension of datafile.
- **Extent management** – how the extent of tablespace are managed (LMT or DMT)
- **Segment management** – track the used and free space in the segment in the tablespace using free lists or bitmap (LMT only) . Pctfree , Pctused, Freelist, Freelist group
- **Autoallocate (d)** – tablespace is system managed, user cannot specify an extent size
  Uniform – tablespace is managed with uniform extents of size 32m or bytes. By default 1mb in size.
- **Reuse –** reuse the existing file.
- **Resize –** enlarging the tablespace size.

Types of Tablespace

1. **System table space**

- Created with the database
- Contain data-dictionary and stored program units

2. **Non-system table space**

- Undo
- Temporary
- User data and easy to manage by dba.

Managing Space in Table spaces.

When oracle allocates space to a segment (table or index), a group of contiguous free blocks (extent), is added to the segment. Metadata regarding the extent allocation and unallocated extents are either stored in the data dictionary are called DMT or stored within the tablespace header are called LMT.

Locally Managed Table space – LMT

A tablespace that manages its own extents maintains a bitmap in each data file to keep track of the free or used status of blocks in that data file. Each bit in the bitmap corresponds to a group of blocks. When an extent is allocated or freed for reuse, Oracle changes the bitmap values to show the new status of the blocks. These changes do not generate rollback information because they do not update tables (like sys.uet$, sys.fet$) in the data dictionary

```
Create tablespace tsb_lmt datafile ‘file spec ‘size 100m
extent management local uniform size 100k ;
```

Advantage

1. Avoid recursive space management. It won't consume or release space in the data-dictionary table.
2. Avoid contention on data-dictionary table.
3. Extents automatically tracks adjacent free space, by eliminating the need of coalesce free extents.
4. Do not generate undo info bcos don't update tables in the data-dictionary.

Autoallocate

Create tablespace tbs_aa datafile 'filespec' size 100m extent management local autoallocate;

UniformSize

Create tablespace tbs_uni datafile 'filespec' size 100m extent management local uniform size 32k;

The SYSTEM Tablespace

Every Oracle database contains a tablespace named SYSTEM, which Oracle creates automatically when the database is created. The SYSTEM tablespace is always online when the database is open.

To take advantage of the benefits of locally managed tablespaces, you can create a locally managed SYSTEM tablespace, or you can migrate an existing dictionary managed SYSTEM tablespace to a locally managed format.

In a database with a locally managed SYSTEM tablespace, dictionary tablespaces cannot be created. It is possible to plug in a dictionary managed tablespace using the transportable feature, but it cannot be made writable.

Undo Tablespaces

Undo tablespaces are special tablespaces used solely for storing undo information. You cannot create any other segment types (for example, tables or indexes) in undo tablespaces. Each database contains zero or more undo tablespaces. In automatic undo management mode, each Oracle instance is assigned one (and only one) undo tablespace. Undo data is managed within an undo tablespace using undo segments that are automatically created and maintained by Oracle.

When the first DML operation is run within a transaction, the transaction is bound (assigned) to an undo segment (and therefore to a transaction table) in the current undo tablespace. In rare circumstances, if the instance does not have a designated undo tablespace, the transaction binds to the system undo segment.

Each undo tablespace is composed of a set of undo files and is locally managed. Like other types of tablespaces, undo blocks are grouped in extents and the status of each extent is represented in the bitmap. At any point in time, an extent is either allocated to (and used by) a transaction table, or it is free.

Default Temporary Tablespace

When the SYSTEM tablespace is locally managed, you must define a default temporary tablespace when creating a database. A locally managed SYSTEM tablespace cannot be used for default temporary storage.

If SYSTEM is dictionary managed and if you do not define a default temporary tablespace when creating the database, then SYSTEM is still used for default temporary storage. However, you will receive a warning in ALERT.LOG saying that a default temporary tablespace is recommended and will be necessary in future releases.

Read-Only Tablespaces

The primary purpose of read-only tablespaces is to eliminate the need to perform backup and recovery of large, static portions of a database. Oracle never updates the files of a read-only tablespace, and therefore the files can reside on read-only media such as CD-ROMs or WORM drives.

Read-only tablespaces cannot be modified. To update a read-only tablespace, first make the tablespace read/write. After updating the tablespace, you can then reset it to be read-only.

Because read-only tablespaces cannot be modified, and as long as they have not been made read-write at any point, they do not need repeated backup. Also, if you need to recover your database, you do not need to recover any read-only tablespaces, because they could not have been modified.

ROLLBACK SEGMENT

Use the CREATE ROLLBACK SEGMENT statement to create a **rollback segment**, which is an object that Oracle uses to store data necessary to reverse, or undo, changes made by transactions.

The information in this section assumes that your database is running in rollback undo mode (the UNDO_MANAGEMENT initialization parameter is set to MANUAL or not set at all).

If your database is running in Automatic Undo Management mode (the UNDO_MANAGEMENT initialization parameter is set to AUTO), then user-created rollback segments are irrelevant. In this case, Oracle returns an error in response to any CREATE ROLLBACK SEGMENT or ALTER ROLLBACK SEGMENT statement.
To suppress these errors, set the UNDO_SUPPRESS_ERRORS parameter to TRUE.

Further, if your database has a locally managed SYSTEM tablespace, then you cannot create rollback segments in any dictionary-managed tablespace. Instead, you must

- Use the Automatic Undo Management feature, which uses undo tablespaces instead of rollback segments to hold undo data, or
- Create locally managed tablespaces to hold the rollback segments.

Oracle Corporation recommends that you use Automatic Undo Management.
To use objects in a tablespace other than the SYSTEM tablespace:

- If you are running the database in rollback undo mode, at least one rollback segment (other than the SYSTEM rollback segment) must be online.
- If you are running the database in Automatic Undo Management mode, at least one UNDO tablespace must be online.

Syntax:

CREATE [PUBLIC] ROLLBACK SEGMENT rollback_segment
[{ TABLESPACE tablespace | storage_clause }
[TABLESPACE tablespace | storage_clause]...];

Parameters
PUBLIC

Specify PUBLIC to indicate that the rollback segment is public and is available to any instance. If you omit this clause, the rollback segment is private and is available only to the instance naming it in its initialization parameter ROLLBACK_SEGMENTS.

rollback_segment

Specify the name of the rollback segment to be created.

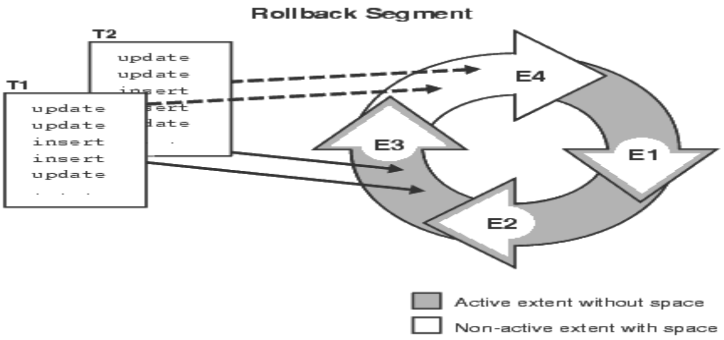
TABLESPACE

Use the TABLESPACE clause to identify the tablespace in which the rollback segment is created. If you omit this clause, Oracle creates the rollback segment in the SYSTEM tablespace.

storage_clause

The *storage_clause* lets you specify storage characteristics for the rollback segment.

CREATE ROLLBACK SEGMENT rbs_one
TABLESPACE rbs_ts STORAGE (INITIAL 10K NEXT 10K MAXEXTENTS UNLIMITED);



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 from user and to user 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.

Loader

SQL*Loader is Oracle’s utility program for loading data into an Oracle table.

Most often, SQL*Loader takes two input files – a control file and a data file – and loads the data into a single Oracle table. The data file contains data, and the control file contains information about the data -- where to load it, what to do if something goes wrong, etc. SQL*Loader has lots and lots of options which can be used to handle various types of data and levels of complexity

The basis for almost everything you do with SQL*Loader is a file known as the control file. The SQL*Loader control file is a text file into which you place a description of the data to be loaded. You also use the control file to tell SQL*Loader which database tables and columns should receive the data that you are loading. Do not confuse SQL*Loader control files with database control files. In a way, it's unfortunate that the same term is used in both cases. Database control files are binary files containing information about the physical structure of your database. They have nothing to do with SQL*Loader. SQL*Loader control files, on the other hand, are text files containing commands that control SQL*Loader's operation.

Once you have a data file to load and a control file describing the data contained in that data file, you are ready to begin the load process. You do this by invoking the

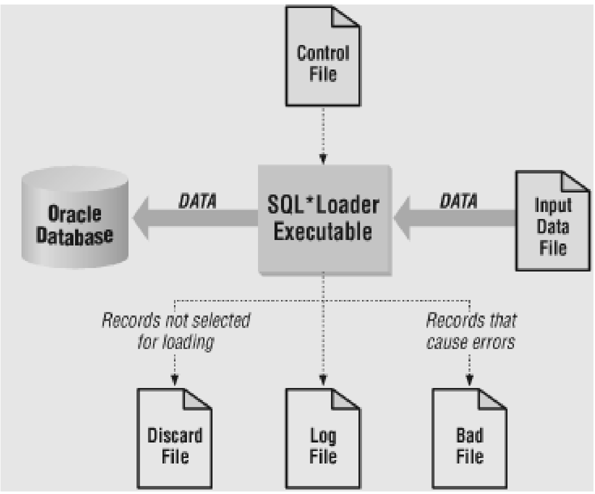
SQL*Loader executable and pointing it to the control file that you have written. SQL*Loader reads the control file to get a description of the data to be loaded. Then it reads the input file and loads the input data into the database.

SQL*Loader is a very flexible utility, and this short description doesn't begin to do it justice. The rest of this chapter provides a more detailed description of the SQL*Loader environment and a summary of SQL*Loader's many capabilities.

The SQL*Loader Environment

When we speak of the SQL*Loader environment, we are referring to the database, the SQL*Loader executable, and all the different files that you need to be concerned with when using SQL*Loader. These are shown in Figure.

The functions of the SQL*Loader executable, the database, and the input data file are rather obvious. The SQL*Loader executable does the work of reading the input file and loading the data. The input file contains the data to be loaded, and the database receives the data.



OPENING AND STOPING DATABASE

Starting the database

A database needs to be started before users can access it. A privileged user should do starting up a database. Users who have been granted the SYSDBA or SYSOPER roles can perform a startup or the database. When a database is started there important steps are executed. The first involves creation of the oracle instance in memory, followed by mounting the database where the control files are read and finally opening the databases where all the data stored in the database is accessible to the users of the databases. To get a complete understanding let us deal with each stage individually.

(1) **Instance Creation:** This is the first step in starting the database. The initialization parameter file is read and the System Global Area(SGA) is configured. The parameter file identified the name of the databases, and various sizes for the memory pools, the optional background processes and so on. During instance creation, the mandatory and optional background processes are started. The alert log file and other trace files are also started.

(2) **Mounting the Databases:** After instance creation, the database is mounted. When a database is mounted a databases administrator can perform certain maintenance or administrative tasks. For E.g. putting a database in archive log mode, renaming data files or performing a full database recovery requires the database to be mounted. During mounting, a database is associated with its previously started instance. The control files of the database is opened and read. The control file contains information about the other files of the database, their status, location and synchronizing information. This information is needed for the next stage of startup, which is opening of the database.

(3) **Opening the database:** This is the last stage in starting a database. This phase has to be performed so that users of the databases can access the data in the database. Once the control file has been read and the location of the physical files of the database identified after mounting, the file are opened and made available to the

users. The files that are opened are the online data files and the online redo log files. If any of the files are unavailable, an error will be reported and the online redo log files. If any of the files are unavailable. An error will be reported and the database will not be opened. It is at this time point that the oracle server verifies the consistency of the database in case the database was shutdown improperly the last time. It will be detected at this point and the SMON background process will perform instance recovery.

Database Startup Options:

Syntax:

STARTUP [FORCE] [RESTRIC] [PFILE= filename]
[OPEN [RECOVER] [database] [READ ONLY] | MOUNT | NOMOUNT]

STARTUP NOMOUNT :

Starts an instance without mounting a database such as during database creation or when recreating control file.

STARTUP MOUNT :

Starts an instance and mount a database without opening it.

We say

1. Add, drop or rename redo log files
2. Enable / Disable redo log archiving options
3. Performs full database recovery and
4. Rename data files.

STARTUP OPEN

Instance is started and the database is opened in normal way.

STARTUP RESTRICTED:

The database is opened but does not allow non-DBA account access and is only available to DBAs. Used to maintain database structure/ Exporting/ importing data etc.

STARTUP FORCE:

Used for instance recovery when database does not start.

SHUTTING DOWN THE DATABASE:

A database may be shutdown to make it unavailable for use. You can either perform a proper or improper shutdown. During a proper shutdown, three phases complementary to the startup are performed in the reverse order. First the database is closed; this involves performing a checkpoint on all available data files and closing the files. Next the database is dismounted. At this time, the control file is synchronized and closed. Finally the instance that was created in memory is released. The SGA no longer exists in memory and all background processed are stopped.

THE SHUTDOWN command:

Syntax :

SHUTDOWN [NORMAL | IMMEDIATE | TRANSACTIONAL | ABORT]

SHUTDOWN NORMAL:

This is the default mode for shutting down the database. The oracle server waits for all currently connected users to discount their sessions. No new connections are permitted. A checkpoint is performed on all the databases and the files are closed. When a database is shutdown is this mode, an instance recovery will not need to be done during the subsequent startup.

SHUTDOWN IMMEDIATE:

When database is shutdown in this mode, the oracle server automatically rolls back all currently action transaction. After the transactions have been rolled back the user sessions are terminated. No new connections are allowed. The database is then closed, dismounted and the instance released. No instance recovery will be performed during subsequent startup.

SHUTDOWN TRANSATIONAL:

Similar to shutdown immediate except it will wait for transaction to be committed before database closes. When the database is shutdown using this option, all currently active transactions will be allowed to complete. As soon as a user's transaction completes the user is automatically disconnected. No new user connections will be allowed. The database is then closed, dismounted and the instance released. No instance recovery will be performed during subsequent startup.

SHUTDOWN ABORT:

When the database is shutdown using this option, the instance is shutdown. This is a case of an improper shutdown. No check pointing is done. All user connections are abnormally terminated. The database is not closed or disconnected. However, the next startup will require an instance recovery to be performed by the SMON background process.

UNIT 5

DATABASE RESOURCE MANAGEMENT AND TASK SCHEDULING

Managing Automated Database Maintain Tasks

Oracle Database has automated several common maintenance tasks typically performed by database administrators. These automated maintenance tasks are performed when the system load is expected to be light. You can enable and disable individual maintenance tasks, and can configure when these tasks run and what resource allocations they are allotted.

1. Automated Maintenance Tasks

Automated maintenance tasks are tasks that are started automatically at regular intervals to perform maintenance operations on the database. An example is a task that gathers statistics on schema objects for the query optimizer.

Automated maintenance tasks run in maintenance windows, which are predefined time intervals that are intended to occur during a period of low system load. You can customize maintenance windows based on the resource usage patterns of your database, or disable certain default windows from running. You can also create your own maintenance windows.

Oracle Database has these predefined automated maintenance tasks:

- **Automatic Optimizer Statistics Collection: -** Collects optimizer statistics for all schema objects in the database for which there are no statistics or only stale statistics. The statistics gathered by this task are used by the SQL query optimizer to improve the performance of SQL execution.
- **Automatic Segment Advisor: -** Identifies segments that have space available for reclamation, and makes recommendations on how to defragment those segments.

  You can also run the Segment Advisor manually to obtain more up to the minute recommendations or to obtain recommendations on segments that the Automatic Segment Advisor did not examine for possible space reclamation.
- **Automatic SQL Tuning Advisor: -** Examines the performance of high-load SQL statements, and makes recommendations on how to tune those statements. You can configure this advisor to automatically implement SQL profile recommendations.
- **SQL Plan Management (SPM) Evolve Advisor: -** Evolves plans that have recently been added to the SQL plan baseline. The advisor simplifies plan evolution by eliminating the requirement to do it manually.

By default, all of these automated maintenance tasks are configured to run in all maintenance windows.

2. Maintenance Windows

A maintenance window is a contiguous time interval during which automated maintenance tasks are run. Maintenance windows are Oracle Scheduler windows that belong to the window group named MAINTENANCE_WINDOW_GROUP.

A Scheduler window can be a simple repeating interval (such as "between midnight and 6 a.m., every Saturday"), or a more complex interval (such as "between midnight and 6 a.m., on the last workday of every month, excluding company holidays").

When a maintenance window opens, Oracle Database creates an Oracle Scheduler job for each maintenance task that is scheduled to run in that window. Each job is assigned a job name that is generated at run time. All automated maintenance task job names begin with ORA$AT. For example, the job for the Automatic Segment Advisor might be called ORA$AT_SA_SPC_SY_26. When an automated maintenance task job finishes, it is deleted from the Oracle Scheduler job system. However, the job can still be found in the Scheduler job history.

In the case of a very long maintenance window, all automated maintenance tasks except Automatic SQL Tuning Advisor are restarted every four hours. This feature ensures that maintenance tasks are run regularly, regardless of window size.

The framework of automated maintenance tasks relies on maintenance windows being defined in the database. Table below lists the maintenance windows that are automatically defined with each new Oracle Database installation.

3. Configuring Automated Maintenance Tasks

To enable or disable specific maintenance tasks in any subset of maintenance windows, you can use the DBMS_AUTO_TASK_ADMIN PL/SQL package.

- **Enabling and Disabling Maintenance Tasks for all Maintenance Windows**

With a single operation, you can disable or enable a particular automated maintenance task for all maintenance windows.

You can disable a particular automated maintenance task for all maintenance windows with a single operation. You do so by calling the DISABLE procedure of the DBMS_AUTO_TASK_ADMIN PL/SQL package without supplying the window_name argument..

EXECUTE DBMS_AUTO_TASK_ADMIN.DISABLE;

- **Enabling and Disabling Maintenance Tasks for Specific Maintenance Windows**

By default, all maintenance tasks run in all predefined maintenance windows. You can disable a maintenance task for a specific window.

4. Configuring Maintenance Windows

You may want to adjust the predefined maintenance windows to a time suitable to your database environment or create a new maintenance window. You can customize maintenance windows using the DBMS_SCHEDULER PL/SQL package.

- **Modifying a Maintenance Window**

The DBMS_SCHEDULER PL/SQL package includes a SET_ATTRIBUTE procedure for modifying the attributes of a window.

- **Creating a New Maintenance Window**

To create a new maintenance window, you must create an Oracle Scheduler window object and then add it to the window group MAINTENANCE_WINDOW_GROUP.

You use the DBMS_SCHEDULER.CREATE_WINDOW package procedure to create the window, and the DBMS_SCHEDULER.ADD_GROUP_MEMBER procedure to add the new window to the window group.

- **Removing a Maintenance Window**

To remove an existing maintenance window, remove it from the MAINTENANCE_WINDOW_GROUP window group.

The window continues to exist but no longer runs automated maintenance tasks. Any other Oracle Scheduler jobs assigned to this window continue to run as usual.

5. Configuring Resource Allocations for Automated Maintenance Tasks

You can reduce or increase resource allocation to the automated maintenance tasks.

1. Resource Allocations for Automated Maintenance Tasks

By default, all predefined maintenance windows use the resource plan DEFAULT_MAINTENANCE_PLAN. Automated maintenance tasks run under its sub plan ORA$AUTOTASK. This sub plan divides its portion of total resource allocation equally among the maintenance tasks.

Any resource allocation that is unused by sessions in SYS_GROUP is then shared by sessions belonging to the other consumer groups and sub plans in the plan. Of that allocation, 5% goes to maintenance tasks and 20% goes to user sessions. The maximum utilization limit for ORA$AUTOTASK is 90. Therefore, even if the CPU is idle, this group/plan cannot be allocated more than 90% of the CPU resources.

To reduce or increase resource allocation to the automated maintenance tasks, you make adjustments to DEFAULT_MAINTENANCE_PLAN.

2. Changing Resource Allocations for Automated Maintenance Tasks

To change the resource allocation for automated maintenance tasks within a maintenance window, you must change the percentage of resources allocated to the sub plan ORA$AUTOTASK in the resource plan for that window. (By default, the resource plan for each predefined maintenance window is DEFAULT_MAINTENANCE_PLAN.) You must also adjust the resource allocation for one or more other sub plans or consumer groups in the window's resource plan such that the resource allocation at the top level of the plan adds up to 100%. For information on changing resource allocations

6. Automated Maintenance Tasks Reference

Oracle Database has predefined maintenance windows. It also has data dictionary views that you can query for information about automated maintenance.

1. Predefined Maintenance Windows

By default there are seven predefined maintenance windows, each one representing a day of the week.

The weekend maintenance windows, SATURDAY_WINDOW and SUNDAY_WINDOW, are longer in duration than the weekday maintenance windows. The window group MAINTENANCE_WINDOW_GROUP consists of these seven windows. The list of predefined maintenance windows is given in Table

Window Name	Description
MONDAY_WINDOW	Starts at 10 p.m. on Monday and ends at 2 a.m.
TUESDAY_WINDOW	Starts at 10 p.m. on Tuesday and ends at 2 a.m.
WEDNESDAY_WINDOW	Starts at 10 p.m. on Wednesday and ends at 2 a.m.
THURSDAY_WINDOW	Starts at 10 p.m. on Thursday and ends at 2 a.m.
FRIDAY_WINDOW	Starts at 10 p.m. on Friday and ends at 2 a.m.
SATURDAY_WINDOW	Starts at 6 a.m. on Saturday and is 20 hours long.
SUNDAY_WINDOW	Starts at 6 a.m. on Sunday and is 20 hours long.

2. Automated Maintenance Tasks Database Dictionary Views

You can query a set of data dictionary views for information about automated maintenance tasks. Below table displays information about database dictionary views for automated maintenance tasks:

View Name	Description
DBA_AUTOTASK_CLIENT_JOB	Contains information about currently running Scheduler jobs created for automated maintenance tasks. It provides information about some objects targeted by those jobs, as well as some additional statistics from previous instantiations of the same task. Some of this additional data is taken from generic Scheduler views.
DBA_AUTOTASK_CLIENT	Provides statistical data for each automated maintenance task over 7-day and 30-day periods.
DBA_AUTOTASK_JOB_HISTORY	Lists the history of automated maintenance task job runs. Jobs are added to this view after they finish executing.
DBA_AUTOTASK_WINDOW_CLIENTS	Lists the windows that belong to MAINTENANCE_WINDOW_GROUP, along with the Enabled or Disabled status for the window for each maintenance task. Primarily used by Cloud Control.
DBA_AUTOTASK_CLIENT_HISTORY	Provides per-window history of job execution counts for each automated maintenance task. This information is viewable in the Job History page of Cloud Control.

Managing resource with Oracle Database Resource Manager

Oracle Database Resource Manager (the Resource Manager) enables you to manage multiple workloads within a database that are contending for system and database resources.

1.1 What Solutions Does the Resource Manager Provide for Workload Management?

Resource Manager allows the database to have more control over how hardware resources are allocated.

When database resource allocation decisions are left to the operating system, you may encounter the following problems with workload management:

- **Excessive overhead**
  Excessive overhead results from operating system context switching between Oracle Database server processes when the number of server processes is high.
- **Inefficient scheduling**
  The operating system de-schedules database servers while they hold latches, which is inefficient.
- **Inappropriate allocation of resources**
  The operating system distributes resources equally among all active processes and cannot prioritize one task over another.
  Inability to manage database-specific resources, such as parallel execution servers and active sessions

The Resource Manager helps to overcome these problems by allowing the database more control over how hardware resources are allocated. In an environment with multiple concurrent user sessions that run jobs with differing priorities, all sessions should not be treated equally. The Resource Manager enables you to classify sessions into groups based on session attributes, and to then allocate resources to those groups in a way that optimizes hardware utilization for your application environment.

With the Resource Manager, you can:

- **Monitor resources**

  When Resource Manager is enabled, Resource Manager automatically records statistics about resource usage, and you can examine these statistics using real-time SQL monitoring and the Resource Manager dynamic performance views (the V$RSRC_* views).
- **Manage runaway sessions or calls in the following ways:**
  - By detecting when a session or call consumes more than a specified amount of CPU, physical I/O, logical I/O, or elapsed time, and then automatically either terminating the session or call, or switching to a consumer group with a lower resource allocation or a limit on the percentage of CPU that the group can use
  - A logical I/O, also known as a buffer I/O, refers to reads and writes of buffers in the buffer cache. When a requested buffer is not found in memory, the database performs a physical I/O to copy the buffer from either disk or the flash cache into memory, and then a logical I/O to read the cached buffer.
  - By recording detailed information about SQL statements that consume more than a specified amount of CPU, physical I/O, logical I/O, or elapsed time with real-time SQL monitoring
  - By using the Automatic Workload Repository (AWR) to analyze a persistent record of SQL statements that consume more than a specified amount of CPU, physical I/O, logical I/O, or elapsed time
  - By logging information about a runaway session without taking any other action related to the session
- Prevent the execution of operations that the optimizer estimates will run for a longer time than a specified limit.
- Limit the amount of time that a session can be idle. This can be further defined to mean only sessions that are blocking other sessions.
- Allow a database to use different resource plans, based on changing workload requirements. You can dynamically change the resource plan, for example, from a daytime resource plan to a nighttime resource plan, without having to shut down and restart the instance. You can also schedule a resource plan change with Oracle Scheduler.

1.2 The Elements of Resource Manager

Resource Manager includes several elements that you can manage.

1.2.1 Elements of Resource Manager

The elements of the Resource Manager include resource consumer groups, resource plans, and resource plan directives.

Element	Description
Resource consumer group	A group of sessions that are grouped together based on resource requirements. The Resource Manager allocates resources to resource consumer groups, not to individual sessions.
Resource plan	A container for directives that specify how resources are allocated to resource consumer groups. You specify how the database allocates resources by activating a specific resource plan.
Resource plan directive	Associates a resource consumer group with a particular plan and specifies how resources are to be allocated to that resource consumer group.

You use the DBMS_RESOURCE_MANAGER PL/SQL package to create and maintain these elements. The elements are stored in tables in the data dictionary. You can view information about them with data dictionary views.

1.2.2 Resource Consumer Groups

A resource consumer group (consumer group) is a collection of user sessions that are grouped together based on their processing needs.

When a session is created, it is automatically mapped to a consumer group based on mapping rules that you set up. As a database administrator (DBA), you can manually switch a session to a different consumer group. Similarly, an application can run a PL/SQL package procedure that switches its session to a particular consumer group. Because the Resource Manager allocates resources (such as CPU) only to consumer groups, when a session becomes a member of a consumer group, its resource allocation is determined by the allocation for the consumer group.

There are special consumer groups that are always present in the data dictionary. They cannot be modified or deleted. They are:

- SYS_GROUP
  This is the initial consumer group for all sessions created by user accounts SYS or SYSTEM. This initial consumer group can be overridden by session-to–consumer group mapping rules.
- OTHER_GROUPS
  This consumer group contains all sessions that have not been assigned to a consumer group. Every resource plan must contain a directive to OTHER_GROUPS.

  There can be no more than 28 resource consumer groups in any active plan.

1.2.3 Resource Plan Directives

The Resource Manager allocates resources to consumer groups according to the set of resource plan directives (directives) that belong to the currently active resource plan.

There is a parent-child relationship between a resource plan and its resource plan directives. Each directive references one consumer group, and no two directives for the currently active plan can reference the same consumer group.

A directive has several ways in which it can limit resource allocation for a consumer group. For example, it can control how much CPU the consumer group gets as a percentage of total CPU, and it can limit the total number of sessions that can be active in the consumer group.

1.2.4 Resource Plans

A resource plan is a container for directives that specify how resources are allocated to resource consumer groups.

In addition to the resource plans that are predefined for each Oracle database, you can create any number of resource plans. However, only one resource plan is active at a time. When a resource plan is active, each of its child resource plan directives controls resource allocation for a different consumer group. Each plan must include a directive that allocates resources to the consumer group named OTHER_GROUPS. OTHER_GROUPS applies to all sessions that belong to a consumer group that is not part of the currently active plan.

Types of Resources Managed by the Resource Manager

Resource plan directives specify how resources are allocated to resource consumer groups or sub-plans. Each directive can specify several different methods for allocating resources to its consumer group or sub-plan.

1. CPU

To manage CPU resources, Resource Manager allocates resources among consumer groups and redistributes CPU resources that were allocated but were not used. You can also set a limit on the amount of CPU resources that can be allocated to a particular consumer group.

1.1 Management Attributes

Management attributes enable you to specify how CPU resources are to be allocated among consumer groups and sub-plans.

Multiple levels of CPU resource allocation (up to eight levels) provide a means of prioritizing CPU usage within a plan. Consumer groups and sub-plans at level 2 get resources that were not allocated at level 1 or that were allocated at level 1 but were not completely consumed by a consumer group or sub-plan at level 1. Similarly, resource consumers at level 3 are allocated resources only when some allocation remains from levels 1 and 2. The same rules apply to levels 4 through 8. Multiple levels not only provide a way of prioritizing, but they provide a way of explicitly specifying how all primary and leftover resources are to be used.

Use the management attributes MGMT_P*n*, where *n* is an integer between 1 and 8, to specify multiple levels of CPU resource allocation. For example, use the MGMT_P1 directive attribute to specify CPU resource allocation at level 1 and MGMT_P2 directive attribute to specify resource allocation at level 2.

Use management attributes with parallel statement directive attributes, such as Degree of Parallelism Limit and Parallel Server Limit, to control parallel statement queuing. When parallel statement queuing is used, management attributes are used to determine which consumer group is allowed to issue the next parallel statement. For example, if you set the MGMT_P1 directive attribute for a consumer group to 80, that group has an 80% chance of issuing the next parallel statement. Table illustrates a simple resource plan with three levels.

Consumer Group	Level 1 CPU Allocation	Level 2 CPU Allocation	Level 3 CPU Allocation
HIGH_GROUP	80%		
LOW_GROUP		50%	
MAINT_SUBPLAN		50%	
OTHER_GROUPS			100%

High priority applications run within HIGH_GROUP, which is allocated 80% of CPU. Because HIGH_GROUP is at level one, it gets priority for CPU utilization, but only up to 80% of CPU. This leaves a remaining 20% of CPU to be shared 50-50 by LOW_GROUP and the MAINT_SUPLAN at level 2. Any unused allocation from levels 1 and 2 are then available to OTHER_GROUPS at level 3. Because OTHER_GROUPS has no sibling consumer groups or sub-plans at its level, 100% is specified.

Within a particular level, CPU allocations are not fixed. If there is not sufficient load in a particular consumer group or sub-plan, residual CPU can be allocated to remaining consumer groups or sub-plans. Thus, when there is only one level, unused allocation by any consumer group or sub-plan can be redistributed to other "sibling" consumer groups or sub-plans. If there are multiple levels, then the unused allocation is distributed to the consumer groups or sub-plans at the next level. If the last level has unused allocations, these allocations can be redistributed to all other levels in proportion to their designated allocations.

As an example of redistribution of unused allocations from one level to another, if during a particular period, HIGH_GROUP consumes only 25% of CPU, then 75% is available to be shared by LOW_GROUP and MAINT_SUBPLAN. Any unused portion of the 75%at level 2 is then made available to OTHER_GROUPS at level 3. However, if OTHER_GROUPS has no session activity at level 3, then the 75% at level 2 can be redistributed to all other consumer groups and sub-plans in the plan proportionally.

1.2 Utilization Limit

Use the UTILIZATION_LIMIT attribute to impose an absolute upper limit on CPU utilization for a resource consumer group. This absolute limit overrides any redistribution of CPU within a plan.

In the previous scenario, suppose that due to inactivity elsewhere, LOW_GROUP acquires 90% of CPU. Suppose that you do not want to allow LOW_GROUP to use 90% of the server because you do not want non-critical sessions to inundate the CPUs. The UTILIZATION_LIMIT attribute of resource plan directives can prevent this situation.

Setting the UTILIZATION_LIMIT attribute is optional. If you omit this attribute for a consumer group, there is no limit on the amount of CPU that the consumer group can use. Therefore, if all the other applications are idle, a consumer group that does not have UTILIZATION_LIMIT set can be allocated 100% of the CPU resources.

You can also use the UTILIZATION_LIMIT attribute as the sole means of limiting CPU utilization for consumer groups, without specifying level limits.

2. Exadata I/O

Management attributes enable you to specify CPU resource allocation for Exadata I/O.

3. Parallel Execution Servers

Resource Manager can manage usage of the available parallel execution servers for a database.

3.1 Degree of Parallelism Limit

You can limit the maximum degree of parallelism for any operation within a consumer group. Use the PARALLEL_DEGREE_LIMIT_P1 directive attribute to specify the degree of parallelism for a consumer group.

The degree of parallelism limit applies to one operation within a consumer group; it does not limit the total degree of parallelism across all operations within the consumer group. However, you can combine both the PARALLEL_DEGREE_LIMIT_P1 and the PARALLEL_SERVER_LIMIT directive attributes to achieve the desired control.

3.2 Parallel Server Limit

Use the PARALLEL_SERVER_LIMIT directive attribute to specify the maximum percentage of the parallel execution server pool that a particular consumer group can use. The number of parallel execution servers used by a particular consumer group is counted as the sum of the parallel execution servers used by all sessions in that consumer group.

It is possible for a single consumer group to launch enough parallel statements to use all of the available parallel execution servers. If this happens when a high-priority parallel statement from a different consumer group is run, then no parallel execution servers are available to allocate to this group. You can avoid such a scenario by limiting the number of parallel execution servers that can be used by a particular consumer group. You can also set the directive PARALLEL_STMT_CRITICAL to BYPASS_QUEUE for the high-priority consumer group so that parallel statements from the consumer group bypass the parallel statement queue.

Managing Parallel Statement Queuing Using Parallel Server Limit

The PARALLEL_SERVER_LIMIT attribute enables you to specify when parallel statements from a consumer group can be queued. Oracle Database maintains a separate parallel statement queue for each consumer group.

A parallel statement from a consumer group is not run and instead is added to the parallel statement queue of that consumer group if the following conditions are met:

- PARALLEL_DEGREE_POLICY is set to AUTO.
  Setting this initialization parameter to AUTO enables automatic degree of parallelism (Auto DOP), parallel statement queuing, and in-memory parallel execution.
- The number of active parallel execution servers across all consumer groups exceeds the PARALLEL_SERVERS_TARGET initialization parameter setting. This condition applies regardless of whether you specify PARALLEL_SERVER_LIMIT. If PARALLEL_SERVER_LIMIT is not specified, then it defaults to 100%.
- The sum of the number of active parallel execution servers for the consumer group and the degree of parallelism of the parallel statement exceeds the target number of active parallel execution servers.
  The target number of active parallel execution servers is computed as follows:
  PARALLEL_SERVER_LIMIT/100 * PARALLEL_SERVERS_TARGET

3.3 Parallel Queue Timeout

The PARALLEL_QUEUE_TIMEOUT directive attribute enables you to specify the maximum time, in seconds, that a parallel statement can wait in the parallel statement queue before it is timed out.

When you use parallel statement queuing, if the database does not have sufficient resources to execute a parallel statement, the statement is queued until the required resources become available. However, there is a chance that a parallel statement may be waiting in the parallel statement queue for longer than is desired. You can prevent such scenarios by specifying the maximum time a parallel statement can wait in the parallel statement queue.

The PARALLEL_QUEUE_TIMEOUT attribute can be set for each consumer group. This attribute is applicable even if you do not specify other management attributes (MGMT_P1, MGMT_P2, and so on) in your resource plan.

When a parallel statement is timed out, the statement execution ends with the following error message:

ORA-07454: queue timeout, n second(s), exceeded

If you want more per-workload management, then you must use the following directive attributes:

- MGMT_P*n*
  Management attributes control how a parallel statement is selected from the parallel statement queue for execution. You can prioritize the parallel statements of one consumer group over another by setting a higher value for the management attributes of that group.
- PARALLEL_SERVER_LIMIT
- PARALLEL_QUEUE_TIMEOUT
- PARALLEL_DEGREE_LIMIT_P1

4. Program Global Area (PGA)

- To manage PGA resources, Resource Manager can limit the amount of PGA memory that can be allocated to each session in a particular consumer group.

- To limit the PGA resources for each session in a consumer group, set the session_pga_limit parameter in the package procedure DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE. The value of this parameter is the maximum amount of PGA memory, in megabytes, allowed for each session in the consumer group. If a session exceeds the limit set for its consumer group, then error ORA-10260 is raised. This limit includes parallel query slaves and job queue processes.

5. Runaway Queries

- Runaway sessions and calls can adversely impact overall performance if they are not managed properly. Resource Manager can take action when a session or call consumes more than a specified amount of CPU, physical I/O, logical I/O, or elapsed time. Resource Manager can either switch the session or call to a consumer group that is allocated a small amount of CPU or terminate the session or call.

6. Active Session Pool with Queuing

- You can control the maximum number of concurrently active sessions allowed within a consumer group. This maximum defines the **active session pool**.
- An **active session** is a session that is actively processing a transaction or SQL statement. Specifically, an active session is either in a transaction, holding a user enqueue, or has an open cursor and has not been idle for over 5 seconds. An active session is considered active even if it is blocked, for example waiting for an I/O request to complete. When the active session pool is full, a session that is trying to process a call is placed into a queue. When an active session completes, the first session in the queue can then be removed from the queue and scheduled for execution. You can also specify a period after which a session in the execution queue times out, causing the call to terminate with an error.
- Active session limits should not be used for OLTP workloads. In addition, active session limits should not be used to implement connection pooling or parallel statement queuing.
- To manage parallel statements, you must use parallel statement queuing with the PARALLEL_SERVER_LIMIT attribute and management attributes (MGMT_P1, MGMT_P2, and so on).

7. Undo Pool

- You can specify an undo pool for each consumer group. An undo pool controls the total amount of undo for uncommitted transactions that can be generated by a consumer group.
- When the total undo generated by a consumer group exceeds its undo limit, the current DML statement generating the undo is terminated. No other members of the consumer group can perform further data manipulation until undo space is freed from the pool.

8. Idle Time Limit

- You can specify an amount of time that a session can be idle, after which it is terminated.
- You can also specify a more stringent idle time limit that applies to sessions that are idle and blocking other sessions.

ORACLE SCHEDULAR CONCEPTS

The Scheduler provides complex enterprise scheduling functionality, which you can use to:

- **Schedule job execution based on time or events**

  The most basic capability of a job scheduler is the ability to schedule a job to run at a particular date and time or when a particular event occurs. The Scheduler enables you to reduce your operating costs by enabling you to schedule execution of jobs. For example, consider the situation where a patch needs to be applied to a database that is in production. To minimize disruptions, this task will need to be performed during non-peak hours. This can be easily accomplished using the Scheduler. Instead of having IT personnel manually carry out this task during non-peak hours, you can instead create a job and schedule it to run at a specified time using the Scheduler.
- **Schedule job processing in a way that models your business requirements**

  The Scheduler enables limited computing resources to be allocated appropriately among competing jobs, thus aligning job processing with your business needs. This is accomplished in the following ways:
  - Jobs that share common characteristic and behavior can be grouped into larger entities called job classes. You can prioritize among the classes by controlling the resources allocated to each class. This enables you to ensure that your critical jobs have priority and have enough resources to complete. For example, if you have a critical project to load a data warehouse, then you can combine all the data warehousing jobs into one class and give priority to it over other jobs by allocating it a high percentage of the available resources.
  - The Scheduler takes prioritization of jobs one step further, by providing you the ability to change the prioritization based on a schedule. Because your definition of a critical job can change over time, the Scheduler enables you to also change the priority among your jobs over that time frame. For example, you may consider the jobs to load a data warehouse to be critical jobs during non-peak hours but not during peak hours. In such a case, you can change the priority among the classes by changing the resource allocated to each class.
  - In addition to running jobs based on a time schedule, the Scheduler enables you start jobs in response to system or business events. Your applications can detect events and then signal the Scheduler. Depending on the type of signal sent, the Scheduler starts a specific job. An example of using events to align your job processing with business needs is to prepare event-based jobs for when a transaction fails, such as someone trying to withdraw more money from a bank account than is available. In this case, you could run jobs that check for suspicious activity in this account.
- **Manage and monitor jobs**

  There are multiple states that a job undergoes from its creation to its completion. Scheduler activity is logged and information such as the status of the job and the last run time of the job can be easily tracked. This information is stored in views and can be easily queried using Enterprise Manager or a SQL query. These views provide valuable information about jobs and their execution that can help you schedule and manage your jobs better. For example, a DBA can easily track all jobs that failed for user scott.

- **Execute and manage jobs in a clustered environment**

  A cluster is a set of database instances that cooperates to perform the same task. Oracle Real Application Clusters (RAC) provides scalability and reliability without any change to your applications. The Scheduler fully supports execution of jobs in such a clustered environment. To balance the load on your system and for better performance, you can also specify the database service where you want a job to run.

Basic Scheduler Concepts

The Scheduler offers a modular approach for managing tasks within the Oracle environment. Advantages of modularity include easier management of your database environment and reusability of scheduler objects when creating new tasks that are similar to existing tasks.

In the Scheduler, most components are database objects like a table, which enables you to use normal Oracle privileges.

The basic elements of the Scheduler are:

1. **Programs**

A Scheduler program object is a collection of metadata about what will be run by the Scheduler. It includes information such as the name of the program object, program action (for example, a procedure or executable name), program type (for example, PL/SQL and Java stored procedures or PL/SQL anonymous blocks) and the number of arguments required for the program.

A program is a separate entity from a job. Jobs run at a certain time or because a certain event occurred, and invoke a certain program. Jobs can be created that point to existing program objects, which means that different jobs can use the same program and run the program at different times and with different settings. Given the right privileges, different users can thus use the same program without having to redefine it. This enables the creation of program libraries, where users can select from a list of existing programs.

Because a Scheduler program can invoke a stored procedure or other executable that requires arguments, a means is provided to store default values for those arguments as program attributes.

2. **Schedules**

A schedule specifies when and how many times a job is executed. Jobs can be scheduled for processing at a later time or immediately. For jobs to be executed at a later time, the user can specify a date and time when the job should start. For jobs that repeat over a period of time, an end date and time can be specified, which indicates when the schedule expires.

A schedule can also specify that a job be executed when a certain event occurs, such as a badge swipe or inventory dropping below a threshold.

Similar to programs, schedules are objects that can be named and saved in the database. Users can then share named schedules. For example, the end of a business quarter may be a common time frame for many jobs. Instead having to define an end-of-quarter schedule each time a new job is defined; job creators can point to a named schedule.

3. **Jobs**

A job is a user-defined task that is scheduled to run one or more times. It is a combination of what needs to be executed (the action) and when (the schedule). Users with the right privileges can create jobs either by:

- Specifying as job attributes both the action to perform (for example, an inline PL/SQL anonymous block) and the schedule by which to perform the action (for example, every day at noon, or when a certain event occurs)
- Specifying as job attributes the names of an existing program object and an existing schedule object

Like programs and schedules, jobs are objects that can be named and saved in the database.

- **Job Arguments**

  You can specify job arguments to customize a named program object. Job arguments override the default argument values in the program object, and provide values for those program arguments that have no default value. In addition, job arguments can provide argument values to an inline action (for example, a stored procedure) that the job specifies.

  A job cannot be enabled until all required program argument values are defined, either as defaults in a referenced program object, or as job arguments.

- **Job Instances**

  A job instance represents a specific run of a job. Jobs that are scheduled to run only once will have only one instance. Jobs that have a repeating schedule will have multiple instances, with each run of the job representing an instance. For example, a job that is scheduled to run on Tuesday, Oct. 8th 2002 will have one instance. A job that runs daily at noon for a week has seven instances, one for each time the job runs.

  When a job is created, only one entry is added to the Scheduler's job table to represent the job. Each time the job runs, an entry is added to the job log. Therefore, if you create a job that has a repeating schedule, you will find one entry in the job views and multiple entries in the job log. Each job instance log entry provides information about a particular run, such as the job completion status and the start and end time. Each run of the job is assigned a unique log id which is used in both the job log and job run details views.

4. Events

An event is a message sent by one application or system process to another to indicate that some action or occurrence has been detected. An event is **raised** (sent) by one application or process, and **consumed** (received) by one or more applications or processes.

There are two kinds of events in the Scheduler:

- **Events raised by the Scheduler**

  The Scheduler can raise an event to indicate state changes that occur within the Scheduler itself. For example, the Scheduler can raise an event when a job starts, when a job completes, when a job exceeds its allotted run time, and so on. The consumer of the event is an application that takes some action in response to the event.

- **Events raised by an application**

  An application can raise an event to be consumed by the Scheduler. The Scheduler reacts to the event by starting a job. You can create a schedule that references an event instead of containing date, time, and recurrence information. If a job is assigned to such a schedule (an **event schedule**), the job runs when the event is raised. You can also create a job that has no schedule assigned and that directly references an event as the means to start the job.

The Scheduler uses Oracle Streams Advanced Queuing to raise and consume events. When raising a job state change event, the Scheduler enqueues a message onto a default event queue. Applications subscribe to this queue, dequeue event messages, and take appropriate action. When raising an event to notify the Scheduler to start a job, an application enqueues a message onto a queue that was specified when setting up the job.

5. Chains

A chain is a grouping of programs that are linked together for a single, combined objective. An example of a chain might be "run program A and then program B, but only run program C if programs A and B complete successfully, otherwise run program D." A Scheduler job can point to a chain instead of pointing to a single program object.

Each position within a chain of interdependent programs is referred to as a step. Typically, after an initial set of chain steps has started, the execution of successive steps depends on the completion of one or more previous steps. Each step can point to one of the following:

- A program
- Another chain (a nested chain)
- An event

  A step that points to an event waits until the specified event is raised. If the event occurs, the step completes successfully.

Multiple steps in the chain can invoke the same program or nested chain.

In a sense, a chain resembles a decision tree, with many possible paths for selecting which steps run and when. A list of rules is used to decide which actions to perform at any particular stage. An example of a rule is "If step 2 fails or step 3 fails, wait an hour and then start step 4."

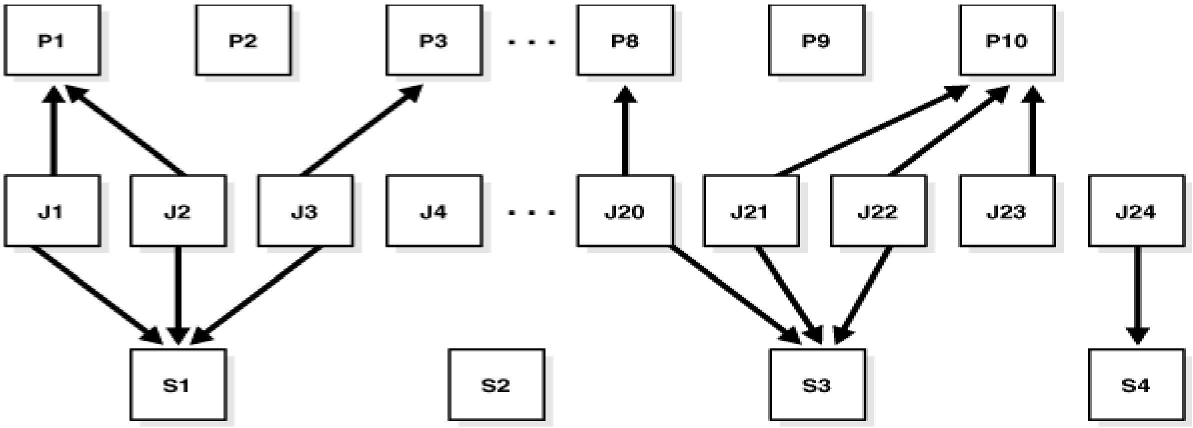
While a job pointing to a chain is running, the current state of all steps of the running chain can be monitored.

A typical situation where you might want to create a chain is to combine the different programs necessary for a successful financial transaction.

How Programs, Jobs, and Schedules are Related

To define what is executed and when, you assign relationships among programs, jobs, and schedules. Figure illustrates examples of such relationships.

Figure Relationships among Programs, Jobs, and Schedules



To understand Figure considers a situation where tables are being analyzed. In this example, P1 would be a program to analyze a table using the DBMS_STATS package. The program has an input parameter for the table name. Two jobs, J1 and J2, both point to the same program, but each supply a different table name. Additionally, schedule S1 could specify a run time of 2:00 a.m. every day. The end result would be that the two tables named in J1and J2 are analyzed daily at 2:00 a.m.

Note that J4 points to no other entity, so it is self-contained with all relevant information defined in the job itself. P2, P9 and S2 illustrate that you can leave a program or schedule unassigned if you want. You could, for example, create a program that calculates a year-end inventory and temporarily leave it unassigned to any job.

Scheduling Jobs with Oracle Scheduler

You can create, run, and manage jobs with Oracle Scheduler. You operate Oracle Scheduler by creating and managing a set of Scheduler objects. Each Scheduler object is a complete database schema object of the form [schema.]Name. Scheduler objects follow the naming rules for database objects exactly and share the SQL namespace with other database objects.

Follow SQL naming rules to name Scheduler objects in the DBMS_SCHEDULER package. By default, Scheduler object names are uppercase unless they are surrounded by double quotes. For example, when creating a job, job_name => 'my_job' is the same as job_name => 'My_Job' and job_name => 'MY_JOB', but different from job_name => '"my_job"'. These naming rules are also followed in those cases where comma-delimited lists of Scheduler object names are used within the DBMS_SCHEDULER package.

Job Tasks and Their Procedures

You use procedures in the DBMS_SCHEDULER package to administer common job tasks. Below Table illustrates common job tasks and their appropriate procedures and privileges:

Task	Procedure	Privilege Needed
Create a job	CREATE_JOB or CREATE_JOBS	CREATE JOB or CREATE ANY JOB
Alter a job	SET_ATTRIBUTE or SET_JOB_ATTRIBUTES	ALTER or CREATE ANY JOB or be the owner
Run a job	RUN_JOB	ALTER or CREATE ANY JOB or be the owner
Copy a job	COPY_JOB	ALTER or CREATE ANY JOB or be the owner
Drop a job	DROP_JOB	ALTER or CREATE ANY JOB or be the owner
Stop a job	STOP_JOB	ALTER or CREATE ANY JOB or be the owner
Disable a job	DISABLE	ALTER or CREATE ANY JOB or be the owner
Enable a job	ENABLE	ALTER or CREATE ANY JOB or be the owner

Creating Jobs

You create jobs using the DBMS_SCHEDULER package or Cloud Control You creates one or more jobs using the DBMS_SCHEDULER.CREATE_JOB or DBMS_SCHEDULER.CREATE_JOBS procedures or Cloud Control.

You use the CREATE_JOB procedure to create a single job. This procedure is overloaded to enable you to create different types of jobs that are based on different objects. You can create multiple jobs in a single transaction using the CREATE_JOBS procedure.

You must have the CREATE JOB privilege to create a job in your own schema, and the CREATEANY JOB privilege to create a job in any schema except SYS.

For each job being created, you specify a job type, an action, and a schedule. You can also optionally specify a credential name, a destination or destination group name, a job class, and other attributes. As soon as you enable a job, it is automatically run by the Scheduler at its next scheduled date and time. By default, jobs are disabled when created and must be enabled with DBMS_SCHEDULER.ENABLE to run. You can also set the enabled argument of the CREATE_JOB procedure to TRUE, in which case the job is ready to be automatically run, according to its schedule, as soon as you create it.

Some job attributes cannot be set with CREATE_JOB, and instead must be set with DBMS_SCHEDULER.SET_ATTRIBUTE. For example, to set the logging_level attribute for a job, you must call SET_ATTRIBUTE after calling CREATE_JOB.

You can create a job in another schema by specifying schema.job_name. The creator of a job is, therefore, not necessarily the job owner. The job owner is the user in whose schema the job is created. The NLS environment of the job, when it runs, is the existing environment at the time the job was created.

Altering Jobs

You alter a job by modifying its attributes. You do so using the SET_ATTRIBUTE, SET_ATTRIBUTE_NULL, or SET_JOB_ATTRIBUTES procedures in the DBMS_SCHEDULER package or Cloud Control.

All jobs can be altered, and, except for the job name, all job attributes can be changed. If there is a running instance of the job when the change is made, it is not affected by the call. The change is only seen in future runs of the job.

In general, you should not alter a job that was automatically created for you by the database. Jobs that were created by the database have the column SYSTEM set to TRUE in job views. The attributes of a job are available in the *_SCHEDULER_JOBS views.

It is valid for running jobs to alter their own job attributes. However, these changes do not take effect until the next scheduled run of the job.

Running Jobs

A job can be run in several different ways. There are three ways in which a job can be run:

- According to the job schedule: - In this case, provided that the job is enabled, the job is automatically picked up by the Scheduler job coordinator and run under the control of a job slave. The job runs as the user who is the job owner, or in the case of a local external job with a credential, as the user named in the credential. To find out whether the job succeeded, you must query the job views (*_SCHEDULER_JOBS) or the job log (*_SCHEDULER_JOB_LO Gand *_SCHEDULER_JOB_RUN_DETAILS).
- When an event occurs: - Enabled event-based jobs start when a specified event is received on an event queue or when a file watcher raises a file arrival event. Event-based jobs also run under the control of a job slave and run as the user who owns the job, or in the case of a local external job with a credential, as the user named in the credential. To find out whether the job succeeded, you must query the job views or the job log.
- By calling DBMS_SCHEDULER.RUN_JOB :- You can use the RUN_JOB procedure to test a job or to run it outside of its specified schedule. You can run the job asynchronously, which is similar to the previous two methods of running a job, or synchronously, in which the job runs in the session that called RUN_JOB, and as the user logged in to that session. The use_current_session argument of RUN_JOB determines whether a job runs synchronously or asynchronously.
  RUN_JOB accepts a comma-delimited list of job names.

Stopping Jobs

You stop one or more running jobs using the STOP_JOB procedure in the DBMS_SCHEDULER package or Cloud Control.

STOP_JOB accepts a comma-delimited list of jobs, job classes, and job destination IDs. A **job destination ID** is a number, assigned by the Scheduler, which represents a unique combination of a job, a credential, and a destination. It serves as a convenient method for identifying a particular child job of a multiple-destination job and for stopping just that child. You obtain the job destination ID for a child job from the *_SCHEDULER_JOB_DESTS views.

If a job class is supplied, all running jobs in the job class are stopped. For example, the following statement stops job job1, all jobs in the job class dw_jobs, and two child jobs of a multiple-destination job:

```
BEGIN
 DBMS_SCHEDULER.STOP_JOB('job1, sys.dw_jobs, 984, 1223');
END;
```

All instances of the designated jobs are stopped. After stopping a job, the state of a one-time job is set to STOPPED, and the state of a repeating job is set to SCHEDULED (because the next run of the job is scheduled). In addition, an entry is made in the job log with OPERATION set to 'STOPPED', and ADDITIONAL_INFO set to 'REASON="Stop job called by user: *username*"'.

By default, the Scheduler tries to gracefully stop a job using an interrupt mechanism. This method gives control back to the slave process, which can collect statistics of the job run. If the force option is set to TRUE, the job is abruptly terminated and certain run-time statistics might not be available for the job run.

Stopping a job that is running a chain automatically stops all running steps (by calling STOP_JOB with the force option set to TRUE on each step).

You can use the commit_semantics argument of STOP_JOB to control the outcome if multiple jobs are specified and errors occur when trying to stop one or more jobs. If you set this argument to ABSORB_ERRORS, the procedure may be able to continue after encountering an error and attempt to stop the remaining jobs. If the procedure indicates that errors occurred, you can query the view SCHEDULER_BATCH_ERRORS to determine the nature of the errors.

Stopping External Jobs

The Scheduler offers implementers of external jobs a mechanism to gracefully clean up after their external jobs when STOP_JOB is called with force set to FALSE.

The mechanism described in this section applies only to remote external jobs on the UNIX and Linux platforms.

On UNIX and Linux, a SIGTERM signal is sent to the process launched by the Scheduler. The implementor of the external job is expected to trap the SIGTERM in an interrupt handler, clean up whatever work the job has done, and exit.

On Windows, STOP_JOB with force set to FALSE is supported. The process launched by the Scheduler is a console process. To stop it, the Scheduler sends a CTRL+BREAK to the process. The CTRL+BREAK can be handled by registering a handler with the SetConsoleCtrlHandler() routine.

Stopping a Chain Job

If a job that points to a running chain is stopped, then all steps of the chain that are running are stopped.

Dropping Jobs

You drop one or more jobs using the DROP_JOB procedure in the DBMS_SCHEDULER package or Cloud Control.

DROP_JOB accepts a comma-delimited list of jobs and job classes. If a job class is supplied, all jobs in the job class are dropped, although the job class itself is not dropped. You cannot use job destination IDs with DROP_JOB to drop the child of a multiple - destination job.

The following statement drops jobs job1 and job3, and all jobs in job classes jobclass1 and jobclass2:

```
BEGIN
 DBMS_SCHEDULER.DROP_JOB ('job1, job3, sys.jobclass1, sys.jobclass2');
END;
```

Dropping Running Jobs

If a job is running at the time of the DROP_JOB procedure call, then attempting to drop the job fails. You can modify this default behavior by setting either the force or defer option.

When you set the force option to TRUE, the Scheduler first attempts to stop the running job by using an interrupt mechanism, calling STOP_JOB with the force option set to FALSE. If the job stops successfully, it is then dropped. Alternatively, you can first call STOP_JOB to stop the job and then call DROP_JOB. If STOP_JOB fails, you can call STOP_JOB with the force option, provided you have the MANAGE SCHEDULER privilege. You can then drop the job. By default, force is set to FALSE for both the STOP_JOB and DROP_JOB procedures.

When you set the defer option to TRUE, the running job is allowed to complete and then dropped. The force and defer options are mutually exclusive; setting both results in an error.

Dropping Multiple Jobs

When you specify multiple jobs to drop, the commit_semantics argument of the DBMS_SCHEDULER.DROP_JOB procedure determines the outcome if an error occurs on one of the jobs.

Possible values for this argument are:

- STOP_ON_FIRST_ERROR, the default:-The call returns on the first error and commits previous successful drop operations to disk.
- TRANSACTIONAL:-The call returns on the first error and rolls back previous drop operations before the error. force must be FALSE.
- ABSORB_ERRORS:-The call tries to absorb any errors, attempts to drop the rest of the jobs, and commits all the drops that were successful.

Setting commit_semantics is valid only when no job classes are included in the job_name list. When you include job classes, default commit semantics (STOP_ON_FIRST_ERROR) are in effect.

Disabling Jobs

You disable one or more jobs using the DISABLE procedure in the DBMS_SCHEDULER package or Cloud Control.

Jobs can also become disabled by other means. For example, dropping a job class disables the class jobs. Dropping either the program or the schedule that jobs point to, disables the jobs. However, disabling either the program or the schedule that jobs point to does not disable the jobs, and therefore, results in errors when the Scheduler tries to run them.

Disabling a job means that, although the metadata of the job is there, it should not run and the job coordinator does not pick up these jobs for processing. When a job is disabled, its state in the job table is changed to disabled.

When a currently running job is disabled with the force option set to FALSE, an error returns. When force is set to TRUE, the job is disabled, but the currently running instance is allowed to finish.

If commit_semantics is set to STOP_ON_FIRST_ERROR, then the call returns on the first error and the previous successful disable operations are committed to disk. If commit_semantics is set to TRANSACTIONAL and force is set to FALSE, then the call returns on the first error and rolls back the previous disable operations before the error. If commit_semantics is set to ABSORB_ERRORS, then the call tries to absorb any errors and attempts to disable the rest of the jobs and commits all the successful disable operations. If the procedure indicates that errors occurred, you can query the view SCHEDULER_BATCH_ERRORS to determine the nature of the errors.

By default, commit_semantics is set to STOP_ON_FIRST_ERROR. You can also disable several jobs in one call by providing a comma-delimited list of job names or job class names to the DISABLE procedure call. For example, the following statement combines jobs with job classes:

```
BEGIN
 DBMS_SCHEDULER.DISABLE('job1, job2, job3, sys.jobclass1, sys.jobclass2');
END;
```

Enabling Jobs

You enable one or more jobs by using the ENABLE procedure in the DBMS_SCHEDULER package or Cloud Control.

The effect of this procedure is that the job will be picked up by the job coordinator for processing. Jobs are created disabled by default, so you must enable them before they can run. When a job is enabled, a validity check is performed. If the check fails, the job is not enabled.

If you enable a disabled job, it begins to run immediately according to its schedule. Enabling a disabled job also resets the job RUN_COUNT, FAILURE_COUNT, and RETRY_COUNT attributes.

If commit_semantics is set to STOP_ON_FIRST_ERROR, then the call returns on the first error and the previous successful enable operations are committed to disk.

If commit_semantics is set to TRANSACTIONAL, then the call returns on the first error and the previous enable operations before the error are rolled back.

If commit_semantics is set to ABSORB_ERRORS, then the call tries to absorb any errors and attempts to enable the rest of the jobs and commits all the successful enable operations. If the procedure indicates that errors occurred, you can query the view SCHEDULER_BATCH_ERRORS to determine the nature of the errors.

By default, commit_semantics is set to STOP_ON_FIRST_ERROR. You can enable several jobs in one call by providing a comma-delimited list of job names or job class names to the ENABLE procedure call. For example, the following statement combines jobs with job classes:

```
BEGIN
 DBMS_SCHEDULER.ENABLE ('job1, job2, job3,
   sys.jobclass1, sys.jobclass2, sys.jobclass3');
END;
```

Copying Jobs

You copy a job using the COPY_JOB procedure in the DBMS_SCHEDULER or Cloud Control.

This call copies all the attributes of the old job to the new job (except job name). The new job is created disabled.

Administering Oracle Scheduler

You can configure, manage, monitor, and troubleshoot Oracle Scheduler.

1. Configuring Oracle Scheduler

Configuring Oracle Scheduler includes tasks such as setting privileges and preferences, and using the Oracle Scheduler agent to run remote jobs.

1.1 Setting Oracle Scheduler Privileges

You must have the SCHEDULER_ADMIN role to perform all Oracle Scheduler administration tasks. Typically, database administrators already have this role with the ADMIN option as part of the DBA role.

For example, users SYS and SYSTEM are granted the DBA role. You can grant this role to another administrator by issuing the following statement:

GRANT SCHEDULER_ADMIN TO username;

Because the SCHEDULER_ADMIN role is a powerful role allowing a grantee to execute code as any user, you should consider granting individual Scheduler system privileges instead. Object and system privileges are granted using regular SQL grant syntax, for example, if the database administrator issues the following statement:

GRANT CREATE JOB TO scott;

After this statement is executed, scott can create jobs, schedules, programs, and file watchers in his schema. As another example, the database administrator can issue the following statement:

GRANT MANAGE SCHEDULER TO adam;

After this statement is executed, adam can create, alter, or drop windows, job classes, or window groups. adam will also be able to set and retrieve Scheduler attributes and purge Scheduler logs.

Setting Chain Privileges

Scheduler chains use underlying Oracle Streams Rules Engine objects along with their associated privileges. To create a chain in their own schema, users must have the CREATE JOB privilege in addition to the Rules Engine privileges required to create rules, rule sets, and evaluation contexts in their own schema. These can be granted by issuing the following statement:

GRANT CREATE RULE, CREATE RULE SET, CREATE EVALUATION CONTEXT TO user;

To create a chain in a different schema, users must have the CREATE ANY JOB privilege in addition to the privileges required to create rules, rule sets, and evaluation contexts in schemas other than their own. These can be granted by issuing the following statement:

GRANT CREATE ANY RULE, CREATE ANY RULE SET,
CREATE ANY EVALUATION CONTEXT TO user;

Altering or dropping chains in schemas other than the users's schema require corresponding system Rules Engine privileges for rules, rule sets, and evaluation contexts.

1.2 Setting Scheduler Preferences

There are several system-wide Scheduler preferences that you can set. You set these preferences by setting Scheduler attributes with the SET_SCHEDULER_ATTRIBUTE procedure in the DBMS_SCHEDULER package.

Setting these attributes requires the MANAGE SCHEDULER privilege. The attributes are:

- **default_timezone**

  It is very important that you set this attribute. Repeating jobs and windows that use the calendaring syntax need to know which time zone to use for their repeat intervals. They normally retrieve the time zone from start_date, but if no start_date is provided (which is not uncommon), they retrieve the time zone from the default_timezone Scheduler attribute.

  The Scheduler derives the value of default_timezone from the operating system environment. If the Scheduler can find no compatible value from the operating system, it sets default_timezone to NULL.

  It is crucial that you verify that default_timezone is set properly, and if not, that you set it. To verify it, run this query:

  ```
  SELECT DBMS_SCHEDULER.STIME FROM DUAL;
   STIME
  ---------------------------------------------------------------------------
  28-FEB-12 09.04.10.308959000 PM UTC
  ```

  To ensure that daylight savings adjustments are followed, it is recommended that you set default_timezone to a region name instead of an absolute time zone offset like '-8:00'. For example, if your database resides in Miami, Florida, USA, issue the following statement:

  ```
  DBMS_SCHEDULER.SET_SCHEDULER_ATTRIBUTE('default_timezone','US
  /Eastern');
  ```

  Similarly, if your database resides in Paris, you would set this attribute to 'Europe/Warsaw'. To see a list of valid region names, run this query:

  ```
  SELECT DISTINCT TZNAME FROM V$TIMEZONE_NAMES;
  ```

  If you do not properly set default_timezone, the default time zone for repeating jobs and windows will be the absolute offset retrieved from SYSTIMESTAMP (the time zone of the operating system environment of the database), which means that repeating jobs and windows that do not have their start_date set will not follow daylight savings adjustments.

- **log_history**

  This attribute controls the number of days that log entries for both the job log and the window log are retained. It helps prevent logs from growing indiscriminately. The range of valid values is 0 through 1000000. If set to 0, no history is kept. Default value is 30. You can override this value at the job class level by setting a value for the log_history attribute of the job class.

1.3 Using the Oracle Scheduler Agent to Run Remote Jobs

The Oracle Scheduler agent can schedule and run remote jobs.

Using the Oracle Scheduler agent, the Scheduler can schedule and run two types of remote jobs:

- Remote database jobs: Remote database jobs must be run through an Oracle Scheduler agent. Oracle recommends that an agent be installed on the same host as the remote database.

If you intend to run remote database jobs, the Scheduler agent must be Oracle Database 11*g* Release 2 (11.2) or later.

- Remote external jobs: Remote external jobs run on the same host that the Scheduler agent is installed on.

  If you intend to run only remote external jobs, Oracle Database 11*g* Release 1 (11.1) of the Scheduler agent is sufficient.

You must install Scheduler agents on all hosts that remote external jobs will run on. You should install Scheduler agents on all hosts running remote databases that remote database jobs will be run on.

Each database that runs remote jobs requires an initial setup to enable secure communications between databases and remote Scheduler agents, Enabling remote jobs involves the following steps:

1.3.1 Enabling and Disabling Databases for Remote Jobs

You can set up databases for remote jobs and disable databases for remote jobs.

1.3.1.1 Setting up Databases for Remote Jobs

Before a database can run jobs using a remote Scheduler agent, the database must be properly configured, and the agent must be registered with the database. This section describes the configuration, including the required agent registration password in the database. You will later register the database

You can limit the number of Scheduler agents that can register, and you can set the password to expire after a specified duration.

Complete the following steps once for each database that creates and runs remote jobs.

1.3.1.2 Disabling Remote Jobs

You can disable remote jobs on a database by dropping the REMOTE_SCHEDULER_AGENT user.

To disable remote jobs:

- Submit the following SQL statement:

DROP USER REMOTE_SCHEDULER_AGENT CASCADE;

Registration of new scheduler agents and execution of remote jobs is disabled until you run prvtrsch.plb again.

1.3.2 Installing and Configuring the Scheduler Agent on a Remote Host

Before you can run remote jobs on a particular host, you must install and configure the Scheduler agent. After installing and configuring the Scheduler agent, you must register and start the Scheduler agent on the host The Scheduler agent must also be installed in its own Oracle home.

1.3.3 Performing Tasks with the Scheduler Agent

The Scheduler agent is a standalone program that enables you to schedule and run external and database jobs on remote hosts. You start and stop the Scheduler agent using the schagent utility on UNIX and Linux, and the OracleSchedulerExecutionAgent service on Windows.

1.3.3.1 About the schagent Utility

The executable utility schagent performs certain tasks for the agent on Windows, UNIX and Linux.

The options for schagent are indicated in Table Use schagent with the appropriate syntax and options as follows:

For example:

UNIX and Linux: AGENT_HOME/bin/schagent -status

Windows: AGENT_HOME/bin/schagent.exe -status

Table schagent options

Option	Description
-start	Starts the Scheduler Agent. *UNIX and Linux only*
-stop	Prompts the Scheduler agent to stop all the currently running jobs and then stop execution gracefully. *UNIX and Linux only*
-abort	Stops the Scheduler agent forcefully, that is, without stopping jobs first. From Oracle Database 11*g* Release 2 (11.2). *UNIX and Linux only*
-status	Returns this information about the Scheduler Agent running locally: version, uptime, total number of jobs run since the agent started, number of jobs currently running, and their descriptions.
-registerdatabase	Register the Scheduler agent with the base database or additional databases that are to run remote jobs on the agent's host computer.
-unregisterdatabase	Unregister an agent from a database.

1.3.3.2 Using the Scheduler Agent on Windows

The Windows Scheduler agent service is automatically created and started during installation. The name of the service ends with OracleSchedulerExecutionAgent.

1.3.3.3 Starting the Scheduler Agent

Starting the Scheduler agent enables the host on which it resides to run remote jobs.

To start the Scheduler agent:

- Do one of the following:
    - On UNIX and Linux, run the following command:

      AGENT_HOME/bin/schagent -start
    - On Windows, start the service whose name ends with OracleSchedulerExecutionAgent.

1.3.3.4 Stopping the Scheduler Agent

Stopping the Scheduler agent prevents the host on which it resides from running remote jobs.

To stop the Scheduler agent:

- Do one of the following:
    - On UNIX and Linux, run the schagent utility with either the -stop or –abort option as described in Table:

      AGENT_HOME/bin/schagent -stop
    - On Windows, stop the service whose name ends with Oracle Scheduler Execution Agent. This is equivalent to the -abort option.

1.3.3.5 Registering Scheduler Agents with Databases

As soon as you have finished configuring the Scheduler Agent, you can register the Agent on one or more databases that are to run remote jobs.

You can also log in later on and register the agent with additional databases.