



# **Manual & Assignment**

## **DATABASE MANAGEMENT SYSTEM LAB**

**IT 691**

**IT 3<sup>rd</sup> Year 6<sup>th</sup> Semester**

## Oracle SQL Tutorial Contents

---

- [1] Introduction to Databases
- [2] CODD'S Rules
- [3] Data types and Creating Tables
- [4] Oracle SQL SELECT Statement
- [5] Formatting Output in SQL \* Plus
- [6] UNION, INTERSECT, MINUS Operators and Sorting Query Result
- [7] Oracle SQL Functions
  - Number Functions (Math Functions)
  - Character Functions
  - Miscellaneous Functions
  - Aggregate Functions
  - Date and Time Functions
- [8] Oracle Join Queries, (Inner Join, Outer Join, Self Join)
- [9] GROUP BY Queries, SUB Queries, CUBE, ROLLUP Functions, WITH, CASE Operators
- [10] Oracle Data Manipulation Language (INSERT, UPDATE, DELETE, INSERT ALL, MERGE)
- [11] Oracle Data Definition Language (CREATE, ALTER, DROP, TRUNCATE, RENAME)
- [12] Oracle Transaction Control Language (COMMIT, ROLLBACK, SAVEPOINT)
- [13] Data Control Language (GRANT, REVOKE)
- [14] Oracle Integrity Constraints (PRIMARY KEY, NOT NULL, CHECK, FOREIGN KEY, UNIQUE)
  - DEFAULT Values
  - Dropping Constraints
  - Disabling and Enabling Constraints
  - Differing Constraints Check
  - Viewing Information about Constraints
- [15] Oracle Date Operators, Functions
- [16] Managing Oracle Views
- [17] Managing Oracle Sequences
- [18] Managing Oracle Synonyms
- [19] Managing Indexes and Clusters
- [20] Introduction to PL/SQL
- [21] What is so great about PL/SQL anyway?
- [22] PL/SQL Architecture
  - a. Overview of PL/SQL Elements Blocks
  - b. Variables and Constants
  - c. Using SQL in PL/SQL
  - d. Branching and Conditional Control
  - e. Looping Statements
  - f. GOTO
  - g. Procedures, Functions and Packages
  - h. Records
  - i. Object Types
  - j. Collections
  - k. Triggers
  - l. Error Handling

## Introduction to Databases

### DATABASE

A *database* is a collection of Data (Information). Examples of databases, which we use in our daily life, are an Attendance Register, Telephone Directory, and Muster Rule.

*Database Management System (DBMS)*: A database management system is a collection of programs written to manage a database. That is, it acts as an interface between user and database.

### RDBMS

A Database Management System based on Relational Data Model is known as *Relational Database Management System (RDBMS)*.

Relational Data Model was developed by *Dr. E.F. CODD*. He developed the relational data model by taking the concept from Relational Algebra in June - 1970.

Relational Data Model has some 12 Rules which are named after Codd as *Codd Rules*. According to Codd a package can be called as RDBMS only if it satisfies the Codd Rules.

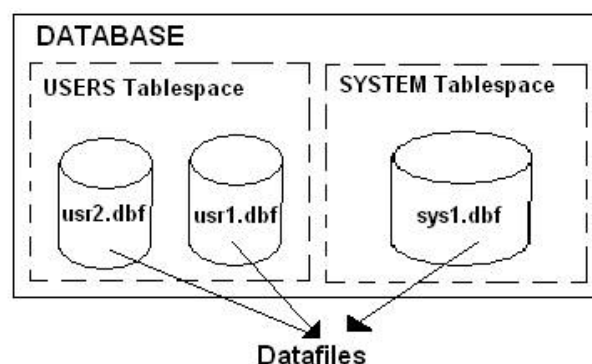
### ORACLE

*Oracle* is an *Object-Relational Database Management System*. It is the leading RDBMS vendor worldwide. Nearly half of RDBMS worldwide market is owned by Oracle.

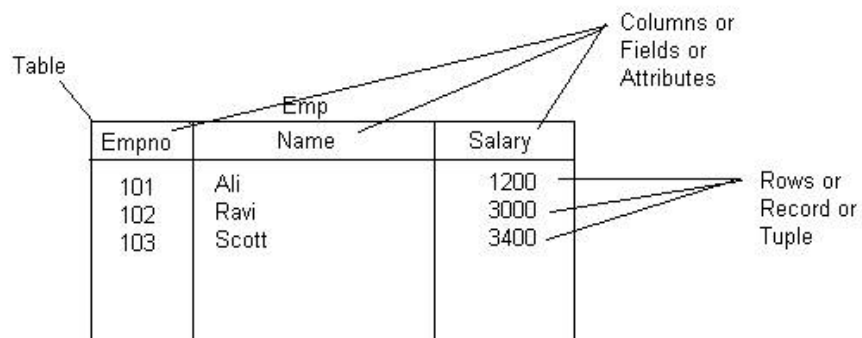
### ORACLE DATABASE

Every Oracle Database Contains Logical and Physical Structures. Logical Structures are table spaces, Schema objects, extents and segments. Physical Structures are Data files, Redo Log Files, Control File.

A database is divided into logical storage units called table spaces, which group related logical structures together. Each Table space in turn consists of one or more data files.



In relational database system all the information is stored in form of tables. A table consists of rows and columns



All the tables and other objects in Oracle are stored in table space logically, but physically they are stored in the data files associated with the table space.

Every Oracle database has a set of two or more redo log files. The set of redo log files for a database is collectively known as the database's redo log. A redo log is made up of redo entries (also called redo records).

The primary function of the redo log is to record all changes made to data. If a failure prevents modified data from being permanently written to the data files, the changes can be obtained from the redo log so work is never lost.

Every Oracle database has a control file. A control file contains the database name and locations of all datafiles and redo log files.

Every Oracle database also has a Parameter File. Parameter file contains the name of the Database, Memory Settings and Location of Control file.

## CODD'S RULES

1. **Information Rule:** *All information in a relational database including table names, column names are represented by values in tables.* This simple view of data speeds design and learning. User productivity is improved since knowledge of only one language is necessary to access all data such as description of the table and attribute definitions, integrity constraints. Action can be taken when the constraints are violated. Access to data can be restricted. All these information are also stored in tables.
2. **Guaranteed Access Rule:** *Every piece of data in a relational database can be accessed by using combination of a table name, a primary key value that identifies the row and column name which identified a cell.* User productivity is improved since there is no need to resort to using physical pointers addresses. Provides data independence. Possible to retrieve each individual

piece of data stored in a relational database by specifying the name of the table in which it is stored, the column and primary key which identified the cell in which it is stored.

3. **Systematic Treatment of Nulls Rule:** *The RDBMS handles records that have unknown or inapplicable values in a pre-defined fashion.* Also, the RDBMS distinguishes between zeros, blanks and nulls in the records and handles such values in a consistent manner that produces correct answers, comparisons and calculations. Through the set of rules for handling nulls, users can distinguish results of the queries that involve nulls, zeros and blanks. Even though the rule doesn't specify what should be done in the case of nulls it specifies that there should be a consistent policy in the treatment of nulls.
4. **Active On-line catalog based on the relational model:** *The description of a database and its contents is database tables and therefore can be queried on-line via the data manipulation language.* The database administrator's productivity is improved since the changes and additions to the catalog can be done with the same commands that are used to access any other table. All queries and reports can also be done as any other table.
5. **Comprehensive Data Sub-language Rule:** *A RDBMS may support several languages. But at least one of them should allow user to do all of the following: define tables and views, query and update the data, set integrity constraints, set authorizations and define transactions.* User productivity is improved since there is just one approach that can be used for all database operations. In a multi-user environment the user does not have to worry about the data integrity things, which will be taken care by the system. Also, only users with proper authorization will be able to access data.
6. **View Updating Rule:** *Any view that is theoretically updateable can be updated using the RDBMS.* Data consistency is ensured since the changes made in the view are transmitted to the base table and vice-versa.
7. **High-Level Insert, Update and Delete:** *The RDBMS supports insertions, updation and deletion at a table level.* The performance is improved since the commands act on a set of records rather than one record at a time.
8. **Physical Data Independence:** *The execution of adhoc requests and application programs is not affected by changes in the physical data access and storage methods.* Database administrators can make changes to the physical access and storage method which improve performance and do not require changes in the application programs or requests. Here the user specified what he wants and need not worry about how the data is obtained.
9. **Logical Data Independence:** *Logical changes in tables and views such adding/deleting columns or changing fields lengths need not necessitate modifications in the programs or in the format of adhoc requests.* The database can change and grow to reflect changes in reality without requiring the user intervention or changes in the applications. For example, adding attribute or column to the base table should not disrupt the programs or the interactive command that have no use for the new attribute.

10. **Integrity Independence:** *Like table/view definition, integrity constraints are stored in the on-line catalog and can therefore be changed without necessitating changes in the application programs.* Integrity constraints specific to a particular RDB must be definable in the relational data sub-language and storable in the catalog. At least the Entity integrity and referential integrity must be supported.
11. **Distribution Independence:** *Application programs and adhoc requests are not affected by change in the distribution of physical data.* Improved systems reliability since application programs will work even if the programs and data are moved in different sites.
12. **No subversion Rule:** *If the RDBMS has a language that accesses the information of a record at a time, this language should not be used to bypass the integrity constraints.* This is necessary for data integrity.

According to **Dr. Edgar. F. Codd**, a relational database management system must be able to manage the database entirely through its relational capabilities.

## Data types and Creating Tables

A table is the data structure that holds data in a relational database. A table is composed of rows and columns.

A table in Oracle Ver. 7.3 can have maximum 255 Columns and in Oracle Ver. 8 and above a table can have maximum 1000 columns. Number of rows in a table is unlimited in all the versions.

A table can represent a single entity that you want to track within your system. This type of a table could represent a list of the employees within your organization, or the orders placed for your company's products.

A table can also represent a relationship between two entities. This type of a table could portray the association between employees and their job skills, or the relationship of products to orders. Within the tables, foreign keys are used to represent relationships.

Although some well designed tables could represent both an entity and describe the relationship between that entity and another entity, most tables should represent either an entity or a relationship.

The following sessions explain how to create, alter, and drop tables. Some simple guidelines to follow when managing tables in your database are included.

## Designing Tables

Consider the following guidelines when designing your tables:

- Use descriptive names for tables, columns, indexes, and clusters.
- Table Names, Columns Names can contain maximum of 30 characters and they should start with an alphabet.

- Be consistent in abbreviations and in the use of singular and plural forms of table names and columns.
- Select the appropriate data type for each column.
- Arrange columns that can contain NULL Values in the last, to save storage space.

Before creating a table, you should also determine whether to use integrity constraints. Integrity constraints can be defined on the columns of a table to enforce the business rules of your database automatically.

Before creating a Table you also have to decide what type of data each column can contain. This is known as data type. Let's discuss what data types are available in Oracle.

## Data types

A data type associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one data type differently from values of another data type. For example, Oracle can add values of NUMBER data type, but not values of RAW data type.

Oracle supplies the following built-in data types:

### *Character data types*

- CHAR
- NCHAR
- VARCHAR2 and VARCHAR
- NVARCHAR2
- CLOB
- NCLOB
- LONG

### *Numeric data types*

- NUMBER

### *Time and date data types*

- DATE
- INTERVAL DAY TO SECOND
- INTERVAL YEAR TO MONTH
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

### *Binary data types*

- BLOB

- BFILE
- RAW
- LONG RAW

Another data type, ROWID, is used for values in the ROWID pseudo column, which represents the unique address of each row in a table.

The following table summarizes the information about each Oracle built-in data type.

Data type	Description	Column Length and Default
CHAR (size [BYTE   CHAR])	Fixed-length character data of length size bytes or characters.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (single-byte or multi byte) before setting size.
VARCHAR2 (size [BYTE   CHAR])	Variable-length character data, with maximum length size bytes or characters.	Variable for each row, up to 4000 bytes per row. Consider the character set (single-byte or multi byte) before setting size. A maximum size must be specified.
NCHAR (size)	Fixed-length Unicode character data of length size characters.	Fixed for every row in the table (with trailing blanks). Column size is the number of characters. (The number of bytes is 2 times this number for the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 2000 bytes per row. Default is 1 character.
NVARCHAR2 (size)	Variable-length Unicode character data of length size characters. A maximum size must be specified.	Variable for each row. Column size is the number of characters. (The number of bytes may be up to 2 times this number for a the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 4000 bytes per row. Default is 1 character.
CLOB	Single-byte character data	Up to 232 - 1 bytes, or 4 gigabytes.
NCLOB	Unicode national character set (NCHAR) data.	Up to 232 - 1 bytes, or 4 gigabytes.
LONG	Variable-length character data.	Variable for each row in the table, up to 232 - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
NUMBER (p, s)	Variable-length numeric data. Maximum precision p and/or scale s is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-RR) specified by the NLS_DATE_FORMAT parameter.
INTERVAL YEAR TO (precision)	A period of time, represented as years and months. The precision	Fixed at 5 bytes.



MONTH	value specifies the number of digits in the YEAR field of the date. The precision can be from 0 to 9, and defaults to 2 for years.	
INTERVAL DAY TO SECOND (precision)	A period of time, represented as days, hours, minutes, and seconds. The precision values specify the number of digits in the DAY and the fractional SECOND fields of the date. The precision can be from 0 to 9, and defaults to 2 for days and 6 for seconds.	Fixed at 11 bytes.
TIMESTAMP (precision)	A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.)  The precision value specifies the number of digits in the fractional second part of the SECOND date field. The precision can be from 0 to 9, and defaults to 6	Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter.
TIMESTAMP (precision) WITH TIME ZONE	A value representing a date and time, plus an associated time zone setting. The time zone can be an offset from UTC, such as '-5:0', or a region name, such as 'US/Pacific'.	Fixed at 13 bytes. The default is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter.
TIMESTAMP (precision) WITH LOCAL TIME ZONE	Similar to TIMESTAMP WITH TIME ZONE, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved.	Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter.
BLOB	Unstructured binary data	Up to 232 - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file	Up to 232 - 1 bytes, or 4 gigabytes.
RAW (size)	Variable-length raw binary data	Variable for each row in the table, up to 2000 bytes per row. A maximum size must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data	Variable for each row in the table, up to 231 - 1 bytes, or 2 gigabytes, per row. Provided for

		backward compatibility.
ROWID	Binary data representing row addresses	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.

## Representing Character Data

Use the character data types to store alphanumeric data:

CHAR and NCHAR data types store fixed-length character strings.

VARCHAR2 and NVARCHAR2 data types store variable-length character strings. (The VARCHAR data type is synonymous with the VARCHAR2 data type.)

NCHAR and NVARCHAR2 data types store Unicode character data only.

CLOB and NCLOB data types store single-byte and multi byte character strings of up to four gigabytes.

The LONG data type stores variable-length character strings containing up to two gigabytes, but with many restrictions.

This data type is provided for backward compatibility with existing applications; in general, new applications should use CLOB and NCLOB data types to store large amounts of character data, and BLOB and BFILE to store large amounts of binary data.

When deciding which data type to use for a column that will store alphanumeric data in a table, consider the following points of distinction:

To store data more efficiently, use the VARCHAR2 data type. The CHAR data type blank-pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 data type does not add any extra blanks.

For example if you define empname as char(20) then if you store names like “Sami” then name will occupy 20 bytes( 4 bytes for characters “Sami” and 16 blank spaces)

And if you define empname as varchar2 (20) then if you store names like “Sami” then oracle will take 4 bytes only.

Use the CHAR data type when you require ANSI compatibility in comparison semantics (when trailing blanks are not important in string comparisons). Use the VARCHAR2 when trailing blanks are important in string comparisons.

The CHAR and VARCHAR2 data types are and will always be fully supported. At this time, the VARCHAR data type automatically corresponds to the VARCHAR2 data type and is reserved for future use.

## Representing Numeric Data

Use the NUMBER data type to store real numbers in a fixed-point or floating-point format. Numbers using this data type are guaranteed to be portable among different Oracle platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude  $1 \times 10^{-130}$  through  $9.99 \times 10^{125}$ , as well as zero, in a NUMBER column.

You can specify that a column contains a floating-point number, for example:

distance NUMBER

Or, you can specify a precision (total number of digits) and scale (number of digits to right of decimal point):

price NUMBER (8, 2)

Although not required, specifying precision and scale helps to identify bad input values. If a precision is not specified, the column stores values as given. The following table shows examples of how data different scale factors affect storage.

<u>Input Data</u>	<u>Specified As</u>	<u>Stored As</u>
4,751,132.79	NUMBER	4751132.79
4,751,132.79	NUMBER (9)	4751133
4,751,132.79	NUMBER (9,2)	4751132.79
4,751,132.79	NUMBER (9,1)	4751132.7
4,751,132.79	NUMBER (6)	(not accepted, exceeds precision)
4,751,132.79	NUMBER (7, -2)	4,751100

## Representing Date and Time Data

Use the DATE data type to store point-in-time values (dates and times) in a table. The DATE data type stores the century, year, month, day, hours, minutes, and seconds.

Use the TIMESTAMP data type to store precise values, down to fractional seconds. For example, an application that must decide which of two events occurred first might use TIMESTAMP. An application that needs to specify the time for a job to execute might use DATE.

### *Date Format*

For input and output of dates, the standard Oracle default date format is DD-MON-RR. For example:

```
'13-NOV-1992'
```

To change this default date format on an instance-wide basis, use the NLS\_DATE\_FORMAT parameter. To change the format during a session, use the ALTER SESSION statement. To enter dates that are not in the current default date format, use the TO\_DATE function with a format mask. For example:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

Be careful using a date format like DD-MON-YY. The YY indicates the year in the current century. For example, 31-DEC-92 is December 31, 2092, not 1992 as you might expect. If you want to indicate years in any century other than the current one, use a different format mask, such as the default RR.

### *Time Format*

Time is stored in 24-hour format, HH24:MI:SS. By default, the time in a date field is 12:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO\_DATE function with a format mask indicating the time portion, as in:

```
INSERT INTO Birthdays_tab (bname, bday) VALUES ('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

## Creating Tables in Oracle

Once you have designed the table and decided about data types use the following SQL command to create a table.

For example, the following statement creates a table named Emp.

```
CREATE TABLE Emp (  
  Empno  NUMBER(5),  
  Ename  VARCHAR2(15),  
  Hiredate DATE,  
  Sal    NUMBER(7,2)  
);
```

To insert rows in the table you can use SQL INSERT command.

For example the following statement creates a row in the above table.

```
SQL> insert into emp values (101,'Sami',3400);
```

To insert rows continuously in SQL Plus you can give the following command.

```
SQL> insert into emp values (&empno,'&name',&sal);
```

These &Empno, &name and &sal are known as substitution variables. That is SQLPlus will prompt you for these values and then rewrites the statement with supplied values.

To see the rows you have inserted give the following command.

```
SQL> Select * from emp;
```

To see the structure of the table i.e. column names and their data types and widths. Give the following command.

```
SQL> desc emp
```

To see how many tables is in your schema give the following command.

```
SQL> select * from cat;
```

or

```
SQL> select * from tab;
```

## Oracle SQL SELECT Statement

Use a SELECT statement or sub query to retrieve data from one or more tables, object tables, views, object views, or materialized views

For example to retrieve all rows from emp table.

```
SQL> select empno, ename, sal from emp;
```

Or (if you want to see all the columns values

You can also give \* which means all columns)

SQL> select \* from emp;

If you want to see only employee names and their salaries then you can type the following statement

SQL> select name, sal from emp;

## Filtering Information using Where Conditions

You can filter information using where conditions like suppose you want to see only those employees whose salary is above 5000 then you can type the following query with where condition.

SQL>select \* from emp where sal > 5000;

To see those employees whose salary is less than 5000 then the query will be

SQL> select \* from emp where sal < 5000;

## Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table below lists logical conditions.

Condition	Operation	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 (sal IS NULL); SELECT * FROM emp WHERE NOT (salary BETWEEN 1000 AND 2000);
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	SELECT * FROM employees WHERE ename ='SAMI' AND sal=3000;
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	SELECT * FROM emp WHERE ename = 'SAMI' OR sal >= 1000;

## Membership Conditions

A membership condition tests for membership in a list or sub query

The following table lists the membership conditions.

Condition	Operation	Example
IN	"Equal to any member of" test. Equivalent to "= ANY".	<pre>SELECT * FROM emp WHERE deptno IN (10,20);</pre> <pre>SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept WHERE city='HYD')</pre>
NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	<pre>SELECT * FROM emp WHERE ename NOT IN ('SCOTT', 'SMITH');</pre>

## Null Conditions

A NULL condition tests for nulls.

What is null?

If a column is empty or no value has been inserted in it then it is called null. Remember 0 is not null and blank string ' ' is also not null.

The following example lists the null conditions.

Condition	Operation	Example
IS [NOT] NULL	Tests for nulls. This is the only condition that you should use to test for nulls.	<pre>SELECT ename FROM emp WHERE deptno IS NULL;</pre> <pre>SELECT * FROM emp WHERE ename IS NOT NULL;</pre>

## EXISTS Conditions

An EXISTS condition tests for existence of rows in a sub query.

The following example shows the EXISTS condition.

Condition	Operation	Example
EXISTS	TRUE if a subquery returns at least one row.	<pre>SELECT deptno FROM dept d WHERE EXISTS (SELECT * FROM emp e WHERE d.deptno = e.deptno);</pre>

## LIKE Conditions

The LIKE conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second. LIKE calculates strings using characters as defined by the input character set.

For example you want to see all employees whose name starts with S char. Then you can use LIKE condition as follows

```
SQL> select * from emp where ename like 'S%';
```

Similarly you want to see all employees whose name ends with "d"

```
SQL>select * from emp where ename like '%d';
```

You want to see all employees whose name starts with 'A' and ends with 'd' like 'Abid', 'Alfred', 'Arnold'.

```
SQL>select * from emp where ename like 'A%d';
```

You want to see those employees whose name contains character 'a' anywhere in the string.

```
SQL> select * from emp where ename like '%a%';
```

To see those employees whose name contains 'a' in second position.

```
SQL>select * from emp where ename like '_a%';
```





To see those employees whose name contains 'a' as last second character.

```
SQL>select * from emp where ename like '%a_';
```

To see those employees whose name contain '%' sign. i.e. '%' sign has to be used as literal not as wild char.

```
SQL> select * from emp where ename like '%\%%%' escape '\';
```

## Formatting Output in Oracle SQL \* Plus

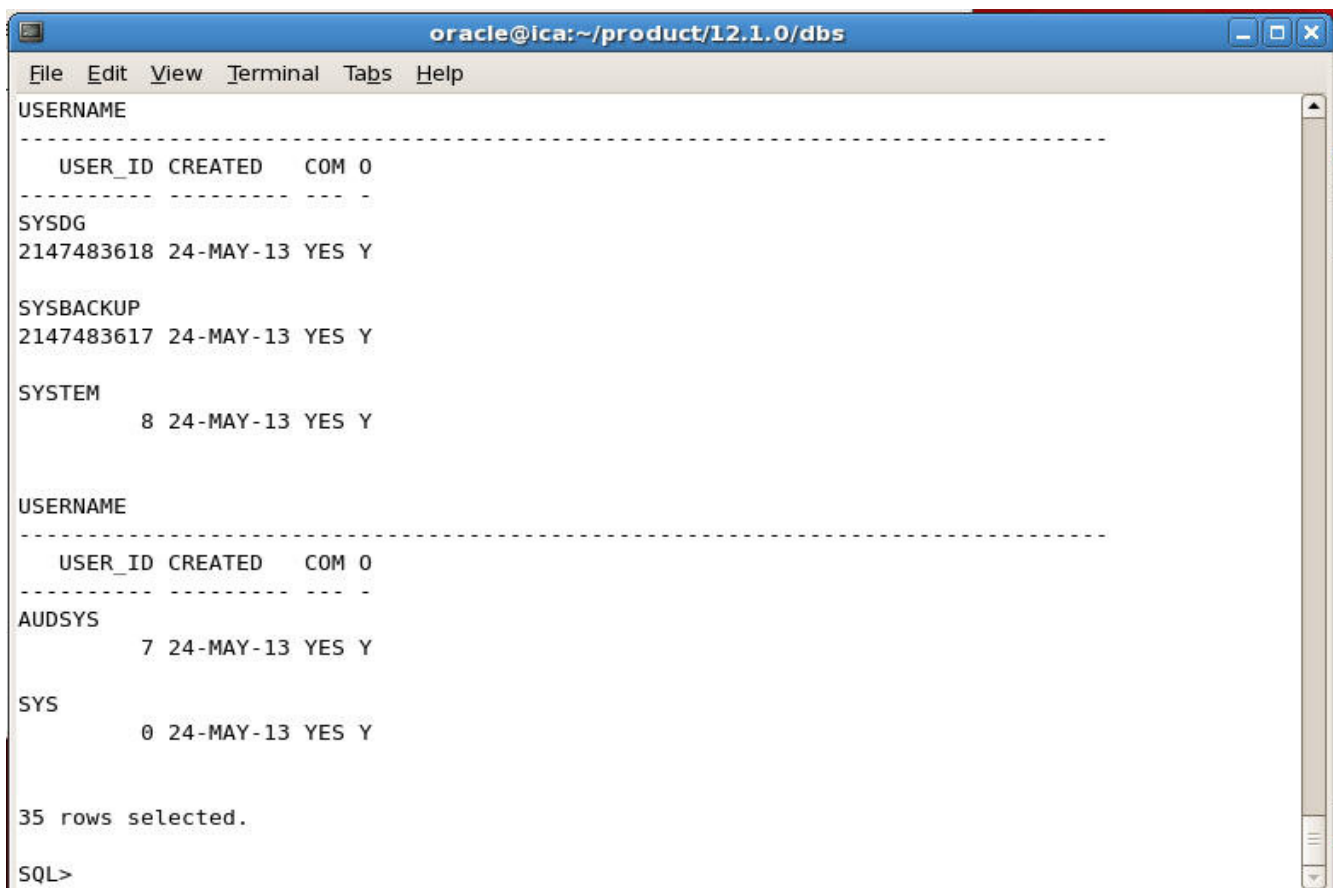
### Simple and useful hints for formatting output in SQL\*Plus

While using Oracle SQL\*Plus for interacting with the database you must have many times seen unstructured output for SQL queries. i.e. the output is hard to interpret.

Like for example if you give a query like this

```
SQL> select * from all users;
```

You will get a output like this



```
oracle@ica:~/product/12.1.0/dbs
File Edit View Terminal Tabs Help
-----
  USERNAME
-----
  USER_ID CREATED   COM 0
-----
SYSDG
2147483618 24-MAY-13 YES Y

SYSBACKUP
2147483617 24-MAY-13 YES Y

SYSTEM
      8 24-MAY-13 YES Y

-----
  USERNAME
-----
  USER_ID CREATED   COM 0
-----
AUDSYS
      7 24-MAY-13 YES Y

SYS
      0 24-MAY-13 YES Y

35 rows selected.

SQL>
```

You can easily structured the output by adjusting the line size and formatting the column by typing the following commands

```
SQL> set linesize 100
```

```
SQL> col username format a30
```

```
35 rows selected.  
  
SQL> set linesize 100  
SQL> col username format a30  
SQL>
```

And then again give the same query; you will see the output in well structured format as shown below

```
oracle@ica:~/product/12.1.0/dbs  
File Edit View Terminal Tabs Help  
35 rows selected.  
  
SQL> set linesize 100  
SQL> col username format a30  
SQL> select * from all_users;  
  
USERNAME                                USER_ID  CREATED    COM 0  
-----  
DVF                                       99  24-MAY-13  YES Y  
APEX_040200                             98  24-MAY-13  YES Y  
APEX_PUBLIC_USER                       95  24-MAY-13  YES Y  
FLOWS_FILES                             94  24-MAY-13  YES Y  
LBACSYS                                  92  24-MAY-13  YES Y  
SPATIAL_CSW_ADMIN_USR                   90  24-MAY-13  YES Y  
SPATIAL_WFS_ADMIN_USR                   87  24-MAY-13  YES Y  
MDDATA                                   85  24-MAY-13  YES Y  
OLAPSYS                                  82  24-MAY-13  YES Y  
DVSYS                                    1279990 24-MAY-13  YES Y  
SI_INFORMTN_SCHEMA                      78  24-MAY-13  YES Y  
-----  
USERNAME                                USER_ID  CREATED    COM 0  
-----  
DBSNMP                                   47  24-MAY-13  YES Y  
ORACLE_OCM                              36  24-MAY-13  YES Y  
DIP                                       23  24-MAY-13  YES Y  
GSMUSER                                  22  24-MAY-13  YES Y  
GSMADMIN_INTERNAL                       21  24-MAY-13  YES Y  
XS$NULL                                 2147483638 24-MAY-13  YES Y  
OUTLN                                    13  24-MAY-13  YES Y  
SYSKM                                    2147483619 24-MAY-13  YES Y  
SYSDG                                    2147483618 24-MAY-13  YES Y  
SYSBACKUP                               2147483617 24-MAY-13  YES Y  
SYSTEM                                   8  24-MAY-13  YES Y  
  
-----  
USERNAME                                USER_ID  CREATED    COM 0  
-----  
AUDSYS                                   7  24-MAY-13  YES Y  
SYS                                       0  24-MAY-13  YES Y  
  
35 rows selected.  
  
SQL> █
```

## Formatting Number Values in SQL Plus

You can also set Number format so see the numeric values with commas for easy reading. For example if you select the rows from scott emp table you will see the output like this

```
oracle@ica:~/product/12.1.0/rdbms/admin
File Edit View Terminal Tabs Help
SQL> select * from emp;

   EMPNO ENAME      JOB              MGR HIREDATE          SAL        COMM         DEPTNO
-----
   7369 SMITH        CLERK            7902 17-DEC-80          800         0             20
   7499 ALLEN        SALESMAN         7698 20-FEB-81         1600        300            30
   7521 WARD          SALESMAN         7698 22-FEB-81         1250        500            30
   7566 JONES        MANAGER          7839 02-APR-81         2975         0             20
   7654 MARTIN      SALESMAN         7698 28-SEP-81         1250       1400            30
   7698 BLAKE        MANAGER          7839 01-MAY-81         2850         0             30
   7782 CLARK        MANAGER          7839 09-JUN-81         2450         0             10
   7788 SCOTT        ANALYST          7566 19-APR-87         3000         0             20
   7839 KING          PRESIDENT                17-NOV-81     5000         0             10
   7844 TURNER      SALESMAN         7698 08-SEP-81         1500         0             30
   7876 ADAMS        CLERK            7788 23-MAY-87         1100         0             20

   EMPNO ENAME      JOB              MGR HIREDATE          SAL        COMM         DEPTNO
-----
   7900 JAMES        CLERK            7698 03-DEC-81          950         0             30
   7902 FORD          ANALYST          7566 03-DEC-81         3000         0             20
   7934 MILLER      CLERK            7782 23-JAN-82         1300         0             10

14 rows selected.

SQL>
```

In the above output the salary column is shown without any formatting which is the default in SQL Plus. If you want to format numeric column values with commas, you can format it like this for example

```
SQL> col sal format $999,99,999.99
```

and now you will get the output like this

```

oracle@ica:~/product/12.1.0/rdbms/admin
File Edit View Terminal Tabs Help
14 rows selected.
SQL> col sal format $999,99,999.99
SQL> /

```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	\$800.00		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	\$1,600.00	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	\$1,250.00	500	30
7566	JONES	MANAGER	7839	02-APR-81	\$2,975.00		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	\$1,250.00	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	\$2,850.00		30
7782	CLARK	MANAGER	7839	09-JUN-81	\$2,450.00		10
7788	SCOTT	ANALYST	7566	19-APR-87	\$3,000.00		20
7839	KING	PRESIDENT		17-NOV-81	\$5,000.00		10
7844	TURNER	SALESMAN	7698	08-SEP-81	\$1,500.00	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	\$1,100.00		20

```


```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7900	JAMES	CLERK	7698	03-DEC-81	\$950.00		30
7902	FORD	ANALYST	7566	03-DEC-81	\$3,000.00		20
7934	MILLER	CLERK	7782	23-JAN-82	\$1,300.00		10

```

14 rows selected.
SQL>

```

Similarly you can also format all numeric values by giving the following command

```
SQL> set numformat "999,99,999.99"
```

Remember the above command will format all numeric values i.e. even empno, deptno etc will be shown in the format, which you don't want in most case.

## **Format DATES in SQL Plus**

Similarly you can also format date values in whatever date format you want by setting the session variable NLS\_DATE\_FORMAT

For example if you set it to the following

```
SQL> alter session set nls_date_format='dd-Mon-yyyy hh:mi:sspm';
```

You will get the output like this

```

oracle@ica:~/product/12.1.0/rdbms/admin
File Edit View Terminal Tabs Help
SQL> alter session set nls_date_format='dd-Mon-yyyy hh:mi:sspm';

Session altered.

SQL> select * from emp;

  EMPNO ENAME      JOB              MGR HIREDATE              SAL        COMM         DEPTNO
-----
 7369 SMITH        CLERK            7902 17-Dec-1980 12:00:00am    $800.00
 7499 ALLEN        SALESMAN         7698 20-Feb-1981 12:00:00am   $1,600.00          300
 7521 WARD          SALESMAN         7698 22-Feb-1981 12:00:00am   $1,250.00          500
 7566 JONES        MANAGER          7839 02-Apr-1981 12:00:00am   $2,975.00
 7654 MARTIN      SALESMAN         7698 28-Sep-1981 12:00:00am   $1,250.00         1400
 7698 BLAKE       MANAGER          7839 01-May-1981 12:00:00am   $2,850.00
 7782 CLARK       MANAGER          7839 09-Jun-1981 12:00:00am   $2,450.00
 7788 SCOTT       ANALYST          7566 19-Apr-1987 12:00:00am   $3,000.00
 7839 KING          PRESIDENT        17-Nov-1981 12:00:00am   $5,000.00
 7844 TURNER      SALESMAN         7698 08-Sep-1981 12:00:00am   $1,500.00          0
 7876 ADAMS       CLERK            7788 23-May-1987 12:00:00am   $1,100.00

  EMPNO ENAME      JOB              MGR HIREDATE              SAL        COMM         DEPTNO
-----
 7900 JAMES        CLERK            7698 03-Dec-1981 12:00:00am    $950.00
 7902 FORD         ANALYST          7566 03-Dec-1981 12:00:00am   $3,000.00
 7934 MILLER      CLERK            7782 23-Jan-1982 12:00:00am   $1,300.00

14 rows selected.

SQL>

```

## Changing SQL Prompt in SQL Plus

You can change the default SQL> prompt in SQL Plus to something more meaningful like you can show username and SID and date in the prompt by giving the following command:

```
SQL> set sqlprompt "_user 'ON' _connect_identifier':'_date> "
```

Then SQL Prompt will change to the following

```

oracle@ica:~/product/12.1.0/rdbms/admin
File Edit View Terminal Tabs Help
SQL> set sqlprompt "_user 'ON' _connect_identifier':'_date> "
SCOTT ON test:01-DEC-15> █

```



This is particularly useful if you work on multiple databases.

## Automatic Setting

What about automatically setting the above formats whenever you login to SQL Plus?

If you want specific settings to be set whenever you login to SQL Plus, then you can write these set commands in glogin.sql or login.sql file located in ORACLE\_HOME/sqlplus/admin folder

For example, you can open or create a new glogin.sql or login.sql file using any text editor and write the following commands:

```
alter session set nls_date_format='dd-Mon-yyyy hh:mi:sspm';  
set sqlprompt "_user 'ON' _connect_identifier:'_date> "
```

Now whenever you login to Oracle using SQL Plus, SQL Plus will show the dates in the above format and SQL Prompt will also change to the above format.



# Oracle UNION, INTERSECT, MINUS OPERATORS AND SORTING QUERY RESULT

## The UNION [ALL], INTERSECT, MINUS Operators

You can combine multiple queries using the set operators UNION, UNION ALL, INTERSECT, and MINUS. All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

### UNION Example

The following statement combines the results with the UNION operator, which eliminates duplicate selected rows.

```
select empno,ename,sal from emp
UNION
select empno,ename,salary from oldemp
```

What if you need to select rows from two tables, but tables have different columns? In this situation you have to use TO\_CHAR function to fill up missing columns.

For Example

This statement shows that you must match datatype (using the TO\_CHAR function) when columns do not exist in one or the other table:

```
select empno, ename, sal, to_char(null) as "Transfer Date" from emp
UNION
select empno,ename,to_char(null) as "Sal",tdate from oldemp;
```

EMPNO	ENAME	SAL	Transfer Date
101	Sami	5000	
102	Smith		11-jul-2000
201	Tamim		10-AUG-2000
209	Ravi	2400	

### UNION ALL Example

The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:



```
select empno,ename from emp
union all
select empno,ename from oldemp;
```

## INTERSECT Example

The following statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
SELECT empno FROM emp
INTERSECT
SELECT empno FROM oldemp;
```

## MINUS Example

The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT empno FROM emp
MINUS
SELECT empno FROM oldemp;
```

## SORTING QUERY RESULTS

To sort query result you can use ORDER BY clause in SELECT statement.

Sorting Examples.

The following query sorts the employees according to ascending order of salaries.

```
select * from emp order by sal;
```

The following query sorts the employees according to descending order of salaries.

```
select * from emp order by sal desc;
```

The following query sorts the employees according to ascending order of names.

```
select * from emp order by ename;
```

The following query first sorts the employees according to ascending order of names. If names are equal then sorts employees on descending order of salaries.

```
select * from emp order by ename, sal desc;
```



You can also specify the positions instead of column names. Like in the following query, which shows employees according to ascending order of their names.

```
select * from emp order by 2;
```

The following query first sorts the employees according to ascending order of salaries.

If salaries are equal then sorts employees on ascending order of names

```
select * from emp order by 3, 2;
```

## SQL Functions in Oracle

SQL functions are built into Oracle and are available for use in various appropriate SQL statements. You can also create your own function using PL/SQL.

### Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, WHERE clauses, START WITH and CONNECT BY clauses, and HAVING clauses.

Oracle SQL Functions can be divided into following categories

- *Number Functions*
- *Character Functions*
- *Miscellaneous Single Row Functions*
- *Aggregate Functions*
- *Date and Time Functions*

Here are the explanation and example of these functions

### Number Functions (also known as Math Functions)

Number functions accept numeric input and return numeric values. Most of these functions return values that are accurate to 38 decimal digits.

The number functions available in Oracle are:

ABS  
ACOS

ASIN  
ATAN  
ATAN2  
BITAND  
CEIL  
COS  
COSH  
EXP  
FLOOR  
LN  
LOG  
MOD  
POWER  
ROUND (number)  
SIGN  
SIN  
SINH  
SQRT  
TAN  
TANH  
TRUNC (number)

## ***ABS***

ABS returns the absolute value of n.

The following example returns the absolute value of -87:

```
SELECT ABS(-87) "Absolute" FROM DUAL;
```

```
Absolute  
-----  
      87
```

## ***ACOS***

ACOS returns the arc cosine of n. Inputs are in the range of -1 to 1, and outputs are in the range of 0 to pi and are expressed in radians.

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3)"Arc_Cosine" FROM DUAL;
```

```
Arc_Cosine
```

-----  
1.26610367

Similar to ACOS, you have ASIN (Arc Sine), ATAN (Arc Tangent) functions.

### *CIEL*

Returns the lowest integer above the given number.

Example:

The following function returns the lowest integer above 3.456;

```
select ciel(3.456) "Ciel" from dual;
```

```
Ciel  
-----  
    4
```

### *FLOOR*

Returns the highest integer below the given number.

Example:

The following function return the highest integer below 3.456;

```
select floor(3.456) "Floor" from dual;
```

```
Floor  
-----  
    3
```

### *COS*

Returns the cosine of an angle (in radians).

Example:

The following example returns the COSINE angle of 60 radians.

```
select cos(60) "Cosine" from dual;
```

## *SIN*

Returns the Sine of an angle (in radians).

Example:

The following example returns the SINE angle of 60 radians.

```
select SIN(60) "Sine" from dual;
```

## *TAN*

Returns the Tangent of an angle (in radians).

Example:

The following example returns the tangent angle of 60 radians.

```
select Tan(60) "Tangent" from dual;
```

Similar to SIN, COS, TAN functions hyperbolic functions SINH, COSH, TANH are also available in oracle.

## *MOD*

Returns the remainder after dividing m with n.

Example

The following example returns the remainder after dividing 30 by 4.

```
Select mod(30,4) "MOD" from dual;
```

```
MOD
```

```
-----
```

```
2
```

## *POWER*

Returns the power of m, raised to n.

Example

The following example returns the 2 raised to the power of 3.

select power(2,3) “Power” from dual;

POWER

-----

8

### *EXP*

Returns the e raised to the power of n.

Example

The following example returns the e raised to power of 2.

select exp(2) “e raised to 2” from dual;

E RAISED TO 2

-----

### *LN*

Returns natural logarithm of n.

Example

The following example returns the natural logarithm of 2.

select ln(2) from dual;

LN

-----

### *LOG*

Returns the logarithm, base m, of n.

Example

The following example returns the log of 100.

select log(10,100) from dual;

LOG

-----  
2

## ***ROUND***

Returns a decimal number rounded of to a given decimal positions.

Example

The following example returns the no. 3.4573 rounded to 2 decimals.

```
select round(3.4573,2) "Round" from dual;
```

Round  
-----  
3.46

## ***TRUNC***

Returns a decimal number Truncated to a given decimal positions.

Example

The following example returns the no. 3.4573 truncated to 2 decimals.

```
select round(3.4573,2) "Round" from dual;
```

Round  
-----  
3.45

## ***SQRT***

Returns the square root of a given number.

Example

The following example returns the square root of 16.

```
select sqrt(16) from dual;
```

SQRT  
-----  
4



## Character Functions available in Oracle

### Character Functions

Character functions operate on values of datatype CHAR or VARCHAR.

#### *LOWER*

Returns a given string in lower case.

```
select LOWER('SAMI') from dual;
```

```
LOWER
```

```
-----
```

```
sami
```

#### *UPPER*

Returns a given string in UPPER case.

```
select UPPER('Sami') from dual;
```

```
UPPER
```

```
-----
```

```
SAMI
```

#### *INITCAP*

Returns a given string with Initial letter in capital.

```
select INITCAP('mohammed sami') from dual;
```

```
INITCAP
```

```
-----
```

```
Mohammed Sami
```

#### *LENGTH*

Returns the length of a given string.

```
select length('mohammed sami') from dual;
```



LENGTH

-----  
13

### ***SUBSTR***

Returns a substring from a given string. Starting from position p to n characters.

For example the following query returns “sam” from the string “mohammed sami”.

```
select substr('mohammed sami',10,3) from dual;
```

Substr  
-----  
sam

### ***INSTR***

Tests whether a given character occurs in the given string or not. If the character occurs in the string then returns the first position of its occurrence otherwise returns 0.

Example

The following query tests whether the character “a” occurs in string “mohammed sami”

```
select instr('mohammed sami','a') from dual;
```

INSTR  
-----  
4

### ***REPLACE***

Replaces a given set of characters in a string with another set of characters.

Example

The following query replaces “mohd” with “mohammed” .

```
select replace('ali mohd khan','mohd','mohammed') from dual;
```

REPLACE  
-----  
ali mohammed khan

## ***TRANSLATE***

This function is used to encrypt characters. For example you can use this function to replace characters in a given string with your coded characters.

Example

The following query replaces characters A with B, B with C, C with D, D with E,...Z with A, and a with b,b with c,c with d, d with e ....z with a.

```
select translate('interface','ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz',  
               'BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz') "Encrypt" from dual;
```

Encrypt

-----  
joufsgbdf

## ***SOUNDEX***

This function is used to check pronunciation rather than exact characters. For example many people write names as “smith” or “smyth” or “smythe” but they are pronounced as smith only.

Example

The following example compare those names which are spelled differently but are pronounced as “smith”.

```
Select ename from emp where soundex(ename)=soundex('smith');
```

ENAME

-----  
Smith  
Smyth  
Smythe

## ***RPAD***

Right pads a given string with a given character to n number of characters.

Example

The following query rights pad ename with '\*' until it becomes 10 characters.

```
select rpad(ename,'*',10) from emp;
```

Ename

-----

```
Smith*****
John*****
Mohammed**
Sami*****
```

### *LPAD*

Left pads a given string with a given character upto n number of characters.

Example

The following query left pads ename with '\*' until it becomes 10 characters.

```
select lpad(ename,'*',10) from emp;
```

Ename

-----

```
*****Smith
*****John
**Mohammed
*****Sami
```

### *LTRIM*

Trims blank spaces from a given string from left.

Example

The following query returns string “ Interface ” left trimmed.

```
select ltrim(' Interface ') from dual;
```

Ltrim

-----

```
Interface
```

### *RTRIM*

Trims blank spaces from a given string from Right.

### Example

The following query returns string “ Interface “ right trimmed.

```
select rtrim(' Interface ') from dual;
```

Rtrim

-----

Interface

### *TRIM*

Trims a given character from left or right or both from a given string.

### Example

The following query removes zero from left and right of a given string.

```
Select trim(0 from '00003443500') from dual;
```

Trim

-----

34435

### *CONCAT*

Combines a given string with another string.

### Example

The following Query combines ename with literal string “ is a “ and jobid.

```
Select concat(concat(ename,' is a '),job) from emp;
```

Concat

-----

Smith is a clerk

John is a Manager

Sami is a G.Manager

## Miscellaneous SQL Functions in Oracle

### Miscellaneous Single Row Functions

#### *COALESCE*

Coalesce function returns the first not null value in the expression list.

Example.

The following query returns salary+commision, if commission is null then returns salary, if salary is also null then returns 1000.

```
select empno,ename,salary,comm,coalesce(salary+comm,salary,1000) "Net Sal" from emp;
```

ENAME	SALARY	COMM	NET SAL
SMITH	1000	100	1100
SAMI	3000		3000
SCOTT		1000	
RAVI	200		1000

#### *DECODE*

DECODE(expr, searchvalue1, result1,searchvalue2,result2,...., defaultvalue)

Decode functions compares an expr with search value one by one. If the expr does not match any of the search value then returns the default value. If the default value is omitted then returns null.

Example

The following query returns the department names according the deptno. If the deptno does not match any of the search value then returns "Unknown Department"

```
select decode(deptno,10,'Sales',20,'Accounts',30,'Production',40,'R&D','Unknown Dept') As DeptName
from emp;
```

DEPTNAME
Sales
Accounts
Unknown Dept.
Accounts

Production  
Sales  
R&D  
Unknown Dept.

## ***GREATEST***

GREATEST(expr1, expr2, expr3,expr4...)

Returns the greatest expr from a expr list.

Example

```
select greatest(10,20,50,20,30) from dual;
```

GREATEST

-----

50

```
select greatest('SAMI','SCOTT','RAVI','SMITH','TANYA') from dual;
```

GREATEST

-----

TANYA

## ***LEAST***

LEAST(expr1, expr2, expr3,expr4...)

It is similar to greatest. It returns the least expr from the expression list.

```
select least(10,20,50,20,30) from dual;
```

LEAST

-----

10

```
select least('SAMI','SCOTT','RAVI','SMITH','TANYA') from dual;
```

LEAST

-----

RAVI

## NVL

NVL2(expr1,expr2)

This function is often used to check null values. It returns expr2 if the expr1 is null, otherwise returns expr1.

Example

The following query returns commission if commission is null then returns 'Not Applicable'.

Select ename,nvl(comm,'Not Applicable') "Comm" from dual;

ENAME	COMM
-----	-----
Scott	300
Tiger	450
Sami	Not Applicable
Ravi	300
Tanya	Not Applicable

## NVL2

NVL2(expr1,expr2,expr3)

NVL2 returns expr2 if expr1 is not null, otherwise return expr3.

Example

The following query returns salary+comm if comm is not null, otherwise just returns salary.

select salary,comm,nvl2(comm,salary+comm,salary) "Income" from emp;

SALARY	COMM	INCOME
-----	-----	-----
1000	100	1100
2000		2000
2300	200	2500
3400		3400

## NULLIF

NULLIF(expr1, expr2)



Nullif compares expr1 with expr2. If they are equal then returns null, otherwise return expr1.

Example:

The following query shows old jobs of those employees who have changed their jobs in the company by comparing the current job with old job in oldemp table.

Select ename,nullif(e.job,o.job) “Old Job” from emp e, oldemp o where e.empno=o.empno;

ENAME	OLD JOB
SMITH	CLERK
SAMI	
SCOTT	MANAGER

### *UID*

Returns the current session ID of user logged on.

Example

select uid from dual;

UID
20

### *USER*

Returns the username of the current user logged on.

select user from dual;

USER
SCOTT

### *SYS\_CONTEXT*

SYS\_CONTEXT returns the value of parameter associated with the context namespace. You can use this function in both SQL and PL/SQL statements.

EXAMPLE



The following query returns the username of the current user.

```
Select sys_context('USERENV','SESSION_USER') "Username" from dual;
```

```
USERNAME
```

```
-----  
SCOTT
```

Similar to SESSION\_USER parameter for namespace USERENV the other important parameters are

ISDBA :To check whether the current user is having DBA privileges or not.

HOST :Returns the name of host machine from which the client is connected.

INSTANCE :The instance identification number of the current instance

IP\_ADDRESS: IP address of the machine from which the client is connected.

DB\_NAME :Name of the database as specified in the DB\_NAME initialization parameter

## ***VSIZE***

```
VSIZE(expr)
```

Returns the internal representation of expr in bytes.

Example

The following query return the representation of ename in bytes.

```
select ename,vsize(ename) as Bytes from emp;
```

```
ENAME BYTES
```

```
-----  
SCOTT 5  
SAMI 4  
RAVI 4  
KIRAN 5
```

# Oracle Aggregate SQL Functions

## Aggregate Functions

Aggregate functions return a single value based on groups of rows, rather than single value for each row. You can use Aggregate functions in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle divides the rows of a queried table or view into groups.

The important Aggregate functions are:

Avg   Sum   Max   Min   Count   Stddev   Variance

### AVG

AVG( ALL /DISTINCT      expr)

Returns the average value of expr.

Example

The following query returns the average salary of all employees.

```
select avg(sal) "Average Salary" from emp;
```

```
Average Salary
-----
2400.40
```

### SUM

SUM(ALL/DISTINCT      expr)

Returns the sum value of expr.

Example

The following query returns the sum salary of all employees.

```
select sum(sal) "Total Salary" from emp;
```

```
Total Salary
-----
26500
```

## MAX

MAX(ALL/DISTINCT expr)

Returns maximum value of expr.

Example

The following query returns the max salary from the employees.

```
select max(sal) "Max Salary" from emp;
```

Maximum Salary

-----  
4500

## MIN

MIN(ALL/DISTINCT expr)

Returns minimum value of expr.

Example

The following query returns the minimum salary from the employees.

```
select min(sal) "Min Salary" from emp;
```

Minimum Salary

-----  
1200

## COUNT

COUNT(\*) OR COUNT(ALL/DISTINCT expr)

Returns the number of rows in the query. If you specify expr then count ignore nulls. If you specify the asterisk (\*), this function returns all rows, including duplicates and nulls. COUNT never returns null.

Example

The following query returns the number of employees.

```
Select count(*) from emp;
```

COUNT

-----  
14

The following query counts the number of employees whose salary is not null.

Select count(sal) from emp;

COUNT

-----  
12

## STDDEV

STDDEV(ALL/DISTINCT expr)

STDDEV returns sample standard deviation of expr, a set of numbers.

Example

The following query returns the standard deviation of salaries.

select stddev(sal) from emp;

Stddev  
-----  
1430

## VARIANCE

VARIANCE(ALL/DISTINCT expr)

Variance returns the variance of expr.

Example

The following query returns the variance of salaries.

select variance(sal) from emp;

Variance  
-----  
1430

## Formating DATES in Oracle

### **Date Functions and Operators.**

To see the system date and time use the following functions:

**CURRENT\_DATE** : returns the current date in the session time zone, in a value in the Gregorian calendar of data type DATE  
**SYSDATE** :Returns the current date and time.  
**SYSTIMESTAMP** :The SYSTIMESTAMP function returns the system date, including fractional seconds and time zone of the database. The return type is **TIMESTAMP WITH TIME ZONE**.

### **SYSDATE Example**

To see the current system date and time give the following query.

```
select sysdate from dual;
```

```
SYSDATE  
-----  
8-AUG-03
```

The format in which the date is displayed depends on **NLS\_DATE\_FORMAT** parameter.

For example set the **NLS\_DATE\_FORMAT** to the following format

```
alter session set NLS_DATE_FORMAT='DD-MON-YYYY HH:MIpm';
```

Then give the give the following statement

```
select sysdate from dual;  
  
SYSDATE  
-----  
8-AUG-2003 03:05pm
```

The default setting of **NLS\_DATE\_FORMAT** is **DD-MON-YY**

### **CURRENT\_DATE Example**

To see the current system date and time with time zone use **CURRENT\_DATE** function



```
ALTER SESSION SET TIME_ZONE = '-4:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

```
SESSIONTIMEZONE CURRENT_DATE
-----
```

```
-04:00      22-APR-2003 14:15:03
```

```
ALTER SESSION SET TIME_ZONE = '-7:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

```
SESSIONTIMEZONE CURRENT_DATE
-----
```

```
-07:00      22-APR-2003 09:15:33
```

### **SYSTIMESTAMP Example**

To see the current system date and time with fractional seconds with time zone give the following statement

```
select systimestamp from dual;
```

```
SYSTIMESTAMP
-----
```

```
22-APR-03 08.38.55.538741 AM -07:00
```

## **DATE FORMAT MODELS**

To translate the date into a different format string you can use TO\_CHAR function with date format. For example to see the current day you can give the following query

```
Select to_char(sysdate,'DAY') "Today" FROM DUAL;
```

```
TODAY
-----
```

```
THURSDAY
```

To translate a character value, which is in format other than the default date format, into a date value you can use TO\_DATE function with date format to date

Like this "DAY" format model there are many other date format models available in Oracle. The following table list date format models.

FORMAT	MEANING
D	Day of the week
DD	Day of the month
DDD	Day of the year
DAY	Full day for ex. 'Monday', 'Tuesday', 'Wednesday'
DY	Day in three letters for ex. 'MON', 'TUE', 'FRI'
W	Week of the month
WW	Week of the year
MM	Month in two digits (1-Jan, 2-Feb,...12-Dec)
MON	Month in three characters like "Jan", "Feb", "Apr"
MONTH	Full Month like "January", "February", "April"
RM	Month in Roman Characters (I-XII, I-Jan, II-Feb,...XII-Dec)
Q	Quarter of the Month
YY	Last two digits of the year.
YYYY	Full year
YEAR	Year in words like "Nineteen Ninety Nine"
HH	Hours in 12 hour format
HH12	Hours in 12 hour format
HH24	Hours in 24 hour format
MI	Minutes
SS	Seconds
FF	Fractional Seconds
SSSSS	Milliseconds
J	Julian Day i.e Days since 1st-Jan-4712BC to till-date
RR	If the year is less than 50 Assumes the year as 21ST Century. If the year is greater than 50 then assumes the year in 20th Century.

### Suffixes

TH	Returns th, st, rd or nd according to the leading number like 1st , 2nd 3rd 4th
SP	Spells out the leading number
AM or PM	Returns AM or PM according to the time
SPTH	Returns Spelled Ordinal number. For. Example First, Fourth

For example to see the today's date in the following format

Friday, 7th March, 2014

Give the following statement

```
select to_char(sysdate,'Day, ddth Month, yyyy')"Today" from dual;
```

TODAY

-----  
Friday, 7th March, 2014

For example you want to see hire dates of all employee in the following format

Friday, 8th August, 2003

Then give the following query.

```
select to_char(hire_date,'Day, ddth Month, yyyy') from emp;
```

### TO\_DATE Example

To\_Date function is used to convert strings into date values. For example you want to see what was the day on 15-aug-1947. The use the to\_date function to first convert the string into date value and then pass on this value to to\_char function to extract day.

```
select to_char(to_date('15-aug-1947','dd-mon-yyyy'),'Day')  
          from dual;
```

TO\_CHAR(  
-----  
Friday

To see how many days have passed since 15-aug-1947 then give the following query

```
select sysdate-to_date('15-aug-1947','dd-mon-yyyy') from dual;
```

Now we want to see which date will occur after 45 days from now

```
select sysdate+45 from dual;
```

SYSDATE  
-----  
06-JUN-2003

### ADD\_MONTHS

To see which date will occur after 6 months from now, we can use ADD\_MONTHS function

```
Select ADD_MONTHS(SYSDATE,6) from dual;
```



ADD\_MONTHS

-----

22-OCT-2003

## MONTHS\_BETWEEN

To see how many months have passed since a particular date, use the MONTHS\_BETWEEN function.

For Example, to see how many months have passed since 15-aug-1947, give the following query.

```
select months_between(sysdate,to_date('15-aug-1947'))
       from dual;
```

Months

-----

616.553

To eliminate the decimal value use truncate function

```
select trunc(months_between(sysdate,to_date('15-aug-1947')))
       from dual;
```

Months

-----

616

## LAST\_DAY

To see the last date of the month of a given date, Use LAST\_DAY function.

```
select LAST_DAY(sysdate) from dual;
```

LAST\_DAY

-----

31-AUG-2003

## NEXT\_DAY

To see when a particular day is coming next, use the NEXT\_DAY function.

For Example to view when next Saturday is coming, give the following query

```
select next_day(sysdate) from dual;
```

NEXT\_DAY

-----  
09-AUG-2003

## EXTRACT

An EXTRACT datetime function extracts and returns the value of a specified datetime field from a datetime or interval value expression. When you extract a TIMEZONE\_REGION or TIMEZONE\_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation

The syntax of EXTRACT function is

EXTRACT ( YEAR / MONTH / WEEK / DAY / HOUR / MINUTE / TIMEZONE FROM DATE)

Example

The following demonstrate the usage of EXTRACT function to extract year from current date.

```
select extract(year from sysdate) from dual;
```

```
EXTRACT
```

```
-----  
2003
```

# Join Queries in Oracle

## Joins

A join is a query that combines rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

## Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a join condition. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

## Equijoins

An equijoin is a join with a join condition containing an equality operator ( = ). An equijoin combines rows that have equivalent values for the specified columns.

For example the following query returns empno,name,sal,deptno and department name and city from department table.

```
select emp.empno,emp.ename,emp.sal,emp.deptno,dept.dname,dept.city from emp,dept where emp.deptno=dept.deptno;
```

The above query can also be written like, using aliases, given below.

```
select e.empno, e.ename, e.sal, e.deptno, d.dname, d.city from emp e, dept d where emp.deptno=dept.deptno;
```

The above query can also be written like given below without using table qualifiers.

```
select empno,ename,sal,dname,city from emp,dept where emp.deptno=dept.deptno;
```

And if you want to see all the columns of both tables then the query can be written like this.

```
select * from emp,dept where emp.deptno=dept.deptno;
```

## Non Equi Joins.

Non equi joins is used to return result from two or more tables where exact join is not possible.

For example we have emp table and salgrade table. The salgrade table contains grade and their low salary and high salary. Suppose you want to find the grade of employees based on their salaries then you can use NON EQUI join.

```
select e.empno, e.ename, e.sal, s.grade from emp e, salgrade s where e.sal between s.lowsal and s.hisal
```

## Self Joins

A self join is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition.

For example the following query returns employee names and their manager names for whom they are working.

```
Select e.empno, e.ename, m.ename "Manager" from emp e,  
emp m where e.mgrid=m.empno
```

## Inner Join

An inner join (sometimes called a "simple join") is a join of two or more tables that returns only those rows that satisfy the join condition.

## Outer Joins

An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

- To write a query that performs an outer join of tables A and B and returns all rows from A (a left outer join), use the ANSI LEFT [OUTER] JOIN syntax, or apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Oracle returns null for any select list expressions containing columns of B.
- To write a query that performs an outer join of tables A and B and returns all rows from B (a right outer join), use the ANSI RIGHT [OUTER] syntax, or apply the outer join operator (+) to all columns of A in the join condition. For all rows in B that have no matching rows in A, Oracle returns null for any select list expressions containing columns of A.
- To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a full outer join), use the ANSI FULL [OUTER] JOIN syntax.

For example the following query returns all the employees and department names and even those department names where no employee is working.

```
select e.empno,e.ename,e.sal,e.deptno,d.dname,d.city from emp e, dept d
where e.deptno(+)=d.deptno;
```

That is specify the (+) sign to the column which is lacking values.

## Cartesian Products

If two tables in a join query have no join condition, Oracle returns their Cartesian product. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product.

## Sub Queries and GROUP BY Queries in Oracle

### SUBQUERIES

A query nested within a query is known as subquery.

For example, you want to see all the employees whose salary is above average salary. For this you have to first compute the average salary using AVG function and then compare employees salaries with this computed salary. This is possible using subquery. Here the sub query will first compute the average salary and then main query will execute.

```
Select * from emp where sal > (select avg(sal) from emp);
```

Similarly we want to see the name and empno of that employee whose salary is maximum.

```
Select * from emp where sal = (select max(sal) from emp);
```

To see second maximum salary

```
Select max(sal) from emp where  
sal < (select max(sal) from emp);
```

Similarly to see the Third highest salary:

```
Select max(sal) from emp where  
sal < (select max(sal) from emp where  
sal < (select max(sal) from emp));
```

We want to see how many employees are there whose salary is above average.

```
Select count(*) from emp where  
sal > (select max(sal) from emp);
```

We want to see those employees who are working in Hyderabad. Remember emp and dept are joined on deptno and city column is in the dept table. Assuming that wherever the department is located the employee is working in that city.

```
Select * from emp where deptno  
in (select deptno from dept where city='HYD');
```

You can also use subquery in FROM clause of SELECT statement.

For example the following query returns the top 5 salaries from employees table.



```
Select sal from (select sal from emp order sal desc)
      where rownum <= 5;
```

To see the sum salary deptwise you can give the following query.

```
Select sum(sal) from emp group by deptno;
```

Now to see the average total salary deptwise you can give a sub query in FROM clause.

```
select avg(depttotal) from (select sum(sal) as depttotal from emp group by deptno);
```

## WITH

The above average total salary department wise can also be achieved from Oracle Version 9i using WITH clause given below

```
WITH
DEPTOT AS (select sum(sal) as dsal from emp
           group by deptno)
select avg(dsal) from deptot;
```

## GROUP BY QUERIES

You can group query results on some column values. When you give a SELECT statement without group by clause then all the resultant rows are treated as a single group.

For Example, we want to see the sum salary of all employees dept wise. Then the following query will achieved the result.

```
Select deptno,sum(sal) from emp group by deptno;
```

Similarly we want to see the average salary dept wise

```
Select deptno,avg(sal) from emp group by deptno;
```

Similarly we want to see the maximum salary in each department.

```
Select deptno,max(sal) from emp group by deptno;
```

Similarly the minimum salary.

```
Select deptno,min(sal) from emp group by deptno;
```

Now we want to see the number of employees working in each department.



Select deptno,count(\*) from emp group by deptno;

Now we want to see total salary department wise where the dept wise total salary is above 5000.

For this you have to use HAVING clause. Remember HAVING clause is used to filter groups and WHERE clause is used to filter rows. You cannot use WHERE clause to filter groups.

```
select deptno,sum(sal) from emp group by deptno
        having sum(sal) >= 5000;
```

We want to see those departments and the number of employees working in them where the number of employees is more than 2.

```
Select deptno, count(*) from emp group by deptno
        having count(*) >=2;
```

Instead of displaying deptno you can also display deptnames by using join conditions.

For example we want to see deptname and average salary of them.

```
Select dname,avg(sal) from emp,dept
        where emp.deptno=dept.deptno group by dname;
```

Similarly to see sum of sal.

```
Select dname,sum(sal) from emp,dept
        where emp.deptno=dept.deptno group by dname;
```

We want to see the cities name and the no of employees working in each city. Remember emp and dept are joined on deptno and city column is in the dept table. Assuming that : wherever the department is located the employee is working in that city.

```
Select dept.city,count(empno) from emp,dept
        where emp.deptno=dept.deptno
        Group by dept.city;
```

## **CUBE, ROLLUP and CASE Expression in Oracle**

### **ROLLUP**

The ROLLUP operation in the simple\_grouping\_clause groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions in the GROUP BY specification, and returns a single row of



summary for each group. You can use the ROLLUP operation to produce subtotal values by using it with the SUM function. When used with SUM, ROLLUP generates subtotals from the most detailed level to the grand total. Aggregate functions such as COUNT can be used to produce other kinds of superaggregates.

For example, given three expressions ( $n=3$ ) in the ROLLUP clause of the simple\_grouping\_clause, the operation results in  $n+1 = 3+1 = 4$  groupings.

Rows grouped on the values of the first 'n' expressions are called regular rows, and the others are called superaggregate rows.

The following query uses rollup operation to show sales amount product wise and year wise. To see the structure of the sales table refer to appendices.

```
Select prod,year,sum(amt) from sales
      group by rollup(prod,year);
```

## CUBE

The CUBE operation in the simple\_grouping\_clause groups the selected rows based on the values of all possible combinations of expressions in the specification, and returns a single row of summary information for each group. You can use the CUBE operation to produce cross-tabulation values.

For example, given three expressions ( $n=3$ ) in the CUBE clause of the simple\_grouping\_clause, the operation results in  $2^n = 2^3 = 8$  groupings. Rows grouped on the values of 'n' expressions are called regular rows, and the rest are called super aggregate rows.

The following query uses CUBE operation to show sales amount product wise and year wise. To see the structure of the sales table refer to appendices.

```
Select prod,year,sum(amt) from sales
      group by CUBE(prod,year);
```

## CASE EXPRESSION

CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures.

For example the following query uses CASE expression to display Department Names based on deptno.

```
select empno,ename,sal,CASE deptno when 10 then
      'Accounts' when 20 then 'Sales'
      when 30 then 'R&D'
      else "Unknown" end
      from emp;
```





The following statement finds the average salary of the employees in the employees table using \$2000 as the lowest salary possible:

```
SELECT AVG(CASE WHEN e.sal > 2000 THEN e.sal  
ELSE 2000 END) "Average Salary" from emp e;
```

## Oracle INSERT, UPDATE, DELETE, MERGE, Multi INSERT Statements

### Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements query and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction.

The following are the DML statements available in Oracle.

- INSERT : Use to Add Rows to existing table.
- UPDATE : Use to Edit Existing Rows in tables.
- DELETE : Use to Delete Rows from tables.
- MERGE : Use to Update or Insert Rows depending on condition.

#### Insert

Use the Insert Statement to Add records to existing Tables.

Examples:

To add a new row to an emp table.

```
Insert into emp values (101,'Sami','G.Manager',  
'8-aug-1998',2000);
```

If you want to add a new row by supplying values for some columns not all the columns then you have to mention the name of the columns in insert statements. For example the following statement inserts row in emp table by supplying values for empno, ename, and sal columns only. The Job and Hiredate columns will be null.

```
Insert into emp (empno,ename,sal) values (102,'Ashi',5000);
```

Suppose you want to add rows from one table to another i.e. suppose we have Old\_Emp table and emp table with the following structure:

Old_emp		Emp	
Column Name	Datatype & Width	Column Name	Datatype & Width
Empno	Number(5)	Empno	Number(5)
Ename	Varchar2(20)	Ename	Varchar2(20)
Sal	Number(10,2)	Sal	Number(10,2)
Tdate	Date	Hiredate	Date
		Job	Varchar2(20)

Now we want to add rows from old\_emp table to emp table. Then you can give the following insert statement:

```
Insert into emp (empno, ename, sal)
select empno, ename, sal from old_emp;
```

### Multi Table Insert

Suppose we have sales table with the following structure.

#### Sales

Prodid	Prodname	Mon_Amt	Tue_Amt	Wed_Amt	Thu_Amt	Fri_Amt	Sat_Amt
101	AIWA	2000	2500	2230	2900	3000	2100
102	AKAI	1900	2100	2130	3100	2800	2120

Now we want to add the rows from SALES table Weekly\_Sales Table in the following Structure:

Prodid	Prodname	WeekDay	Amount
101	AIWA	Mon	2000
101	AIWA	Tue	2500
101	AIWA	Wed	2230
101	AIWA	Thu	2900
101	AIWA	Fri	3000
101	AIWA	Sat	2100
102	AKAI	Mon	1900
102	AKAI	Tue	2100
102	AKAI	Wed	2130
102	AKAI	Thu	3100
102	AKAI	Fri	2800
102	AKAI	Sat	2120

To achieve the above we can give a multi table INSERT statement given below-

Insert all

```
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Mon',mon_amt)
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Tue',tue_amt)
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Wed',wed_amt)
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Thu',thu_amt)
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Fri',fri_amt)
Into week_sales(prodid,prodname,weekday,amount)
Values (prodid,prodname,'Sat',sat_amt)
Select prodid,prodname,mon_amt,tue_amt,wed_amt,thu_amt
Fri_amt,sat_amt from sales;
```

## Update

Update statement is used to update rows in existing tables which is in your own schema or if you have update privilege on them.

For example to raise the salary by Rs.500 of employee number 104, you can give the following statement:

```
update emp set sal=sal+500 where empno = 104;
```

In the above statement if we did not give the where condition then all employees salary will be raised by Rs. 500. That's why always specify proper WHERE condition if don't want to update all employees.

For example : we want to change the name of employee no 102 from 'Sami' to 'Mohd Sami' and to raise the salary by 10%. Then the statement will be.

```
update emp set name='Mohd Sami',
sal=sal+(sal*10/100) where empno=102;
```

Now we want to raise the salary of all employees by 5%.

```
update emp set sal=sal+(sal*5/100);
```

Now to change the names of all employees to uppercase:

```
update emp set name=upper(name);
```

Suppose we have a student table with the following structure:

Rollno	Name	Maths	Phy	Chem	Total	Average
101	Sami	99	90	89		
102	Scott	34	77	56		
103	Smith	45	82	43		

Now to compute total which is sum of Maths, Phy and Chem and Average.

```
update student set total=maths+phy+chem,
average=(maths+phy+chem)/3;
```

Using Sub Query in the Update Set Clause.

Suppose we added the city column in the employee table and now we want to set this column with corresponding city column in department table which is join to employee table on deptno.

```
update emp set city=(select city from dept
where deptno= emp.deptno);
```

## Delete

Use the DELETE statement to delete the rows from existing tables which are in your schema or if you have DELETE privilege on them.

For example to delete the employee whose empno is 102:

```
delete from emp where empno=102;
```

If you don't mention the WHERE condition then all rows will be deleted.

Suppose we want to delete all employees whose salary is above 2000. Then give the following DELETE statement.

```
delete from emp where salary > 2000;
```

The following statement has the same effect as the preceding example, but uses a sub query:

```
DELETE FROM (SELECT * FROM emp)
WHERE sal > 2000;
```

To delete all rows from emp table.

```
delete from emp;
```

## Merge

Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause. It is a new feature of Oracle Ver. 9i. It is also known as UPSERT i.e. combination of UPDATE and INSERT.

For example suppose we are having sales and sales\_history table with the following structure.

**SALES**

Prod	Month	Amount
SONY	JAN	2200
SONY	FEB	3000
SONY	MAR	2500
SONY	APR	3200
SONY	MAY	3100
SONY	JUN	5000

**SALES HISTORY**

Prod	Month	Amount
SONY	JAN	2000
SONY	MAR	2500
SONY	APR	3000
AKAI	JAN	3200

Now we want to update sales\_history table from sales table i.e. those rows which are already present in sales\_history, their amount should be updated and those rows which are not present in sales\_history table should be inserted.

```
merge into sales_history sh
using sales s
on (s.prod=sh.prod and s.month=sh.month)
when matched then update set sh.amount=s.amount
when not matched then insert values (prod,month,amount);
```

After the statement is executed sales\_history table will look like this.

**SALES\_HISTORY**

Prod	Month	Amount
SONY	JAN	2200
SONY	FEB	3000
SONY	MAR	2500
SONY	APR	3200
AKAI	JAN	3200
SONY	MAY	3100
SONY	JUN	5000

# Oracle DDL Commands

## Data Definition Language (DDL) Statements

Data definition language (DDL) statements enable you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary

The CREATE, ALTER, and DROP commands require exclusive access to the specified object. For example, an ALTER TABLE statement fails if another user has an open transaction on the specified table.

The GRANT, REVOKE, ANALYZE, AUDIT, and COMMENT commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement. Many DDL statements may cause Oracle to recompile or reauthorize schema objects.

DDL Statements are:

**CREATE** : Use to create objects like CREATE TABLE, CREATE FUNCTION, CREATE SYNONYM, CREATE VIEW. Etc.

**ALTER** : Use to Alter Objects like ALTER TABLE, ALTER USER, ALTER TABLESPACE, ALTER DATABASE. Etc.

**DROP** : Use to Drop Objects like DROP TABLE, DROP USER, DROP TABLESPACE, DROP FUNCTION. Etc.

**REPLACE** : Use to Rename table names.

**TRUNCATE** : Use to truncate (delete all rows) a table.

## Create

Creating tables, views, synonyms, sequences, functions, procedures, packages etc.

For example: to create a table, you can give the following statement

```
create table emp (empno number(5) primary key,
    name varchar2(20),
    sal number(10,2),
    job varchar2(20),
    mgr number(5),
    Hiredate date,
    comm number(10,2));
```

Now Suppose you have emp table now you want to create a TAX table with the following structure and also insert rows of those employees whose salary is above 5000.

Tax	
Empno	Number(5)
Tax	Number(10,2)

To do this we can first create TAX table by defining column names and data types and then use INSERT into EMP SELECT .... Statement to insert rows from emp table given below:

```
create table tax (empno number(5), tax number(10,2));
```

```
insert into tax select empno,(sal-5000)*0.40
    from emp where sal > 5000;
```

Instead of executing the above two statements the same result can be achieved by giving a single CREATE TABLE AS statement.

```
create table tax as select empno,(sal-5000)*0.4
    as tax from emp where sal>5000
```

You can also use CREATE TABLE AS statement to create copies of tables. Like to create a copy EMP table as EMP2 you can give the following statement.

```
create table emp2 as select * from emp;
```

To copy tables without rows i.e. to just copy the structure give the following statement

```
create table emp2 as select * from emp where 1=2;
```

## Temporary Tables (From Oracle Ver. 8i)

It is also possible to create a temporary table. The definition of a temporary table is visible to all sessions, but the data in a temporary table is visible only to the session that inserts the data into the table. You use the CREATE GLOBAL TEMPORARY TABLE statement to create a temporary table. The ON COMMIT keywords indicate if the data in the table is transaction-specific (the default) or session-specific:

- ON COMMIT DELETE ROWS specifies that the temporary table is transaction specific and Oracle truncates the table (delete all rows) after each commit.
- ON COMMIT PRESERVE ROWS specifies that the temporary table is session specific and Oracle truncates the table when you terminate the session.

This example creates a temporary table that is transaction specific:

```
CREATE GLOBAL TEMPORARY TABLE taxable_emp
(empno number(5),
ename varchar2(20),
sal number(10,2),
tax number(10,2))
ON COMMIT DELETE ROWS;
```

Indexes can also be created on temporary tables. They are also temporary and the data in the index has the same session or transaction scope as the data in the underlying table.

## Alter

Use the ALTER TABLE statement to alter the structure of a table.

Examples:

To add new columns addr, city, pin, ph, fax to employee table you can give the following statement

```
alter table emp add (addr varchar2(20), city varchar2(20),
pin varchar2(10),ph varchar2(20));
```

For example we you want to increase the length of the column ename from varchar2(20) to varchar2(30) then give the following command.

```
alter table emp modify (ename varchar2(30))
```

To decrease the width of a column the column can be decreased up to largest value it holds.

```
alter table emp modify (ename varchar2(15));
```





The above is possible only if you are using Oracle ver 8i and above. In Oracle 8.0 and 7.3 you cannot decrease the column width directly unless the column is empty.

To change the datatype the column must be empty in All Oracle Versions.

## Drop columns

From Oracle Ver. 8i you can drop columns directly it was not possible in previous versions.

For example to drop PIN, CITY columns from emp table.

```
alter table emp drop column (pin, city);
```

Remember you cannot drop the column if the table is having only one column.

If the column you want to drop is having primary key constraint on it then you have to give cascade constraint clause.

```
alter table emp2 drop column (empno) cascade constraints;
```

To drop columns in previous versions of Oracle 8.0 and 7.3 and to change the column name in all Oracle versions do the following:

For example we want to drop pin and city columns and to change SAL column name to SALARY.

Step 1: Create a temporary table with desired columns using subquery.

```
create table temp as select empno, ename,  
sal AS salary, addr, ph from emp;
```

Step 2: Drop the original table.

```
drop table emp;
```

Step 3: Rename the temporary table to the original table.

```
rename temp to emp;
```

## Rename

Use the RENAME statement to rename a table, view, sequence, or private synonym for a table, view, or sequence.

- Oracle automatically transfers integrity constraints, indexes, and grants on the old object to the new object.

- Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

Example:

To rename table emp2 to employee2 give the following command.

```
rename emp2 to employee2
```

## Drop

Use the drop statement to drop tables, functions, procedures, packages, views, synonym, sequences, table spaces etc.

Example: The following command drops table emp2

```
drop table emp2;
```

If emp2 table is having primary key constraint, to which other tables refer to, then you have to first drop referential integrity constraint and then drop the table. Or if you want to drop table by dropping the referential constraints then give the following command

```
drop table emp2 cascade constraints;
```

## Truncate

Use the Truncate statement to delete all the rows from table permanently. It is same as “DELETE FROM <table\_name>” except

- Truncate does not generate any rollback data hence, it cannot be roll backed.
- If any delete triggers are defined on the table. Then the triggers are not fired
- It deallocates free extents from the table. So that the free space can be use by other tables.

Example

```
truncate table emp;
```

If you do not want free space and keep it with the table. Then specify the REUSE storage clause like this

```
truncate table emp reuse storage;
```

## Transaction Control Language (TCL)

Transaction control statements manage changes made by DML statements.

### What is a Transaction?

A transaction is a set of SQL statements which Oracle treats as a Single Unit. i.e. all the statements should execute successfully or none of the statements should execute.

To control transactions Oracle does not make permanent any DML statements unless you commit it. If you don't commit the transaction and power goes off or system crashes then the transaction is rolled back.

TCL Statements available in Oracle are

**COMMIT** : Make changes done in transaction permanent.

**ROLLBACK**: Rollbacks the state of database to the last commit point.

**SAVEPOINT**: Use to specify a point in transaction to which later you can rollback.

### COMMIT

To make the changes done in a transaction permanent issues the COMMIT statement.

The syntax of COMMIT Statement is

```
COMMIT [WORK] [COMMENT 'your comment'];
```

WORK is optional.

COMMENT is also optional, specify this if you want to identify this transaction in data dictionary DBA\_2PC\_PENDING.

Example

```
insert into emp (empno,ename,sal) values (101,'Abid',2300);
```

```
commit;
```

### ROLLBACK

To rollback the changes done in a transaction give rollback statement. Rollback restore the state of the database to the last commit point.

Example:

delete from emp;

rollback; /\* undo the changes \*/

## SAVEPOINT

Specify a point in a transaction to which later you can roll back.

Example

```
insert into emp (empno,ename,sal) values (109,'Sami',3000);
savepoint a;
insert into dept values (10,'Sales','Hyd');
savepoint b;
insert into salgrade values ('III',9000,12000);
```

Now if you give

```
rollback to a;
```

Then row from salgrade table and dept will be roll backed. At this point you can commit the row inserted into emp table or rollback the transaction.

If you give

```
rollback to b;
```

Then row inserted into salgrade table will be roll backed. At this point you can commit the row inserted into dept table and emp table or rollback to savepoint a or completely roll backed the transaction.

If you give

```
rollback;
```

Then the whole transactions is roll backed.

If you give

```
commit;
```

Then the whole transaction is committed and all savepoints are removed.

# How to Grant and Revoke privileges in Oracle

## Data Control Language (DCL) Statements

Data Control Language Statements are used to grant privileges on tables, views, sequences, synonyms, procedures to other users or roles.

The DCL statements are

- GRANT** : Use to grant privileges to other users or roles.
- REVOKE** : Use to take back privileges granted to other users and roles.

Privileges are of two types:

- System Privileges
- Object privileges

System Privileges are normally granted by a DBA to users. Examples of system privileges are CREATE SESSION, CREATE TABLE, and CREATE USER etc.

Object privileges means privileges on objects such as tables, views, synonyms, procedure. These are granted by owner of the object.

Object Privileges are:

ALTER	Change the table definition with the ALTER TABLE statement.
DELETE	Remove rows from the table with the DELETE statement.  Note: You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.
SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement.  Note: You must grant the SELECT privilege on the table along with the UPDATE privilege.

### Grant

Grant is use to grant privileges on tables, view, procedure to other users or roles.



Example: Suppose you own emp table. Now you want to grant select, update, insert privilege on this table to other user “SAMI”.

```
grant select, update, insert on emp to sami;
```

Suppose you want to grant all privileges on emp table to sami. Then

```
grant all on emp to sami;
```

Suppose you want to grant select privilege on emp to all other users of the database. Then

```
grant select on emp to public;
```

Suppose you want to grant update and insert privilege on only certain columns not on all the columns then include the column names in grant statement. For example you want to grant update privilege on ename column only and insert privilege on empno and ename columns only. Then give the following statement

```
grant update (ename),insert (empno, ename) on emp to sami;
```

To grant select statement on emp table to sami and to make sami be able further pass on this privilege you have to give WITH GRANT OPTION clause in GRANT statement like this.

```
grant select on emp to sami with grant option;
```

## REVOKE

Use to revoke privileges already granted to other users.

For example to revoke select, update, insert privilege you have granted to Sami then give the following statement.

```
revoke select, update, insert on emp from sami;
```

To revoke select statement on emp granted to public give the following command:

```
revoke select on emp from public;
```

To revoke update privilege on ename column and insert privilege on empno and ename columns give the following revoke statement.

```
revoke update, insert on emp from sami;
```



Note: You cannot take back column level privileges. Suppose you just want to take back insert privilege on ename column then you have to first take back the whole insert privilege and then grant privilege on empno column.

## ROLES

A role is a group of Privileges. A role is very handy in managing privileges, particularly in such situation when number of users should have the same set of privileges.

For example you have four users: Sami, Scott, Ashi, Tanya in the database. To these users you want to grant select, update privilege on emp table, select, delete privilege on dept table. To do this first create a role by giving the following statement

```
create role clerks
```

Then grant privileges to this role.

```
grant select,update on emp to clerks;  
grant select,delete on dept to clerks;
```

Now grant this clerks role to users like this

```
grant clerks to sami, scott, ashi, tanya ;
```

Now Sami, Scott, Ashi and Tanya have all the privileges granted on clerks role.

Suppose after one month you want grant delete on privilege on emp table all these users then just grant this privilege to clerks role and automatically all the users will have the privilege.

```
grant delete on emp to clerks;
```

If you want to take back update privilege on emp table from these users just take it back from clerks role.

```
revoke update on emp from clerks;
```

To Drop a role

```
Drop role clerks;
```

## LISTING INFORMATION ABOUT PRIVILEGES

To see which table privileges are granted by you to other users.

```
SELECT * FROM USER_TAB_PRIVS_MADE
```

To see which table privileges are granted to you by other users

```
SELECT * FROM USER_TAB_PRIVS_RECD;
```

To see which column level privileges are granted by you to other users.

```
SELECT * FROM USER_COL_PRIVS_MADE
```

To see which column level privileges are granted to you by other users

```
SELECT * FROM USER_COL_PRIVS_RECD;
```

To see which privileges are granted to roles

```
SELECT * FROM USER_ROLE_PRIVS;
```

## How to use Primary key, Foreign Key, Check, Not Null, Unique Integrity constraints in Oracle

### INTEGRITY CONSTRAINTS

Integrity Constraints are used to prevent entry of invalid information into tables. There are five Integrity Constraints available in Oracle. They are:

- Not Null
- Primary Key
- Foreign Key
- Check
- Unique

### Not Null

By default all columns in a table can contain null values. If you want to ensure that a column must always have a value, i.e. it should not be left blank, then define a NOT NULL constraint on it.

Always be careful in defining NOT NULL constraint on columns, for example in employee table some employees might have commission and some employees might not have any commission. If you put





NOT NULL constraint on COMM column then you will not be able insert rows for those employees whose commission is null. Only put NOT NULL constraint on those column which are essential for example in EMP table ENAME column is a good candidate for NOT NULL constraint.

## Primary Key

Each table can have one primary key, which uniquely identifies each row in a table and ensures that no duplicate rows exist. Use the following guidelines when selecting a primary key:

- Whenever practical, use a column containing a sequence number. It is a simple way to satisfy all the other guidelines.
- Minimize your use of composite primary keys. Although composite primary keys are allowed, they do not satisfy all of the other recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.
- Choose a column whose data values are unique, because the purpose of a primary key is to uniquely identify each row of the table.
- Choose a column whose data values are never changed. A primary key value is only used to identify a row in the table, and its data should never be used for any other purpose. Therefore, primary key values should rarely or never be changed.
- Choose a column that does not contain any nulls. A PRIMARY KEY constraint, by definition, does not allow any row to contain a null in any column that is part of the primary key.
- Choose a column that is short and numeric. Short primary keys are easy to type. You can use sequence numbers to easily generate numeric primary keys.

For example in EMP table EMPNO column is a good candidate for PRIMARY KEY.

To define a primary key on a table give the following command.

```
alter table emp add constraint empck primary key (empno);
```

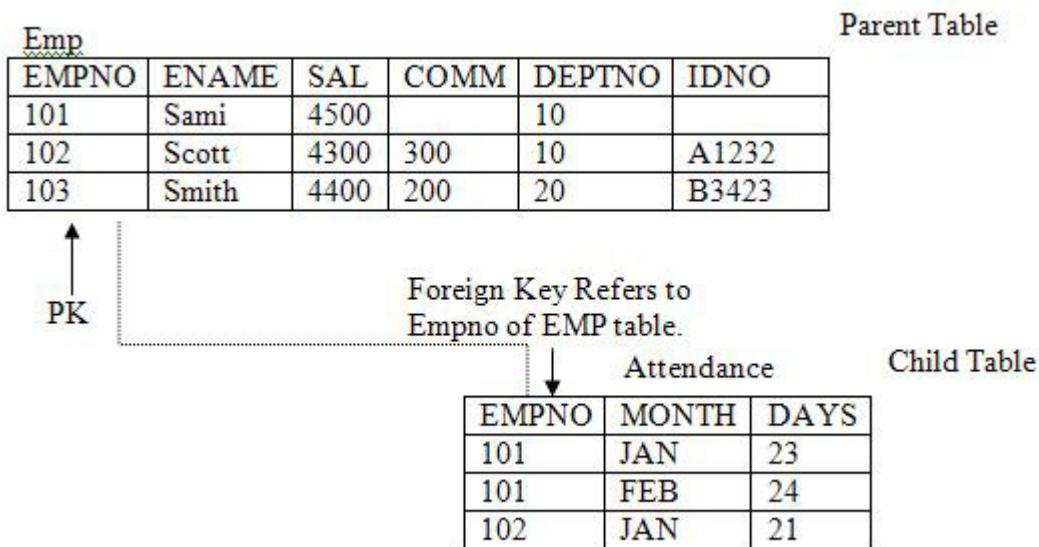
The above command will succeed only if the existing values are compliant i.e. no duplicates are there in EMPNO column. If EMPNO column contains any duplicate value then the above command fails and Oracle returns an error indicating of non compliant values.

Whenever you define a PRIMARY KEY Oracle automatically creates a index on that column. If an Index already exist on that column then Oracle uses that index.

## FOREIGN KEY

On whichever column you put FOREIGN KEY constraint then the values in that column must refer to existing values in the other table. A foreign key column can refer to primary key or unique key column of other tables. This Primary key and Foreign key relationship is also known as PARENT-CHILD relationship i.e. the table which has Primary Key is known as PARENT table and the table which has Foreign key is known as CHILD table. This relationship is also known as REFERENTIAL INTEGRITY.

The following shows an example of parent child relationship:



Here EMPNO in attendance table is a foreign key referring to EMPNO of EMP table.

```
alter table attendance add constraint empno_fk
foreign key (empno) references emp(empno);
```

The above command succeeds only if EMPNO column in ATTENDANCE table contains values which are existing in EMPNO column of EMP table. If any value does not exist then the above statement fails and Oracle returns an error indicating non compliant values.

Some points to remember for referential integrity

- You cannot delete a parent record if any existing child record is there. If you have to first delete the child record before deleting the parent record. In the above example you cannot delete row of employee no. 101 since it's child exist in the ATTENDANCE table. However, you can delete the row of employee no. 103 since no child record exist for this employee in ATTENDANCE table. If you define the FOREIGN KEY with ON DELETE CASCADE option then you can delete the parent record and if any child record exist it will be automatically deleted.

To define a foreign key constraint with ON DELETE CASCADE option give the following command.

```
ALTER TABLE attendance ADD CONSTRAINT empno_fk
FOREIGN KEY (empno) REFERENCES emp(empno)
ON DELETE CASCADE;
```

From Oracle version 9i, Oracle has also given a new feature i.e. ON DELETE SET NULL . That is it sets the value for foreign key to null whenever the parent record is deleted.



To define a foreign key constraint with ON DELETE SET NULL option give the following command.

```
ALTER TABLE attendance ADD CONSTRAINT empno_fk  
FOREIGN KEY (empno) REFERENCES emp(empno)  
ON DELETE SET NULL;
```

- You also cannot drop the parent table without first dropping the FOREIGN KEY constraint from attendance table. However if you give CASCADE CONSTRAINTS option in DROP TABLE statement then Oracle will automatically drop the references and then drops the table.

## CHECK

Use the check constraint to validate values entered into a column. For example in the above ATTENDANCE table, the DAYS column should not contain any value more than 31. For this you can define a CHECK constraint as given below:

```
alter table attendance add constraint dayscheck  
check (days <= 31);
```

Similarly if you want the salaries entered in to SAL column of employee table should be between 1000 and 20000 then you can define a CHECK constraint on EMP table as follows

```
alter table emp add constraint sal_check  
check (sal between 1000 and 20000);
```

You can define as many check constraints on a single column as you want there are no restrictions on number of check constraints.

## UNIQUE KEY

Unique Key constraint is same as primary key i.e. it does not accept duplicate values, except the following differences

- There can be only one Primary key per table. Whereas, you can have as many Unique Keys per table as you want.
- Primary key does not accept NULL values whereas; unique key columns can be left blank.
- You can also refer to Unique key from Foreign key of other tables.

On which columns you should put Unique Key Constraint?

It depends on situations, first situation is suppose you have already defined a Primary key constraint on one column and now you have another column which also should not contain any duplicate values, Since a table can have only one primary key, you can define Unique Key constraint on these columns. Second situation is when a column should not contain any duplicate value but it should also be left



blank. For example in the EMP table IDNO is a good candidate for Unique Key because all the IDNO's are unique but some employees might not have ID Card so you want to leave this column blank.

To define a UNIQUE KEY constraint on an existing table give the following command.

```
alter table emp add constraint id_unique unique (idno);
```

Again the above command will execute successfully if IDNO column contains complying values otherwise you have to remove non complying values and then add the constraint.

## Default Values and Managing Constraints in Oracle

### DEFAULT

You can also specify the DEFAULT value for columns i.e. when user does not enter anything in that column then that column will have the default value. For example in EMP table suppose most of the employees are from Hyderabad, then you can put this as default value for CITY column. Then while inserting records if user doesn't enter anything in the CITY column then the city column will have Hyderabad.

To define default value for columns create the table as given below

```
create table emp (empno number(5),
                 name varchar2(20),
                 sal number(10,2),
                 city varchar2(20) default 'Hyd');
```

Now, when user inserts record like this

```
insert into emp values (101,'Sami',2000,'Bom');
```

Then the city column will have value 'Bom '. But when user inserts a record like this

```
insert into emp (empno,name,sal) values (102,'Ashi',4000);
```

Then the city column will have value 'Hyd'. Since it is the default.

Example:

Defining Constraints in CREATE TABLE statement:

```
create table emp (empno number(5) constraint emppk
    Primary key,
    ename varchar2(20) constraint namenn
    not null,
    sal number(10,2) constraint salcheck
    check (sal between 1000 and 20000)
    idno varchar2(20) constraint id_unique
    unique );
```

```
create table attendance (empno number(5) constraint empfk
    references emp (empno)
    on delete cascade,
    month varchar2(10),
    days number(2) constraint dayscheck
    check (days <= 31) );
```

The name of the constraints are optional. If you don't define the names then oracle generates the names randomly like 'SYS\_C1234'

Another way of defining constraint in CREATE TABLE statement:

```
create table emp (empno number(5),
    ename varchar2(20) not null,
    sal number(10,2),
    idno varchar2(20),
    constraint emppk Primary key (empno)
    constraint salcheck check (sal between 1000 and 20000)
    constraint id_unique unique (idno) );
```

```
create table attendance (empno number(5),
    month varchar2(10),
    days number(2),
    constraint empfk foreign key (empno)
    references emp (empno)
    on delete cascade
    constraint dayscheck
    check (days <= 31) );
```

## Deferring Constraint Checks

You may wish to defer constraint checks on UNIQUE and FOREIGN keys if the data you are working with has any of the following characteristics:

- Tables are snapshots
- Tables that contain a large amount of data being manipulated by another application, which may or may not return the data in the same order
- Update cascade operations on foreign keys

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

Ensure Constraints Are Created Deferrable.

After you have identified and selected the appropriate tables, make sure their FOREIGN, UNIQUE and PRIMARY key constraints are created deferrable. You can do so by issuing a statement similar to the following:

```
create table attendance (empno number(5),
month varchar2(10),
days number(2),
constraint empfk foreign key (empno)
references emp (empno)
on delete cascade
DEFERRABLE
constraint dayscheck
check (days <= 31) );
```

Now give the following statement

```
set constraint empfk deferred;
update attendance set empno=104 where empno=102;
insert into emp values (104,'Sami',4000,'A123');
commit;
```

You can check for constraint violations before committing by issuing the SET CONSTRAINTS ALL IMMEDIATE statement just before issuing the COMMIT. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

## ENABLING AND DISABLING CONSTRAINTS

You can enable and disable constraints at any time.

To enable and disable constraints the syntax is:

```
ALTER TABLE <TABLE_NAME> ENABLE/DISABLE
CONSTRAINT <CONSTRAINT_NAME>
```

For example to disable primary key of EMP table give the following statement

```
alter table emp disable constraint emppk;
```

And to enable it again, give the following statement

```
alter table emp enable constraint emppk;
```

## Dropping constraints

You can drop constraint by using ALTER TABLE DROP constraint statement.

For example to drop Unique constraint from emp table, give the following statement

```
alter table emp drop constraint id_unique;
```

To drop primary key constraint from emp table.

```
alter table emp drop constraint emppk;
```

The above statement will succeed only if the foreign key is first dropped otherwise you have to first drop the foreign key and then drop the primary key. If you want to drop primary key along with the foreign key in one statement then CASCADE CONSTRAINT statement like this:

```
alter table emp drop constraint emppk cascade;
```

## Viewing Information about constraints

To see information about constraints, you can query the following data dictionary tables.

```
select * from user_constraints;  
select * from user_cons_columns;
```

# Default Values and Managing Constraints in Oracle

## DEFAULT

You can also specify the DEFAULT value for columns i.e. when user does not enter anything in that column then that column will have the default value. For example in EMP table suppose most of the employees are from Hyderabad, then you can put this as default value for CITY column. Then while inserting records if user doesn't enter anything in the CITY column then the city column will have Hyderabad.

To define default value for columns create the table as given below:

```
create table emp (empno number(5),
                 name varchar2(20),
                 sal number(10,2),
                 city varchar2(20) default 'Hyd');
```

Now, when user inserts record like this

```
insert into emp values (101,'Sami',2000,'Bom');
```

Then the city column will have value 'Bom '. But when user inserts a record like this

```
insert into emp (empno,name,sal) values (102,'Ashi',4000);
```

Then the city column will have value 'Hyd'. Since it is the default.

Examples:

Defining Constraints in CREATE TABLE statement:

```
create table emp (empno number(5) constraint emp_pk
                 Primary key,
                 ename varchar2(20) constraint nam_enn
                 not null,
                 sal number(10,2) constraint sal_check
                 check (sal between 1000 and 20000)
                 idno varchar2(20) constraint id_unique
                 unique );
```

```
create table attendance (empno number(5) constraint emp_fk
                         references emp (empno)
                         on delete cascade,
                         month varchar2(10),
                         days number(2) constraint days_check
                         check (days <= 31) );
```

The name of the constraints is optional. If you don't define the names then oracle generates the names randomly like 'SYS\_C1234'

Another way of defining constraint in CREATE TABLE statement:

```
create table emp (empno number(5),
                 ename varchar2(20) not null,
                 sal number(10,2),
                 idno varchar2(20),
                 constraint emp_pk Primary key (empno)
```



```
constraint salcheck check (sal between 1000 and 20000)
constraint id_unique unique (idno) );
```

```
create table attendance (empno number(5),
                        month varchar2(10),
                        days number(2),
constraint empfk foreign key (empno)
references emp (empno)
on delete cascade
constraint dayscheck
check (days <= 31) );
```

### Deferring Constraint Checks

You may wish to defer constraint checks on UNIQUE and FOREIGN keys if the data you are working with has any of the following characteristics:

- Tables are snapshots
- Tables that contain a large amount of data being manipulated by another application, which may or may not return the data in the same order
- Update cascade operations on foreign keys

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

Ensure Constraints Are Created Deferrable

After you have identified and selected the appropriate tables, make sure their FOREIGN, UNIQUE and PRIMARY key constraints are created deferrable. You can do so by issuing a statement similar to the following:

```
create table attendance (empno number(5),
                        month varchar2(10),
                        days number(2),
constraint empfk foreign key (empno)
references emp (empno)
on delete cascade
DEFERRABLE
constraint dayscheck
check (days <= 31) );
```

Now give the following statement

```
set constraint empfk deferred;
update attendance set empno=104 where empno=102;
```

```
insert into emp values (104,'Sami',4000,'A123');  
commit;
```

You can check for constraint violations before committing by issuing the SET CONSTRAINTS ALL IMMEDIATE statement just before issuing the COMMIT. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

## ENABLING AND DISABLING CONSTRAINTS

You can enable and disable constraints at any time.

To enable and disable constraints the syntax is

```
ALTER TABLE <TABLE_NAME> ENABLE/DISABLE  
CONSTRAINT <CONSTRAINT_NAME>
```

For example to disable primary key of EMP table give the following statement

```
alter table emp disable constraint emppk;
```

And to enable it again, give the following statement

```
alter table emp enable constraint emppk;
```

## Dropping constraints

You can drop constraint by using ALTER TABLE DROP constraint statement.

For example to drop Unique constraint from emp table, give the following statement

```
alter table emp drop constraint id_unique;
```

To drop primary key constraint from emp table.

```
alter table emp drop constraint emppk;
```

The above statement will succeed only if the foreign key is first dropped otherwise you have to first drop the foreign key and then drop the primary key. If you want to drop primary key along with the foreign key in one statement then CASCADE CONSTRAINT statement like this

```
alter table emp drop constraint emppk cascade;
```

## Viewing Information about constraints

To see information about constraints, you can query the following data dictionary tables.

```
select * from user_constraints;  
select * from user_cons_columns;
```

## Default Values and Managing Constraints in Oracle

### DEFAULT

You can also specify the DEFAULT value for columns i.e. when user does not enter anything in that column then that column will have the default value. For example in EMP table suppose most of the employees are from Hyderabad, then you can put this as default value for CITY column. Then while inserting records if user doesn't enter anything in the CITY column then the city column will have Hyderabad.

To define default value for columns create the table as given below

```
create table emp (empno number(5),  
                 name varchar2(20),  
                 sal number(10,2),  
                 city varchar2(20) default 'Hyd');
```

Now, when user inserts record like this-

```
insert into emp values (101,'Sami',2000,'Bom');
```

Then the city column will have value 'Bom '. But when user inserts a record like this

```
insert into emp (empno,name,sal) values (102,'Ashi',4000);
```

Then the city column will have value 'Hyd'. Since it is the default.

Examples:

Defining Constraints in CREATE TABLE statement:

```
create table emp (empno number(5) constraint emp_pk  
                 Primary key,  
                 ename varchar2(20) constraint namenn  
                 not null,  
                 sal number(10,2) constraint salcheck  
                 check (sal between 1000 and 20000))
```

```
idno varchar2(20) constraint id_unique  
unique );
```

```
create table attendance (empno number(5) constraint empfk  
references emp (empno)  
on delete cascade,  
month varchar2(10),  
days number(2) constraint dayscheck  
check (days <= 31) );
```

The name of the constraints is optional. If you don't define the names then oracle generates the names randomly like 'SYS\_C1234'

Another way of defining constraint in CREATE TABLE statement:

```
create table emp (empno number(5),  
ename varchar2(20) not null,  
sal number(10,2),  
idno varchar2(20),  
constraint empfk Primary key (empno)  
constraint salcheck check (sal between 1000 and 20000)  
constraint id_unique unique (idno) );
```

```
create table attendance (empno number(5),  
month varchar2(10),  
days number(2),  
constraint empfk foreign key (empno)  
references emp (empno)  
on delete cascade  
constraint dayscheck  
check (days <= 31) );
```

## Deferring Constraint Checks

You may wish to defer constraint checks on UNIQUE and FOREIGN keys if the data you are working with has any of the following characteristics:

- Tables are snapshots
- Tables that contain a large amount of data being manipulated by another application, which may or may not return the data in the same order
- Update cascade operations on foreign keys

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

Ensure Constraints Are Created Deferrable.

After you have identified and selected the appropriate tables, make sure their FOREIGN, UNIQUE and PRIMARY key constraints are created deferrable. You can do so by issuing a statement similar to the following:

```
create table attendance (empno number(5),
  month varchar2(10),
  days number(2),
  constraint empfk foreign key (empno)
  references emp (empno)
  on delete cascade
  DEFERRABLE
  constraint dayscheck
  check (days <= 31) );
```

Now give the following statement-

```
set constraint empfk deferred;
update attendance set empno=104 where empno=102;
insert into emp values (104,'Sami',4000,'A123');
commit;
```

You can check for constraint violations before committing by issuing the SET CONSTRAINTS ALL IMMEDIATE statement just before issuing the COMMIT. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

## ENABLING AND DISABLING CONSTRAINTS

You can enable and disable constraints at any time.

To enable and disable constraints the syntax is

```
ALTER TABLE <TABLE_NAME> ENABLE/DISABLE
  CONSTRAINT <CONSTRAINT_NAME>
```

For example to disable primary key of EMP table give the following statement

```
alter table emp disable constraint empfk;
```

And to enable it again, give the following statement

```
alter table emp enable constraint empfk;
```

## Dropping constraints

You can drop constraint by using ALTER TABLE DROP constraint statement.

For example to drop Unique constraint from emp table, give the following statement-

```
alter table emp drop constraint id_unique;
```

To drop primary key constraint from emp table.

```
alter table emp drop constraint emppk;
```

The above statement will succeed only if the foreign key is first dropped otherwise you have to first drop the foreign key and then drop the primary key. If you want to drop primary key along with the foreign key in one statement then CASCADE CONSTRAINT statement like this:

```
alter table emp drop constraint emppk cascade;
```

## Viewing Information about constraints

To see information about constraints, you can query the following data dictionary tables.

```
select * from user_constraints;  
select * from user_cons_columns;
```

# How to Create and Manage Views in Oracle

## Views

Views are known as logical tables. They represent the data of one or more tables. A view derives its data from the tables on which it is based. These tables are called base tables. Views can be based on actual tables or another view also.

Whatever DML operations you performed on a view they actually affect the base table of the view. You can treat views same as any other table. You can Query, Insert, Update and delete from views, just as any other table.

Views are very powerful and handy since they can be treated just like any other table but do not occupy the space of a table.

The following sections explain how to create, replace, and drop views using SQL commands.

## Creating Views

Suppose we have EMP and DEPT table. To see the empno, ename, sal, deptno, department name and location we have to give a join query like this.

```
select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc  
      From emp e, dept d where e.deptno=d.deptno;
```

So everytime we want to see emp details and department names where they are working we have to give a long join query. Instead of giving this join query again and again, we can create a view on this table by using a CREATE VIEW command given below

```
create view emp_det as select e.empno,  
e.ename,e.sal,e.deptno,d.dname,d.loc  
      from emp e, dept d where e.deptno=d.deptno;
```

Now to see the employee details and department names we don't have to give a join query, we can just type the following simple query.

```
select * from emp_det;
```

This will show same result as you have type the long join query. Now you can treat this EMP\_DET view same as any other table.

For example, suppose all the employee working in Department No. 10 belongs to accounts department and most of the time you deal with these people. So every time you have to give a DML or Select statement you have to give a WHERE condition like .....WHERE DEPTNO=10. To avoid this, you can create a view as given below:

```
CREATE VIEW accounts_staff AS  
  SELECT Empno, Ename, Deptno  
  FROM Emp  
  WHERE Deptno = 10  
  WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Now to see the account people you don't have to give a query with where condition you can just type the following query.

```
select * from accounts_staff;
```

```
select sum(sal) from accountst_staff;
```

```
select max(sal) from accounts_staff;
```

As you can see how views make things easier.



The query that defines the `ACCOUNTS_STAFF` view references only rows in department 10. Furthermore, `WITH CHECK OPTION` creates the view with the constraint that `INSERT` and `UPDATE` statements issued against the view are not allowed to create or result in rows that the view cannot select.

Considering the example above, the following `INSERT` statement successfully inserts a row into the `EMP` table through the `ACCOUNTS_STAFF` view:

```
INSERT INTO Accounts_staff VALUES (110, 'ASHI', 10);
```

However, the following `INSERT` statement is rolled back and returns an error because it attempts to insert a row for department number 30, which could not be selected using the `ACCOUNTS_STAFF` view:

```
INSERT INTO Accounts_staff VALUES (111, 'SAMI', 30);
```

## Creating FORCE VIEWS

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.

## Replacing/Altering Views

To alter the definition of a view, you must replace the view using one of the following methods:

- A view can be dropped and then re-created. When a view is dropped, all grants of corresponding view privileges are revoked from roles and users. After the view is re-created, necessary privileges must be regranted.
- A view can be replaced by redefining it with a `CREATE VIEW` statement that contains the `OR REPLACE` option. This option replaces the current definition of a view, but preserves the present security authorizations.

For example, assume that you create the `ACCOUNTS_STAFF` view, as given in a previous example. You also grant several object privileges to roles and other users. However, now you realize that you must redefine the `ACCOUNTS_STAFF` view to correct the department number specified in the





WHERE clause of the defining query, because it should have been 30. To preserve the grants of object privileges that you have made, you can replace the current version of the ACCOUNTS\_STAFF view with the following statement:

```
CREATE OR REPLACE VIEW Accounts_staff AS
  SELECT Empno, Ename, Deptno
  FROM Emp
  WHERE Deptno = 30
  WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Replacing a view has the following effects:

- Replacing a view replaces the view's definition in the data dictionary. All underlying objects referenced by the view are not affected.
- If previously defined but not included in the new view definition, then the constraint associated with the WITH CHECK OPTION for a view's definition is dropped.
- All views and PL/SQL program units dependent on a replaced view become invalid.

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the EMP table using the ACCOUNTS\_STAFF view:

```
INSERT INTO Accounts_staff
  VALUES (199, 'ABID', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with WITH CHECK OPTION, then a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

The constraint created by WITH CHECK OPTION of the ACCOUNTS\_STAFF view only allows rows that have a department number of 10 to be inserted into, or updated in, the EMP table. Alternatively, assume that the ACCOUNTS\_STAFF view is defined by the following statement (that is, excluding the DEPTNO column):

```
CREATE VIEW Accounts_staff AS
  SELECT Empno, Ename
  FROM Emp
```

```
WHERE Deptno = 10  
WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Considering this view definition, you can update the EMPNO or ENAME fields of existing records, but you cannot insert rows into the EMP table through the ACCOUNTS\_STAFF view because the view does not let you alter the DEPTNO field. If you had defined a DEFAULT value of 10 on the DEPTNO field, then you could perform inserts.

If you don't want any DML operations to be performed on views, create them WITH READ ONLY option. Then no DML operations are allowed on views.

## Referencing Invalid Views

When a user attempts to reference an invalid view, Oracle returns an error message to the user.

```
ORA-04063: view 'view_name' has errors
```

This error message is returned when a view exists but is unusable due to errors in its query (whether it had errors when originally created or it was created successfully but became unusable later because underlying objects were altered or dropped).

## Dropping Views

Use the SQL command DROP VIEW to drop a view. For example:

```
DROP VIEW Accounts_staff;
```

## Modifying a Join View

Oracle allows you, with some restrictions, to modify views that involve joins. Consider the following simple view:

```
CREATE VIEW Emp_view AS  
SELECT Ename, Empno, deptno FROM Emp;
```

This view does not involve a join operation. If you issue the SQL statement:

```
UPDATE Emp_view SET Ename = 'SHAHRYAR' WHERE Empno = 109;
```

then the EMP base table that underlies the view changes, and employee 109's name changes from ASHI to SHAHRYAR in the EMP table.

However, if you create a view that involves a join operation, such as:

```
CREATE VIEW Emp_dept_view AS
SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
FROM Emp e, Dept d /* JOIN operation */
WHERE e.Deptno = d.Deptno
AND d.Loc IN ('HYD', 'BOM', 'DEL');
```

There are restrictions on modifying either the EMP or the DEPT base table through this view.

A modifiable join view is a view that contains more than one table in the top-level FROM clause of the SELECT statement, and that does not contain any of the following:

- DISTINCT operator
- Aggregate functions: AVG, COUNT, GLB, MAX, MIN, STDDEV, SUM, or VARIANCE
- Set operations: UNION, UNION ALL, INTERSECT, MINUS
- GROUP BY or HAVING clauses
- START WITH or CONNECT BY clauses
- ROWNUM pseudo column

Any UPDATE, INSERT, or DELETE statement on a join view can modify only one underlying base table.

The following example shows an UPDATE statement that successfully modifies the EMP\_DEPT\_VIEW view:

```
UPDATE Emp_dept_view
SET Sal = Sal * 1.10
WHERE Deptno = 10;
```

The following UPDATE statement would be disallowed on the EMP\_DEPT\_VIEW view:

```
UPDATE Emp_dept_view
SET Loc = 'BOM'
WHERE Ename = 'SAMI';
```

This statement fails with an ORA-01779 error ("cannot modify a column which maps to a non key-preserved table"), because it attempts to modify the underlying DEPT table, and the DEPT table is not key preserved in the EMP\_DEPT view.

In general, all modifiable columns of a join view must map to columns of a key-preserved table. If the view is defined using the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are not modifiable.

So, for example, if the EMP\_DEPT view were defined using WITH CHECK OPTION, then the following UPDATE statement would fail:

```
UPDATE Emp_dept_view  
SET Deptno = 10  
WHERE Ename = 'SAMI';
```

The statement fails because it is trying to update a join column.

## Deleting from a Join View

You can delete from a join view provided there is one and only one key-preserved table in the join.

The following DELETE statement works on the EMP\_DEPT view:

```
DELETE FROM Emp_dept_view  
WHERE Ename = 'SMITH';
```

This DELETE statement on the EMP\_DEPT view is legal because it can be translated to a DELETE operation on the base EMP table, and because the EMP table is the only key-preserved table in the join.

In the following view, a DELETE operation cannot be performed on the view because both E1 and E2 are key-preserved tables:

```
CREATE VIEW emp_emp AS  
SELECT e1.Ename, e2.Empno, e1.Deptno  
FROM Emp e1, Emp e2  
WHERE e1.Empno = e2.Empno;
```

If a view is defined using the WITH CHECK OPTION clause and the key-preserved table is repeated, then rows cannot be deleted from such a view. For example:

```
CREATE VIEW Emp_mgr AS  
SELECT e1.Ename, e2.Ename Mname  
FROM Emp e1, Emp e2  
WHERE e1.mgr = e2.Empno  
WITH CHECK OPTION;
```

No deletion can be performed on this view because the view involves a self-join of the table that is key preserved.

## Inserting into a Join View

The following INSERT statement on the EMP\_DEPT view succeeds, because only one key-preserved base table is being modified (EMP), and 40 is a valid DEPTNO in the DEPT table (thus satisfying the FOREIGN KEY integrity constraint on the EMP table).

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
VALUES ('ASHU', 119, 40);
```

The following INSERT statement fails for the same reason: This UPDATE on the base EMP table would fail: the FOREIGN KEY integrity constraint on the EMP table is violated.

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
VALUES ('ASHU', 110, 77);
```

The following INSERT statement fails with an ORA-01776 error ("cannot modify more than one base table through a view").

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
VALUES (110, 'TANNU', 'BOMBAY');
```

An INSERT cannot, implicitly or explicitly, refer to columns of a non-key-preserved table. If the join view is defined using the WITH CHECK OPTION clause, then you cannot perform an INSERT to it.

## Listing Information about VIEWS

To see how many views are there in your schema. Give the following query.

```
select * from user_views;
```

To see which columns are updatable in join views.

Data Dictionaries which shows which columns are updatable.

View Name	Description
USER_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the user's schema that are modifiable
DBA_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the DBA schema that are modifiable
ALL_UPDATABLE_VIEWS	Shows all columns in all tables and views that are modifiable

If you are in doubt whether a view is modifiable, then you can SELECT from the view USER\_UPDATABLE\_COLUMNS to see if it is. For example:

```
SELECT * FROM USER_UPDATABLE_COLUMNS WHERE TABLE_NAME =
'EMP_DEPT_VIEW';
```

This might return:

```
OWNER    TABLE_NAME  COLUMN_NAME  UPD
-----  -
SCOTT    EMP_DEPT     EMPNO        NO
```

SCOTT	EMP_DEPT	ENAME	NO
SCOTT	EMP_DEPT	DEPTNO	NO
SCOTT	EMP_DEPT	DNAME	NO
SCOTT	EMP_DEPT	LOC	NO

5 rows selected.

## Using Sequences in Oracle (Auto Increment Feature)

### SEQUENCES

A sequence is used to generate numbers in sequence. You can use sequences to insert unique values in Primary Key and Unique Key columns of tables. To create a sequence gives the CREATE SEQUENCE statement.

### CREATING SEQUENCES

```
create sequence bills
  start with 1
  increment by 1
  minvalue 1
  maxvalue 100
  cycle
  cache 10';
```

The above statement creates a sequence bills it will start with 1 and increment by 1. Its maxvalue is 100 i.e. after 100 numbers are generated it will stop if you say NOCYCLE, otherwise if you mention CYCLE then again it will start with no. 1. You can also specify NOMAXVALUE in that case the sequence will generate infinite numbers.

The CACHE option is used to cache sequence numbers in System Global Area (SGA). If you say CACHE 10 then Oracle will cache next 10 numbers in SGA. If you access a sequence number then oracle will first try to get the number from cache, if it is not found then it reads the next number from disk. Since reading the disk is time consuming rather than reading from SGA it is always recommended to cache sequence numbers in SGA. If you say NOCACHE then Oracle will not cache any numbers in SGA and every time you access the sequence number it reads the number from disk.

### Accessing Sequence Numbers

To generate Sequence Numbers you can use NEXTVAL and CURRVAL for example to get the next sequence number of bills sequence type the following command.

```
Select bills.nextval from dual;
```

## BILLS

-----

1

NEXTVAL gives the next number in sequence. Whereas, CURRVAL returns the current number of the sequence. This is very handy in situations where you have insert records in Master Detail tables. For example to insert a record in SALES master table and SALES\_DETAILS detail table.

```
insert into sales (billno,custname,amt)
  values (bills.nextval,'Sami',2300);
```

```
insert into sales_details (billno,itemname,qty,rate) values
  (bills.currval,'Onida',10,13400);
```

Sequences are usually used as DEFAULT Values for table columns to automatically insert unique numbers. For Example,

```
create table invoices (invoice_no number(10) default bills.nextval,
  invoice_date date default sysdate,
  customer varchar2(100),
  invoice_amt number(12,2));
```

Now whenever you insert rows into invoices table ommiting invoice\_no as follows

```
insert into invoices (customer,invoice_amt) values ('A to Z Traders',5000);
```

Oracle will insert invoice\_no from bills sequence

## ALTERING SEQUENCES

To alter sequences use ALTER SEQUENCE statement. For example to alter the bill sequence MAXVALUE give the following command.

```
ALTER SEQUENCE BILLS
  MAXVALUE 200;
```

Except Starting Value, you can alter any other parameter of a sequence. To change START WITH parameter you have to drop and recreate the sequence.

## DROPPING SEQUENCES

To drop sequences use DROP SEQUENCE command. For example to drop bills sequence gives the following statement

```
drop sequence bills;
```

## Listing Information about Sequences

To see how many sequences are there in your schema and what are their settings give the following command.

```
select * from user_sequences;
```

## Using Synonyms in Oracle

### SYNONYMS

A synonym is an alias for a table, view, snapshot, sequence, procedure, function, or package.

There are two types to SYNONYMS they are

PUBLIC SYNONYM  
PRIVATE SYNONYM

If you create a synonym as public then it can be accessed by any other user with qualifying the synonym name i.e. the user doesn't have to mention the owner name while accessing the synonym. Nevertheless the other user should have proper privilege to access the synonym. Private synonyms need to be qualified with owner names.

### CREATING SYNONYMS

To create a synonym for SCOTT emp table give the following command.

```
create synonym employee for scott.emp;
```

A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced. For example, if a synonym named EMPLOYEE refers to a table or view, then the following statement is valid:

```
select * from employee;
```

Suppose you have created a function known as TODAY which returns the current date and time. Now you have granted execute permission on it to every other user of the database. Now these users can execute this function but when they call they have to give the following command:

```
select scott.today from dual;
```

Now if you create a public synonym on it then other users don't have to qualify the function name with owner's name. To define a public synonym gives the following command.



```
create public synonym today for scott.today;
```

Now the other users can simply type the following command to access the function.

```
select today from dual;
```

## Dropping Synonyms

To drop a synonym uses the DROP SYNONYM statement. For example, to drop EMPLOYEE synonym gives the statement

```
drop synonym employee;
```

## Listing information about synonyms

To see synonyms information give the following statement.

```
select * from user_synonyms;
```

# Managing Indexes and Clusters in Oracle

## INDEXES

Use indexes to speed up queries. Indexes speeds up searching of information in tables. So create indexes on those columns which are frequently used in WHERE conditions. Indexes are helpful if the operations return only small portion of data i.e. less than 15% of data is retrieved from tables.

Follow these guidelines for creating indexes

- Do not create indexes on small tables i.e. where number of rows is less. (Full table scan itself will be faster if table is small)
- Do not create indexes on those columns which contain many null values.
- Do not create BTree index on those columns which contain many repeated values. In this case create BITMAP indexes on these columns.
- Limit the number of indexes on tables because, although they speed up queries, but at the same time DML operations becomes very slow as all the indexes have to updated whenever an Update, Delete or Insert takes place on tables.

## Creating Indexes

To create an Index give the create index command. For example the following statement creates an index on empno column of emp table.

```
create index empno_ind on emp (empno);
```

If two columns are frequently used together in WHERE conditions then create a composite index on these columns. For example, suppose we use EMPNO and DEPTNO oftenly together in WHERE condition. Then create a composite index on this column as given below:

```
create index empdept_ind on emp (empno,deptno);
```

The above index will be used whenever you use empno or deptno column together, or you just use empno column in WHERE condition. The above index will not be used if you use only deptno column alone.

## BITMAP INDEXES

Create Bitmap indexes on those columns which contains many repeated values and when tables are large. City column in EMP table is a good candidate for Bitmap index because it contains many repeated values. To create a composite index gives the following command.

```
create bitmap index city_ind on emp (city);
```

## FUNCTION BASED Indexes

Function Based indexes are built on expressions rather than on column values. For example if you frequently use the expression SAL+COMM in WHERE conditions then create a Function base index on this expression like this:

```
create index salcomm_ind on emp (sal+comm);
```

Now, whenever you use the expression SAL+COMM in where condition then oracle will use SALCOMM\_IND index.

## DROPPING INDEXES

To drop indexes use DROP INDEX statement. For example to drop SALCOMM\_IND give the following statement:

```
drop index salcomm_ind;
```

## Listing Information about indexes

To see how many indexes are there in your schema and its information, give the following statement:

```
select * from user_indexes;
```

## CLUSTERS

If you two or more tables are joined together on a single column and most of the time you issue join queries on them, then consider creating a cluster of these tables.

A cluster is a group of tables that share the same data blocks i.e. all the tables are physically stored together.

For example EMP and DEPT table are joined on DEPTNO column. If you cluster them, Oracle physically stores all rows for each department from both the emp and dept tables in the same data blocks.

- Since cluster stores related rows of different tables in same data blocks, Disk I/O is reduced and access time improves for joins of clustered tables.
- Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value.

Therefore, less storage might be required to store related table and index data in a cluster than is necessary in non-clustered table format.

### CREATING A CLUSTER

To create clustered tables, first, create a cluster and create index on it. Then create tables in it.

For example to create a cluster of EMP and DEPT tables in which the DEPTNO will be cluster key, first create the cluster by typing the following command.

```
create cluster emp_dept (deptno number(2));
```

Then create index on it.

```
create index on cluster emp_dept;
```

Now create table in the cluster like this

```
create table dept (deptno number(2),  
                 name varchar2(20),  
                 loc varchar2(20))  
                 cluster emp_dept (deptno);
```

```
create table emp (empno number(5),  
                 name varchar2(20),
```

```
sal number(10,2),  
deptno number(2)) cluster emp_dept (deptno)
```

## Dropping Clusters

To drop a cluster use DROP CLUSTER statement. For example to drop the emp\_dept cluster give the following command:

```
drop cluster emp_dept;
```

This will drop the cluster, if the cluster is empty i.e. no tables are existing it. If tables are there in the cluster first drop the tables and then drop the cluster. If you want to drop the cluster even when tables are there then give the following command.

```
drop cluster emp_dept including tables;
```

## Listing Information about Clusters

To see how many clusters are there in your schema, give the following statement:

```
select * from user_clusters;
```

To see which tables are parts of a cluster, give the following command:

```
select * from tab
```

TABLE_NAME	TYPE	CLUSTER_ID
EMP	TABLE	1
SALGRADE	TABLE	
CUSTOMER	TABLE	
DEPT	TABLE	1

In the above example notice the CLUSTER\_ID column, for EMP and DEPT table the cluster\_id is 1. That means these tables are in cluster whose cluster\_id is 1. You can see the cluster\_id's name in USER\_CLUSTERS table.

## Assignment

1. For the following relation schema:

*employee(employee-name, street, city)*

*works(employee-name, company-name, salary)*

*company(company-name, city)*

*manages(employee-name, manager-name)*

Give an expression in SQL for each of the following queries:

- a) Find the names, street address, and cities of residence for all employees who work for 'First Bank Corporation' and earn more than \$10,000.
- b) Find the names of all employees in the database who live in the same cities as the companies for which they work.
- c) Find the names of all employees in the database who live in the same cities and on the same streets as do their managers.
- d) Find the names of all employees in the database who do not work for 'First Bank Corporation'. Assume that all people work for exactly one company.
- e) Find the names of all employees in the database who earn more than every employee of 'Small Bank Corporation'. Assume that all people work for at most one company.
- f) Assume that the companies may be located in several cities. Find all companies located in every city in which 'Small Bank Corporation' is located.
- g) Find the names of all employees who earn more than the average salary of all employees of their company. Assume that all people work for at most one company.
- h) Find the name of the company that has the smallest payroll.

2. Let  $R=(A, B, C)$ ,  $S=(C, D, E)$  and let  $q$  and  $r$  be relations on schema  $R$  and  $s$  be a relation on schema  $S$ . Convert the following queries to SQL:

- a)  $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r \wedge b = 10) \}$
- b)  $q - r$
- c)  $\{ t \mid \exists p \in r \exists q \in s (t[A] = p[A] \wedge t[E] = q[E] \wedge p[C] = q[D]) \}$
- d)  $\angle_A, c(r) \bowtie \angle_{C, D}(s)$
- e)  $r \cdot s$

## PL/SQL (procedural language extension to Structured Query Language)

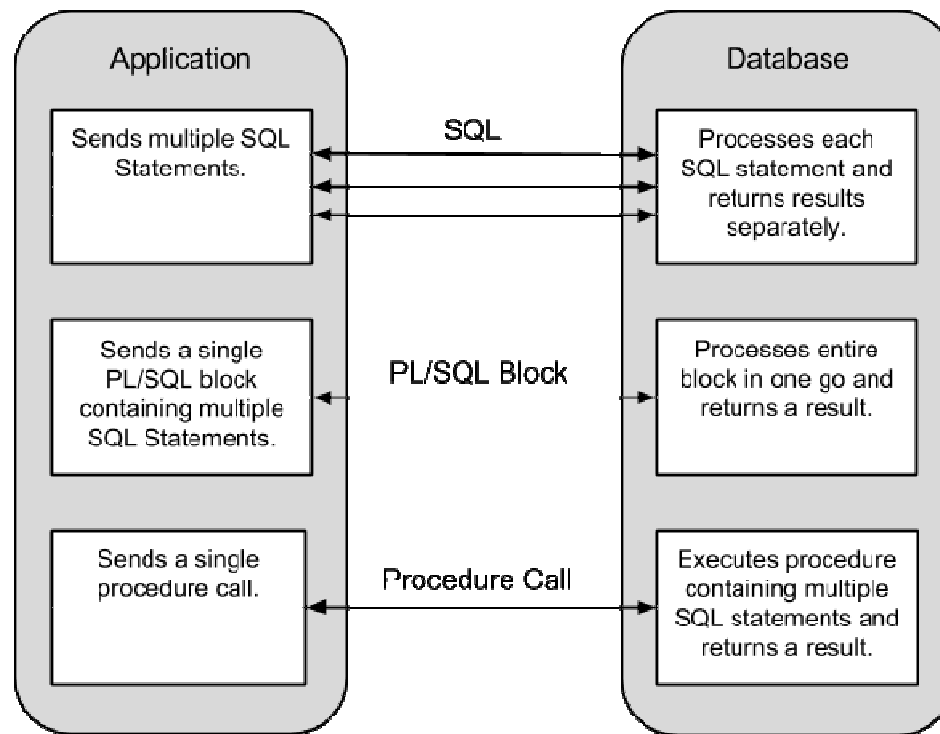
In Oracle database management, PL/SQL is a procedural language extension to Structured Query Language (SQL). The purpose of PL/SQL is to combine database language and procedural programming language. The basic unit in PL/SQL is called a block, which is made up of three parts: a declarative part, an executable part, and an exception-building part.

Because PL/SQL allows you to mix SQL statements with procedural constructs, it is possible to use PL/SQL blocks and subprograms to group SQL statements before sending them to Oracle for execution. Without PL/SQL, Oracle must process SQL statements one at a time and, in a network environment, this can affect traffic flow and slow down response time. PL/SQL blocks can be compiled once and stored in executable form to improve response time.

A PL/SQL program that is stored in a database in compiled form and can be called by name is referred to as a stored procedure. A PL/SQL stored procedure that is implicitly started when an INSERT, UPDATE or DELETE statement is issued against an associated table is called a trigger.

### What is so great about PL/SQL anyway?

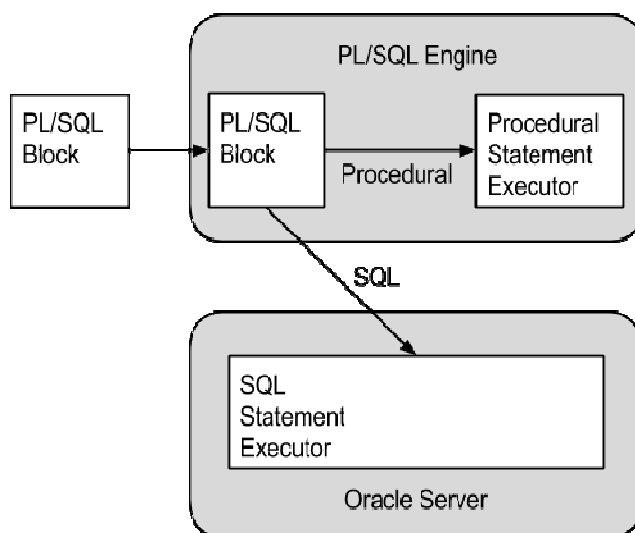
- PL/SQL is a procedural extension of SQL, making it extremely simple to write procedural code that includes SQL as if it were a single language. In comparison, most other programming languages require mapping data types, preparing statements and processing result sets, all of which require knowledge of specific APIs.
- The data types in PL/SQL are a super-set of those in the database, so you rarely need to perform data type conversions when using PL/SQL. Ask your average Java or .NET programmer how they find handling date values coming from a database. They can only wish for the simplicity of PL/SQL.
- When coding business logic in middle tier applications, a single business transaction may be made up of multiple interactions between the application server and the database. This adds a significant overhead associated with network traffic. In comparison, building all the business logic as PL/SQL in the database means client code needs only a single database call per transaction, reducing the network overhead significantly.



- Oracle is a multi-platform database, making PL/SQL and incredibly portable language. If your business logic is located in the database, you are protecting yourself from operating system lock-in.
- Programming languages go in and out of fashion continually. Over the last 35+ years Oracle databases have remained part of the enterprise landscape. Suggesting that any language is a safer bet than PL/SQL is rather naive. Placing your business logic in the database makes changing your client layer much simpler if you like to follow fashion.
- Centralizing application logic enables a higher degree of security and productivity. The use of Application Program Interfaces (APIs) can abstract complex data structures and security implementations from client application developers, leaving them free to do what they do best.

## PL/SQL Architecture

The PL/SQL language is actually made up of two distinct languages. Procedural code is executed by the PL/SQL engine, while SQL is sent to the SQL statement executor.



For the most part, the tight binding between these two languages make PL/SQL look like a single language to most developers.

## Overview of PL/SQL Elements

### Blocks

Blocks are the organizational unit for all PL/SQL code, whether it is in the form of an anonymous block, procedure, function, trigger or type. A PL/SQL block is made up of three sections (declaration, executable and exception), of which only the executable section is mandatory.

```

[DECLARE
  -- delarations]
BEGIN
  -- statements
[EXCEPTION
  -- handlers
END;
  
```

Based on this definition, the simplest valid block is shown below, but it doesn't do anything.

```

BEGIN
  NULL;
  
```



```
END;
```

The optional declaration section allows variables, types, procedures and functions do be defined for use within the block. The scope of these declarations is limited to the code within the block itself, or any nested blocks or procedure calls. The limited scope of variable declarations is shown by the following two examples. In the first, a variable is declared in the outer block and is referenced successfully in a nested block. In the second, a variable is declared in a nested block and referenced from the outer block, resulting in an error as the variable is out of scope.

```
DECLARE  
  l_number NUMBER;
```

```
BEGIN
```

```
  l_number := 1;
```

```
  BEGIN
```

```
    l_number := 2;
```

```
  END;
```

```
END;
```

```
/
```

PL/SQL procedure successfully completed.

```
BEGIN
```

```
  DECLARE
```

```
    l_number NUMBER;
```

```
  BEGIN
```

```
    l_number := 1;
```

```
  END;
```

```
  l_number := 2;
```

```
END;
```

```
/
l_number := 2;
*
ERROR at line 8:
ORA-06550: line 8, column 3:
PLS-00201: identifier 'L_NUMBER' must be declared
ORA-06550: line 8, column 3:
PL/SQL: Statement ignored

SQL>
```

The main work is done in the mandatory executable section of the block, while the optional exception section is where all error processing is placed. The following two examples demonstrate the usage of exception handlers for trapping error messages. In the first, there is no exception handler so a query returning no rows results in an error. In the second the same error is trapped by the exception handler, allowing the code to complete successfully.

```
DECLARE
l_date DATE;
BEGIN
SELECT SYSDATE
INTO l_date
FROM dual
WHERE 1=2; -- For zero rows
END;
/
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 4
```

```
DECLARE
  l_date DATE;
BEGIN
  SELECT SYSDATE
  INTO l_date
  FROM dual
  WHERE 1=2; -- For zero rows
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    NULL;
END;
/

PL/SQL procedure successfully completed.

SQL>
```

## Variables and Constants

Variables and constants must be declared for use in procedural and SQL code, although the datatypes available in SQL are only a subset of those available in PL/SQL. All variables and constants must be declared before they are referenced. The declarations of variables and constants are similar, but constant definitions must contain the **CONSTANT** keyword and must be assigned a value as part of the definition. Subsequent attempts to assign a value to a constant will result in an error. The following example shows some basic variable and constant definitions, along with a subsequent assignment of a value to a constant resulting in an error.

```
DECLARE
  l_string VARCHAR2(20);
  l_number NUMBER(10);
```

```
l_con_string CONSTANT VARCHAR2(20) := 'This is a constant.';
BEGIN
l_string := 'Variable';
l_number := 1;

l_con_string := 'This will fail';
END;
/
l_con_string := 'This will fail';
*
ERROR at line 10:
ORA-06550: line 10, column 3:
PLS-00363: expression 'L_CON_STRING' cannot be used as an assignment target
ORA-06550: line 10, column 3:
PL/SQL: Statement ignored

SQL>
```

In addition to standard variable declarations used within SQL, PL/SQL allows variable data types to match the data types of existing columns, rows or cursors using the %TYPE and %ROWTYPE qualifiers, making code maintenance much easier. The following code shows each of these definitions in practice.

```
DECLARE
-- Specific column from table.
l_username all_users.username%TYPE;

-- Whole record from table.
l_all_users_row all_users%ROWTYPE;
```

```
CURSOR c_user_data IS
  SELECT username,
         created
  FROM   all_users
  WHERE  username = 'SYS';

-- Record that matches cursor definition.
l_all_users_cursor_row c_user_data%ROWTYPE;
BEGIN
  -- Specific column from table.
  SELECT username
  INTO   l_username
  FROM   all_users
  WHERE  username = 'SYS';

  DBMS_OUTPUT.put_line('l_username=' || l_username);

  -- Whole record from table.
  SELECT *
  INTO   l_all_users_row
  FROM   all_users
  WHERE  username = 'SYS';

  DBMS_OUTPUT.put_line('l_all_users_row.username=' ||
                      l_all_users_row.username);
  DBMS_OUTPUT.put_line('l_all_users_row.user_id=' ||
                      l_all_users_row.user_id);
```

```
DBMS_OUTPUT.put_line('l_all_users_row.created=' ||
    l_all_users_row.created);

-- Record that matches cursor definition.
OPEN c_user_data;
FETCH c_user_data
INTO l_all_users_cursor_row;
CLOSE c_user_data;

DBMS_OUTPUT.put_line('l_all_users_cursor_row.username=' ||
    l_all_users_cursor_row.username);
DBMS_OUTPUT.put_line('l_all_users_cursor_row.created=' ||
    l_all_users_cursor_row.created);
END;
/
l_username=SYS
l_all_users_row.username=SYS
l_all_users_row.user_id=0
l_all_users_row.created=18-MAR-2004 08:02:17
l_all_users_cursor_row.username=SYS
l_all_users_cursor_row.created=18-MAR-2004 08:02:17

PL/SQL procedure successfully completed.

SQL>
```

The %TYPE qualifier signifies that the variable datatype should match that of the specified table column, while the %ROWTYPE qualifier signifies that the variable datatype should be a record structure that matches the specified table or cursor structure. Notice that the record structures use the dot notation (variable.column) to reference the individual column data within the record structure.

Values can be assigned to variables directly using the "==" assignment operator, via a SELECT ... INTO statement or when used as OUT or IN OUT parameter from a procedure. All three assignment methods are shown in the example below.

```
DECLARE
  l_number NUMBER;

  PROCEDURE add(p1 IN NUMBER,
               p2 IN NUMBER,
               p3 OUT NUMBER) AS
  BEGIN
    p3 := p1 + p2;
  END;
BEGIN
  -- Direct assignment.
  l_number := 1;

  -- Assignment via a select.
  SELECT 1
  INTO l_number
  FROM dual;

  -- Assignment via a procedure parameter.
  add(1, 2, l_number);
END;
/
```

## Using SQL in PL/SQL

The SQL language is fully integrated into PL/SQL, so much so that they are often mistaken as being a single language by newcomers. It is possible to manually code the retrieval of data using explicit cursors, or let Oracle do the hard work and use implicit cursors. Examples of both explicit implicit cursors are presented below, all of which rely on the following table definition table.

```
CREATE TABLE sql_test (  
  id      NUMBER(10),  
  description VARCHAR2(10)  
);  
  
INSERT INTO sql_test (id, description) VALUES (1, 'One');  
INSERT INTO sql_test (id, description) VALUES (2, 'Two');  
INSERT INTO sql_test (id, description) VALUES (3, 'Three');  
COMMIT;
```

The SELECT ... INTO statement allows data from one or more columns of a specific row to be retrieved into variables or record structures using an implicit cursor.

```
SET SERVEROUTPUT ON  
DECLARE  
  l_description VARCHAR2(10);  
BEGIN  
  SELECT description  
  INTO l_description  
  FROM sql_test  
  WHERE id = 1;  
  
  DBMS_OUTPUT.put_line('l_description=' || l_description);  
END;  
/  
l_description=One  
  
PL/SQL procedure successfully completed.
```



```
SQL>
```

The previous example can be recoded to use an explicit cursor as shown below. Notice that the cursor is now defined in the declaration section and is explicitly opened and closed, making the code larger and a little ugly.

```
SET SERVEROUTPUT ON
DECLARE
  l_description VARCHAR2(10);

  CURSOR c_data (p_id IN NUMBER) IS
    SELECT description
    FROM   sql_test
    WHERE  id = p_id;
BEGIN
  OPEN c_data (p_id => 1);
  FETCH c_data
  INTO l_description;
  CLOSE c_data;

  DBMS_OUTPUT.put_line('l_description=' || l_description);
END;
/
l_description=One

PL/SQL procedure successfully completed.

SQL>
```

When a query returns multiple rows it can be processed within a loop. The following example uses a cursor FOR-LOOP to cycle through multiple rows of an implicit cursor. Notice there is no need for a variable definition as "cur\_rec" acts as a pointer to the current record of the cursor.

```
SET SERVEROUTPUT ON
BEGIN
  FOR cur_rec IN (SELECT description
                  FROM sql_test)
  LOOP
    DBMS_OUTPUT.put_line('cur_rec.description=' || cur_rec.description);
  END LOOP;
END;
/
cur_rec.description=One
cur_rec.description=Two
cur_rec.description=Three

PL/SQL procedure successfully completed.

SQL>
```

The explicit cursor version of the previous example is displayed below. Once again the cursor management is all done manually, but this time the exit from the loop must be managed manually also.

```
SET SERVEROUTPUT ON
DECLARE
  l_description VARCHAR2(10);

  CURSOR c_data IS
    SELECT description
    FROM sql_test;
BEGIN
  OPEN c_data;
  LOOP
```

```
FETCH c_data
INTO l_description;
EXIT WHEN c_data%NOTFOUND;

DBMS_OUTPUT.put_line('l_description=' || l_description);
END LOOP;
CLOSE c_data;
END;
/
l_description=One
l_description=Two
l_description=Three

PL/SQL procedure successfully completed.

SQL>
```

In most situations the implicit cursors provide a faster and cleaner solution to data retrieval than their explicit equivalents.

## Branching and Conditional Control

The IF-THEN-ELSE and CASE statements allow code to decide on the correct course of action for the current circumstances. In the following example the IF-THEN-ELSE statement is used to decide if today is a weekend day.

```
SET SERVEROUTPUT ON
DECLARE
  l_day VARCHAR2(10);
BEGIN
  l_day := TRIM(TO_CHAR(SYSDATE, 'DAY'));
```

```
IF l_day IN ('SATURDAY', 'SUNDAY') THEN
    DBMS_OUTPUT.put_line('It's the weekend!');
ELSE
    DBMS_OUTPUT.put_line('It's not the weekend yet!');
END IF;
END;
/
```

First, the expression between the IF and the THEN is evaluated. If that expression equates to TRUE the code between the THEN and the ELSE is performed. If the expression equates to FALSE the code between the ELSE and the END IF is performed. The IF-THEN-ELSE statement can be extended to cope with multiple decisions by using the ELSIF keyword. The example below uses this extended form to produce a different message for Saturday and Sunday.

```
SET SERVEROUTPUT ON
DECLARE
    l_day VARCHAR2(10);
BEGIN
    l_day := TRIM(TO_CHAR(SYSDATE, 'DAY'));

    IF l_day = 'SATURDAY' THEN
        DBMS_OUTPUT.put_line('The weekend has just started!');
    ELSIF l_day = 'SUNDAY' THEN
        DBMS_OUTPUT.put_line('The weekend is nearly over!');
    ELSE
        DBMS_OUTPUT.put_line('It's not the weekend yet!');
    END IF;
END;
/
```

SQL CASE expressions were introduced in the later releases of Oracle 8i, but Oracle 9i included support for CASE statements in PL/SQL for the first time. The CASE statement is the natural replacement for

large IF-THEN-ELSIF-ELSE statements. The following code gives an example of a matched CASE statement.

```
SET SERVEROUTPUT ON
DECLARE
  l_day VARCHAR2(10);
BEGIN
  l_day := TRIM(TO_CHAR(SYSDATE, 'DAY'));

  CASE l_day
    WHEN 'SATURDAY' THEN
      DBMS_OUTPUT.put_line('The weekend has just started!');
    WHEN 'SUNDAY' THEN
      DBMS_OUTPUT.put_line('The weekend is nearly over!');
    ELSE
      DBMS_OUTPUT.put_line('It"s not the weekend yet!');
  END CASE;
END;
/
```

The WHEN clauses of a matched CASE statement simply state the value which is to be compared. If the value of the variable specified after the CASE keyword matches this comparison value the code after the THEN keyword is performed.

A searched CASE statement has a slightly different format, with each WHEN clause containing a full expression, as shown below.

```
SET SERVEROUTPUT ON
DECLARE
  l_day VARCHAR2(10);
BEGIN
  l_day := TRIM(TO_CHAR(SYSDATE, 'DAY'));
```

```
CASE
  WHEN l_day = 'SATURDAY' THEN
    DBMS_OUTPUT.put_line('The weekend has just started!');
  WHEN l_day = 'SUNDAY' THEN
    DBMS_OUTPUT.put_line('The weekend is nearly over!');
  ELSE
    DBMS_OUTPUT.put_line('It's not the weekend yet!');
END CASE;
END;
/
```

## Looping Statements

Loops allow sections of code to be processed multiple times. In its most basic form a loop consists of the LOOP and END LOOP statement, but this form is of little use as the loop will run forever.

```
BEGIN
  LOOP
    NULL;
  END LOOP;
END;
/
```

Typically you would expect to define an end condition for the loop using the EXIT WHEN statement along with a Boolean expression. When the expression equates to true the loop stops. The example below uses this syntax to print out numbers from 1 to 5.

```
SET SERVEROUTPUT ON
DECLARE
  i NUMBER := 1;
BEGIN
  LOOP
```

```
EXIT WHEN i > 5;
DBMS_OUTPUT.put_line(i);
i := i + 1;
END LOOP;
END;
/
```

The placement of the EXIT WHEN statement can affect the processing inside the loop. For example, placing it at the start of the loop means the code within the loop may be executed "0 to many" times, like a while-do loop in other language. Placing the EXIT WHEN at the end of the loop means the code within the loop may be executed "1 to many" times, like a do-while loop in other languages.

The FOR-LOOP statement allows code within the loop to be repeated a specified number of times based on the lower and upper bounds specified in the statement. The example below shows how the previous example could be recorded to use a FOR-LOOP statement.

```
SET SERVEROUTPUT ON
BEGIN
FOR i IN 1 .. 5 LOOP
DBMS_OUTPUT.put_line(i);
END LOOP;
END;
/
```

The WHILE-LOOP statement allows code within the loop to be repeated until a specified expression equates to TRUE. The following example shows how the previous examples can be re-coded to use a WHILE-LOOP statement.

```
SET SERVEROUTPUT ON
DECLARE
i NUMBER := 1;
BEGIN
WHILE i <= 5 LOOP
DBMS_OUTPUT.put_line(i);
```

```
i := i + 1;  
END LOOP;  
END;  
/
```

In addition to these loops a special cursor FOR-LOOP is available as seen previously.

## GOTO

The GOTO statement allows a program to branch unconditionally to a predefined label. The following example uses the GOTO statement to repeat the functionality of the examples in the previous section.

```
SET SERVEROUTPUT ON  
DECLARE  
  i NUMBER := 1;  
BEGIN  
  LOOP  
    IF i > 5 THEN  
      GOTO exit_from_loop;  
    END IF;  
    DBMS_OUTPUT.put_line(i);  
    i := i + 1;  
  END LOOP;  
  
  << exit_from_loop >>  
  NULL;  
END;  
/
```

In this example the GOTO has been made conditional by surrounding it with an IF statement. When the GOTO is called the program execution immediately jumps to the appropriate label, defined using double-angled brackets.



## Procedures, Functions and Packages

Procedures and functions allow code to be named and stored in the database, making code reuse simpler and more efficient. Procedures and functions still retain the block format, but the DECLARE keyword is replaced by PROCEDURE or FUNCTION definitions, which are similar except for the additional return type definition for a function. The following procedure displays numbers between upper and lower bounds defined by two parameters, then shows the output when it's run.

```
CREATE OR REPLACE PROCEDURE display_numbers (  
  p_lower IN NUMBER,  
  p_upper IN NUMBER)  
AS  
BEGIN  
  FOR i IN p_lower .. p_upper LOOP  
    DBMS_OUTPUT.put_line(i);  
  END LOOP;  
END;  
/  
  
SET SERVEROUTPUT ON  
EXECUTE display_numbers(2, 6);  
2  
3  
4  
5  
6  
  
PL/SQL procedure successfully completed.  
  
SQL>
```

The following function returns the difference between upper and lower bounds defined by two parameters.

```
CREATE OR REPLACE FUNCTION difference (
```

```
  p_lower IN NUMBER,
```

```
  p_upper IN NUMBER)
```

```
  RETURN NUMBER
```

```
AS
```

```
BEGIN
```

```
  RETURN p_upper - p_lower;
```

```
END;
```

```
/
```

```
VARIABLE l_result NUMBER
```

```
BEGIN
```

```
  :l_result := difference(2, 6);
```

```
END;
```

```
/
```

```
PL/SQL procedure successfully completed.
```

```
PRINT l_result
```

```
  L_RESULT
```

```
-----
```

```
      4
```

```
SQL>
```

Packages allow related code, along with supporting types, variables and cursors, to be grouped together. The package is made up of a specification that defines the external interface of the package, and a body that contains all the implementation code. The following code shows how the previous procedure and function could be grouped into a package.

```
CREATE OR REPLACE PACKAGE my_package AS
```

```
PROCEDURE display_numbers (
```

```
  p_lower IN NUMBER,
```

```
  p_upper IN NUMBER);
```

```
FUNCTION difference (
```

```
  p_lower IN NUMBER,
```

```
  p_upper IN NUMBER)
```

```
  RETURN NUMBER;
```

```
END;
```

```
/
```

```
CREATE OR REPLACE PACKAGE BODY my_package AS
```

```
PROCEDURE display_numbers (
```

```
  p_lower IN NUMBER,
```

```
  p_upper IN NUMBER)
```

```
AS
```

```
BEGIN
```

```
  FOR i IN p_lower .. p_upper LOOP
```

```
    DBMS_OUTPUT.put_line(i);
```

```
  END LOOP;
```

```
END;
```

```
FUNCTION difference (
```

```
  p_lower IN NUMBER,
```

```
p_upper IN NUMBER)
RETURN NUMBER
AS
BEGIN
    RETURN p_upper - p_lower;
END;

END;
/
```

Once the package specification and body are compiled they can be executed as before, provided the procedure and function names are prefixed with the package name.

```
SET SERVEROUTPUT ON
EXECUTE my_package.display_numbers(2, 6);
2
3
4
5
6
```

PL/SQL procedure successfully completed.

```
VARIABLE l_result NUMBER
BEGIN
    :l_result := my_package.difference(2, 6);
END;
/
```

PL/SQL procedure successfully completed.

```
PRINT l_result
```

```
L_RESULT
```

```
-----
```

```
4
```

```
SQL>
```

Since the package specification defines the interface to the package, the implementation within the package body can be modified without invalidating any dependent code, thus breaking complex dependency chains. A call to any element in the package causes the whole package to be loaded into memory, improving performance compared to loading several individual procedures and functions.

## Records

Record types are composite data structures, or groups of data elements, each with its own definition. Records can be used to mimic the row structures of tables and cursors, or as a convenient way to pass data between subprograms without listing large number of parameters.

When a record type must match a particular table or cursor structure it can be defined using the `%ROWTYPE` attribute, removing the need to define each column within the record manually. Alternatively, the record can be specified manually. The following code provides an example of how records can be declared and used in PL/SQL.

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
-- Define a record type manually.
```

```
TYPE t_all_users_record IS RECORD (
```

```
  username VARCHAR2(30),
```

```
  user_id NUMBER,
```

```
  created DATE
```

```
);
```

```
-- Declare record variables using the manual and %ROWTYPE methods.
```

```
l_all_users_record_1 t_all_users_record;
l_all_users_record_2 all_users%ROWTYPE;
BEGIN
-- Return some data into once record structure.
SELECT *
INTO l_all_users_record_1
FROM all_users
WHERE username = 'SYS';

-- Display the contents of the first record.
DBMS_OUTPUT.put_line('l_all_users_record_1.username=' ||
    l_all_users_record_1.username);
DBMS_OUTPUT.put_line('l_all_users_record_1.user_id=' ||
    l_all_users_record_1.user_id);
DBMS_OUTPUT.put_line('l_all_users_record_1.created=' ||
    l_all_users_record_1.created);

-- Assign the values to the second record structure in a single operation.
l_all_users_record_2 := l_all_users_record_1;

-- Display the contents of the second record.
DBMS_OUTPUT.put_line('l_all_users_record_2.username=' ||
    l_all_users_record_2.username);
DBMS_OUTPUT.put_line('l_all_users_record_2.user_id=' ||
    l_all_users_record_2.user_id);
DBMS_OUTPUT.put_line('l_all_users_record_2.created=' ||
    l_all_users_record_2.created);
```

```
l_all_users_record_1 := NULL;

-- Display the contents of the first record after deletion.
DBMS_OUTPUT.put_line('l_all_users_record_1.username=' ||
    l_all_users_record_1.username);
DBMS_OUTPUT.put_line('l_all_users_record_1.user_id=' ||
    l_all_users_record_1.user_id);
DBMS_OUTPUT.put_line('l_all_users_record_1.created=' ||
    l_all_users_record_1.created);

END;
/
l_all_users_record_1.username=SYS
l_all_users_record_1.user_id=0
l_all_users_record_1.created=18-MAR-2004 08:02:17
l_all_users_record_2.username=SYS
l_all_users_record_2.user_id=0
l_all_users_record_2.created=18-MAR-2004 08:02:17
l_all_users_record_1.username=
l_all_users_record_1.user_id=
l_all_users_record_1.created=

PL/SQL procedure successfully completed.

SQL>
```

Notice how the records can be assigned to each other directly, and how all elements within a record can be initialized with a single assignment of a NULL value.

## Object Types

Oracle implements Objects through the use of TYPE declarations, defined in a similar way to packages. Unlike packages where the instance of the package is limited to the current session, an instance of an object type can be stored in the database for later use. The definition of the type contains a comma separated list of attributes/properties, defined in the same way as package variables, and member functions/procedures. If a type contains member functions/procedures, the procedural work for these elements is defined in the TYPE BODY.

To see how objects can be used let's assume we want to create one to represent a person. In this case, a person is defined by three attributes (first\_name, last\_name, date\_of\_birth). We would also like to be able to return the age of the person, so this is included as a member function (get\_age).

```
CREATE OR REPLACE TYPE t_person AS OBJECT (  
  first_name  VARCHAR2(30),  
  last_name   VARCHAR2(30),  
  date_of_birth DATE,  
  MEMBER FUNCTION get_age RETURN NUMBER  
);  
/
```

Type created.

SQL>

Next the type body is created to implement the get\_age member function.

```
CREATE OR REPLACE TYPE BODY t_person AS  
  MEMBER FUNCTION get_age RETURN NUMBER AS  
  BEGIN  
    RETURN TRUNC(MONTHS_BETWEEN(SYSDATE, date_of_birth)/12);  
  END get_age;  
END;  
/
```



Type body created.

SQL>

Once the object is defined it can be used to define a column in a database table.

```
CREATE TABLE people (  
  id    NUMBER(10) NOT NULL,  
  person t_person  
);
```

Table created.

SQL>

To insert data into the PEOPLE table we must use the t\_person() constructor. This can be done as part of a regular DML statement, or using PL/SQL.

```
INSERT INTO people (id, person)  
VALUES (1, t_person('John', 'Doe', TO_DATE('01/01/2000','DD/MM/YYYY')));
```

1 row created.

```
COMMIT;
```

Commit complete.

```
DECLARE
```

```
  l_person t_person;
```

```
BEGIN
```

```
  l_person := t_person('Jane','Doe', TO_DATE('01/01/2001','DD/MM/YYYY'));
```

```
INSERT INTO people (id, person)
VALUES (2, l_person);
COMMIT;
END;
/

PL/SQL procedure successfully completed.

SQL>
```

Once the data is loaded it can be queried using the dot notation.

```
alias.column.attribute
alias.column.function()
```

The query below shows this in action.

```
SELECT p.id,
       p.person.first_name,
       p.person.get_age() AS age
FROM   people p;
```

ID	PERSON.FIRST_NAME	AGE
1	John	5
2	Jane	4

2 rows selected.

SQL>

## Collections

Oracle uses collections in PL/SQL the same way other languages use arrays.

## Triggers

Database triggers are stored programs associated with a specific table, view or system events, such that when the specific event occurs the associated code is executed. Triggers can be used to validate data entry, log specific events, perform maintenance tasks or perform additional application logic. The following example shows how a table trigger could be used to keep an audit of update actions.

```
-- Create and populate an items table and creat an audit log table.
CREATE TABLE items (
  id      NUMBER(10),
  description VARCHAR2(50),
  price   NUMBER(10,2),
  CONSTRAINT items_pk PRIMARY KEY (id)
);

CREATE SEQUENCE items_seq;

INSERT INTO items (id, description, price) VALUES (items_seq.NEXTVAL, 'PC', 399.99);

CREATE TABLE items_audit_log (
  id      NUMBER(10),
  item_id NUMBER(10),
  description VARCHAR2(50),
  old_price NUMBER(10,2),
  new_price NUMBER(10,2),
  log_date DATE,
  CONSTRAINT items_audit_log_pk PRIMARY KEY (id)
);
```

```
CREATE SEQUENCE items_audit_log_seq;

-- Create a trigger to log price changes of items.
CREATE OR REPLACE TRIGGER items_aru_trg
  AFTER UPDATE OF price ON items
  FOR EACH ROW
BEGIN
  INSERT INTO items_audit_log (id, item_id, description, old_price, new_price, log_date)
  VALUES (items_audit_log_seq.NEXTVAL, :new.id, :new.description, :old.price, :new.price, SYSDATE);
END;
/

-- Check the current data in the audit table, should be no rows.
COLUMN description FORMAT A10
SELECT * FROM items_audit_log;

no rows selected

-- Update the price of an item.
UPDATE items
SET price = 499.99
WHERE id = 1;

-- Check the audit table again.
COLUMN description FORMAT A10
SELECT * FROM items_audit_log;

ID ITEM_ID DESCRIPTIO OLD_PRICE NEW_PRICE LOG_DATE
-----
```

```
1      1 PC      399.99  499.99 19-AUG-2005 10:14:11
```

1 row selected.

-- Clean up.

```
DROP TABLE items_audit_log;
```

```
DROP TABLE items;
```

From this you can see that the trigger fired when the price of the record was updated, allowing us to audit the action.

The following trigger sets the `current_schema` parameter for each session logging on as the `APP_LOGON` user, making the default schema that of the `SCHEMA_OWNER` user.

```
CREATE OR REPLACE TRIGGER APP_LOGON.after_logon_trg AFTER
LOGON ON APP_LOGON.SCHEMA BEGIN
    EXECUTE IMMEDIATE 'ALTER SESSION SET current_schema=SCHEMA_OWNER';
END;
/
```

## Error Handling

When PL/SQL detects an error normal execution stops and an exception is raised, which can be captured and processed within the block by the exception handler if it is present. If the block does not contain an exception handler section the exception propagates outward to each successive block until a suitable exception handler is found, or the exception is presented to the client application.

Oracle provides many predefined exceptions for common error conditions, like `NO_DATA_FOUND` when a `SELECT ... INTO` statement returns no rows. The following example shows how exceptions are trapped using the appropriate exception handler. Assume we want to return the username associated with a specific `user_id` value, we might do the following.

```
SET SERVEROUTPUT ON
DECLARE
    l_user_id all_users.username%TYPE := 0;
    l_username all_users.username%TYPE;
```

```
BEGIN
  SELECT username
  INTO l_username
  FROM all_users
  WHERE user_id = l_user_id;

  DBMS_OUTPUT.put_line('l_username=' || l_username);
END;
/
l_username=SYS

PL/SQL procedure successfully completed.

SQL>
```

That works fine for user\_id values that exist, but look what happens if we use one that doesn't.

```
SET SERVEROUTPUT ON
DECLARE
  l_user_id all_users.username%TYPE := 999999;
  l_username all_users.username%TYPE;
BEGIN
  SELECT username
  INTO l_username
  FROM all_users
  WHERE user_id = l_user_id;

  DBMS_OUTPUT.put_line('l_username=' || l_username);
END;
/
```

```
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5

SQL>
```

This is not a very user friendly message, so we can trap this error and produce something more meaningful to the users.

```
SET SERVEROUTPUT ON
DECLARE
  l_user_id all_users.username%TYPE := 999999;
  l_username all_users.username%TYPE;
BEGIN
  SELECT username
  INTO l_username
  FROM all_users
  WHERE user_id = l_user_id;

  DBMS_OUTPUT.put_line('l_username=' || l_username);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.put_line('No users have a user_id=' || l_user_id);
END;
/
No users have a user_id=999999

PL/SQL procedure successfully completed.
```

SQL>

It is possible to declare your own exceptions for application specific errors, or associate them with Oracle "ORA-" messages, which are executed using the RAISE statement. The example below builds on the previous example using a user defined exception to signal an application specific error.

```
SET SERVEROUTPUT ON
DECLARE
  l_user_id  all_users.username%TYPE := 0;
  l_username all_users.username%TYPE;

  ex_forbidden_users EXCEPTION;
BEGIN
  SELECT username
  INTO   l_username
  FROM   all_users
  WHERE  user_id = l_user_id;

  -- Signal an error is the SYS or SYSTEM users are queried.
  IF l_username IN ('SYS', 'SYSTEM') THEN
    RAISE ex_forbidden_users;
  END IF;

  DBMS_OUTPUT.put_line('l_username=' || l_username);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.put_line('No users have a user_id=' || l_user_id);
  WHEN ex_forbidden_users THEN
    DBMS_OUTPUT.put_line('Don"t mess with the ' || l_username || ' user, it is forbidden!');
END;
```



```

/
Don't mess with the SYS user, it is forbidden!

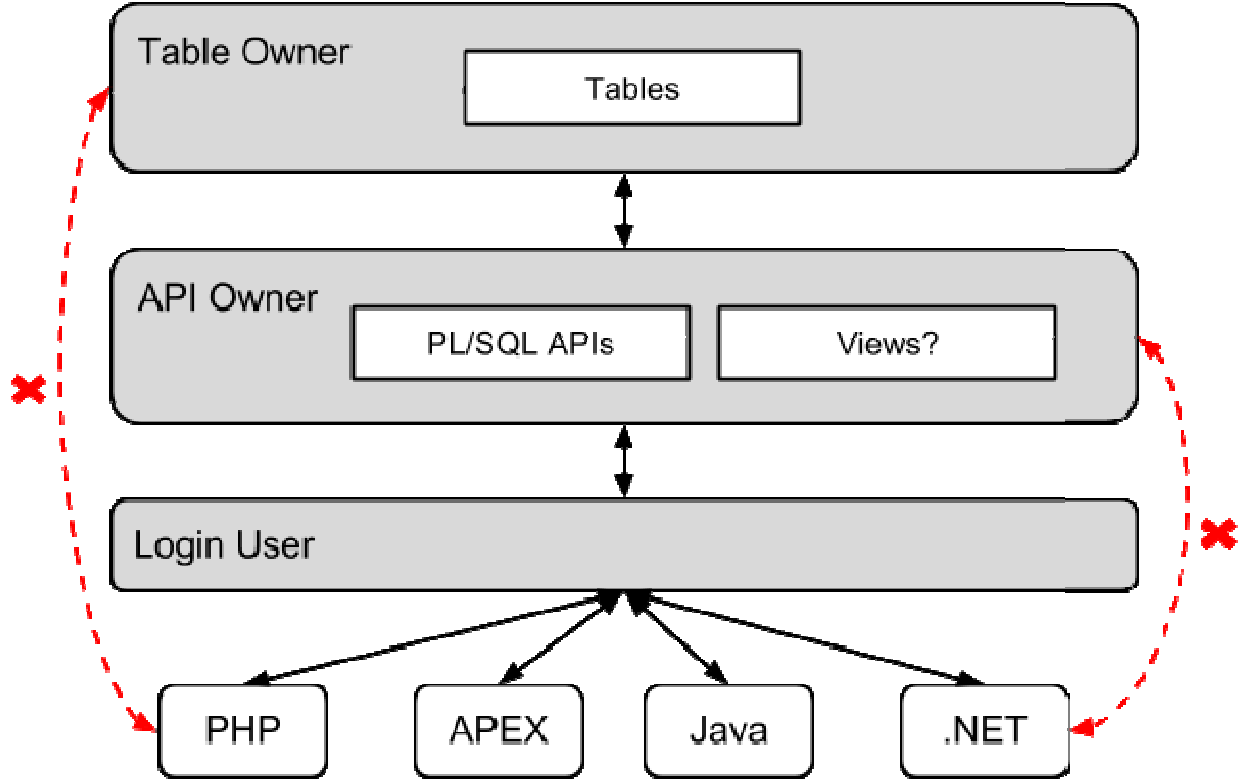
PL/SQL procedure successfully completed.

SQL>

```

The code still handles users that don't exist, but now it also raises an exception if the user returned is either SYS or SYSTEM.

The use of PL/SQL Application Program Interfaces (APIs) should be compulsory. Ideally, client application developers should have no access to tables, but instead access data via PL/SQL APIs, or possibly views if absolutely necessary.



This has a number of beneficial effects, including:

- Security and auditing mechanisms can be implemented and maintained at the database level, with little or no impact on the client application layer.
- It removes the need for triggers as all inserts, updates and deletes are wrapped in APIs. Instead of writing triggers you simply add the code into the API.

- It prevents people who don't understand SQL writing inefficient queries. All SQL should be written by PL/SQL developers or DBAs, reducing the likelihood of bad queries.
- The underlying structure of the database is hidden from the client application developers, so it hides complexity and structural changes can be made without client applications being changed.
- The API implementation can be altered and tuned without affecting the client application layer. Reducing the need for redeployments of applications.
- The same APIs are available to all applications that access the database. Resulting in reduced duplication of effort.

This sounds a little extreme, but this approach has paid dividends for me again and again. Let's elaborate on these points to explain why this approach is so successful.

It's a sad fact that auditing and security are often only brought into focus after something bad has happened. Having the ability to revise and refine these features is a massive bonus. If this means you have to re-factor your whole application you are going to have problems. If on the other hand it can be revised in your API layer you are on to a winner.

Over-reliance on database triggers is a bad thing in my opinion. It seems every company I've worked for has at one time or another used triggers to patch a "hole" or implement some business functionality in their application. Every time I see this my heart sinks. Invariably these triggers get disabled by accident and bits of functionality go AWOL, or people forget they exist and re-code some of their functionality elsewhere in the application. It's far easier to wrap the transactional processing in an API that includes all necessary functionality, thereby removing the need for table triggers entirely.

Many client application developers have to be able to work with several database engines, and as a result are not always highly proficient at coding against Oracle databases. Added to that, some development architectures such as J2EE positively discourage developers from working directly with the database. You wouldn't ask an inexperienced person to fix your car, so why would you ask one to write SQL for you? Abstracting the SQL in an API leaves the client application developers to do what they do best, while your PL/SQL programmers can write the most efficient SQL and PL/SQL possible.

During the lifetime of an application many changes can occur in the physical implementation of the database. It's nice to think that the design will be perfected before application development starts, but in reality this seldom seems to be the case. The use of APIs abstracts the developers from the physical implementation of the database, allowing change without impacting on the application.

In the same way, it is not possible to foresee all possible performance problems during the coding phase of an application. Many times developers will write and test code with unrealistic data, only to find the code that was working perfectly in a development environment works badly in a production environment. If the data manipulation layer is coded as an API it can be tuned without re-coding sections of the application, after all the implementation has changed, not the interface.

A problem I see time and time again is that companies invest heavily in coding their business logic into a middle tier layer on an application server, then want to perform data loads either directly into the database, or via a tool that will not link to their middle tier application. As a result they have to re-code sections of their business logic into PL/SQL or some other client language. Remember, it's not just the duplication of effort during the coding, but also the subsequent maintenance. Since every language worth using can speak to Oracle via OCI, JDBC, ODBC or web services, it makes sense to keep your logic in the database and let every application or data load use the same programming investment.

## Assignment

1.	Write a simple PL/SQL <b>script</b> that displays “Hello World”.
2.	Write a PL/SQL <b>stored procedure</b> to display “Hello World”.
3.	Write a PL/SQL script that performs simple arithmetic like Addition, Subtraction, and Multiplication & Division of input numbers.
4.	<p>Create two tables as shown below:            Table 1 : product (product_id, product_name, supplier_name, unit_price)            Table 2: product_price_history(product_id, product_name, supplier_name, unit_price)</p> <p>Insert appropriate data into Table 1 i.e. the “product” table.            Now write a <b>PL/SQL trigger</b> that automatically copies a row from product table to product_price_history table whenever the unit price of a product is changed in the product table.            Note: „product“ table contains new updated value of unit price while „product_price_history“ table contains the old value.</p>
5.	Write a PL-SQL script to compare three given numbers and display them in ascending order.
6.	<p>Create the following table:            Emp(E_ID, E_Name, E_Dept, E_Salary)            Insert appropriate data into Emp table.            The attribute E_Dept contains values like ( I.T. , Accounts, Sales)..            Write a <b>PL-SQL cursor</b> that increments the salary of employees of I.T. Dept. by 20%.</p>