Tech Mahindra

Oracle

SQL Study Material

for ***Oracle*** and ***BI*** Streams

Tech Mahindra

Table of contents

Introduction to Oracle

It is a database management system (DBMS), which manages a large amount of data in a multi-user environment so that many users concurrently access the data. It also provides security and Recovery. It stores and manages data using relational model.

Oracle is the name of database management system developed by **Oracle Corporation**.

Oracle server manages data in the database. Users access Oracle server using SQL commands. So Oracle server receives SQL commands from users and executes them on the database to produce the desired results as requested by the user.

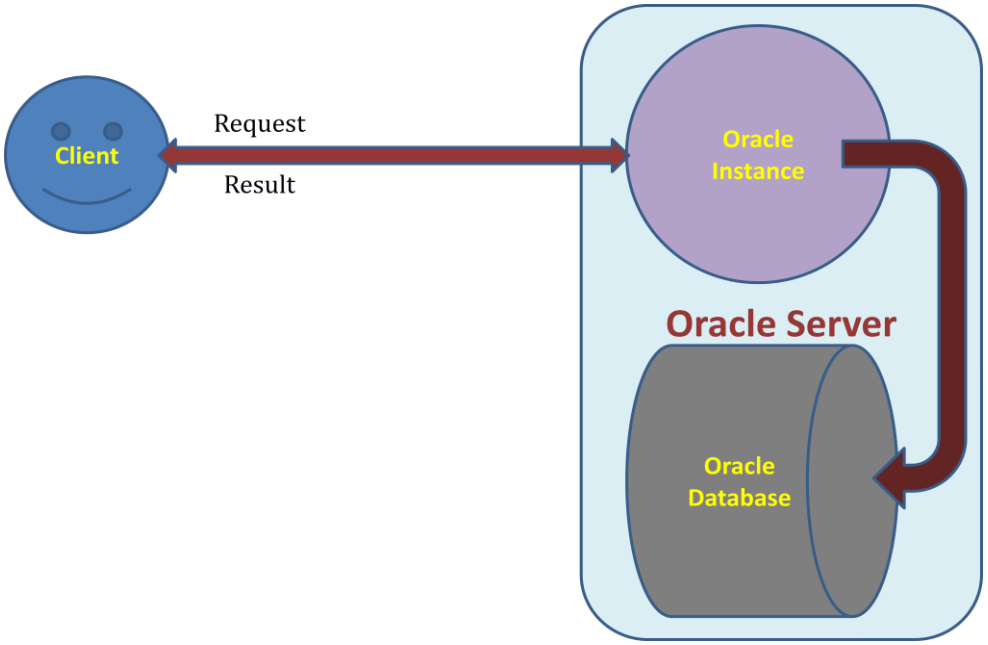

Figure 1: Oracle Architecture

Features of Oracle

The following are some of the important features of Oracle Server.

Large Database Support

Oracle supports largest database, potentially hundreds of pita bytes in size. It also allows efficient usage of space by providing full control on space management.

Data Concurrence

Oracle supports concurrent access to database by multiple users. It automatically locks and unlocks rows to maintain integrity of the data.

Industry acceptance standards

Oracle server is 100% compliant with Entry of the ANSI / ISO standards. Oracle adheres to industry standards for data access language, network protocols etc. This makes Oracle an 'open' system, which protects the investment of customer. It is easy to port Oracle applications.

Portability

Oracle software can be ported to different operating systems and it is the name on all systems. Application development in Oracle can be ported to any operating system with little or no modifications.

Oracle server runs on different platforms. The following are some of the platforms on which Oracle runs.

- Windows NT.
- Novel Netware
- Unix

Enforced Integrity

Oracle allows users to define business rules and enforce them. These rules need not be included at the application level.

Data Security

Oracle provides security in different levels – system level and object level. It also makes implementation of security easier through Roles.

Support for Client / Server environment

Oracle allows process to be split between client and server. Oracle server does all database management whereas Client does user interface. Oracle server allows code to be stored in the database in the form of procedures and functions. This allows centralization of the code and reduces network traffic.

Oracle's Role in Client / Server Computing

Client/Server computing is a method in which

- Database is stored on the server in the network
- A dedicated program, called back-end, runs on the server to manage database, which is also stored on the server.
- User access the data in database by running application, also called as front-end from clients, that accesses back-end running on the server.
- Applications running on the clients interact with the user.
- Back-end takes care of total database management.
- Client application and back-end run on different machines, which may be of different types. For example, back-end may run on mainframe and front-end may be on a PC.

Oracle is a database system that runs on the server, and used to manage the data. The other name to database server is Back-End.

What is Personal Oracle?

Personal Oracle is one of the flavors of Oracle. In this Oracle server and client both run on the same machine. This is unlike other flavors where Oracle Server runs on Server and Front-end runs on Client.

It is possible to develop an application using Personal Oracle and deploy it in a Client / Server environment. Personal Oracle can support up to 15 database connections.

Physical Database Architecture

A database contains any length of information. But for the end user, we have to show only required information by hiding the unwanted information. This data hiding can be done using various data abstraction methods.

In any RDBMS we can use 3 levels of data abstractions.

- Physical level
- Logical Level
- View level

Physical Level

The Physical structure of the database is placed in Physical level. It is physically a set of three operating system files. These files automatically create when database is created.

- Data Files
- Redo log files
- Control files

Data Files

It contains the data of the database. Every table that is stored in the database is a part of these files. Only Oracle Server can interpret these data files.

Redo Log Files

Every database has a set of two or more Redo Log files. The set of redo log files is known as databases redo log. Redo Log files are used in failure recovery. All changes made to the database are written to redo log file. (Filenames redo01.log)

Control Files

Contain information required to verify the integrity of the database, including the names of the other files in the database (Extension of file is ctl)

- Database Name
- Names and locations of data files and redo log files.

Path: We can use this Oracle\oradata\orcl path in the server to see all the 3 types of files

Logical Database Architecture

Logical Structure is independent of Physical structure. Each Oracle database contains the following components.

- Tablespaces
- Segments
- Extents
- Blocks

Tablespace

Each Database is a collection of tablespaces. For example we can use a table space called PAYROLL to store all the data related to payroll application.

Every database contains SYSTEM tablespace. This is automatically created when a database is created. SYSTEM tablespace contains the data dictionary tables.

It is possible to make tablespace temporarily unavailable by making it off-line and makes it available again by changing it to on-line. By making a tablespace off-line, DBA can take the backup.

Segments

Data into table spaces comes in the form of segments. Example Table is a segment. An Oracle database requires up to 4 types of segments

Data segments	It is used to store data of tables
Index Segments	Used to store indexes
Rollback segments	Undo information is stored
Temporary segments	Oracle stores Temporary tables

Extents

The storage space is allocated to segments is in the form of Extents. Each Tablespace contains 65536 data files N number of such Table spaces creates a database. An extent is made with in a data file. N Number of continuous db blocks makes up an Extent.

Data block

A data block is the amount of data that is transferred between disc to memory and memory to disk when a read or write operation is performed.

Overall System Structure

A database system is partitioned into modules, which handles different responsibilities of over all system.

The functional components of a database system are

- Query processor Component
- Storage manager component

Query Processor Component

This component is a collection of the following processes.

- **DML Compiler** : It translates DML statements into a lower level instructions that the query evaluation engine understands
- **Embedded DML precompiler** It converts DML statements embedded in an application program into normal procedure calls in the host language.
- **DDL Interpreter** It interprets DDL statements and records them in a set of tables
- **Query evaluation engine** It executes lower level instructions generated by the DML compiler

Storage manager component

It is an Interface between the data stored in the database, application programs and queries submitted to the system.

- **Authorization and Integrity manager** It tests for satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction Manager** It ensures concurrent transaction executions processed without conflicting.
- **File manager** It manages the allocation of space on disk and the data structures used to represent information.
- **Buffer manager** This is responsible for fetching data from disk storage into main memory.

Oracle Instance

Every oracle database is associated with an Oracle Instance. Every time a database is started, a memory area called System Global Area (SGA) or Shared Global Area is allocated and one or more processes are started.

The combination of SGA and Oracle processes is called as Oracle Instance. SGA consists of several memory structures:

- The shared pool is used to store the most recently executed SQL statements and the most recently used data from the data dictionary. These SQL statements may be submitted by a user process or, in the case of stored procedures, read from the data dictionary.
- The database buffer cache is used to store the most recently used data. The data is read from, and written to, the data files.
- The redo log buffer is used to track changes made to the database by the server and background processes.

Query process

When User connects to the database, it automatically creates two different processes called as User process and Server Process. The user process is the application program that originates SQL statements. The server process executes the statements sent from the user process.

There are three main stages in the processing of a query:

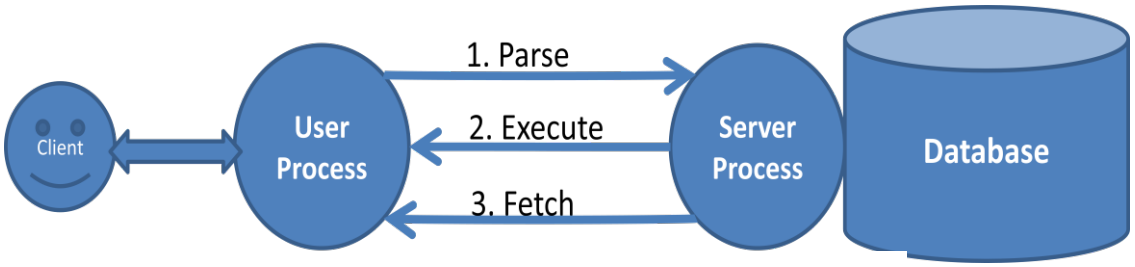

Figure 2: Oracle Query Processing Steps

Parsing

During the *parse* stage, the SQL statement is passed from the user process to the server process, and a parsed representation of the SQL statement is loaded into a shared SQL area.

During the parse, the server process performs the following functions:

- Searches for an existing copy of the SQL statement in the shared pool
- validates the SQL statement by checking its syntax
- Performs data dictionary lookups to validate table and column definitions

Execute

Identify rows selected. The steps to be taken when executing the request. The optimizer is the function in the Oracle Server that determines the optimal execution plan.

Fetch

Return rows to user process. With each fetch process, it can fetch 20 records at a time.

DML Processing Steps

A data manipulation language (DML) statement requires only two phases of processing:

- Parse is the same as the parse phase used for processing a query
- Execute requires additional processing to make data changes

DML Execute Phase

The server process records the before image to the rollback block and updates the data block. Both of these changes are done in the database buffer cache. Any changed blocks in the buffer cache are marked as dirty buffers: that is, buffers that are not the same as the corresponding blocks on the disk.

The processing of a `DELETE` or `INSERT` command uses similar steps. The before image for a `DELETE` contains the column values in the deleted row, and the before image of an `INSERT` contains the row location information.

Because the changes made to the blocks are only recorded in memory structures and are not written immediately to disk, a computer failure that causes the loss of the SGA can also lose these changes.

Oracle Versions

Table 1: Versions and features of Oracle

Oracle 6.0	1990	
Oracle 7.0	1995	
Oracle 7.1	1996	
Oracle 7.2	1997	
Oracle 7.3	1998	Object based
Oracle 8.0	1999	ORDBMS
Oracle 8i	2000	Internet based Application
Oracle9i	2001	Application server
Oracle10g	2004	Grid Computing
Oracle 11g	2009	482 new features
Oracle 12i	2009 Aug	1500 new features

Oracle 9i

Features

Oracle offers a comprehensive high performance infrastructure for e-business. It is called Oracle9i. It provides everything needed to develop, deploy and manage Internet applications.

Benefits

- Scalability from departments to enterprise e-business sites
- Reliable, available and secure architecture
- One development model, easy development options
- Common skill sets including SQL, PL/SQL,JAVA and XML
- One Management interface for all applications

9i Products

There are two products. They provide a complete and simple infrastructure for Internet applications.

Application Server

9i Application server runs all the applications and 9i database stores our data. Oracle 9i Application server runs

- Portals or web sites
- Java Transactional Applications
- Provides integration between users, applications and data

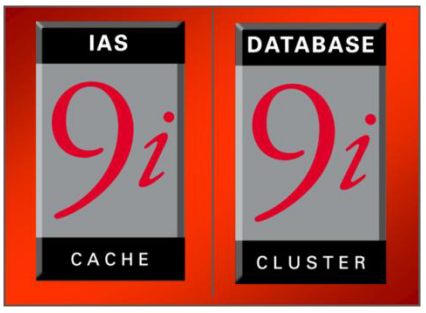

Figure 3: Products of Oracle 9i

Oracle9i: ORDBMS

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

Oracle 9i supports

- User-Defined data types and objects
- Fully compatible with relational database (It supports all the CODD rules)
- Support of multimedia and Large objects
- It also support client server and web based applications

Oracle 9i can scale tens of thousands of concurrent users and support up to 512 peta bytes of data (One peta byte = 1000 tera bytes and One terabyte = 1000 GB).

Data Concurrency and Consistency

Data concurrency allows many users to access the same data at the same time. One way of managing data concurrency is by making other users wait until their turn comes, when a user is accessing the data. But the goal of the database system is to reduce wait time so that it is negligible to each user. At the same time data integrity should not be compromised.

Locking

Oracle uses locking mechanism and multi-versioning to increase data concurrency while maintaining data integrity.

Oracle uses Locks to control data concurrency. Locks are used to prevent destructive interference. For instance, when user X is modifying a row, it is locked so that other users cannot modify it until X completes modification. However, it doesn't stop users querying the row. That means users reading the data will not be interrupted by user modifying and vice-versa.

Note: it is the responsibility of the application developer to unlock rows that are locked by committing or rolling back the transaction.

Read Consistency

For a query, Oracle returns a time point-based version of data. That means, the data retrieved is consistent with the time at which the query started. So any changes made to database since query started will not be available**.**

Ready consistency is made possible In Oracle using Rollback segment. Rollback segment keeps a copy of unchanged data. This data is substituted for the data that has changed since query started, in the query result.

This ensures readers do not wait for writers and vice-versa. And ensures that writers only wait for other writers, if they attempt to update identical rows at the same time.

Oracle 10g Grid Computing

1. Grid computing enables groups of networked computers to be pooled on demand to meet the changing needs of business.
2. Grid computing enables the creation of a single IT infrastructure that can be shared by multiple business processes.
3. Grid computing also employs special software infrastructure to monitor resource usage and allocate requests to the most appropriate resource.

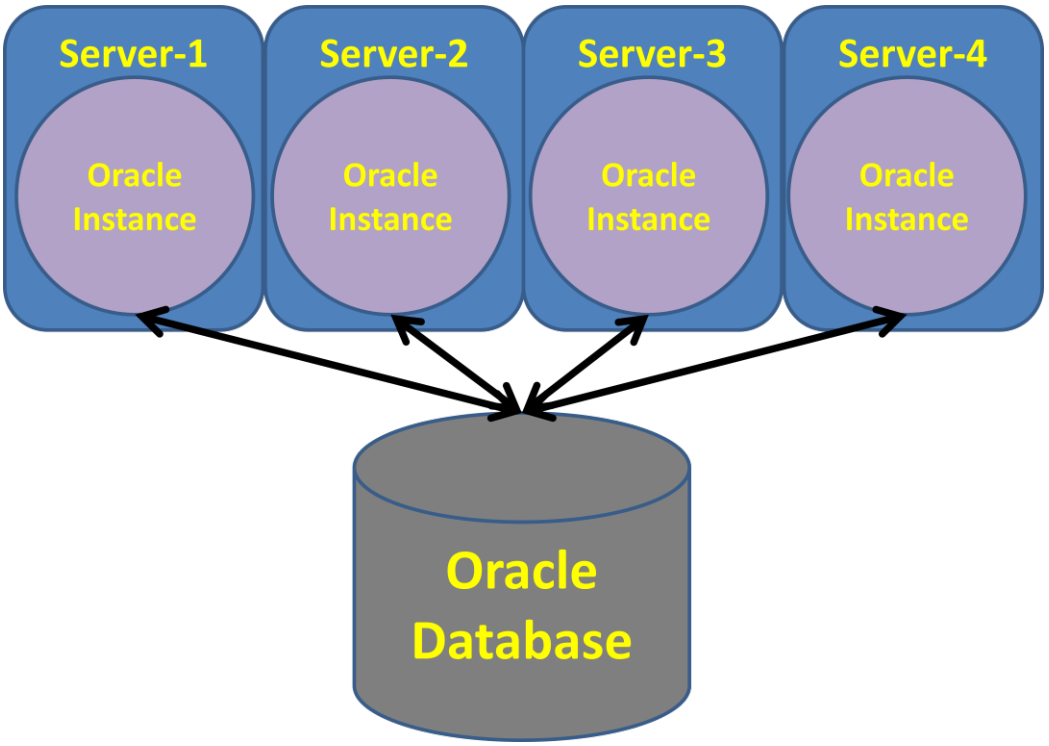

Figure 4: Oracle RAC Cluster

Summary

Oracle is RDBMS. In a Client/Server environment, Oracle runs on the server as back-end to manage the data. The logical structure of the database is independent of physical structure of the database. User is concerned with only the logical structure of the database.

An Oracle Instance is the combination of SGA and Oracle process. Oracle instance contains a collection of background processes, where each process does a specific job.

Oracle uses locking mechanism to manage data concurrency. It copies the data of a row, before the row is changed, to rollback segment to provide read consistency.

Exercises

1. SGA stands for _____
2. _____ is the name of the tablespace that is automatically created when a database is created.
3. In which segment the data of a table is stored?
4. What is the difference between Personal Oracle and Client/Server Oracle
5. Redo Log files are also called as _____
6. File extension of control file is _____
7. _____ Number of data files are there in each table space
8. What is Oracle Instance?
9. What is the maximum storage capacity of Oracle database
10. What is Application server?

Answers:

1. System Global Area or Shared Global Area
2. SYSTEM tablespace
3. DATA SEGMENT
4. PERSONAL ORACLE is for single user purpose where as Cleint/Server Oracle is for Enterprise.
5. Databases redo log files.
6. CTL
7. 65536
8. Oracle Instance is a component of oracle server that manages one and only one DATABASE.
9. 512 PETA Bytes.
10. It is a part of Oracle 9i which runs Web Sites, Java Transactional applications and also provide user integration and security features.

SQL – Structured Query Language

Environment

Clients of Oracle can use two different environments for executing SQL statements. SQL*plus and iSQL*plus.

iSQL*plus is (Available only from Oracle 9i) a web based client tool.

- An Environment
- Oracle proprietary
- Keywords can be abbreviated
- Runs on a browser
- Centrally loaded, does not have to be implemented on each machine

Difference between SQL*Plus and ISQL*plus

- SQL*Plus is a CUI and iSQL*Plus runs on a browser
- SQL*plus should be loaded in each every client system, where as iSQL*plus, centrally loaded, doesn't have to be implemented on each machine

Introduction to SQL

A Brief History of SQL

The history of SQL begins in an IBM laboratory in San Jose, California, where SQL was developed in the late 1970s. The initials stand for Structured Query Language, and the language itself is often referred to as "sequel." It was originally developed for IBM's DB2 product (a relational database management system, or RDBMS, that can still be bought today for various platforms and environments). In fact, SQL makes an RDBMS possible. SQL is a nonprocedural language, in contrast to the procedural or third generation languages (3GLs) such as COBOL and C that had been created up to that time.

NOTE: Nonprocedural means what rather than how. For example, SQL describes what data to retrieve, delete, or insert, rather than how to perform the operation.

The characteristic that differentiates a DBMS from an RDBMS is that the RDBMS provides a set-oriented database language. For most RDBMS, this set-oriented database language is SQL. Set oriented means that SQL processes sets of data in groups.

Two standards organizations, the American National Standards Institute (ANSI) and the International Standards Organization (ISO), currently promote SQL standards to industry. The ANSI-92 standard is the standard for the SQL used throughout this book. Although these standard-making bodies prepare standards for database system designers to follow, all database products differ from the ANSI standard to some degree. In technology in a single-user business application positions the application for future growth.

An Overview of SQL

SQL is the standard language used to manipulate and retrieve data from these relational databases. SQL enables a programmer or database administrator to do the following:

- Modify a database's structure
- Change system security settings
- Add user permissions on databases or tables
- Query a database for information
- Update the contents of a database

SQL in Application Programming

SQL was originally made an ANSI standard in 1986. The ANSI 1989 standard (often called SQL-89) defines three types of interfacing to SQL within an application program:

Module Language-- Uses procedures within programs. These procedures can be called by the application program and can return values to the program via parameter passing.

Embedded SQL--Uses SQL statements embedded with actual program code. This method often requires the use of a precompiler to process the SQL statements. The standard defines statements for Pascal, FORTRAN, COBOL, and PL/1.

Direct Invocation--Left up to the implementer.

Before the concept of dynamic SQL evolved, embedded SQL was the most popular way to use SQL within a program. Embedded SQL, which is still used, uses *static* SQL—meaning that the SQL statement is compiled into the application and cannot be changed at runtime. The principle is much the same as a compiler versus an interpreter. The performance for this type of SQL is good; however, it is not flexible--and cannot always meet the needs of today's changing business environments.

Structured Query Language

Oracle server supports ANSI standard SQL and contains extensions. Using SQL, you can communicate with the Oracle server. SQL has the following advantages

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

SQL Statements and Categorization

Structured query language is a collection of various types of statements shown in the table below.

Table 2: Various statements of SQL

SELECT	Data retrieval or Data query
INSERT UPDATE DELETE MERGE	Data manipulation language(DML)
CREATE ALTER DROP RENAME TRUNCATE	Data definition language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction control
GRANT REVOKE	Data control language (DCL)

Oracle SQL Datatypes

Each column value and constant in a SQL statement has a data type, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a data type for each of its columns. Oracle provides the following built-in data types.

- Character Data types
  - CHAR Data type
  - VARCHAR2 and VARCHAR Data types
  - NCHAR and NVARCHAR2 Data types
- LONG Data type
- NUMBER Data type
- DATE Data type
- LOB Data types
  - BLOB data type
  - CLOB and NCLOB data types
  - BFILE Data type
- RAW and LONG RAW Data types

Character Datatypes

The character data types store character (alphanumeric) data in strings, with byte values corresponding to the character.

CHAR datatype

Fixed length character data of length size in bytes. (Default size is 1 and maximum size is 2000). Padded on right with blanks to full length of size.

VARCHAR2 (size)

Variable length characters strings having a maximum size of 4000 bytes (Default size is 1). Truncates leftover blank spaces.

NVARCHAR2(size)

Variable length characters strings having a maximum size of 4000 bytes (Default size is 1) Or characters, depending on the choice of national character set. Truncates leftover blank spaces.

NUMBER

The NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle Database, up to 38 digits of precision.

The following numbers can be stored in a NUMBER column:

- Positive numbers in the range 1 x 10-130 to 9.99...9 x 10125 with up to 38 significant digits
- Negative numbers from -1 x 10-130 to 9.99...99 x 10125 with up to 38 significant digits
- Zero
- Positive and negative infinity (generated only by importing from an Oracle Database, Version

LONG

Character data of variable size up to 2GB in length. Only one LONG column is allowed in a table. Long column cannot be used in sub queries, functions, expressions, where clause or indexes.

DATE

Valid date ranges from January 1,4712 BC to December 31,9999 AD. (Default date format DD-MON-YY)

TIMESTAMP(precision)

Date plus time, where precision is the number of digits in the fractional part of the seconds field (default is 6).

TIMESTAMP(precision) WITH TIME ZONE

Timestamp plus time zone displacement value.

TIMESTAMP(precision) WITHLOCAL TIME ZONE

TIMESTAMP, with normalized to local time zone.

RAW(size)

Raw binary date, size bytes long. Maximum size is 2000 bytes.

LONG RAW

Raw binary data, otherwise the same as LONG. Raw and Long Raw data types allow storing pictures.

CLOB

Character Large object, up to 4GB in length.

BLOB

Binary large object, up to 4GB in length.

BFILE

Pointer to a binary operating system file.

Rules for writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.

Data Retrieval or Data Query using SELECT statement

It is a building block for data retrieval in SQL. This statement helps to retrieve data from one or multiple tables and produce output in a well formatted manner.

Syntax : SELECT <COLUMNS> FROM <TABLE>;

Your First Query

INPUT:

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

OUTPUT:

```
EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
------ ---------- --------- ---------- --------- ---------- ---------- ----------
7369 SMITH      CLERK            7902 17-DEC-80        800                    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                    20
7654 MARTIN     SALESMAN         7698 28-SEP-81       1250       1400         30
7698 BLAKE      MANAGER          7839 01-MAY-81       2850                    30
7782 CLARK      MANAGER          7839 09-JUN-81       2450                    10
7788 SCOTT      ANALYST          7566 09-DEC-82       3000                    20
7839 KING       PRESIDENT             17-NOV-81       5000                    10
7844 TURNER     SALESMAN         7698 08-SEP-81       1500          0         30
7876 ADAMS      CLERK            7788 12-JAN-83       1100                    20
7900 JAMES      CLERK            7698 03-DEC-81        950                    30
7902 FORD       ANALYST          7566 03-DEC-81       3000                    20
7934 MILLER     CLERK            7782 23-JAN-82       1300                    10
```

ANALYSIS:

Notice that columns 6 and 8 in the output statement are right justified and that columns 2 and 3 are left justified. This format follows the alignment convention in which numeric data types are right justified and character data types are left justified.

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

A full table scan is used whenever there is no where clause on a query.

Terminating an SQL Statement

In some implementations of SQL, the semicolon at the end of the statement tells the interpreter that you are finished writing the query. For example, Oracle's SQL*PLUS won't execute the query until it finds a semicolon (or a slash). On the other hand, some implementations of SQL do not use the semicolon as a terminator. For example, Microsoft Query and Borland's ISQL don't require a terminator, because your query is typed in an edit box and executed when you push a button.

Changing the Order of the Columns

We can change the order of selection of columns

INPUT:

```
SQL> SELECT empno, ename, sal, job, comm FROM emp;
```

OUTPUT:

```
     EMPNO ENAME              SAL JOB              COMM
---------- ---------- ---------- --------- ----------
      7369 SMITH              800 CLERK
      7499 ALLEN             1600 SALESMAN          300
      7521 WARD              1250 SALESMAN          500
      7566 JONES             2975 MANAGER
      7654 MARTIN            1250 SALESMAN         1400
      7698 BLAKE             2850 MANAGER
      7782 CLARK             2450 MANAGER
      7788 SCOTT             3000 ANALYST
      7839 KING              5000 PRESIDENT
      7844 TURNER            1500 SALESMAN            0
      7876 ADAMS             1100 CLERK
      7900 JAMES              950 CLERK
      7902 FORD              3000 ANALYST
      7934 MILLER            1300 CLERK

14 rows selected.
```

ANALYSIS:

Observe that the column sequence specified in the select command is not the original sequence followed during table creation. Also as per sql a column may be selected any number of times in the same select command.

Expressions, Conditions, and Operators

Expressions

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

SELECT sal FROM emp;

In the following statement **NAME, DESIGNATION, SAL** are expressions:

SELECT ENAME, DESIGNATION, SAL FROM EMP;

Now, examine the following expression:

WHERE ENAME = 'KING'

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

Conditions

If you ever want to find a particular item or group of items in your database, you need one or more conditions. Conditions are contained in the **WHERE** clause. In the preceding example, the condition is ENAME = 'KING'

To find everyone in your organization who worked more than **100** hours last month, your condition would be SAL > 2000

Conditions enable you to make selective queries. In there most common form, conditions comprise a variable, a constant, and a comparison operator. In the first example the variable is E**NAME**, the constant is **'KING'**, and the comparison operator is **=**.

In the second example the variable is **SAL**, the constant is **100**, and the comparison operator is **>**. You need to know about two more elements before you can write conditional queries: the **WHERE** clause and operators.

The WHERE Clause

Syntax: SELECT <COLUMNS> FROM <TABLE> WHERE <SEARCH CONDITION>;

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

If you want a particular EMPLOYEE details, you could type

INPUT:

```
SQL> SELECT * FROM emp WHERE ename = 'KING';
```

OUTPUT:

```
EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
------ ---------- --------- ---------- --------- ---------- ---------- ----------
7839 KING       PRESIDENT            17-NOV-81       5000                    10
```

ANALYSIS:

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

If you want a particular EMPLOYEE, you could type

INPUT:

```
SQL> SELECT * FROM emp WHERE ename != 'KING';
                OR
SQL> SELECT * FROM emp WHERE ename <> 'KING';
```

OUTPUT:

```
EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
------- ---------- --------- ---------- --------- ---------- ---------- ----------
7369 SMITH      CLERK            7902 17-DEC-80        800                    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                    20
7654 MARTIN     SALESMAN         7698 28-SEP-81       1250       1400         30
7698 BLAKE      MANAGER          7839 01-MAY-81       2850                    30
7782 CLARK      MANAGER          7839 09-JUN-81       2450                    10
7788 SCOTT      ANALYST          7566 09-DEC-82       3000                    20
7844 TURNER     SALESMAN         7698 08-SEP-81       1500          0         30
7876 ADAMS      CLERK            7788 12-JAN-83       1100                    20
7900 JAMES      CLERK            7698 03-DEC-81        950                    30
7902 FORD       ANALYST          7566 03-DEC-81       3000                    20
7934 MILLER     CLERK            7782 23-JAN-82       1300                    10
```

ANALYSIS:

Displays all the employees other than KING.

Operators

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

Arithmetic Operators

The arithmetic operators are plus (**+**), minus (-), divide (**/**), multiply (*****).

The first four are self-explanatory. Modulo returns the integer remainder of a division.

Comparison Operators

True to their name, comparison operators compare expressions and return one of three values: **TRUE**, **FALSE**, or **Unknown**. See the following examples.

```
SELECT * FROM emp WHERE sal >= 2000;

SELECT * FROM emp WHERE sal >= 3000 AND sal <= 4000;

SELECT * FROM emp WHERE sal BETWEEN 3000 AND 4000;

SELECT * FROM emp WHERE sal NOT BETWEEN 3000 AND 4000;
```

To understand how you could get an **Unknown**, you need to know a little about the concept of **NULL**. In database terms **NULL** is the absence of data in a field. It does not mean a column has a zero or a blank in it. A zero or a blank is a value. **NULL** means nothing is in that field. If you make a comparison like **Field = 9** and the only value for **Field** is **NULL**, the comparison will come back **Unknown**. Because **Unknown** is an uncomfortable condition, most flavors of SQL change **Unknown** to **FALSE** and provide a special operator, **IS NULL**, to test for a **NULL** condition.

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

```
SELECT * FROM emp WHERE comm IS NULL;

SELECT * FROM emp WHERE comm IS NOT NULL;
```

Character Operators

You can use character operators to manipulate the way character strings are represented, both in the output of data and in the process of placing conditions on data to be retrieved. This section describes two character operators: the **LIKE** operator and the || operator, which conveys the concept of character concatenation.

LIKE operator

What if you wanted to select parts of a database that fit a pattern but weren't quite exact matches? You could use the equal sign and run through all the possible cases, but that process would be time-consuming. Instead, you could use **LIKE**.

Consider the following:

INPUT:
```
SQL> SELECT * FROM emp WHERE ename LIKE 'A%';
```

ANALYSIS:
```
Displays all the employees whose names begins with letter A
```

INPUT:
```
SQL> SELECT * FROM emp WHERE ename NOT LIKE 'A%';
```

ANALYSIS:
```
Displays all the employees whose names not beginning with letter A
```

INPUT:
```
SQL> SELECT * FROM emp WHERE ename LIKE '%A%';
```

ANALYSIS:
```
Displays all the employees whose names contains letter A (Any number of A's)
```

INPUT:
```
SQL> SELECT * FROM emp WHERE ename LIKE '%A%A%';
```

ANALYSIS:
```
Displays all the names whose name contains letter A more than one time
```

INPUT:

```
SQL> SELECT * FROM emp WHERE hiredate LIKE '%DEC%';
```

ANALYSIS:

Displays all the employees who joined in the month of December.

INPUT:

```
SQL> SELECT * FROM emp WHERE hiredate LIKE '%81';
```

ANALYSIS:

Displays all the employees who joined in the year 81.

INPUT:

```
SQL> SELECT * FROM emp WHERE sal LIKE '4%';
```

ANALYSIS:

Displays all the employees whose salary begins with number 4. (Implicit data conversion takes place).

Underscore (_)

The underscore is the single-character wildcard with in the LIKE operator.

INPUT:

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

OUTPUT:

```
     EMPNO ENAME
---------- ----------
      7521 WARD
      7654 MARTIN
      7900 JAMES
```

ANALYSIS:

Displays all the employees whose second letter is A

INPUT:

```
SQL> SELECT * FROM emp WHERE ename LIKE '__A%';
```

OUTPUT:

```
ENAME
----------
BLAKE
CLARK
ADAMS
```

ANALYSIS:

Displays all the employees whose third letter is A (Two underscores followed by A)

INPUT:

```
SQL> SELECT * FROM emp WHERE ename LIKE 'A%\_%' ESCAPE '\';
```

OUTPUT:

```
ENAME
----------
AVINASH_K
ANAND_VARDAN
ADAMS_P
```

ANALYSIS:

Displays all the employees with underscore (_). '\' Escape character. Underscore is used to identify a position in the string. To treat _ as a character we have to use Escape (\) character,

Concatenation (||) operator

Used to combine two given strings

INPUT:

```
SQL> SELECT ename || job FROM emp;
```

OUTPUT:

```
ENAME||JOB
------------------
SMITHCLERK
ALLENSALESMAN
WARDSALESMAN
JONESMANAGER
MARTINSALESMAN
BLAKEMANAGER
CLARKMANAGER
SCOTTANALYST
KINGPRESIDENT
TURNERSALESMAN
ADAMSCLERK
JAMESCLERK
FORDANALYST
MILLERCLERK
```

ANALYSIS:

Combines both name and designation as a single string.

INPUT:

```
SQL>SELECT ename || ' , ' || job FROM emp;
```

OUTPUT:

```
ENAME||','||JOB
----------------------
SMITH , CLERK
ALLEN , SALESMAN
WARD , SALESMAN
JONES , MANAGER
MARTIN , SALESMAN
BLAKE , MANAGER
CLARK , MANAGER
SCOTT , ANALYST
KING , PRESIDENT
TURNER , SALESMAN
ADAMS , CLERK
JAMES , CLERK
FORD , ANALYST
MILLER , CLERK
```

ANALYSIS:

Combines both name and designation as a single string separated by,

Logical Operators

AND, OR and NOT are the logical operators used in SQL.

INPUT:

```
SQL> SELECT ename FROM emp WHERE ename LIKE '%A%' AND ename NOT LIKE '%A%A%'
```

OUTPUT:

```
ENAME
----------
ALLEN
WARD
MARTIN
BLAKE
CLARK
JAMES
```

ANALYSIS:

```
Displays all the employees whose names contains letter A exactly one time.
```

```
SELECT * FROM emp WHERE sal >= 3000 AND sal <= 4000;

SELECT * FROM emp WHERE sal BETWEEN 3000 AND 4000;

SELECT * FROM emp WHERE sal NOT BETWEEN 3000 AND 4000;
```

Miscellaneous Operators: IN, BETWEEN and DISTINCT

The two operators **IN** and **BETWEEN** provide shorthand for functions you already know how to do. You could type the following:

INPUT:

```
SQL> SELECT ename, job FROM emp WHERE job='CLERK' OR job='MANAGER' OR
job='SALESMAN';
```

OUTPUT:

```
ENAME      JOB
---------- ---------
SMITH      CLERK
ALLEN      SALESMAN
WARD       SALESMAN
JONES      MANAGER
MARTIN     SALESMAN
BLAKE      MANAGER
CLARK      MANAGER
TURNER     SALESMAN
ADAMS      CLERK
JAMES      CLERK
MILLER     CLERK
```

ANALYSIS:

```
Display employees with designations manager, clerk, and salesman,
```

The above statement takes more time to parse it, which reduces the efficiency.

INPUT:

```
SQL> SELECT * FROM emp WHERE job IN('CLERK','SALESMAN','MANAGER');
```

OUTPUT:

```
ENAME      JOB
---------- ---------
SMITH      CLERK
ALLEN      SALESMAN
WARD       SALESMAN
JONES      MANAGER
MARTIN     SALESMAN
BLAKE      MANAGER
CLARK      MANAGER
TURNER     SALESMAN
ADAMS      CLERK
JAMES      CLERK
MILLER     CLERK
```

ANALYSIS:

Display employees with designations manager, clerk, and salesman,

INPUT:

```
SQL> SELECT ename, job FROM emp WHERE job NOT IN('CLERK','SALESMAN','MANAGER');
```

OUTPUT:

```
ENAME      JOB
---------- ---------
SCOTT      ANALYST
KING       PRESIDENT
FORD       ANALYST
```

ANALYSIS:

Display designations other than manager, clerk, and salesman

INPUT:

```
SQL> SELECT ename,hiredate FROM emp WHERE hiredate IN ('01-MAY-1981','09-DEC-1982');
```

OUTPUT:

```
ENAME      HIREDATE
---------- ---------
BLAKE      01-MAY-81
SCOTT      09-DEC-82
```

ANALYSIS:

Display employees who joined on two different dates.

Distinct Operator

INPUT:
```
SQL> SELECT DISTINCT job FROM emp;
```

OUTPUT:
```
JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

ANALYSIS:
Distinct operator displays unique designations. Distinct operator by default displays the information in ascending order.

ORDER BY CLAUSE

Display the information in a particular order (Ascending or descending order)

Syntax: SELECT <COLUMNS> FROM <TABLE> WHERE <CONDITION> ORDER BY <COLUMN(S)>;

INPUT:
```
SQL> SELECT ename FROM emp ORDER BY ename;
```

OUTPUT:
```
ENAME
----------
ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD
```

ANALYSIS:
Display employees in ascending order of names.

Ordering on multiple columns is also available as shown below.

INPUT:

```
SQL> SELECT job,ename,sal FROM emp ORDER BY job,ename;
```

OUTPUT:

```
JOB       ENAME          SAL
--------- ---------- ----------
ANALYST   FORD             3000
ANALYST   SCOTT            3000
CLERK     ADAMS            1100
CLERK     JAMES             950
CLERK     MILLER           1300
CLERK     SMITH             800
MANAGER   BLAKE            2850
MANAGER   CLARK            2450
MANAGER   JONES            2975
PRESIDENT KING             5000
SALESMAN  ALLEN            1600
SALESMAN  MARTIN           1250
SALESMAN  TURNER           1500
SALESMAN  WARD             1250
```

ANALYSIS:

Display employees in ascending order of jobs. With each job it places the information in ascending order of names.

INPUT:

```
SQL> SELECT * FROM emp ORDER BY job, ename desc;
```

ANALYSIS:

Display employees in ascending order by jobs. With each job it places the information in descending order of names.

INPUT:

```
SQL> SELECT * FROM emp ORDER BY job desc, ename DESC;
```

ANALYSIS:

Display employees in descending order by jobs. With each job it places the information in descending order of names.

INPUT:

```
SQL> SELECT * FROM emp WHERE job != 'CLERK' ORDER BY job;
```

OUTPUT:

Display employees in ascending order of jobs other than clerks.

ANALYSIS:

When we are executing the query, it is divided into two different parts.

1) SELECT * FROM emp WHERE job != 'CLERK'
2) ORDER BY job;

First part is going to execute first, and selects all the employees whose designation is other than clerk and places them in a temporary table.

On the temporary table, order by clause is applied, places the information in ascending order by jobs in the shadow page, from where end user can able to see the output.

We can also use order by clause as

INPUT:

```
SQL> SELECT * FROM EMP ORDER BY 3;
```

ANALYSIS:

It places the information in the order of third column in the table.

INPUT:

```
SQL> SELECT deptno, job, sal, empno FROM emp ORDER BY 3;
```

ANALYSIS:

The information is displayed in the order of the 3rd column of the selected and not the 3rd column of the table

INPUT:

```
SQL> SELECT deptno, job, sal, empno, comm FROM emp ORDER BY comm;
```

ANALYSIS:

Since the comm column contain nulls, you observe that the rows containing null in the comm column are displayed at the bottom of the output.

INPUT:

```
SQL> SELECT empno, ename, sal, sal*12 as ann_sal FROM emp ORDER BY ann_sal;
```

ANALYSIS:

Data can be ordered by using alias name.

Exercises

1. _____ is the operator used to compare a column with null value.
2. _____is used to compare one value with a set of values.
3. The maximum number of characters that can be stored in CHAR type is
4. Assume there is a table student(sname char(6), sname1 varchar2(6)). Assume that value placed in both columns is RAVI. What is the size of sname and sname1.
5. How many LONG columns can a table contain? _____
6. SQL commands are to be terminated with _____
7. Display list of employees that start with letter C
8. Display employees in ascending order of 5th column in the table
9. Examine the trace instance chart for employee table. You want to display each employee hiredate from earliest to latest. Which SQL statement will you use?
    a. SELECT hiredate FROM emp;
    b. SELECT hiredate FROM emp ORDER BY hiredate;
    c. SELECT emp FROM emp ORDER BY hiredate;
    d. SELECT hiredate FROM emp ORDER BY hiredate DESC;
10. Which of the following data type should you use for interest rates with varying and unpredictable decimal places such as 1.234, 1.3, 4?
    a. Long
    b. Number
    c. Number(p,s)
    d. None
11. Which SQL Query generates the alias Annual Salary for the calculated column SALARY *12?
    a. SELECT ename, salary*12 ANNUAL_SALARY FROM employees;
    b. SELECT ename, salary*12 ANNUAL SALARY FROM employees;
    c. SELECT ename, salary*12 AS “ANNUAL SALARY” FROM employees;
    d. SELECT ename, salary*12 AS INITCAP(ANNUAL SALARY)FROM employees
12. The EMP table has these columns. ENAME VARCHAR2(35), SALARY NUMBER(8,2), HIRE_DATE DATE. Management wants a list of names of

employees, who have been with the company for more than 5 years. Which of the following Select commands will get the required result?

a. SELECT ename FROM emp WHERE SYSDATE – hire_date > 5;
b. SELECT ename FROM emp WHERE hire_date – SYSDATE > 5;
c. SELECT ename FROM emp WHERE (SYSDATE – hire_date)/365 > 5;
d. SELECT ename FROM emp WHERE (SYSDATE – hire_date)*365 > 5;

13. The employee table contains these columns. LAST_NAME VARCHAR2(25), FIRST_NAME VARCHAR2(25) DEPT_ID NUMBER(9) You need to display the names of the employees that are not assigned to the department. Evaluate this SQL statement. **SELECT last_name, first_name FROM employee WHERE dept_id IS NULL;** which change should you make to achieve the desired result?

a. Create an outer join
b. Change the Column in the Where Condition
c. Query executes Successfully
d. Add a second condition to the where condition

14. Which statement about SQL is true?

a. Null values are displayed last in the ascending sequences
b. Data values are displayed in the descending or by default
c. You cannot specify a column alias in an ORDER BY clause
d. You cannot sort query result by a column that is not included in the SELECT list.

Answers

1. IS NULL or IS NOT NULL
2. IN() operator
3. 2000 Characters
4. 6 and 4 respectively (can be verified using vsize() function)
5. 1 (one)
6. ; (semicolon)
7. SELECT * FROM emp WHERE ename like('C%');
8. SELECT * FROM emp ORDER BY 5
9. b (SELECT hiredate FROM emp ORDER BY hiredate)
10. b (NUMBER)
11. a) and c)
12. c (SELECT ename FROM emp WHERE (SYSDATE – hire_date)/365 > 5;)
13. c (Query executes successfully)
14. a (Null values are displayed last in the ascending sequences).

FUNCTIONS

A function is a sub program, which executes whenever we call it and returns a value to the calling place. Oracle has a large collection of predefined functions. Each function has a name and some parameters on which the function will work and return some answer.

These functions are classified into two types

- Predefined functions
- User defined functions

Predefined functions

These functions are again classified into two types

- Group or Aggregate Functions
- Single row Functions

Now let us have some detailed description of each of these functions, their use and application.

Aggregate Functions

These functions are also referred to as group functions. They return a value based on the non-null values in a column.

***COUNT*([DISTINCT] <column name>|*|<expression>)**

The function **COUNT** returns the number of rows that satisfy the condition in the **WHERE** clause.

Say you wanted to know how many employees are there.

INPUT:

```
SQL> SELECT COUNT(*) FROM emp;
```

OUTPUT:

```
COUNT(*)
--------
14
```

ANALYSIS:

```
It counts the number of rows in that table.
```

To make the code more readable, try an alias:

INPUT:

```
SQL> SELECT COUNT(*) NUM_OF_EMP FROM emp;
```

OUTPUT:

```
NUM_OF_EMP
--------------------
        14
```

INPUT:

```
SQL> SELECT COUNT(comm) FROM emp;
```

OUTPUT:

```
COUNT(comm)
--------
4
```

ANALYSIS:

```
It counts only those when there is a value in comm Column
Note: Count (*) faster than count(comm) Count(*) count the row when a row present in
the table where as Count(comm) counts the row only when there is a value in the
column.
```

INPUT:

```
SQL> SELECT COUNT(*) FROM emp WHERE job = 'MANAGER';
```

OUTPUT:

```
COUNT(*)
--------
4
```

ANALYSIS:

It counts only managers

INPUT:

```
SQL> SELECT COUNT(DISTINCT job) FROM emp;
```

OUTPUT:

```
COUNT (*)
--------
4
```

ANALYSIS:

It counts only distinct jobs

***SUM*(<column name>|<expression>)**

SUM does just that. It returns the sum of all values in a column.

INPUT:

```
SQL> SELECT SUM(sal) TOTAL_SALARY FROM emp;
```

OUTPUT:

```
TOTAL_SALARY
-------------
       29025
```

ANALYSIS:

Find the total salary drawn by all the employees

INPUT:

```
SQL> SELECT SUM(sal) TOTAL_SALARY, SUM(comm) TOTAL_COMM FROM emp;
```

OUTPUT:

```
TOTAL_SALARY   TOTAL_COMM
-------------  ----------
        29025        2200
```

ANALYSIS:

The totals of sal column and the comm column are calculated and displayed

INPUT:

```
SQL> SELECT SUM(sal) TOTAL_SALARY,SUM(comm) TOTAL_COMM FROM emp WHERE
job='SALESMAN';
```

OUTPUT:

```
TOTAL_SALARY   TOTAL_COMM
-------------  ----------
        5600         2200
```

AVG(<column name>|<expression>)

The **AVG** function computes the average of a column.

INPUT:

```
SQL> SELECT AVG(sal) AVERAGE_SALARY FROM emp;
```

OUTPUT:

```
AVERAGE_SALARY
---------------
2073.21429
```

ANALYSIS:

```
Find the average salary of all the employees
```

INPUT:

```
SQL> SELECT AVG(comm) AVERAGE_COMM FROM emp;
```

OUTPUT:

```
AVERAGE_COMM
------------
         550
```

ANALYSIS:

```
Functions ignores null rows
```

***MAX*(<column name>|<expression>)**

Returns the Maximum value in the given column of values

INPUT:

```
SQL> SELECT MAX(sal) FROM emp;
```

OUTPUT:

```
MAX(SAL)
--------
     5000
```

ANALYSIS:

Takes the value from one different rows from one particular column

INPUT:

```
SQL> SELECT MAX(ename) FROM emp;
```

OUTPUT:

```
MAX(ENAME)
--------
WARD
```

ANALYSIS:

Max of name is identified based on ASCII value when a char column is given

INPUT:

```
SQL> SELECT MAX (hiredate) FROM emp;
```

OUTPUT:

```
 MAX(HIREDATE)
 -------------
     12-JAN-83
```

ANALYSIS:

Can find the maximum date in the given column

***MIN*(<column name>|<expression>)**

Finds the minimum value in the given column of values. This example shows the use of min function with a numeric column.

INPUT:

```
SQL> SELECT MIN(sal) FROM emp;
```

OUTPUT:

```
 MIN(SAL)
 --------
      800
```

Using **MIN** with char column

INPUT:

```
SQL> SELECT MIN(ename) FROM emp;
```

OUTPUT:

```
 MIN (ENAME)
 --------
     ADAMS
```

The following example shows the use of all aggregate functions together.

INPUT:

```
SQL> SELECT SUM(sal),AVG(sal),MIN(sal),MAX(sal),COUNT(*) FROM emp;
```

OUTPUT:

```
  SUM(SAL)    AVG(SAL)    MIN(SAL)    MAX(SAL)    COUNT(*)
  ---------  ---------  --------    --------    --------
     29025  2073.21429         800        5000          14
```

ANALYSIS:

All the aggregate functions can be used together in a single SQL statement

SINGLE ROW FUNCTIONS

These functions work on each and every row and return a value to the calling places.

These functions are classified into different types

- Arithmetic Functions
- Character Functions
- Date functions
- Miscellaneous Functions

Arithmetic Functions

Many of the uses you have for the data you retrieve involve mathematics. Most Implementations of SQL provide arithmetic functions similar to that of operators covered here.

ABS(<column name>|<expression>)

The **ABS** function returns the absolute value of the number you point to. For example:

INPUT:

```
SQL> SELECT ABS(-10) ABSOLUTE_VALUE FROM dual;
```

OUTPUT:

```
ABSOLUTE_VALUE
---------------
             10
```

ANALYSIS:

ABS changes all the negative numbers to positive and leaves positive numbers alone.

Dual is a system table or dummy table from where we can display system information (i.e. system date and username etc) or we can make our own calculations.

CEIL(<column name>|<expression>) *and* *FLOOR*(<column name>|<expression>)

CEIL returns the smallest integer greater than or equal to its argument. **FLOOR** does just the reverse, returning the largest integer equal to or less than its argument.

INPUT:
```
SQL> SELECT CEIL(12.145) FROM dual;
```

OUTPUT:
```
     CEIL(12.145)
     ------------
               13
```

INPUT:
```
SQL> SELECT CEIL(12.000) FROM dual;
```

OUTPUT:
```
     CEIL(12.000)
     ------------
               12
```

ANALYSIS:
Minimum we require one decimal place, to get the next higher integer number

INPUT:
```
SQL> SELECT FLOOR(12.678) FROM dual;
```

OUTPUT:
```
     FLOOR(12.678)
     -----------------
               12
```

INPUT:
```
SQL> SELECT FLOOR(12.000) FROM dual;
```

OUTPUT:
```
     FLOOR(12.000)
     -----------------
               12
```

***MOD*(<column name>|<expression>,<column name>|<expression>)**

It returns remainder when we divide one value with another value

INPUT:

```
SQL> SELECT MOD(5,2) FROM dual;
```

OUTPUT:

```
       MOD(5,2)
     ----------
              1
```

INPUT:

```
SQL> SELECT MOD(2,5) FROM dual;
```

OUTPUT:

```
       MOD(2,5)
     ----------
              2
```

ANALYSIS:

When numerator value less than denominator, it returns numerator value as remainder.

***POWER*(<column name>|<expression>,<column name>|<expression>)**

To raise one number to the power of another, use **POWER**. In this function the first argument is raised to the power of the second:

INPUT:

```
SQL> SELECT POWER(5,3) FROM dual;
```

OUTPUT:

```
POWER(5,3)
----------
       125
```

***SIGN*(<column name>|<expression>)**

SIGN returns **-1** if its argument is less than **0**, **0** if its argument is equal to **0**, and **1** if its argument is greater than **0**,

INPUT:

```
SQL> SELECT SIGN(-10), SIGN(10),SIGN(0) FROM dual;
```

OUTPUT:

```
SIGN(-10)    SIGN(10)     SIGN(0)
---------- ---------- ----------
        -1          1          0
```

***SQRT*(<column name>|<expression>)**

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

INPUT:

```
SQL> SELECT SQRT(4) FROM dual;
```

OUTPUT:

```
SQRT(4)
---------
        2
```

***Round*(expression1,expression2) *and Trunc*((expression1,expression2)**

These functions are useful for performing numeric and date operations. Study the below examples and analyse.

Test the following

1. Select Round(12.1567,2) from dual;
2. Select round(12.1567,-1) from dual;
3. Select round(51.782,-2) from dual;
4. Select round(sysdate,'YEAR') from dual;
5. Select round(sysdate,'MONTH') from dual;
6. Select round(Sysdate,'DAY') from dual;

Note : Test the above with TRUNC function

CHARACTER FUNCTIONS

Many implementations of SQL provide functions to manipulate characters and strings of characters.

CHR(<column name>|<expression>)

CHR returns the character equivalent of the number it uses as an argument. The character it returns depends on the character set of the database. For this example the database is set to ASCII.

INPUT:

```
SQL> SELECT CHR(65) FROM dual;
```

OUTPUT:

```
CHR(65)
-------
A
```

CONCAT(<column name>|<expression>,<column name>|<expression>)

It is similar to that of concatenate operator (| |)

INPUT:

```
SQL> SELECT CONCAT('KRISHNA', ' KANTH') FROM dual;
```

OUTPUT:

```
CONCAT('KRISHNA', ' KANTH')
---------------------------
KRISHNA KANTH
```

INITCAP(<column name>|<expression>)

INITCAP capitalizes the first letter of a word and makes all other characters lowercase.

INPUT:

```
SQL> SELECT ename "BEFORE",INITCAP(ename) "AFTER" FROM emp;
```

OUTPUT:

```
BEFORE     AFTER
---------- ----------
SMITH      Smith
ALLEN      Allen
WARD       Ward
JONES      Jones
MARTIN     Martin
BLAKE      Blake
CLARK      Clark
SCOTT      Scott
KING       King
TURNER     Turner
ADAMS      Adams
JAMES      James
FORD       Ford
MILLER     Miller
```

ANALYSIS:

Observe that the first letter of every word is capital and remaining smalls.

***LOWER*(<column name>|<expression>) *and UPPER*(<column name>|<expression>)**

As you might expect, **LOWER** changes all the characters to lowercase; **UPPER** does just the changes all the characters to uppercase.

INPUT:

```
SQL>SELECT ename,UPPER(ename) UPPER_CASE,LOWER(ename) LOWER_CASE FROM emp;
```

OUTPUT:

```
ENAME      UPPER_CASE LOWER_CASE
---------- ---------- ----------
SMITH      SMITH      smith
ALLEN      ALLEN      allen
WARD       WARD       ward
JONES      JONES      jones
MARTIN     MARTIN     martin
BLAKE      BLAKE      blake
CLARK      CLARK      clark
SCOTT      SCOTT      scott
KING       KING       king
TURNER     TURNER     turner
ADAMS      ADAMS      adams
JAMES      JAMES      james
FORD       FORD       ford
MILLER     MILLER     miller
```

LPAD(expr1,expr2[,expr3]) *and RPAD*(expr1,expr2[,expr3])

LPAD and **RPAD** take a minimum of two and a maximum of three arguments. The first argument is the character string to be operated on. The second is the number of characters to pad it with, and the optional third argument is the character to pad it with. The third argument defaults to a blank, or it can be a single character or a character string.

The following statement adds five pad characters, assuming that the field **LASTNAME** is defined as a 15-character field:

INPUT:

```
SQL> SELECT LPAD(ename,15,'*') FROM emp;
```

OUTPUT:

```
LPAD(ENAME,15,'
---------------
**********SMITH
**********ALLEN
***********WARD
**********JONES
*********MARTIN
**********BLAKE
**********CLARK
**********SCOTT
***********KING
*********TURNER
**********ADAMS
**********JAMES
***********FORD
*********MILLER
```

ANALYSIS:

15 locations allocated to display ename, out of that, name is occupying some space and in the remaining space to the left side of the name pads with *.

INPUT:

```
SQL> SELECT RPAD(5000,10,'*') FROM dual;
```

OUTPUT:

```
RPAD(5000,10,'*')
--------------------
5000******
```

LTRIM(expr1[,expr2]) *and RTRIM*(expr1[,expr2])

LTRIM and RTRIM take at least one and at most two arguments. The first argument, like LPAD and RPAD, is a character string. The optional second element is either a character or character string or defaults to a blank. If you use a second argument that is not a blank, these trim functions will trim that character the same way they trim the blanks in the following examples.

INPUT:

```
SQL> SELECT ename, RTRIM(ename,'R') FROM emp;
```

OUTPUT:

```
ENAME      RTRIM(ENAM
---------- ----------
SMITH      SMITH
ALLEN      ALLEN
WARD       WARD
JONES      JONES
MARTIN     MARTIN
BLAKE      BLAKE
CLARK      CLARK
SCOTT      SCOTT
KING       KING
TURNER     TURNE
ADAMS      ADAMS
JAMES      JAMES
FORD       FORD
MILLER     MILLE
```

ANALYSIS:

```
Removes the rightmost character
```

Note: Try TRIM() function

***REPLACE*(<expr1>,<expr2>,[<expr3>])**

REPLACE does just that. Of its three arguments, the first is the string to be searched. The second is the search key. The last is the optional replacement string. If the third argument is left out or **NULL**, each occurrence of the search key on the string to be searched is removed and is not replaced with anything.

Syntax: REPLACE(STRING,SEARCH_STRING,REPLACE_STRING)

INPUT:

```
SQL> SELECT REPLACE ('RAMANA','MA', VI') FROM dual;
```

OUTPUT:

```
REPLACE ('RAMANA','MA', VI')
----------------------------
RAVINA
```

INPUT:

```
SQL> SELECT REPLACE('RAMANA','MA') FROM dual;
```

OUTPUT:

```
REPLACE('RAMANA','MA')
----------------------
RANA
```

ANALYSIS:

When the replace string is missing, search string removed from the given string

INPUT:

```
SQL> SELECT REPLACE ('RAMANA','MA', NULL) FROM dual;
```

OUTPUT:

```
REPLACE ('RAMANA','MA', NULL)
-----------------------------
RANA
```

TRANSLATE(<expr1>,<expr2>,<expr3>)

The function **TRANSLATE** takes three arguments: the target string, the **FROM** string, and the **TO** string. Elements of the target string that occur in the **FROM** string are translated to the corresponding element in the **TO** string.

INPUT:

```
SQL> SELECT TRANSLATE('RAMANA','MA','CD') FROM dual;
```

OUTPUT:

```
TRANSLATE('RAMANA','MA','CD')
------------------------------
RDCDND
```

ANALYSIS:

Notice that the function is case sensitive. When search string matches, it replaces with corresponding replace string and if any one character is matching in the search string, it replaces with corresponding replace character.

***SUBSTR*(expr1,expr2[,expr3])**

This three-argument function enables you to take a piece out of a target string. The first argument is the target string. The second argument is the position of the first character to be output. The third argument is the number of characters to show.

Syntax: **SUBSTR**(STRING,STARTING_POSITION[,NO_OF_CHARACTERS])

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',1,3) FROM dual;
```

OUTPUT:

```
SUBSTR('RAMANA',1,3)
--------------------
RAM
```

ANALYSIS:

It takes first 3 characters from first character

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',3,3) FROM DUAL;
```

OUTPUT:

```
SUBSTR('RAMANA',3,3)
--------------------
MAN
```

ANALYSIS:

It takes 3 characters from third position

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',-2,2) FROM dual;
```

OUTPUT:

```
SUBSTR('RAMANA',-2,2)
---------------------
NA
```

ANALYSIS:

You use a negative number as the second argument, the starting point is determined by counting backwards from the right end.

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',1,2) || SUBSTR('RAMANA',-2,2) FROM dual;
```

OUTPUT:

```
SUBSTR('RAMANA
--------------
RANA
```

ANALYSIS:

First two characters and last two characters are combined together as a single string

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',3) FROM dual;
```

OUTPUT:

```
SUBSTR('RAMANA',3)
------------------
MANA
```

ANALYSIS:

When third argument is missing, it takes all the character from starting position

INPUT:

```
SQL> SELECT * FROM emp WHERE SUBSTR(hiredate,4,3) = SUBSTR(SYSDATE,4,3);
```

ANALYSIS:

Displays all the employees who joined in the current month SYSDATE is a single row function, which gives the current date.

INPUT:

```
SQL> SELECT SUBSTR('RAMANA',1,2) || SUBSTR('RAMANA',-2,2) FROM DUAL;
```

OUTPUT:

```
SUBSTR('RAMANA',1,2)
--------------------
RANA
```

ANALYSIS:

First two characters and Last two characters are combined together as a single string

***INSTR*(<expr1>,<expr2>[,<expr3>[,expr4]])**

To find out where in a string a particular pattern occurs, use **INSTR**. Its first argument is the target string. The second argument is the pattern to match. The third and forth are numbers representing where to start looking and which match to report. This example returns a number representing the first occurrence of O starting with the second

INPUT:
```
SQL> SELECT INSTR('RAMANA','A') FROM DUAL;
```

OUTPUT:
```
INSTR('RAMANA','A')
-------------------
                  2
```

ANALYSIS:
Find the position of the first occurrence of letter A

INPUT:
```
SQL> SELECT INSTR('RAMANA','A',1,2) FROM dual;
```

OUTPUT:
```
INSTR('RAMANA','A',1,2)
-----------------------
                      4
```

ANALYSIS:
Find the position of the second occurrence of letter A from the beginning of the string. Third argument represents from which position, Fourth argument represents, which occurrence.

INPUT:
```
SQL> SELECT INSTR('RAMANA','a') FROM dual;
```

OUTPUT:
```
INSTR('RAMANA','a')
-------------------
                  0
```

ANALYSIS:
Function is case sensitive; it returns 0 (zero) when the given character is not found.

INPUT:
```
SQL> SELECT INSTR('RAMANA','A',3,2) FROM dual;
```

OUTPUT:
```
INSTR('RAMANA','A',3,2)
-----------------------
                      6
```

ANALYSIS:
Find the position of the second occurrence of letter A from 3rd position of the string

Conversion Functions

These functions provide a handy way of converting one type of data to another. They are mainly useful for changing date formats and number formats.

TO_CHAR

The primary use of **TO_CHAR** is to convert a number into a character. Different Implementations may also use it to convert other data types, like Date, into a character, or to include different formatting arguments.

The following example illustrates the primary use of **TO_CHAR**:

INPUT:

```
SQL> SELECT sal, TO_CHAR(sal) FROM emp;
```

OUTPUT:

```
       SAL TO_CHAR(SAL)
---------- ----------------------------------------
       800 800
      1600 1600
      1250 1250
      2975 2975
      1250 1250
      2850 2850
      2450 2450
      3000 3000
      5000 5000
      1500 1500
      1100 1100
       950 950
      3000 3000
      1300 1300
```

ANALYSIS:

After conversion, Converted information is left aligned. So we can say that it is a string.

The main usage of this function is, to change the date formats and number formats

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'DD/MM/YYYY') FROM dual;
```

OUTPUT:

```
SYSDATE        TO_CHAR(SYSDATE,'DD/MM/YYYY')
---------      ------------------------------
24-MAR-07       24/03/2007
```

ANALYSIS:

Convert the default date format to DD/MM/YYYY format

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'DD-MON-YY') FROM dual;
```

OUTPUT:

```
SYSDATE        TO_CHAR(SYSDATE,'DD-MON-YY')
---------      ------------------------------
24-MAR-07       24-MAR-07
```

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'DY-MON-YY') FROM dual;
```

OUTPUT:

```
SYSDATE        TO_CHAR(SYSDATE,'DY-MON-YY')
---------      ------------------------------
24-MAR-07       SAT-MAR-07
```

ANALYSIS:

DY displays the first 3 letters from the day name

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'DAY MONTH YEAR') FROM dual;
```

OUTPUT:

```
SYSDATE        TO_CHAR(SYSDATE,'DAYMONTHYEAR')
---------      ------------------------------
24-MAR-07       SATURDAY  MARCH TWO THOUSAND SEVEN
```

ANALYSIS:

DAY	gives the total day name
MONTH	gives the total month name
YEAR	writes the year number in words

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'DDSPTH MONTH YEAR') FROM dual;
```

OUTPUT:

```
SYSDATE   TO_CHAR(SYSDATE,'DDSPTHMONTHYEAR')
--------- ------------------------------------------------------------------
24-MAR-07 TWENTY-FOURTH MARCH     TWO THOUSAND SEVEN
```

ANALYSIS:

DD	gives the day number
DDSP	Writes day number in words
TH	is the format. Depends upon the number it gives either ST / RD/ST/ND format

INPUT:

```
SQL> SELECT hiredate,TO_CHAR(hiredate,'DDSPTH MONTH YEAR') FROM emp;
```

OUTPUT:

```
HIREDATE  TO_CHAR(HIREDATE,'DDSPTHMONTHYEAR')
--------- ------------------------------------------------------------------
17-DEC-80 SEVENTEENTH DECEMBER  NINETEEN EIGHTY
20-FEB-81 TWENTIETH FEBRUARY  NINETEEN EIGHTY-ONE
22-FEB-81 TWENTY-SECOND FEBRUARY  NINETEEN EIGHTY-ONE
02-APR-81 SECOND APRIL     NINETEEN EIGHTY-ONE
28-SEP-81 TWENTY-EIGHTH SEPTEMBER NINETEEN EIGHTY-ONE
01-MAY-81 FIRST MAY NINETEEN EIGHTY-ONE
09-JUN-81 NINTH JUNE NINETEEN EIGHTY-ONE
09-DEC-82 NINTH DECEMBER NINETEEN EIGHTY-TWO
17-NOV-81 SEVENTEENTH NOVEMBER  NINETEEN EIGHTY-ONE
08-SEP-81 EIGHTH SEPTEMBER NINETEEN EIGHTY-ONE
12-JAN-83 TWELFTH JANUARY   NINETEEN EIGHTY-THREE
03-DEC-81 THIRD DECEMBER  NINETEEN EIGHTY-ONE
03-DEC-81 THIRD DECEMBER  NINETEEN EIGHTY-ONE
23-JAN-82 TWENTY-THIRD JANUARY   NINETEEN EIGHTY-TWO
```

ANALYSIS:

Converts all hire dates in EMP table into Words

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'Q') FROM dual;
```

OUTPUT:

```
SYSDATE       TO_CHAR(SYSDATE,'Q')
---------     ------------------------------
24-MAR-07      1
```

ANALYSIS:

Gives in the quarter the given date falls

INPUT:

```
SQL> SELECT TO_CHAR(TO_DATE('10-SEP-2005'),'Q') FROM dual;
```

OUTPUT:

```
TO_CHAR(TO_DATE('10-SEP-2005'),'Q')
----------------------------------------
3
```

ANALYSIS:

To_date is data conversion function, which converts given string into date type

INPUT:

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'W') FROM dual;
```

OUTPUT:

```
SYSDATE       TO_CHAR(SYSDATE,'W')
---------     ------------------------------
24-MAR-07     4
```

ANALYSIS:

Gives the week number in the current month (In which week given date falls in the current month)

INPUT:
```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE,'WW') FROM dual;
```

OUTPUT:
```
SYSDATE     TO_CHAR(SYSDATE)
---------   ----------------------------
24-MAR-07   12
```

ANALYSIS:
```
Returns no. of weeks worked during the year.
```

INPUT:
```
SQL> SELECT TO_CHAR(SYSDATE,'HH:MI:SS AM') FROM dual;
```

OUTPUT:
```
TO_CHAR(SYS
-----------
08:40:17 PM
```

ANALYSIS:
```
HH     returns Hours         }
MI     returns Minutes       } Returns time from current date
SS     returns Seconds       }
AM     returns AM / PM depends on Time
```

INPUT:
```
SQL> SELECT TO_CHAR(SYSDATE,'HH24:MI:SS') FROM dual;
```

OUTPUT:
```
TO_CHAR(
--------
20:43:12
```

ANALYSIS:
```
HH24     returns Hours in 24 hour format      }
MI       returns Minutes                      } Returns time from current date
SS       returns Seconds                      }
```

INPUT:
```
SQL> SELECT TO_CHAR(12567,'99,999.99') FROM dual;
```

OUTPUT:
```
TO_CHAR(12567,'99,999.99')
------------------------------
 12,567.00
```

ANALYSIS:
```
Converts the given number into comma format with two decimal places
```

INPUT:

```
SQL> SELECT TO_CHAR(12567,'L99,999.99') FROM dual;
```

OUTPUT:

```
TO_CHAR(12567,'L99,999.99')
----------------------------
 $12,567.00
```

ANALYSIS:

Display the local currency symbol

INPUT:

```
SQL> SELECT TO_CHAR(-12567,'L99,999.99PR') FROM dual;
```

OUTPUT:

```
TO_CHAR(-12567,'L99,999.99PR')
-----------------------------------
<$12,567.00>
```

ANALYSIS:

PR Parenthesis negative number

TO_NUMBER

TO_NUMBER is the companion function to TO_CHAR, and of course, it converts a string into a number.

INPUT:

```
SQL> SELECT sal, TO_NUMBER((TO_CHAR(sal)) FROM emp;
```

OUTPUT:

```
       SAL TO_NUMBER(TO_CHAR(SAL))
---------- ------------------------
       800                      800
      1600                     1600
      1250                     1250
      2975                     2975
      1250                     1250
      2850                     2850
      2450                     2450
      3000                     3000
      5000                     5000
      1500                     1500
      1100                     1100
       950                      950
      3000                     3000
      1300                     1300
```

ANALYSIS:

After conversion, Converted information is right aligned. So we can say that it is a number.

Date and Time Functions

We live in a civilization governed by times and dates, and most major implementations of SQL have functions to cope with these concepts.

It demonstrates the time and date functions.

ADD_MONTHS(expr1,expr2)

This function adds a number of months to a specified date.

For example, say a customer deposited some amount on a particular date for a period of 6 months. To find the maturity date of the deposit

INPUT:

```
SQL> SELECT ADD_MONTHS (SYSDATE, 6) MATURITY_DATE FROM dual;
```

OUTPUT:

```
MATURITY_DATE
--------------------
24-SEP-07
```

ANALYSIS:

```
It adds 6 months to the system date
```

INPUT:

```
SQL> SELECT hiredate, TO_CHAR(ADD_MONTHS(hiredate,33*12),'DD/MM/YYYY') RETIRE_DATE
FROM emp;
```

OUTPUT:

```
HIREDATE  RETIRE_DATE
--------- ---------------
17-DEC-80 17/12/2013
20-FEB-81 20/02/2014
22-FEB-81 22/02/2014
02-APR-81 02/04/2014
28-SEP-81 28/09/2014
01-MAY-81 01/05/2014
09-JUN-81 09/06/2014
09-DEC-82 09/12/2015
17-NOV-81 17/11/2014
08-SEP-81 08/09/2014
12-JAN-83 12/01/2016
03-DEC-81 03/12/2014
03-DEC-81 03/12/2014
23-JAN-82 23/01/2015
```

ANALYSIS:

```
Displaying the retirement date with century.
```

INPUT:

```
SQL> SELECT hiredate, ADD_MONTHS(hiredate,33*12) RETIRE_DATE FROM emp;
```

OUTPUT:

```
HIREDATE  RETIRE_DATE
--------- ---------------
17-DEC-80 17-DEC-13
20-FEB-81 20-FEB-14
22-FEB-81 22-FEB-14
02-APR-81 02-APR-14
28-SEP-81 28-SEP-14
01-MAY-81 01-MAY-14
09-JUN-81 09-JUN-14
09-DEC-82 09-DEC-15
17-NOV-81 17-NOV-14
08-SEP-81 08-SEP-14
12-JAN-83 12-JAN-16
03-DEC-81 03-DEC-14
03-DEC-81 03-DEC-14
23-JAN-82 23-JAN-15
```

ANALYSIS:

Find the retirement date of an employee Assume, 33 years of service from date of join is retirement date

LAST_DAY(expr1)

LAST_DAY returns the last day of a specified month.

For example, you need to know what the last day of the month by issueing the command on 12/1/11 you get the following result

INPUT:

```
SELECT LAST_DAY(sysdate) FROM DUAL;
```

OUTPUT:

```
LAST_DAY(SYSDATE)
-----------------
31-JAN-11
```

ANALYSIS:

The last_date function automatically calculates the last of the current month on which the function is applied and return that date.

MONTHS_BETWEEN(expr1,expr2)

Used to find the number of months between two given months

INPUT:

```
SQL> SELECT ename,MONTHS_BETWEEN(SYSDATE,hiredate)/12 EXPERIENCE FROM emp;
```

OUTPUT:

```
ENAME      EXPERIENCE
---------- ----------
SMITH      26.2713494
ALLEN      26.0966182
WARD       26.0912419
JONES      25.9783387
MARTIN     25.4917795
BLAKE      25.8976935
CLARK      25.7928548
SCOTT      24.2928548
KING       25.3546827
TURNER      25.545543
ADAMS      24.2014569
JAMES      25.3089838
FORD       25.3089838
MILLER      25.171887
```

ANALYSIS:

Finds number of months between sysdate and hiredate. Result is divided with 12 to get the experience

LOCALTIMESTAMP

Returns Local timestamp in the active time zone, with no time zone information shown

INPUT:

```
SQL> SELECT LOCALTIMESTAMP FROM dual;
```

OUTPUT:

```
LOCALTIMESTAMP
---------------------------------
25-MAR-07 06.21.02.312000 PM
```

***NEW_TIME*(expr1,expr2,expr3)**

This function is used to adjust the time according to the time zone you are in.

Here are the time zones you can use with this function:

Abbreviation Time Zone

- AST or ADT Atlantic standard or daylight time
- BST or BDT Bering standard or daylight time
- CST or CDT Central standard or daylight time
- EST or EDT Eastern standard or daylight time
- GMT Greenwich mean time
- HST or HDT Alaska-Hawaii standard or daylight time
- MST or MDT Mountain standard or daylight time
- NST Newfoundland standard time
- PST or PDT Pacific standard or daylight time
- YST or YDT Yukon standard or daylight time

You can adjust your time like this:

INPUT:

```
SQL>  SELECT  TO_CHAR(NEW_TIME(LOCALTIMESTAMP,'EsT','PDT'),'DD/MM/YYYY  HH  :  MI  :SS
PM') FROM dual;
```

OUTPUT:

```
TO_CHAR(NEW_TIME(LOCALTIM
-------------------------
25/03/2007 04 : 32 :55 PM
```

INPUT:

```
SQL> SELECT hiredate,NEW_TIME(hiredate,'EsT','PDT') FROM emp;
```

OUTPUT:

```
HIREDATE  NEW_TIME(
--------- ---------
17-DEC-80 16-DEC-80
20-FEB-81 19-FEB-81
22-FEB-81 21-FEB-81
02-APR-81 01-APR-81
28-SEP-81 27-SEP-81
01-MAY-81 30-APR-81
09-JUN-81 08-JUN-81
09-DEC-82 08-DEC-82
17-NOV-81 16-NOV-81
08-SEP-81 07-SEP-81
12-JAN-83 11-JAN-83
03-DEC-81 02-DEC-81
03-DEC-81 02-DEC-81
23-JAN-82 22-JAN-82
```

ANALYSIS:

```
Like magic, all the times are in the new time zone and the dates are adjusted.
```

NEXT_DAY(expr1,expr2)

NEXT_DAY finds the name of the first day of the week that is equal to or later than another specified date.

INPUT:

```
SQL> SELECT NEXT_DAY(SYSDATE,'MONDAY') FROM dual;
```

OUTPUT:

```
NEXT_DAY(
---------
26-MAR-07
```

ANALYSIS:

If the sysdate is Saturday, March 24, 2007, It display the date of the next coming Monday.

EXTRACT

We can use Extract function in the place of to_char function from Oracle 9i.

```
SELECT EXTRACT(MONTH FROM SYSDATE) FROM dual;

SELECT EXTRACT(DAY FROM SYSDATE) FROM dual;

SELECT EXTRACT(YEAR FROM SYSDATE) FROM dual
```

Interval Data types (Only from 9i)

Oracle supports two interval datatypes. Both were introduced in Oracle9i Database, and both conform to the ISO SQL standard.

INTERVAL YEAR to MONTH

Allows you to define and interval of time in terms of years and months

INTERVAL DAY TO SECOND

Allows you to define an interval of time in terms of days, hours, minutes and seconds (including fractional seconds).

INPUT:

```
SQL> SELECT (SYSDATE - TO_DATE('10-jan-2004')) YEAR TO MONTH FROM dual;
```

OUTPUT:

```
(SYSDATE-TO_DATE('10-JAN-2004'))YEARTOMONTH
---------------------------------------------------------------------------
+08-00
```

ANALYSIS:

```
Returns the difference. No of years and months between both the dates.
```

INPUT:

```
SQL> SELECT (SYSDATE - TO_DATE('10-NOV-2007')) DAY TO SECOND FROM dual;
```

OUTPUT:

```
(SYSDATE - TO_DATE('10-NOV-2007'))DAYTOSECOND
---------------------------------------------------------------------------
+000000009 11:04:07
```

ANALYSIS:

```
Find the difference with days and time. For sysdate it takes 0.00 hrs as starting
time
```

Note :- We can also change the default format for a given session using the ALTER SESSION command

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'DD/MM/YYYY';

SQL> SELECT SYSDATE FROM dual:
```

Interpreting Two-digit Years

Oracle provides the RR format element to interpret two-digit years.

If the Current year is in the first half of the century (years 0 through 49), then:

- If you enter a date in the first half of the century (i.e. from 0 through 49), RR returns the current century.
- If you enter a date in the latter half of the century (i.e. from 50 through 99), RR returns the previous century.

INPUT:

```
SQL> SELECT TO_CHAR(SYSDATE,'DD/MM/YYYY') "CURRENT DATE",
TO_CHAR(TO_DATE('14-OCT-88','DD-MON-RR'),'YYYY') "YEAR 88",
TO_CHAR(TO_DATE('14-OCT-18','DD-MON-RR'),'YYYY') "YEAR 18" FROM dual
```

OUTPUT:

```
CURRENT DATE               YEAR 88              YEAR 18
-------------------------- -------------------- --------------------
19/11/2007                 1988                 2018

When we reach the year 2050, RR will interpret the same dates differently

CURRENT DATE               YEAR 88              YEAR 18
-------------------------- -------------------- --------------------
19/11/2050                 2088                 2118
```

CONVERTING NUMBER TO INTERVALS

The NUMTOYMINTERVAL and NUMTODSINTERVAL functions allow you to convert a single numeric value to one of the interval data types. The function NUMTOYMINTERVAL (pronounced " num to Y M interval") converts a numeric value an interval of type INTERVAL YEAR TO MONTH.

INPUT:

```
SQL> SELECT NUMTOYMINTERVAL (10.5,'YEAR') FROM dual;
```

OUTPUT:

```
NUMTOYMINTERVAL (10.5,'YEAR')
---------------------------------------------------------------------------
+000000010-06
```

INPUT:

```
SQL> SELECT NUMTOYMINTERVAL (10.3,'YEAR') FROM dual;
```

OUTPUT:

```
NUMTOYMINTERVAL (10.3,'YEAR')
---------------------------------------------------------------------------
+000000010-03
```

Try the following formats

- SELECT NUMTOYMINTERVAL(10.5,'MONTH') FROM DUAL;
- SELECT NUMTOYMINTERVAL(10.5,'DAY') FROM DUAL;

Name	Description
YEAR	Some number of years ranging from 1 through 999,999,999
MONTH	Some number of months ranging from 0 through 11
DAY	Some number of days ranging from 0 through 999,999,999
HOUR	Some number of hours ranging from 0 through 23
MINUTE	Some number of minutes ranging from 0 through 59
SECOND	Some number of seconds ranging from 0 through 59.999999999

NUMTODSINTERVAL

This function allows to convert a single numeric value to one of the interval data types.

This function (pronounced "num to D S interval") likewise converts a numeric value to an interval of type INTERVAL DAY TO SECOND.

INPUT:

```
SQL> SELECT NUMTODSINTERVAL(1440,'MINUTE') FROM dual;
```

OUTPUT:

```
NUMTODSINTERVAL(1440,'MINUTE')
------------------------------
+01 00:00.00.000000
```

ANALYSIS:

Oracle automatically taken care of normalizing the input value of 1440 minutes to an interval value of 1 day.

Miscellaneous Functions

Here are three miscellaneous functions you may find useful.

***GREATEST*(list) *and* *LEAST*(list)**

INPUT:

```
SQL> SELECT GREATEST(10,1,83,2,9,67) FROM dual;
```

OUTPUT:

```
GREATEST
---------
   83
```

ANALYSIS:

```
Displays the greatest of the given set of values
```

Difference between GREATEST AND MAX IS

GREATEST IS SINGLE ROW FUNCTION, MAX IS A GROUP FUNCTION. GREATEST TAKES VALUES FROM DIFFERENT COLUMNS FROM EACH ROW, WHERE AS MAX TAKES VALUES FROM DIFFERENT ROWS FROM A COLUMN.

Assume there is a student table

ROLLNO	NAME	SUB1	SUB2	SUB3	SUB4
1	RAVI	55	22	86	45
2	KRIS	78	55	65	12
3	BABU	55	22	44	77
4	ANU	44	55	66	88

To find the greatest and Least Mark we can use the GREATEST and LEAST functions as follows.

INPUT:

```
SQL> SELECT name,sub1,sub2,sub3,sub4, GREATEST(sub1,sub2,sub3,sub4) GREATEST_MARK,
LEAST(sub1,sub2,sub3,sub4) LEAST_MARK FROM student;
```

OUTPUT:

```
ROLLNO     NAME      SUB1     SUB2     SUB3     SUB4     GREATEST_MARK     LEAST_MARK
------     ------    ----     ----     ----     ----     -------------     ----------
     1     RAVI        55       22       86       45                86             22
     2     KRIS        78       55       65       12                78             12
     3     BABU        55       22       44       77                77             22
     4     ANU         44       55       66       88                88             44
```

USER

USER returns the character name of the current user of the database.

INPUT:

```
SQL> SELECT USER FROM dual;
```

OUTPUT:

```
USER
--------
SCOTT
```

ANALYSIS:

```
Displays the current sessions user name We can also display username using
environment command

SQL> SHOW USER
```

***DECODE*(expr1,expr2,expr3[,...])**

The DECODE function is one of the most powerful commands in SQL*Plus--and perhaps the most powerful. The standard language of SQL lacks procedural functions that are contained in languages such as COBOL and C. The DECODE statement is similar to an IF...THEN statement in a procedural programming language. Where flexibility is required for complex reporting needs, DECODE is often able to fill the gap between SQL and the functions of a procedural language.

SYNTAX: DECODE (column1, value1, output1, value2, output2, output3)

The syntax example performs the DECODE function on column1. If column1 has a value of value1, then display output1 instead of the column's current value. If column1 has a value of value2, then display output2 instead of the column's current value. If column1 has a value of anything other than value1 or value2, then display output3 instead of the column's current value.

INPUT:

```
SQL> SELECT ename,job,DECODE(job,'CLERK','EXEC','SALESMAN',
'S.OFFICER','ANALYST','PM','MANAGER','VP',JOB) PROMOTION FROM emp;
```

OUTPUT:

```
ENAME      JOB       PROMOTION
---------- --------- ---------
SMITH      CLERK     EXEC
ALLEN      SALESMAN  S.OFFICER
WARD       SALESMAN  S.OFFICER
JONES      MANAGER   VP
MARTIN     SALESMAN  S.OFFICER
BLAKE      MANAGER   VP
CLARK      MANAGER   VP
SCOTT      ANALYST   PM
KING       PRESIDENT PRESIDENT
TURNER     SALESMAN  S.OFFICER
ADAMS      CLERK     EXEC
JAMES      CLERK     EXEC
FORD       ANALYST   PM
MILLER     CLERK     EXEC
```

ANALYSIS:

```
When JOB has a value CLERK , then display EXEC instead of CLERK
When JOB has a value SALESMAN , then display S.OFFICER instead of SALESMAN
When JOB has a value ANALYST , then display PM instead of ANALYST
When JOB has a value MANAGER , then display VP instead of MANAGER
OTHERWISE DISPLAY SAME JOB
```

INPUT:

```
SQL> SELECT ename, job, sal,DECODE(job,'CLERK',SAL*1.1,'SALESMAN',
sal*1.2,'ANALYST',sal*1.25,'MANAGER',sal*1.3,SAL)  NEW_SAL FROM emp;
```

OUTPUT:

ENAME	JOB	SAL	NEW_SAL
SMITH	CLERK	800	880
ALLEN	SALESMAN	1600	1920
WARD	SALESMAN	1250	1500
JONES	MANAGER	2975	3867.5
MARTIN	SALESMAN	1250	1500
BLAKE	MANAGER	2850	3705
CLARK	MANAGER	2450	3185
SCOTT	ANALYST	3000	3750
KING	PRESIDENT	5000	5000
TURNER	SALESMAN	1500	1800
ADAMS	CLERK	1100	1210
JAMES	CLERK	950	1045
FORD	ANALYST	3000	3750
MILLER	CLERK	1300	1430

ANALYSIS:

```
When JOB has a value CLERK , then giving 10% increment
When JOB has a value SALESMAN , then giving 20% increment
When JOB has a value ANALYST , then giving 25% increment
When JOB has a value MANAGER , then giving 30% increment
OTHERWISE no increment
```

Assume there is a table with empno, ename, sex

INPUT:

```
SQL> SELECT ename,sex,DECODE(sex,'MALE','MR.'||ename,'MS.'||ename) FROM emp;
```

ANALYSIS:

```
Adding Mr.' or 'Ms.' before the name based on their Gender
```

CASE

As of Oracle 9i, you can use the CASE function in place of DECODE. The CASE function uses the keywords when, then, else, and end to indicate the logic path followed, which may make the resulting code easier to follow than an equivalent DECODE.

INPUT:

```
SQL> SELECT job,
    CASE job
    WHEN 'MANAGER' then 'VP'
    WHEN 'CLERK'   THEN 'EXEC'
    WHEN 'SALESMAN' THEN 'S.OFFICER'
    ELSE
       job
    END
    FROM emp;
```

OUTPUT:

```
JOB       CASEJOBWH
--------- ---------
CLERK     EXEC
SALESMAN  S.OFFICER
SALESMAN  S.OFFICER
MANAGER   VP
SALESMAN  S.OFFICER
MANAGER   VP
MANAGER   VP
ANALYST   ANALYST
PRESIDENT PRESIDENT
SALESMAN  S.OFFICER
CLERK     EXEC
CLERK     EXEC
ANALYST   ANALYST
CLERK     EXEC
```

ANALYSIS:

Works similar to that of DECODE

NVL

If the value is NULL, this function is equal to substitute. If the value is not NULL, this function is equal to value. Substitute can be a literal number, another column, or a computation.

NVL is not restricted to numbers, it can be used with CHAR, VARCHAR2, DATE, and other data types, but the value and substitute must be the same data type.

INPUT:

```
SQL> SELECT  empno,sal,comm.,sal+comm TOTAL FROM emp;
```

OUTPUT:

```
     EMPNO        SAL       COMM      TOTAL
---------- ---------- ---------- ----------
      7369        800
      7499       1600        300       1900
      7521       1250        500       1750
      7566       2975
      7654       1250       1400       2650
      7698       2850
      7782       2450
      7788       3000
      7839       5000
      7844       1500          0       1500
      7876       1100
      7900        950
      7902       3000
      7934       1300
```

ANALYSIS:

```
Arithmetic operation is possible only when value is there in both columns
```

INPUT:

```
SQL> SELECT empno, sal, comm, sal + NVL(comm,0) TOTAL FROM emp;
```

OUTPUT:

```
     EMPNO        SAL       COMM      TOTAL
---------- ---------- ---------- ----------
      7369        800                   800
      7499       1600        300       1900
      7521       1250        500       1750
      7566       2975                  2975
      7654       1250       1400       2650
      7698       2850                  2850
      7782       2450                  2450
      7788       3000                  3000
      7839       5000                  5000
      7844       1500          0       1500
      7876       1100                  1100
      7900        950                   950
      7902       3000                  3000
      7934       1300                  1300
```

ANALYSIS:

```
Using NVL, we are substituting 0 if COMM is NULL.
```

LENGTH

Finds the length of the given information

```
SQL> SELECT ename, LENGTH(ename)FROM emp;

SQL> SELECT LENGTH(SYSDATE) FROM emp;

SQL> SELECT sal, LENGTH(sal) FROM emp;
```

ASCII

Finds the ASCII value of the given character

```
SQL> SELECT ASCII('A') FROM dual;
```

CAST

Converts one type of information into another type

```
SQL> SELECT 50 NUMB, CAST(50 as VARCHAR2(2)) VALUE FROM dual;
```

Exercises

1. _____ function performs one to one character substitution.
2. _____ format option is used to get complete year spelled out in TO_CHAR function.
3. _____ symbol is used to combine tow given strings
4. What happens if “replace string” is not given for REPLACE function
5. Can a number be converted to DATE?
6. Convert the value of name in the EMP table to lower case letters
7. Display the names of the employees who have more than 4 characters in the name.
8. Print *’s as number of thousands are there in the number
9. Display the ename, comm. If the commission is NULL, print as NO COMM
10. Add number of days to the given date
11. Display the first and last two characters from a given name and combine them as a single string (Use only functions)
12. Find the difference between two given dates
13. Display all the names which contain underscore
14. subtract number of months from given date

GROUP BY clause with SELECT statement

Syntax: SELECT [column,] group_function(column)...
FROM table
[WHERE condition]
[GROUP BY [CUBE | ROLLUP] group_by_expression]
[HAVING having_expression]
[ORDER BY column];

Group by statement groups all the rows with the same column value. Use to generate summary output from the available data. Whenever we use a group function in the SQL statement, we have to use a group by clause.

INPUT:

```
SQL> SELECT job, COUNT (*) FROM emp GROUP BY job;
```

OUTPUT:

```
JOB          COUNT(*)
--------- ----------
ANALYST             2
CLERK               4
MANAGER             3
PRESIDENT           1
SALESMAN            4
```

ANALYSIS:

Counts number of employees under each and every job. When we are grouping on job, initially jobs are placed in ascending order in a temporary segment. On the temporary segment, group by clause is applied, so that on each similar job count function applied.

INPUT:

```
SQL> SELECT job, SUM(sal) FROM emp GROUP BY job;
```

OUTPUT:

```
JOB          SUM(SAL)
--------- ----------
ANALYST          6000
CLERK            4150
MANAGER          8275
PRESIDENT        5000
SALESMAN         5600
```

ANALYSIS:

With each job, it finds the total salary

ERROR with GROUP BY Clause

Note :

- Only grouped columns allowed in the group by clause
- Whenever we are using a group function in the SQL statement, we have to use group by clause.

INPUT:

```
SQL> SELECT job,COUNT(*) FROM emp;
```

OUTPUT:

```
SELECT job, COUNT (*) FROM emp
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

ANALYSIS:

This result occurs because the group functions, such as SUM and COUNT, are designated to tell you something about a group or rows, not the individual rows of the table. This error is avoided by using JOB in the group by clause, which forces the COUNT to count all the rows grouped within each job.

INPUT:

```
SQL> SELECT job,ename,COUNT(*) FROM emp GROUP BY job;
```

OUTPUT:

```
SELECT JOB,ENAME,COUNT(*) FROM EMP GROUP BY JOB
           *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

ANALYSIS:

In the above query, JOB is only the grouped column where as ENAME column is not a grouped column. Whatever the columns we are grouping, the same column is allowed to display.

INPUT:

```
SQL> SELECT job, MIN(sal),MAX(sal) FROM emp GROUP BY job;
```

OUTPUT:

```
JOB          MIN(SAL)   MAX(SAL)
--------- ---------- ----------
ANALYST          3000       3000
CLERK             800       1300
MANAGER          2450       2975
PRESIDENT        5000       5000
SALESMAN         1250       1600
```

ANALYSIS:

With each job, it finds the MINIMUM AND MAXIMUM SALARY

For displaying Total summary information from the table.

INPUT:

```
SQL> SELECT job, SUM(sal),AVG(sal),MIN(sal),MAX(sal) ,COUNT(*) FROM emp GROUP BY
job;
```

OUTPUT:

```
JOB          SUM(SAL)    AVG(SAL)    MIN(SAL)    MAX(SAL)    COUNT(*)
--------- ---------- ---------- ---------- ---------- ----------
ANALYST          6000       3000       3000       3000          2
CLERK            4150     1037.5        800       1300          4
MANAGER          8275 2758.33333       2450       2975          3
PRESIDENT        5000       5000       5000       5000          1
SALESMAN         5600       1400       1250       1600          4
```

ANALYSIS:

With each job, finds the total summary information.

To display the output Designation wise, Department wise total salaries With a matrix style report.

INPUT:

```
SQL>   SELECT   job,SUM(DECODE(deptno,10,sal))   DEPT10,   SUM(DECODE(deptno,20,sal))
DEPT20,
     SUM(DECODE(deptno,30,sal)) DEPT30, SUM(sal) TOTAL FROM emp GROUP BY job;
```

OUTPUT:

```
JOB           DEPT10     DEPT20     DEPT30      TOTAL
--------- ---------- ---------- ---------- ----------
ANALYST                    6000                  6000
CLERK           1300       1900        950       4150
MANAGER         2450       2975       2850       8275
PRESIDENT       5000                             5000
SALESMAN                                5600       5600
```

ANALYSIS:

When we apply group by, initially all the designations are placed in ascending order of designations. Then group by clause groups similar designations, then DECODE function (Single row function) applies on each and every row of that group and checks the DEPTNO. If DEPTNO=10, it passes corresponding salary as an argument to SUM() .

INPUT:

```
SQL> SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job;
```

OUTPUT:

```
    DEPTNO JOB          COUNT(*)
---------- --------- ----------
        10 CLERK              1
        10 MANAGER            1
        10 PRESIDENT          1
        20 CLERK              2
        20 ANALYST            2
        20 MANAGER            1
        30 CLERK              1
        30 MANAGER            1
        30 SALESMAN           4
```

ANALYSIS:

Department wise, Designation wise , counts the number of employees

To display the DEPTNO only one time (output with breaks)

INPUT:

```
SQL> BREAK ON DEPTNO SKIP 1
SQL> SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job;
```

OUTPUT:

```
    DEPTNO JOB          COUNT(*)
---------- --------- ----------
        10 CLERK              1
           MANAGER            1
           PRESIDENT          1

        20 CLERK              2
           ANALYST            2
           MANAGER            1

        30 CLERK              1
           MANAGER            1
           SALESMAN           4
```

ANALYSIS:

Break is Environment command , which breaks the information on repetitive column and displays them only once.
SKIP 1 used with BREAK to leave one blank line after completion of each Deptno.

To remove the given break , we have to use an Environment command

```
SQL> CLEAR BREAK;
```

Group by with ROLLUP and CUBE Operators

- Use Rollup or CUBE with Group by to produce super aggregate rows by cross-referencing columns.
- ROLLUP grouping produces a result set containing the regular grouped rows and the subtotal values.
- CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows

The ROLLUP and CUBE operators are available only in Oracle8i and later releases.

CUBE function

We can use CUBE function to generate subtotals for all combinations of the values in the group by clause.(CUBE and ROLLUP are available only from 9i)

INPUT:

```
SQL> SELECT deptno,job,COUNT(*) FROM emp GROUP BY CUBE(deptno,job);
```

OUTPUT:

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

ANALYSIS:

Cube displays the out with all the permutation and combination of all the columns given a CUBE function.

ROLLUP FUNCTION

It is similar to that of CUBE function

INPUT:

```
SQL> SELECT deptno,job,COUNT(*) FROM emp GROUP BY ROLLUP(deptno,job)
```

OUTPUT:

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

ANALYSIS:

Observe the output. The count(*) column total is automatically displayed with every change in the deptno.

COLLECT(<colname>) from oracle 10g

This function enables us to aggregate data into a collection, retaining multiple records of data within a single row (like a nested table). One of the main benefits of this function is that it makes "string aggregation" (one of the web's most-requested Oracle technique) very simple. This article will introduce the COLLECT function and then demonstrate how it can be used to aggregate multiple records into a single value (a technique known as "string aggregation").

INPUT:

```
SQL> SELECT deptno, COLLECT(ename) AS emps FROM emp GROUP BY deptno;

DEP EMPS
---     ------------------------------------------------------------------------
10  SYSTPXeCjDqbWSqWrshgYrRPR4Q==('CLARK', 'KING')
20  SYSTPXeCjDqbWSqWrshgYrRPR4Q==('SMITH', 'JONES', 'SCOTT', 'ADAMS', 'FORD')
30  SYSTPXeCjDqbWSqWrshgYrRPR4Q==('ALLEN','WARD','MARTIN','BLAKE','TURNER', 'JAMES')
40  SYSTPXeCjDqbWSqWrshgYrRPR4Q==('MILLER')

4 rows selected.
```

ANALYSIS:

we can see that the COLLECT function has aggregated the employee names per department as requested.

Oracle has created a collection type to support the COLLECT function each name startin with SYSTP. This informaiton is available in the data dictionary USER_TYPES

HAVING CLAUSE

Whenever we are using a group function in the condition, we have to use having clause. Having clause is used along with group by clause.

For example, to display Designation wise total salaries

INPUT:

```
SQL> SELECT job,SUM(sal) FROM emp GROUP BY job;
```

OUTPUT:

```
JOB          SUM(SAL)
--------- ----------
ANALYST          6000
CLERK            4150
MANAGER          8275
PRESIDENT        5000
SALESMAN         5600
```

To Display only those designations, whose total salary is more than 5000

INPUT:

```
SQL> SELECT job,SUM(sal) FROM emp WHERE SUM(sal) > 5000 GROUP BY job;
```

OUTPUT:

```
SELECT JOB,SUM(SAL) FROM EMP WHERE SUM(SAL) > 5000 GROUP BY JOB
                               *
ERROR at line 1:
ORA-00934: group function is not allowed here
```

ANALYSIS:

Where clause doesn't allow using group function in the condition.
When we are using group function in the condition, we have to use having clause.

INPUT:

```
SQL> SELECT job,SUM(sal) FROM emp GROUP BY job HAVING SUM(sal) > 5000;
```

OUTPUT:

```
JOB          SUM(SAL)
--------- ----------
ANALYST          6000
MANAGER          8275
SALESMAN         5600
```

ANALYSIS:

Displays all the designations whose total salary is more than 5000.

INPUT:

```
SQL> SELECT job,COUNT(*) FROM emp GROUP BY job HAVING COUNT(*) BETWEEN 3 AND 5;
```

OUTPUT:

```
JOB          COUNT(*)
--------- ----------
CLERK                4
MANAGER              3
SALESMAN             4
```

ANALYSIS:

Displays all the designations whose number where employees between 3 and 5

INPUT:

```
SQL> SELECT sal FROM emp GROUP BY sal HAVING COUNT(sal) > 1;
```

OUTPUT:

```
       SAL
----------
      1250
      3000
```

ANALYSIS:

Displays all the salaries, which are appearing more than one time in the table.

POINTS TO REMEMBER

- WHERE clause can be used to check for conditions based on values of columns and expressions but not the result of GROUP functions.
- HAVING clause is specially designed to evaluate the conditions that are based on group functions such as SUM, COUNT etc.
- HAVING clause can be used only when GROUP BY clause is used.

ORDER OF EXECUTION

Here are the rules ORCALE uses to execute different clauses given in SELECT command

- Selects rows based on Where clause
- Groups rows based on GROUP BY clause
- Calculates results for each group
- Eliminate groups based on HAVING clause
- Then ORDER BY is used to order the results

INPUT:
```
SQL> SELECT job,SUM(sal) FROM emp WHERE job != 'CLERK'
     GROUP BY job HAVING SUM(sal) > 5000  ORDER BY job DESC;
```

Nested Sub queries

Oracle has the ability to execute queries which are more complex than the ones that we have seen till now. In some situations, we may have to embed queries with in another query. This system called sub-query. Nesting is the act of embedding a sub query within another query.

Syntax: SELECT * FROM <tablename> WHERE (SELECT … (SELECT … (SELECT …)));

Whenever particular information is not accessible through a single query, then we have to write different queries one included in another.

Sub queries can be nested as deeply as your implementation of SQL allows. We can write different types sub queries

- Single row sub queries
- Multi row sub queries
- Multi column sub queries
- Correlated sub queries.

Single row sub query

A Sub query which returns only one value. For example**,** To get the employee, who is drawing maximum salary?

INPUT:
```
SQL> SELECT ENAME,SAL FROM EMP WHERE SAL = ( SELECT MAX(SAL) FROM EMP);
```

OUTPUT:
```
ENAME              SAL
------------       ----------
KING                     5000
```

ANALYSIS:
Right side query is called as child query and left side query is called parent query. In nested queries, child query executes first before executing parent query.

INPUT:

```
SQL> SELECT ename, hiredate FROM emp WHERE hiredate = (SELECT MAX(hiredate) FROM
emp);
```

OUTPUT:

```
ENAME      HIREDATE
---------- ---------
ADAMS      12-JAN-83
```

ANALYSIS:

Display the least experienced employee

INPUT:

```
SQL> SELECT ename,sal FROM emp WHERE sal < (SELECT MAX(sal) FROM emp);
```

OUTPUT:

```
ENAME                SAL
---------- ----------
SMITH                800
ALLEN               1600
WARD                1250
JONES               2975
MARTIN              1250
BLAKE               2850
CLARK               2450
SCOTT               3000
TURNER              1500
ADAMS               1100
JAMES                950
FORD                3000
MILLER              1300
```

ANALYSIS:

Display all the employees whose salary is less than the maximum salary of all the employees.

Try the following?

To display all the employees whose salary lines between minimum and maximum salaries

Answer:

```
SQL> SELECT * FROM emp WHERE sal BETWEEN (SELECT MIN(sal) FROM emp) AND (SELECT
MAX(sal) FROM emp);
```

Display all the employees who are getting maximum commission in the organization

Answer:

```
SQL> SELECT * FROM emp WHERE comm = (SELECT MAX(comm) FROM emp);
```

Display all the employees from department 30 whose salary is less than maximum salary of department 20.

Answer:

```
SQL> SELECT empno, ename, sal FROM emp WHERE deptno=30
     AND sal < (SELECT MAX(sal) FROM emp WHERE deptno = 20);
```

Multi row Sub queries

A sub query, which returns more than one value.

INPUT:

```
SQL> SELECT ename,sal FROM emp WHERE sal IN(SELECT sal FROM emp GROUP BY sal HAVING
     COUNT(*)> 1);
```

OUTPUT:

```
ENAME              SAL
----------         ---------
WARD                    1250
MARTIN                  1250
SCOTT                   3000
FORD                    3000
```

ANALYSIS:

```
Displays all the employees who are drawing similar salaries
```

When child query returns more than one value, we have to use IN operator for comparison.

Multi Column Sub Queries

When sub queries returns values from different columns.

INPUT:

```
SQL> SELECT empno,ename,deptno,sal FROM emp WHERE (deptno,sal)
     IN (SELECT deptno,MAX(sal) FROM emp GROUP BY deptno);
```

OUTPUT:

```
     EMPNO ENAME           DEPTNO        SAL
---------- ---------- ---------- ----------
      7839 KING               10       5000
      7788 SCOTT              20       3000
      7902 FORD               20       3000
      7698 BLAKE              30       2850
```

ANALYSIS:

Display all the employees who are drawing maximum salaries in each department

Correlated Sub Queries

A correlated sub query is a sub query that receives a value from the main query and then sends a value back to main query.

For example, Display all the employees whose salary is less than maximum salary of each department

INPUT:

```
SQL> SELECT empno,ename,deptno,sal FROM emp X WHERE sal < (SELECT MAX(sal)
     FROM emp WHERE deptno = x.deptno);
```

OUTPUT:

```
     EMPNO ENAME           DEPTNO        SAL
---------- ---------- ---------- ----------
      7369 SMITH              20        800
      7499 ALLEN              30       1600
      7521 WARD               30       1250
      7566 JONES              20       2975
      7654 MARTIN             30       1250
      7782 CLARK              10       2450
      7844 TURNER             30       1500
      7876 ADAMS              20       1100
      7900 JAMES              30        950
      7934 MILLER             10       1300
```

ANALYSIS:

```
Find department wise maximum salaries and display the employees whose salary is less
than that value for each department
```

Execution Sequence of steps in Correlated sub queries

- A row from main query is retrieved
- Executes sub query with the value retrieved from main query
- Sub query returns a value to main query
- Main query's current row is either selected or not, depending upon the value passed by sub query.
- This continues until all rows of main query are retrieved

To display the nth highest paid employee

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp X WHERE &N = (SELECT COUNT(DISTINCT sal) FROM
emp WHERE sal >= x.sal);
```

OUTPUT:

```
     EMPNO ENAME              SAL
---------- ---------- ----------
      7788 SCOTT            3000
      7902 FORD             3000
```

ANALYSIS:

```
It selects each row from emp table from parent query and finds the distinct count
for each salary whose salary >=  the salary returned by main query.
```

The following example produces same output in two different methods with a comparison of their performance in time.

Least Efficient (156 Mill Sec)

```
SQL>SELECT SYSTIMESTAMP FROM dual;

SQL>SELECT * FROM emp E WHERE sal >=5000 AND job = 'MANAGER' AND
           25 < (SELECT COUNT(*) FROM emp WHERE mgr = e.empno);

SQL> SELECT SYSTIMESTAMP FROM dual;
```

Most Efficient (110 Mill Sec)

```
SQL>SELECT SYSTIMESTAMP FROM dual;

SQL>SELECT * FROM emp E WHERE
           25 < (SELECT COUNT(*) FROM emp WHERE mgr = e.empno)  and sal
                    >=5000 AND job = 'MANAGER'

SQL> SELECT SYSTIMESTAMP FROM dual;
```

Table joins should be written first before any condition of WHERE clause. And the conditions which filter out the maximum records should be placed at the end after the joins as the parsing is done from **BOTTOM** to **TOP.**

ANY and ALL Operators

Both are used for comparing one value against a set of values. The operator can be any one of the standard relational operators (=, >=, >, <. <= , !=) and list is a series of values.

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp WHERE sal > ANY(SELECT sal FROM emp);
```

OUTPUT:

```
     EMPNO ENAME              SAL
---------- ---------- ----------
      7499 ALLEN            1600
      7521 WARD             1250
      7566 JONES            2975
      7654 MARTIN           1250
      7698 BLAKE            2850
      7782 CLARK            2450
      7788 SCOTT            3000
      7839 KING             5000
      7844 TURNER           1500
      7876 ADAMS            1100
      7900 JAMES             950
      7902 FORD             3000
      7934 MILLER           1300
```

ANALYSIS:

```
ANY displays greater than any values in the list.
```

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp WHERE sal < ANY (SELECT sal FROM emp);
```

OUTPUT:

```
     EMPNO ENAME              SAL
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250
      7566 JONES            2975
      7654 MARTIN           1250
      7698 BLAKE            2850
      7782 CLARK            2450
      7788 SCOTT            3000
      7844 TURNER           1500
      7876 ADAMS            1100
      7900 JAMES             950
      7902 FORD             3000
      7934 MILLER           1300
```

ANALYSIS:

```
Less than ANY of the list of values
```

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp WHERE sal >ALL(SELECT sal FROM emp);
```

OUTPUT:

```
no rows selected
```

ANALYSIS:

Greater than Maximum of list

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp WHERE sal >ALL(3000,2000,4000);
```

OUTPUT:

```
     EMPNO ENAME              SAL
---------- ---------- ----------
      7839 KING             5000
```

ANALYSIS:

Greater than maximum of List

INPUT:

```
SQL> SELECT empno,ename,sal FROM emp WHERE sal <ALL(3000,2000,4000);
```

OUTPUT:

```
     EMPNO ENAME              SAL
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250
      7654 MARTIN           1250
      7844 TURNER           1500
      7876 ADAMS            1100
      7900 JAMES             950
      7934 MILLER           1300
```

ANALYSIS:

Less than minimum of List

DML STATEMENTS IN SUB QUERIES

To modify the salary of an employee who is drawing minimum salary with the salary of the employee who is drawing maximum salary

INPUT:

```
SQL> UPDATE emp SET sal = (SELECT MAX(sal) FROM emp) WHERE Empno = (SELECT empno
FROM emp WHERE sal = (SELECT MIN (sal) FROM emp));
```

ANALYSIS:

```
Identify the employee who is drawing minimum salary and update with the maximum
salary of all the employees.
```

To insert selected rows from emp table to emp1 table

INPUT:

```
SQL> INSERT INTO emp1 SELECT * FROM emp ;
```

ANALYSIS:

```
Assuming that EMP1 is an existing table.  Inserts all the selected rows into EMP1
table.
```

EXISTS And NOT EXISTS Operators

These two operators are exclusively used in correlated sub query. EXISTS checks whether any row is existing in the sub query, and NOT EXISTS does the opposite.

EXISTS is different from other operators like IN , ANY etc, because it doesn't compare values of columns, instead. It checks any row is retrieved from sub query or not. If any row is retrieved from sub query the EXISTS returns true otherwise it returns False.

INPUT:

```
SQL> SELECT empno,ename,sal,mgr FROM emp X WHERE EXISTS(SELECT mgr FROM emp WHERE
X.mgr = empno);
```

OUTPUT:

```
     EMPNO ENAME                SAL        MGR
---------- ---------- ---------- ----------
      7369 SMITH                800       7902
      7499 ALLEN               1600       7698
      7521 WARD                1250       7698
      7566 JONES               2975       7839
      7654 MARTIN              1250       7698
      7698 BLAKE               2850       7839
      7782 CLARK               2450       7839
      7788 SCOTT               3000       7566
      7844 TURNER              1500       7698
      7876 ADAMS               1100       7788
      7900 JAMES                950       7698
      7902 FORD                3000       7566
      7934 MILLER              1300       7782
```

Note: Use **EXISTS** in place of **IN** for Base Tables to improve the performance.

Remember

The following important points to be remembered while dealing with sub queries

- Sub query can not use ORDER BY clause. Because ORDER BY clause must be the last clause of SELECT
- BETWEEN … AND operator can not be used with Sub queries

Exercise based on sub-queries

1. In department 20, one employee is drawing minimum salary and is having some designation. Display the employees from other departments whose designation is matching with the designation of the above employee.
2. Display all the employees whose salary is within ±1000 from the average salary of all the employees.
3. Display the employees who reported to KING
4. Display all the employees whose salary is less than the minimum salary of MANAGERS.
5. Display the details of students who have paid the highest amount so far in their course.
6. Display the details of subjects that have been taken by more than two students

INTEGRITY CONSTRAINTS

Constraints are used to implement standard rules such as uniqueness in the key filed and business rule such as AGE column should contain a value between 15 and 60 etc.

Oracle server makes sure that the constraints are not violated whenever a row is inserted, deleted or updated. If constraint is not satisfied the operation will fail.

Constraints are normally defined at the time of creating table. But it is also possible to define constraints after the table is created.

Constraint Guidelines

- Name a constraint or the Oracle server generates a name by using the SYS_Cn format
- Create a constraint either:
  - At the same time as the table is created, or
  - After the table has been created.
- Define a constraint at the column or table level.
- View a constraint in the Data dictionary for verification

TYPES OF CONSTRAINTS

Constraints are classified into two types

- Table Constraints
- Column Constraints

Table Constraint

A constraint given at the table level is called as Table Constraint. It may refer to more than one column of the table.

A typical example is PRIMARY KEY constraint that is used to define composite primary key.

Column Constraint

A constraint given at the column level is called as Column constraint. It defines a rule for a single column. It cannot refer to column other than the column, at which it is defined.

A typical example is PRIMARY KEY constraint when a single column is the primary key of the table.

Various types of Integrity constraints

- PRIMARY KEY
- UNIQUE
- NOT NULL
- CHECK

PRIMARY KEY

It is used to uniquely identify rows in a table. There can be only one primary key in a table. It may consist of more than one column, if so, it is called as composite primary key. (It maintains uniqueness in the data and null values are not acceptable) i.e. UNIQUE + NOT NULL = PRIMARY KEY

- Automatically creates unique index to enforce uniqueness.

UNIQUE

Maintains unique and NULL values are acceptable.

- Oracle automatically creates a unique index for the column.

Example : EmailID

A UNIQUE key integrity constraint requires that every value in a column or set of columns (key) be unique- that is, no two rows of table can have duplicate values in a specified column or set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the unique key. If the UNIQUE constraint comprises more than one column, the group of columns is called a composite unique key.

NOT NULL

Uniqueness not maintained and null values are not acceptable.

Note: The NOT NULL constraint can be specified only at the column level, not at the table level.

CHECK

Defines the condition that should be satisfied before insertion and updating is done.

- Defines a condition that each row must satisfy
- The following expressions are not allowed
    - References to CURRVAL, NEXTVAL and ROWNUM pseudocolumns
    - Calls to SYSDATE,UID,USER functions
    - Queries that refer to other values in other rows

Note: - Pseudocolumns are not actual columns in a table but they behave like columns. For example, you can select values from pseudocolumns. However, you cannot insert into, update, or delete from a pseudocolumn.

Guidelines for Primary Keys and Foreign Keys

- You cannot use duplicate values in a primary key.
- Primary keys generally cannot be changed.
- Foreign keys are based on data values and are purely logical, not physical, pointers.
- A foreign key value must match an existing primary key value or unique key value, or else be null.
- A foreign key must reference either a primary key or unique key column.

DDL (Data Definition language)

CREATE, ALTER, DROP commands used in SQL are called the DDL commands. DDL STATEMENTS COMMITS AUTOMATICALLY. There is no need to save explicitly.

Create Table

Syntax: CREATE TABLE <TABLE-NAME>
(COLUMN DEFINITION1, COLUMN DEFINITION2);

Column Def:

<Name> Data type [Default Value] [constraint <name> constraint type]

Note: Min. Column in a table = 1
Max. Columns in a table = 1000

Naming Rules in oracle

1. A table or a column name must never start a number but they can contain numbers in them
2. They can't consist of any special characters other than "$", "#", "-"

i.e. $,# are used mainly for system tables.

```
SQL>CREATE TABLE empl4747(
    empno  NUMBER (3) CONSTRAINT pk_empl47473_empno PRIMARY KEY,
    ename  VARCHAR2 (10) NOT NULL,
    gender CHAR(1) CONSTRAINT chk_empl47473_gender CHECK(UPPER(gender) IN('M','F')),
    email_id VARCHAR2(30) UNIQUE,
    designation VARCHAR2(15),
    salary NUMBER(7,2) CHECK (salary BETWEEN 10000 AND 70000)
    );
```

Note:

- Constraint name is useful for manipulating the given constraint
- When the constraint name is not given at the time of defining constraints, system creates a constraint with the name SYS_Cn.
- Constraints defined on a particular table are store in a data dictionary table USER_CONSTRAINTS, USER_CONS_COLUMNS.
- Tables defined by a user are stored in a data dictionary table USER_TABLES

These definitions can be viewed by giving the following command.

INPUT:

```
SQL> DESCRIBE  USER_CONSTRAINTS

SQL> SELECT CONSTRAINT_NAME,   CONSTRAINT_TYPE, SEARCH_CONDITION
     FROM USER_CONSTRAINTS
     WHERE TABLE_NAME = 'EMPL47473';
```

OUTPUT:

```
CONSTRAINT_NAME              CONSTRAINTTYPE      SEARCH_CONDITION
---------------              ---------------     --------------------------------
SYS_C003018                  C                   "ENAME" IS NOT NULL
CHK_EMPL47473_GENDER         C                    UPPER (GENDER) IN ('M','F')
SYS_C003020                  C                    SALARY BETWEEN 10000 AND 70000
PK_EMPL47473_EMPNO           P
SYS_C003022                  U
```

ANALYSIS:

Describe displays structure of the data dictionary table.
Select statement is used to view the constraints defined on the table

INPUT:

```
SQL> DESCRIBE  USER_CONS_COLUMNS
SQL> SELECT CONSTRAINT_NAME,COLUMN_NAME
     FROM USER_CONS_COLUMNS
     WHERE TABLE_NAME = 'EMPL47473';
```

OUTPUT:

```
CONSTRAINT_NAME                 COLUMN_NAME
-----------------------------   --------------------------------
CHK_EMPL47473_GENDER            GENDER
PK_EMPL47473_EMPNO              EMPNO
SYS_C003018                     ENAME
SYS_C003020                     SALARY
SYS_C003022                     EMAIL_ID
```

ANALYSIS:

Describe displays structure of the data dictionary table.
Select statement is used to view the constraints defined on the column

ALTER TABLE

Used to modify the structure of a table

Syntax: ALTER TABLE <TABLENAME> [ADD | MODIFY | DROP | RENAME] (COLUMN(S));

ADD - for adding new columns into the table

MODIFY - for modifying the structure of columns

DROP - for removing a column in the table (8i)

RENAME - for renaming the column name (Only from 9i)

```
SQL> ALTER TABLE empl47473 ADD(address VARCHAR2(30), doj DATE,pincode VARCHAR2(7));

SQL> ALTER TABLE empl47473 MODIFY(ename CHAR (15), salary NUMBER (8,2));

SQL> ALTER TABLE empl47473 DROP COLUMN pincode;

SQL> ALTER TABLE empl47473 DROP (designation,address);

SQL> ALTER TABLE empl47473 RENAME COLUMN ename TO empname;
```

Note: This command is also useful for manipulating constraints

INPUT:

```
SQL> ALTER TABLE emplL47473 DROP PRIMARY KEY;
```

ANALYSIS:

To remove the primary key from table. Other constraints are removed only by referring constraint name.

INPUT:

```
SQL>ALTER TABLE empl47473 ADD PRIMARY KEY(empno);
```

ANALYSIS:

To add primary key to the table without constraint name. It creates constraint name with SYS_Cn.

INPUT:

```
SQL>ALTER TABLE empl47473 ADD CONSTRAINT pk_empl47473_empno PRIMARY KEY(empno);
```

ANALYSIS:

To add primary key in the table with constraint name

DATA MANIPULATION

Data manipulation means performing operations on the data in the tables of the dataase. We perform mainly 3 types of data manipulation operations. They are

- ⇨ Insertion
- ⇨ Updation
- ⇨ Deletion

INSERTING ROWS

Syntax: INSERT INTO tablename(colname1,colname2,…)
VALUES(VALUE1,VALUE2,…);

```
SQL> INSERT INTO empl47473 VALUES(101,'RAVI','M',
          'RAMESH_B@YAHOO.COM',5000,'10-JAN-2001');
                                    OR
SQL> INSERT INTO empl47473 VALUES(&empno,
&ename','&gender','&email_id',&salary,'&doj');
```

To insert data into specific columns of the table instead of all columns

```
SQL> INSERT INTO empl47473(empno,empname,salary) VALUES(101,'RAVI', 5000);
                                    OR
SQL> INSERT INTO empl47473(empno,empname,salary) VALUES(&empno,'&empname',&salary);

ANALYSIS:
We can't skip primary key and NOT NULL columns
```

Note: Changes made on the database are recorded only in the shadow page. For saving the information we have to use a command COMMIT, ROLLBACK.SAVEPOINT (Called as Transactional processing statements)

```
SQL>COMMIT;

ANALYSIS:
Information from shadow page flushed back to the table and shadow page gets
destroyed automatically.
```

```
SQL> ROLLBACK;

ANALYSIS:
Shadow page destroys automatically without transferring the information back to the
table.
```

SAVEPOINT

We can use save points to roll back portions of your current set of transactions

For example

```
SQL> INSERT INTO empl47473 VALUES(105,'KIRAN','M','KIRAN_B@YAHOO.COM',5000,'10-JAN-
2001');
SQL> SAVEPOINT A
SQL> INSERT INTO empl47473 VALUES(106,'LATHA','F', 'LATHA_D@YAHOO.COM',5000,'15-JAN-
2002');
SQL> SAVEPOINT B
SQL> INSERT INTO empl47473 VALUES(107,'RADHA','F','RADHA_V@GMAIL.COM',15000,'15-JAN-
2002');
```

When we SELECT data from the table

```
SQL> SELECT * FROM empl47473;

     EMPNO EMPNAME    G EMAIL_ID                SALARY DOJ
---------- ---------- - ------------------ ---------- ---------
       105 KIRAN      M KIRAN_B@YAHOO.COM         5000 10-JAN-01
       106 LATHA      F LATHA_D@YAHOO.COM         5000 15-JAN-02
       107 RADHA      F RADHA_V@GMAIL.COM        15000 15-JAN-02
```

The output shows the three new records we've added . Now roll back just the last insert:

```
SQL> ROLLBACK TO B;
```

IMPLICIT COMMIT

The actions that will force a commit to occur, even without your instructing it to, or quit, exit (the equivalent to exit), any DDL command forces a commit.

AUTO ROLLBACK

If you've completed a series of inserts, updates or deletes, but not yet explicitly or implicitly committed them, and you experience serious difficulties, such as a computer failure, Oracle automatically roll back any uncommitted work. If the machine or database goes down, it does this as cleanup work the next time the database is brought back up.

Note :

- Rollback works only on uncommitted data
- A DDL transaction after a DML transaction, automatically commits.
- We can use an Environment command SET VERIFY OFF to remove the old and new messages while inserting data.

ISSUE Frequent commit statements

Whenever possible, issue frequent COMMIT statements in all your programs. By issuing frequent COMMIT statements, the **performance** of the program is **enhanced** & its resource requirements are minimized as **COMMIT frees** up the following **resources:**

- Information held in the rollback segments to undo the transaction if necessary.
- All locks acquired during statement processing
- Space in the redo log buffer cache
- Overhead associated with any internal Oracle mechanisms to manage the resources in the previous three items.

CREATING A TABLE FROM ANOTHER TABLE

Syntax: CREATE TABLE <tablename> AS SELECT <columns> FROM <tablename> [WHERE <condition>];

```
SQL> CREATE TABLE empl47473 AS SELECT empno,ename,sal,job FROM  emp;
```

To add a new column in the table

```
SQL> ALTER TABLE empl7473 ADD(sex CHAR(1));

SQL> SELECT * FROM empl47473;
```

UPDATING ROWS

This command is used to change the data of the table

Syntax: UPDATE <tablename> SET column1 = expression, column2 = expression WHERE <condition>;

```
SQL> UPDATE empl47473 SET sal = sal * 1.1;

SQL>  COMMIT / ROLLBACK;

ANALYSIS
To give uniform increments to all the employees
```

```
SQL> UPDATE empl47473 SET
     sal = DECODE(job,'CLERK',sal*1.1, 'SALESMAN',sal*1.2,sal*1.15);

SQL> COMMIT / ROLLBACK;
```

```
SQL> UPDATE empl47473 SET sex = 'M' WHERE ename IN ('KING','MILLER','BLAKE');

SQL> COMMIT / ROLLBACK;

SQL> SELECT * FROM empl47473;
```

```
SQL> UPDATE empl47473 SET sex = 'F' WHERE sex IS NULL;

SQL> COMMIT / ROLLBACK;

SQL> SELECT * FROM empl47473 ;
```

```
SQL> UPDATE empl47473 SET ename = DECODE(sex,'M','Mr.'||ename,'Ms.'||ename);

SQL> COMMIT / ROLLBACK;

ANALYSIS:
ADD Mr. or Ms. Before the existing name as per the SEX value
```

DELETING ROWS

Syntax: DELETE [FROM] <table-name> WHERE <condition>;

- From is an optional keyword.
- Omitting the where clause will delete all rows from the table.
- You have to commit or rollback if the autocommit is not set to true.

```
SQL> DELETE FROM empl47473 WHERE sex = 'M';

SQL> COMMIT | ROLLBACK;
```

TRUNCATING A TABLE

Syntax: TRUNCATE TABLE <TABLENAME>

Note : Removes all the rows from table. Deleting specified rows is not possible. Once the table is truncated, it automatically commits. It is a DDL statement.

DROPPING A TABLE OR REMOVING A TABLE

Syntax: DROP TABLE <TABLENAME>

Note : Table is dropped permanently. It is a DDL statement. It removes the data along with table definitions and the table.

ADDING COMMENTS TO A TABLE.

You can add comments up to 2000 bytes about a column, table, view by using the **COMMENT** statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

Syntax: COMMENT ON TABLE table | COLUMN table.column IS 'text' ;

REFERENTIAL INTEGRITY CONSTRAINTS

This constraint is useful for maintaining relation with other table. Various referential integrity constraints we can use in Oracle are

- Foreign Key
- References
- On delete cascade
- On Delete Set NULL

Foreign Key

Defines the column in the child table at the table constraint level

References

Identifies the table and column in the parent table. Reference key accepts NULL and duplicate values.

On delete cascade

Deletes the dependent rows in the child table when a row in the parent table is deleted.

On Delete Set NULL

Converts dependent foreign key values to null.

For example we want following table structures and constraints imposed on those tables as shown below. Give the commands to create those tables with the specified constraints on those tables.

Department47473 (Deptno , dname)
Employee47473 (Empno, ename, salary, dno)
Deptno of Department47473 is a primary key
Empno of Employee47473 is a primary key
Dno of Employee47473 is a reference key

Solution

```
SQL> CREATE TABLE department47473 (
     deptno     NUMBER(3) PRIMARY KEY,
     dname      VARCHAR2(20) NOT NULL
     );

SQL> CREATE TABLE employee47473(
     empno      NUMBER(3) PRIMARY KEY,
     ename      VARCHAR2(10) NOT NULL,
     salary     NUMBER(7,2) CHECK(salary > 0),
     dno        NUMBER(3) REFERENCES department47473(deptno) ON DELETE CASCADE
     );
```

Assume the case where supermarket selling various items and customers order the items. Items may be returned by people who purchased it.

```
SQL> CREATE TABLE itemmaster (
     itemno      NUMBER (3) PRIMARY KEY,
     itemname    VARCHAR2 (10),
     stock       NUMBER (3) CHECK (stock > 0)
     );

SQL> CREATE TABLE itemtran (
     trnno       NUMBER (3),
     itemno      NUMBER (3) REFERENCES itemmaster (itemno),
     trndate     DATE,
     trntype     CHAR (1) CHECK (UPPER (trntype) IN ('R','I')),
     quantity    NUMBER (3) CHECK (quantity > 0),
     PRIMARY KEY (trnno, itemno)
     );

SQL> CREATE TABLE itemrefund(
     trnno       NUMBER (3),
     itemno      NUMBER (3),
     quantity    NUMBER (3),
     FOREIGN KEY (trnno,itemno) REFERENCES itemtran
     );
```

```
ALTER TABLE <TABLENAME> DISABLE PRIMARY KEY

ALTER TABLE <tablename> DISABLE PRIMARY KEY CASCADE;

Note : It is not possible to enable using cascade
```

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

ANALYSIS:
Removing the primary key along with Reference key
```

```
DROP TABLE <TABLENAME> CASCADE CONSTRAINTS

ANALYSIS:
Dropping the table along with constraints
```

Exercise

- Consider a training institute conducting different courses, into which the students are joining for various courses (Also, assume the case where same student can join in more than one course)
- The students may pay the fee in installments

Identify the tables, attributes and define them with relations

JOINS

Objectives

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

- Write SELECT statements to access data from more than one table using equality and non-equality join.
- View Data that generally does not meet a join condition by using outer joins
- Join a table itself by using self join

Join will enable you to gather and manipulate data across several tables. By

One of the most powerful features of SQL is its capability to gather and manipulate data from across several tables. Without this feature you would have to store all the data elements necessary for each application in one table. Without common tables you would need to store the same data in several tables.

TYPES OF JOINS

Oracle supports different types of joins as defined in the following table.

Oracle Proprietary Joins(8i and prior)	**SQL: 1999 Complaint Joins**
Equi join	Cross join
Non-Equi join	Natural Join
Outer join	Using clause
Self Join	Full or two side outer joins
	Arbitrary join conditions for outer joins

The Oracle 9i database offers join syntax that is SQL: 1999 compliant. Prior to 9i release, the join syntax was different from the ANSI standards. The new SQL: 1999 compliant join syntax does not offer any performance benefits over the Oracle proprietary join syntax that existed in prior releases.

Joining tables using Oracle Syntax

Use a join to query data from more than one table

Syntax: SELECT table1.column1, table2.column2 FROM table1,table2
WHERE table1.column1 = table2.column2;

- Write the join condition in the WHERE clause.
- Prefix the column name with table name when the same column name appears in more than one table.

Guidelines

- When writing a SELECT statement that joins tables, precede the common column name with the table name for clarify and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join n tables together, you need a minimum of n-1 join conditions. This rule many not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

Consider the table structures and data given below to understand various types of joins.

Departments Table has the various department name of the company as shown below.

DEPARTMENT_ID	DEPARTMENT_NAME
10	ADMINISTRATION
20	MARKETING
50	SHIPPING
60	IT
80	SALES

Employees table has the details of the employees working in the above mentioned departments.

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

Equi join

Extracting the information from more than one table by comparing (=) the common information.

Note : Equi Joins are also called as simple joins or Inner Joins

To display common column information

INPUT:

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

OUTPUT:

```
     EMPNO ENAME      JOB               SAL DNAME
---------- ---------- --------- ---------- --------------
      7782 CLARK      MANAGER         2450 ACCOUNTING
      7839 KING       PRESIDENT       5000 ACCOUNTING
      7934 MILLER     CLERK           1300 ACCOUNTING
      7369 SMITH      CLERK            800 RESEARCH
      7876 ADAMS      CLERK           1100 RESEARCH
      7902 FORD       ANALYST         3000 RESEARCH
      7788 SCOTT      ANALYST         3000 RESEARCH
      7566 JONES      MANAGER         2975 RESEARCH
      7499 ALLEN      SALESMAN        1600 SALES
      7698 BLAKE      MANAGER         2850 SALES
      7654 MARTIN     SALESMAN        1250 SALES
      7900 JAMES      CLERK            950 SALES
      7844 TURNER     SALESMAN        1500 SALES
      7521 WARD       SALESMAN        1250 SALES
```

ANALYSIS:

Efficiency is more when we compare the information from lower data table(master table) to Higher data table(child table).

When Oracle processes multiple tables, it uses an internal sort/merge procedure to join those tables. First, it scans & sorts the first table (the one specified last in FROM clause). Next, it scans the second table (the one prior to the last in the FROM clause) and merges all of the retrieved from the second table with those retrieved from the first table. It takes around 0.96 seconds

INPUT:

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

ANALYSIS:

Here driving table is EMP. It takes around 26.09 seconds
So, Efficiency is less.

Non-Equi joins

Getting the information from more than one table without using comparison (=) operator.

INPUT:

```
SQL> SELECT empno,ename,sal,grade,losal,hisal FROM salgrade g,emp e
           WHERE e.sal BETWEEN g.losal and g.hisal;
```

ANALYSIS:

Displays all the employees whose salary lies between any pair of low and high salary ranges.

INPUT:

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

OUTPUT:

```
DEPTNO     DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON
```

ANALYSIS:

Displays the details of the department where there are no employees

We can also get above output using relational algebra operators.

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

SQL> SELECT deptno FROM dept UNION SELECT deptno FROM emp;

SQL> SELECT deptno FROM dept UNION ALL SELECT deptno FROM emp;
```

OUTER JOIN

It is a join, which forcibly joins multiple tables even without having the common information. It is represented by +.

INPUT:

```
SQL> SELECT empno,ename,job,sal,dname FROM dept,emp WHERE dept.deptno =
emp.deptno(+);
```

OUTPUT:

```
     EMPNO ENAME      JOB              SAL DNAME
---------- ---------- --------- ---------- --------------
      7782 CLARK      MANAGER          2450 ACCOUNTING
      7839 KING       PRESIDENT        5000 ACCOUNTING
      7934 MILLER     CLERK            1300 ACCOUNTING
      7369 SMITH      CLERK             800 RESEARCH
      7876 ADAMS      CLERK            1100 RESEARCH
      7902 FORD       ANALYST          3000 RESEARCH
      7788 SCOTT      ANALYST          3000 RESEARCH
      7566 JONES      MANAGER          2975 RESEARCH
      7499 ALLEN      SALESMAN         1600 SALES
      7698 BLAKE      MANAGER          2850 SALES
      7654 MARTIN     SALESMAN         1250 SALES
      7900 JAMES      CLERK             950 SALES
      7844 TURNER     SALESMAN         1500 SALES
      7521 WARD       SALESMAN         1250 SALES
                                            OPERATIONS
```

LEFT, RIGHT AND FULL OUTER JOIN

As of Oracle 9i, you can use the ANSI SQL standard syntax for outer joins. In the FROM clause, you can tell Oracle to perform a LEFT, RIGHT or FULL OUTER join.

INPUT:

```
SQL> SELECT empno, ename, job, dept.deptno, dname FROM emp
    LEFT OUTER JOIN dept ON dept.deptno = emp.deptno;
```

OUTPUT:

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

ANALYSIS:

Gets the common information, and forcibly joins from left side table to right side table.

INPUT:

```
SQL> SELECT empno, ename, job, dept.deptno, dname FROM dept
     LEFT OUTER JOIN emp ON dept.deptno = emp.deptno;
```

OUTPUT:

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

ANALYSIS:

Gets the common information from both tables, then forcibly joins from dept table to emp table.

INPUT:

```
SQL> SELECT empno, ename, job, dept.deptno, dname FROM dept
     RIGHT OUTER JOIN emp ON dept.deptno = emp.deptno;
```

OUTPUT:

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

ANALYSIS:

Gets the common information from both tables, and then forcibly joins from dept table to emp table.

INPUT:

```
SQL> SELECT empno, ename, job, dept.deptno, dname FROM emp
     RIGHT OUTER JOIN dept ON dept.deptno.deptno = emp.deptno;
```

OUTPUT:

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

ANALYSIS:

Gets the common information from both tables, then forcibly joins from dept table to emp table.

Position of Joins in where clause

Table joins should be written first before any condition of WHERE clause. And the conditions which filter out the maximum records should be placed at the end after the joins as the parsing is done from BOTTOM to TOP

Least Efficient (Total CPU = 153.6 Sec)

```
SQL> SELECT ename, job, mgr FROM emp e WHERE sal > 50000 AND  job = 'MANAGER'
     AND 25 < (SELECT COUNT(*) FROM emp WHERE mgr = e.empno);
```

Most Efficient (Total CPU time = 10.6 sec)

```
SQL> SELECT ename, job, mgr FROM emp e WHERE 25 < (SELECT COUNT(*) FROM emp
     WHERE mgr = e.empno) AND sal > 5000 AND job = 'MANAGER';
```

SELF JOIN

Joining the table from itself is called as self join.

INPUT:

```
SQL> SELECT worker.ename || ' IS WORKING UNDER ' || manager.ename
     FROM emp worker, emp manager WHERE worker.mgr = manager.empno;
```

OUTPUT:

```
WORKER.ENAME||'ISWORKINGUNDER'||MANAGE
--------------------------------------
SCOTT IS WORKING UNDER JONES
FORD IS WORKING UNDER JONES
ALLEN IS WORKING UNDER BLAKE
WARD IS WORKING UNDER BLAKE
JAMES IS WORKING UNDER BLAKE
TURNER IS WORKING UNDER BLAKE
MARTIN IS WORKING UNDER BLAKE
MILLER IS WORKING UNDER CLARK
ADAMS IS WORKING UNDER SCOTT
JONES IS WORKING UNDER KING
CLARK IS WORKING UNDER KING
BLAKE IS WORKING UNDER KING
SMITH IS WORKING UNDER FORD
```

ANALYSIS:

It displays who is working under whom MGR number appearing against employee is the employee number of manager

NATURAL AND INNER JOINS (Introduced in 9i)

We can use natural keyword to indicate that a join should be performed based on all columns that have the same name in the two tables being joined.

Creating Natural Joins

- The Natural join clause is based on all columns in the two tables that have the same name,
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

To get the common information from two tables,

INPUT:

```
SQL> SELECT empno, ename, job, deptno, dname FROM dept NATURAL JOIN emp;
```

OUTPUT:

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

Joins with Using Clause

- If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with USING clause to specify the columns that should be used for an equijoin.
- Use the USING clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns
- The NATURAL JOIN and USING clauses are mutually exclusive.

INNER JOIN

Support for INNER join syntax was introduced in Oracle9i, inner joins are the default – they return the rows the two tables have in common, and are the alternative to outer joins. Note that they support ON clause, so that you can specify join criteria.

```
SQL> SELECT empno, ename, job, dept.deptno, dname FROM emp
     INNER JOIN dept ON emp.deptno = dept.deptno;
```

CROSS JOIN (CARTESIAN PRODUCT)

Joining tables without giving proper join condition.

INPUT:

```
SQL> SELECT * FROM dept, emp;
                          OR
SQL> SELECT * FROM emp CROSS JOIN dept;
```

ANALYSIS:

```
It multiplies the rows from both tables and displays the output i.e. 14 rows (emp
table) X 4 rows(dept table) = 56 rows.
```

OTHER OBJECTS

SEQUENCE OBJECT

Used to generate sequence(Unique) Integers for use of primary keys.

Syntax:
```
CREATE SEQUENCE sequence
[INCREMENT BY n]
[START WITH n]
[{MAXVALUE n | NOMAXVALUE}]
[{MINVALUE n | NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];
```

Sequence	is the name of the sequence generator
INCREMENT BY *n*	specifies the interval between sequence numbers where *n* is an integer (If this clause is omitted, the sequence increments by 1.)
START WITH *n*	specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
MAXVALUE *n*	specifies the maximum value the sequence can generate N
NOMAXVALUE	specifies a maximum value of 10^27 for an ascending sequence and –1 for a descending sequence (This is the default option.)
MINVALUE *n*	*s*pecifies the minimum sequence value
NOMINVALUE	specifies a minimum value of 1 for an ascending sequence and – (10^26) for a descending sequence (This is the default option.)
CYCLE | NOCYCLE	specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default Option.)

CACHE *n* | NOCACHE specifies how many values the Oracle server preallocates and keep in memory (By default, the Oracle server caches 20 values.) The value set must be less than MAXVALUE minus

Example

```
CREATE SEQUENCE sqno47473
    START WITH 1
    INCREMENT BY 1
    MAXVALUE 10;
```

```
CREATE SEQUENCE sqno47473
    START WITH 1
    INCREMENT BY 1
    MAVALUE 10
    CACHE 3
    CYCLE;
```

Note: - These sequences are stored in a data dictionary table SER_SEQUENCES.

This sequence object provides two public member functions

NEXTVAL and CURRVAL

NEXTVAL is a function which generate next value from sequence object. CURRVAL is a function, which gives the current value of the sequence object. Assume there is a table SAMPLE47473(EMPNO,ENAME,SAL) columns. To insert the values into the table we are using the sequence

```
SQL> INSERT INTO sample47473 VALUES(sqno47473.NEXTVAL,
           '&ENAME', &SAL);
```

TO MODIFY THE SEQUNECE OBJECT

```
SQL> ALTER SEQUENCE sqno47473 INCREMENT BY 2 MAXVALUE 40;
Note : We can't change starting value
```

To remove the sequence object

```
SQL>  DROP SEQUENCE <SEQUENCE_NAME>;
```

VIEWS

- A view is an object, which is a logical representation of a table
- A view contains no data on its own
- It is derived from tables
- Changes made in tables are automatically reflected in views
- As a view does not store any data the redundancy problem does not arise.
- Critical data in the base table is safeguarded as access to such data can be controlled.
- It is used to reduce the complexity of the query

In Oracle Oracle we can create different types of views

- SIMPLE
- COMPLEX
- INLINE

SIMPLE view is a view, which is created using only one base table.

COMPLEX view is a view, which is created using more than one table or using group functions

INLINE view is a view, which is created using sub query (it is not a schema object). It is a named sub query in the FROM clause of the main query. Generally used in TOP N Analysis.

Syntax: CREATE OR REPLACE [FORCE] VIEW <viewname>
AS SELECT <columns> FROM <table > [WITH READ ONLY];

The table on which a view is based is called as **base table**

FORCE option allows view to be created even if the base table doesn't exist. However, the base table should exist before the view is used.

Changing Base Table through view

A view can also be used to change the data of base table

A view can be used to delete, insert and update rows in the base table.

However for each operation certain conditions are to be satisfied.

Rules for deleting row

The following should NOT be used in the query.

- More than one table
- GROUP BY clause
- GROUP Function
- DISTINCT clause
- Pseudo column ROWNUM

Rules for updating rows

The following are the rules that are to be satisfied to update base table through view.

- All the rules of Delete are applicable
- The column being updated should not be derived from an expression.

Rules for insertion of rows

- All the rules of UPDATE
- Al NOT NULL columns should be included in the view.

```
SQL> CREATE OR REPLACE VIEW testview47473 AS SELECT empno, ename, sal FROM
empl47473;
```

Note: - These views are stored in a data dictionary table USER_VIEWS

```
SQL> CREATE OR REPLACE VIEW testview47473 AS
     SELECT empno, ename, sal FROM empl47473 WITH READ ONLY;

ANALYSIS:
View becomes a read only view
```

WITH CHECK OPTION

This option is used to prevent any changes to base table through view. Insertion and updating is not allowed into base table through view.

```
SQL> CREATE OR REPLACE VIEW chkview AS SELECT * FROM emp WHERE deptno = 20 WITH
CHECK OPTION;
```

It doesn't allow you to update the condition column as well as it doesn't allow you to insert the details of employees with DEPTNO other than 20.

We can also create a view using group functions. Such views are called as INLINE views. They are by default read only.

```
SQL> CREATE OR REPLACE VIEW simpleview47473 AS SELECT
     SUBSTR(hiredate,-2) YEAR, COUNT(*) NUMB FROM empl47473
     GROUP BY job;
```

To remove a view

```
SQL> DROP VIEW <VIEWNAME>;
```

INDEX

The concept indexing in Oracle is same as a book index. Just like how book index is sorted in the ascending order of topics, an index in Oracle is a list of values of a column in the ascending order. Page number in book index is similar to ROWID if Oracle index.

An oracle index is a database object. It contains the values of the indexed column(s) in the ascending order along with address of each row. The address of rows are obtained using ROWID pseudo column.

Why to Use An INDEX

INDEXES in ORACLE are used for two purposes

- To speed up data retrieval and thereby improving performance of query
- To enforce uniqueness

Note: A UNIQUE index is automatically created when you use PRIMARY KEY and UNIQUE constraints

An index can have up to 32 columns.

Syntax: CREATE [UNIQUE] INDEX index_name ON table(column1,column2,…);

Note :- Indexes are stored in the data dictionary table USER_INDEXES.

When Oracle Does Not Use Index

Oracle index is completely automatic. I.e., you never have to open or close an index. Oracle server decides whether to use an index or not.

The following are the cases in which Oracle does NOT use index.

- SELECT doesn't contain WHERE clause
- When the data size is less
- SELECT contains WHERE clause, but WHERE clause doesn't refer to indexed column.
- SELECT contains WHERE clause and WHERE clause uses indexed columns but indexed column is modified in the WHERE clause.

Negative Side of an Index

INDEX plays an important role in improving performance. But at the same time it may also degrade performance, if not designed carefully.

Having many indexes may have negative impact on the performance because whenever there is a change in the table, immediately index is to reflect that change. A new row's insertion will effect index and Oracle server implicitly updates index. So this will put more burden on the machine if more number of indexes are created on a table.

FUNCTIONAL INDEX (Function based indexes)

As of Oracle8i, we can create functional-based indexes. When we are storing alpha-numeric information, we may store the information in any case. When we create index on such columns, information is placed in different ranges of indexes. So, before creating index, we can convert them in to single case.

```
SQL> CREATE INDEX idx_name ON emp(UPPER(ename));
```

Dropping an Index

Syntax: DROP INDEX <indexname>;

Removing an index doesn't invalidate existing applications, because applications are not directly dependent on index, but at the same time not having an index may effects performance.

PSEUDO COLUMN

A pseudo-column is a column that yields a value when selected but which is not an actual column of the table.

- ROWID
- ROWNUM
- SYADATE
- NEXTVAL
- CURRVAL
- NULL
- LEVEL

Are called as Pseudo-columns.

```
SELECT ROWNUM, empno, ename FROM emp;
```

TO DISPLAY 3 HIGHEST PAID EMPLOYEES

```
SQL> SELECT ROWNUM,empno,ename,sal FROM (SELECT empno,
     ename,sal FROM emp ORDER BY sal DESC)
     WHERE ROWNUM <= 3;
```

Exercise

1. Display the string SATYAM in the format

   S
   A
   T
   Y
   A
   M
2. Display only even rows from the table
3. Display one year calendar
4. Display how many a's are there in the given string
5. Remove duplicate rows from the given table

Empno	ename
1	x
2	y
3	z
1	x
3	z

1. Find out how many columns are there in a given table(Use the data dictionary table USER_TAB_COLUMNS)

ADVANCED QUERIES

ANALYTICAL QUERIES

Analytical functions are used mainly for the analysis of data as required by decision-making managers.

Oracle has embedded analytical functions in SQL statement to cater to most of the requirements of data mining.

These functions are listed below

Ranking: For calculating ranks, percentiles, and n-tiles of the values in a result set.

For example, to find out top three salaried employees

INPUT:

```
SQL> SELECT RANK() OVER(ORDER BY SAL DESC) DEFAULT_RANK, SAL FROM EMP;
```

OUTPUT:

```
DEFAULT_RANK        SAL
------------ ----------
           1   855945.6
           1   855945.6
           1   855945.6
           4   641959.2
           5  196867.32
           6  184563.24
           7  153802.68
           7  153802.68
           9  129461.88
          10   111807.6
          11   66870.77
          12      10000
          13    6810.91
```

ANALYSIS:

When different employees salary is same, they get the same rank. There is a gap between ranks. In the example, see the ranks between 1 and 4.

DENSE_RANK

The main difference between RANK() and DENSE_RANK() is that in the DENSE_RANK () there is no gap between ranks.

INPUT:

```
SQL> SELECT DENSE_RANK() OVER(ORDER BY SAL DESC) DEFAULT_RANK, SAL FROM EMP;
```

OUTPUT:

```
DEFAULT_RANK        SAL
------------ ----------
           1   855945.6
           1   855945.6
           1   855945.6
           2   641959.2
           3  196867.32
           4  184563.24
           5  153802.68
           5  153802.68
           6  129461.88
           7   111807.6
           8   66870.77
           9      10000
          10    6810.91
```

ROW_NUMBER

The function row number assigns unique rank to each row even if they are having the same value of order by expression. The rows will get the same rank if their order by expression has the same value.

INPUT:

```
SQL> SELECT RANK() OVER(ORDER BY SAL) DEFAULT_RANK, ROW_NUMBER() OVER(ORDER BY SAL)
RW_NUM,SAL FROM EMP
```

OUTPUT:

```
DEFAULT_RANK     RW_NUM        SAL
------------ ---------- ----------
           1          1    6810.91
           2          2      10000
           3          3   66870.77
           4          4   111807.6
           5          5  129461.88
           6          6  153802.68
           6          7  153802.68
           8          8  184563.24
           9          9  196867.32
          10         10   641959.2
          11         11   855945.6
          11         12   855945.6
          11         13   855945.6
```

NULLIF FUNCTION

This function produces NULL value if the expression has a specified value. It produces NULL value if the expression has a specified value. This is like a complement of the NVL function.

INPUT:

```
SQL> SELECT ENAME,JOB,COMM,NULLIF(JOB,'MANAGER') FROM EMP;
```

OUTPUT:

```
ENAME           JOB              COMM NULLIF(JO
--------------- --------- ---------- ---------
ss ss           CLERK            1234 CLERK
allen           SALESMAN        746.5 SALESMAN
WARD            SALESMAN      1244.16 SALESMAN
JONES           MANAGER
MARTIN          SALESMAN      3483.65 SALESMAN
BLAKE           MANAGER
CLARK           MANAGER
SCOTT           ANALYST                ANALYST
TURNER          SALESMAN            0 SALESMAN
ADAMS           CLERK                  CLERK
JAMES           CLERK                  CLERK
FORD            ANALYST                ANALYST
phani           se                20000 se
```

ANALYSIS:

It is just opposite to NVL(). NVL() substitute value, if its is NULL. Where as NULLIF produces NULL, if value matches.

NVL2 Function

It is an extended form of NVL.

Syntax: NVL2(expr1,expr2,expr3)

In NVL2, expr1 can never be returned; either expr2 or expr3 will be returned. If expr1 is not NULL, NVL2 returns expr2 otherwise returns expr3. The expr1 can have any data type. The arguments expr2 and expr3 can have any datatype except LONG.

INPUT:

```
SQL> SELECT COMM,NVL2(COMM,COMM,0) FROM EMP;
```

OUTPUT:

```
      COMM NVL2(COMM,COMM,0)
---------- ------------------
                            0
       300                300
       500                500
                            0
      1400               1400
                            0
                            0
                            0
                            0
                            0
```

Coalesce Function

This function takes n arguments and produces first argument, which is having the first NOT NULL value.

INPUT:

```
SQL> SELECT ENAME, COMM, SAL, DEPTNO, MGR, COALESCE (COMM, SAL, DEPTNO, MGR) COAL
FROM EMP;
```

OUTPUT:

```
ENAME                  COMM        SAL     DEPTNO        MGR       COAL
-------------- ---------- ---------- ---------- ---------- ----------
ss ss                  1234    6810.91         20       7902       1234
allen                 746.5  196867.32         30       7698      746.5
WARD                1244.16  153802.68         30       7698    1244.16
JONES                        855945.6         20       7839   855945.6
MARTIN              3483.65  153802.68         30       7698    3483.65
BLAKE                        855945.6         30       7839   855945.6
CLARK                        855945.6         10       7839   855945.6
SCOTT                        641959.2         20       7566   641959.2
TURNER                 0  184563.24         30       7698          0
ADAMS                        129461.88         20       7788  129461.88
JAMES                         111807.6         30       7698   111807.6
FORD                         66870.77         20       7566   66870.77
phani                20000       10000         90       9822      20000
```

Multiple Insert and Merge statements

Oracle 9i provides enhanced facility of loading data in the table using select statement.

Using this facility you

- Load data in multiple tables in single insert statement
- Load multiple rows in the same table in single insert statement
- Load data conditionally in the table.

Multiple insert statement

There are four types of multiple inert statements

- Unconditional insert statement
- Pivoting insert statement
- Conditional insert statement
- Insert first statement

Unconditional Insert Statement

This statement enables to you insert data in multiple tables using a single insert statement. To understand this query consider the three tables from Annexure

To insert the data into two tables

```
Example1(pname,title,salary)
Example2(pname,course,cost)
```

```
Insert all
Into example1 values(pname,title,salary)
Into example2 values(pname,course,cost)
Select p.pname,title,course,cost,salary from programmer p,software s, studies st
Where p.pname = s.pname and s.pname = st.pname;
```

Conditional Insert statement

It inserts the data into the table only when condition is satisfied.

Assume there are 3 tables table1, table2 and table3 with same structure ***(empno, ename, job, sal, deptno)***. We can load the data conditionally into three tables by using following statement

```
SQL> insert all
     When deptno = 10 then
         Insert into table1 values(empno,ename,job,sal,deptno);
     When deptno = 20 then
         Insert into table2 values(empno,ename,job,sal,deptno);
     When deptno = 30 then
         Insert into table3 values(empno,ename,job,sal,deptno);
     Select empno,ename,sal,deptno from emp
```

The difference between ***Insert First*** and ***Insert All*** is that in the former, at the most one row is inserted into the table while in later rows may be inserted into multiple tables.

LOCKING MECHANISMS

To ensure data integrity oracle uses locks. Locks are used to prevent destructive interaction between processes accessing the same resource.

Oracle uses two locking types

- DML locks - used to protect data. Can be either table or row level.
- Dictionary locks – used to protect the database structure

Locks may be explicit or implicit. A process or user instigates explicit locks. Implicit locks are undertaken by Oracle. Oracle will lock any resource when it detects a valid attempt to update the resource. Dictionary locks are always implicit. DML locks may be implicit or explicit.

There are five locking modes. They are

EXCLUSIVE(X)	LOCK ALLOWS QUERIES BUT NOTHING ELSE
SHARE(S)	Lock allows queries but not updates
ROW SHARE(RS)	Lock allows concurrent process access to table. Resource may not be locked exclusively.
ROW EXCLUSIVE (RX)	Same as row share but no share mode locking. Updates. Deletes and inserts use this lock mode.

Locks are held until either a transaction is committed / rolled back.

Locking using SELECT for UPDATE

```
SELECT * FROM EMP WHERE EMPNO = 7521 FOR UPDATE OF SAL NOWAIT;
```

With this SELECT we lock all the rows in the result set for later update. The FOR UPDATE tells Oracle to lock each row as it processes it.

The **OF** keyword prefixes the column identification area that specifies which columns are going to be updated by us at a later date. The NOWAIT keyword specifies that we don't want the statement to wait until, and current locks on the table are removed.

Assume that there are two users SCOTT AND X

In the Scott user there is one table with the name EMP. Scott has given some privileges on EMP table to X user. If X wants to lock the EMP table, then he has to issue a command

```
LOCK TABLE SCOTT.EMP IN EXCLUSIVE MODE NOWAIT;
```

Now from SCOTT user, if he tries to the lock the table, which is already locked by X

```
LOCK TABLE EMP IN EXCLUSIVE MODE;
```

It results an error

```
LOCK TABLE EMP IN EXCLUSIVE MODE NOWAIT
           *
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

SECURITY

What is a privilege

A privilege is a right to access an object such as a table, view etc., or to execute a particular type of SQL command such as CREATE TABLE.

Privileges are classified into two categories depending upon what type of right they give to the user.

- Object Privileges
- System privileges

Object privilege

An object privilege is a right to perform a particular operation on an object. An object is a table, view, sequence, procedure, function or package.

System privilege

A system privilege is a right to perform certain operation in the system. For example, the privilege to create a table is a system privilege.

Object Privileges

User owns the object that he/she creates. Unless otherwise specified, only the owner and DBA can access the object. But, if user wants to share his object with other users, he has to grant privileges on the object to other users.

The following are the list of object privileges available in Oracle

ALTER, DELETE, EXECUTE, INDEX, INSERT, SELECT, UPDATE

These privileges are given on various objects, such as

TABLE, VIEW, SEQUENCE, PROCEDURE,FUNCTION, PACKAGE AND OBJECT TYPE

Syntax: GRANT <PRIVILEGES> | ALL [(COLUMN1,COLUMN2)] ON <OBJECT> TO (USER | PUBLIC | ROLE) [WITH GRANT OPTION];

Privileges	any object privileges
ALL	to grant all privileges
PUBLIC	to grant privilege to all the users of the system.

Example

```
GRANT SELECT,UPDATE ON EMP TO X;
```

Note :- X is a user

Now from X user, he can select as well as update the information in EMP table.

```
SELECT * FROM SCOTT.EMP;
UPDATE SCOTT.EMP SET AL = 11000 WHERE EMPNO = 7521;
```

Restricting privilege to certain columns

```
GRANT UPDATE(SAL) ON EMP TO X;
```

So, X user can modify only sal value in the EMP table.

REVOKE OBJECT PRIVILEGES

To remove the given privileges, we can use REVOKE command

```
REVOKE <PRIVILEGES> ON <OBJECT> FROM <USER>;
```

SYNONYMS

To simplify accessing tables owned by other users, create a SYNONYM. A synonym is another name (alias) to a table or view. By creating a synonym you can avoid giving the owner name while accessing table of other users.

We can create two types of synonyms

- Private Synonyms
- Public Synonyms

Any user can create private synonym, where as public synonym is created by only DBA using a keyword called public.

Creating private synonym.

```
CREATE SYNONYM EMPL FOR SCOTT.EMP;
```

Now we can access EMP table which is there in SCOTT user using synonym EMPL as

```
SQL> SELECT * FROM EMPL;
```

Public Synonym

Public synonym is available for all the users.

```
SQL>CREATE PUBLIC SYNONYM EMPL FOR SCOTT.EMP;
```

Bulk loading

What is SQL*Loader?

SQL*Loader is Oracle's utility program for loading data into an Oracle table. Most often, SQL*Loader takes two input files – a control file and a data file – and loads the data into a single Oracle table. The data file contains data, and the control file contains information about the data -- where to load it, what to do if something goes wrong, etc.

SQL*Loader has lots and lots of options which can be used to handle various types of data and levels of complexity. SQL*Loader is fully described in the Oracle Server Utilities User's Guide. This document is just about getting started. SQL*Loader runs on Unix, mainframes, and PC's. This document is just about running it from a Windows PC.

Why Use SQL*Loader From Your PC?

If you need to transfer quite a lot of data from your machine to an Oracle database table, you might want to use SQL*Loader. If you already have the data in some other format, it may be worthwhile to use SQL*Loader. If you need to transfer local data to a remote database on some recurring basis, it may be preferable to use SQL*Loader rather than something like FTP. At the end of this document, there is a brief comparison of FTP versus SQL*Loader.

Getting Started, an Example

Say, for example, that you've got an Excel spreadsheet with State data already in it. You've got 50 rows of data – each containing the State Abbreviation, State Name, an [optional] unofficial State Slogan, and the number of State Residents Who Drink Bottled Water. (Is 50 rows of data really sufficient to justify this exercise? That's debatable, but let's say you've thought it over and you DO want to SQL*load your data into a 4-column Oracle table at UW-Stevens Point. The remote table is called sp.mystates.)

Here's what you do:

1. **Create your data file.** This is easy. Save your Excel spreadsheet data AS a Comma-Separated-Variable (*.csv) file. This will automatically put commas between each of the four data elements. In addition, if any of the data elements already contain a comma, the Save AS *.csv step will optionally and automatically enclose that data in double quotes. So, after your Save AS command, you might have a file named `C:\MyStates.csv` that contains data like this:

```
AR,Arkansas,We are sure proud of Bill,0
CO,Colorado,,3000
WI,Wisconsin,Rose Bowl Champions Again!,5
CA,California,"Dude? You want, like, another hit of Oxygen?",90203049
```

2. **Create your control file.** Using any text editor, create a file (say, C:\mystates.ctl) containing these lines:

```
LOAD DATA
INFILE 'C:\EMPLOYEE.csv'
REPLACE
INTO TABLE EMPL
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(EMPNO,
 ENAME
INTEGER EXTERNAL)
```

The `REPLACE` keyword says, "remove any existing rows before starting the load." There's also an `INSERT [into empty table]` and `APPEND [to existing rows]` option.

`State_Abbrev, State_Name, State_Slogan, and Nbr_Residents_WDBW` are the actual column names defined in the sp.mystates table.

Because the first three items are of character datatype, it was not necessary to further describe them – character is the default. The fourth column is numeric data – it totals the number of state residents who drink bottled water. The INTEGER EXTERNAL describes the datatype in the C:\mystates.csv input file.

Notice there is some missing data in the data file -- Colorado has no state slogan. The TRAILING NULLCOLS statement handles the missing data; it tells SQL*Loader to load any missing data as NULL values. There are, as we said earlier, *lots* of available options described in the Utilities User's Guide.

3. Create a table with the name referred in ctl file with required columns
4. **Run SQL*Loader.**

Prerequisites:

- You must have SQL*Loader and SQL*Net installed on your machine. The SQL*Loader program may have a version number included as part of its name, something like `sqlldr73.exe` or `sqlldr80.exe`. Or maybe it will be just `sqlldr.exe`. You can look for it in your ORAWIN95 or ORANT \BIN directory. If it's not installed, you can get the Oracle Client Software installation CD and install "`UTILITIES`".
- You must have the target database (say, it's called 'ELTP') configured as SQL*Net service in your local tnsnames.ora file. This is pretty standard stuff; it's probably already there.
- You must have authorization to modify the sp.mystates table (INSERT, or DELETE and INSERT if you're using the REPLACE option in the control file. In the example below we assume that user `SCOTT` with password `TIGER` has appropriate authorization.

At an MS-DOS prompt (or the Start, Run menu) , execute SQL*Loader as follows:

```
sqlldr scott/tiger@ELTP control=C:\mystates.ctl
```

When the load completes, look in the file C:\mystates.log. This log file will contain information about how many rows were loaded, how many rows -- if any -- were NOT loaded, and other information that may be useful to reassure or debug.

How to convert Excel Sheet into CSV(Comma separated variable) file

This article describes how to convert a single column of addresses in a Microsoft Excel worksheet into a comma-separated value (CSV) file that you can import into another program (for example, Microsoft Word).

Note For the address example in this article, the Excel worksheet contains the following address information:

A	B
1	ravi
2	kris
3	babu

1. On the **File** menu, click **Save As**.
   Note In Excel 2007, click the **Microsoft Office Button**, and then click **Save As**.
2. In the **Save As** dialog box:
   a. In the **Save as type** box, click **CSV (Comma delimited) (*.csv)**.
   b. In the **File name** box, type a name for your CSV file (for example, **Address.csv**), and then click **Save**.
   c. Click **OK** when you receive the following message:
   The selected file type does not support workbooks that contain multiple sheets.
   • To save only the active sheet, click OK.
   • To save all sheets, save them individually using a different file name for each, or select a file type that supports multiple sheets.
   d. Click **Yes** when you receive the following message:
   Address.csv may contain features that are incompatible with CSV (comma delimited). Do you want to keep the workbook in this format?
   • To keep this format, which leaves out any incompatible features, click Yes.
   • To preserve the features, click No. Then save a copy in the latest Excel format.
   • To see what might be lost, click Help.
3. On the **File** menu, click **Close**, and then exit Microsoft Excel.
   Note In Excel 2007, click the **Microsoft Office Button**, click **Close**, and then click **Exit Excel**.
   Note You may be prompted to save the file again. When you are prompted, you can click **Yes**, repeat steps c and d, and then exit Excel.

Edit the CSV File in Microsoft Word

1. Start Microsoft Word.
2. On the **File** menu, click **Open**.

Note In Word 2007, click the **Microsoft Office Button**, and then click **Open**.

3. In the **Files of type** box, click **All Files (*.*)**.
4. Click the CSV file that you saved in step 4 of the "Edit the Excel Worksheet" section, and then click **Open**.
5. On the **Tools** menu, click **Options**.
   Note In Word 2007, skip this step.
6. On the **View** tab, click to select the **All** check box, and then click **OK**.
   Note In Word 2007, follow these steps:
   a. Click the **Microsoft Office Button**, and then click **Word Options**.
   b. Click **Display**.
   c. Click **Paragraph marks** under the **Always show these formatting marks on the screen**.

Date and time formats in Oracle

The following table gives the list of number formats supported by Oracle

Element	Example	Description
, (comma)	`9,999`	Returns a comma in the specified position. You can specify multiple commas in a number format model. Restrictions: • A comma element cannot begin a number format model. • A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	`99.99`	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	`$9999`	Returns value with a leading dollar sign.

0	`0999`
	
	`9990`
		Returns leading zeros.
		
		Returns trailing zeros.

9	`9999`	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	`B9999`	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0"s in the format model).
C	`C999`	Returns in the specified position the ISO currency symbol (the current value of the `NLS_ISO_CURRENCY` parameter).
D	`99D99`	Returns in the specified position the decimal character, which is the current value of the `NLS_NUMERIC_CHARACTER` parameter. The default is a period (.). **Restriction:** You can specify only one decimal character in a number format model.
EEEE	`9.9EEEE`	Returns a value using in scientific notation.
FM	`FM90.9`	Returns a value with no leading or trailing blanks.
G	`9G999`	Returns in the specified position the group separator (the current value of the `NLS_NUMERIC_CHARACTER` parameter). You can specify multiple group separators in a number format model. **Restriction:** A group separator cannot appear to the right of a decimal character or period in a number format model.
L	`L999`	Returns in the specified position the local currency symbol (the current value of the `NLS_CURRENCY` parameter).
MI	`9999MI`	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing blank.

Element	Example	Description
		Restriction: The MI format element can appear only in the last position of a number format model.
PR	`9999PR`	Returns negative value in <angle brackets>. Returns positive value with a leading and trailing blank. **Restriction:** The PR format element can appear only in the last position of a number format model.

RN
 	
rn
	`RN`
	
	`Rn`
		Returns a value as Roman numerals in uppercase.
		
		Returns a value as Roman numerals in lowercase.
		
		Value can be an integer between 1 and 3999.

S	`S9999`
	
	`9999S`
		Returns negative value with a leading minus sign (-).
		
		Returns positive value with a leading plus sign (+).
		
		Returns negative value with a trailing minus sign (-).
		
		Returns positive value with a trailing plus sign (+).
		
		Restriction: The S format element can appear only in the first or last position of a number format model.

TM	`TM`	"Text minimum". Returns (in decimal output) the smallest number of characters possible. This element is case-insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If output exceeds 64 characters, then Oracle automatically returns the number in scientific notation. **Restrictions:** • You cannot precede this element with any other element. • You can follow this element only with 9 or E (only one) or e (only one).
U	`U9999`	Returns in the specified position the "Euro" (or other) dual currency symbol (the current value of the `NLS_DUAL_CURRENCY` parameter).
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where n is the number of 9's after the "V".

X	`XXXX`
	
	`Xxxx`
		Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then Oracle rounds it to an integer.
		
		Restrictions:
		
		• This element accepts only positive values or 0. Negative values return an error.
		• You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has 1 leading blank.

The following are the date formats provided by oracle.

Format mask	Description
CC	Century
SCC	Century BC prefixed with -
YYYY	Year with 4 numbers
SYYY	Year BC prefixed with -
IYYY	ISO Year with 4 numbers
YY	Year with 2 numbers
RR	Year with 2 numbers with Y2k compatibility
YEAR	Year in characters
SYEAR	Year in characters, BC prefixed with -
BC	BC/AD Indicator *
Q	Quarter in numbers (1,2,3,4)
MM	Month of year 01, 02...12
MONTH	Month in characters (i.e. January)
MON	JAN, FEB
WW	Weeknumber (i.e. 1)
W	Weeknumber of the month (i.e. 5)
IW	Weeknumber of the year in ISO standard.
DDD	Day of year in numbers (i.e. 365)
DD	Day of the month in numbers (i.e. 28)
D	Day of week in numbers(i.e. 7)

DAY	Day of the week in characters (i.e. Monday)
FMDAY	Day of the week in characters (i.e. Monday)
DY	Day of the week in short character description (i.e. SUN)
J	Julian Day (number of days since January 1 4713 BC, where January 1 4713 BC is 1 in Oracle)
HH	Hournumber of the day (1-12)
HH12	Hournumber of the day (1-12)
HH24	Hournumber of the day with 24Hours notation (0-23)
AM	AM or PM
PM	AM or PM
MI	Number of minutes (i.e. 59)
SS	Number of seconds (i.e. 59)
SSSSS	Number of seconds this day.
DS	Short date format. Depends on NLS-settings. Use only with timestamp.
DL	Long date format. Depends on NLS-settings. Use only with timestamp.
E	Abbreviated era name. Valid only for calendars: Japanese Imperial, ROC Official and Thai Buddha.. (Input-only)
EE	The full era name
FF	The fractional seconds. Use with timestamp.
FF1..FF9	The fractional seconds. Use with timestamp. The digit controls the number of decimal digits used for fractional seconds.
FM	Fill Mode: suppresses blianks in output from conversion
FX	Format Exact: requires exact pattern matching between data and format model.
IYY or IY or I	the last 3,2,1 digits of the ISO standard year. Output only
RM	The Roman numeral representation of the month (I .. XII)

RR	The last 2 digits of the year.
RRRR	The last 2 digits of the year when used for output. Accepts fout-digit years when used for input.
SCC	Century. BC dates are prefixed with a minus.
CC	Century
SP	Spelled format. Can appear of the end of a number element. The result is always in english. For example month 10 in format MMSP returns "ten"
SPTH	Spelled and ordinal format; 1 results in first.
TH	Converts a number to it's ordinal format. For example 1 becoms 1st.
TS	Short time format. Depends on NLS-settings. Use only with timestamp.
TZD	Abbreviated time zone name. ie PST.
TZH	Time zone hour displacement.
TZM	Time zone minute displacement.
TZR	Time zone region
X	Local radix character. In america this is a period (.)

Function list of Oracle

SQL Functions as per oracle documentation: Oracle 10g

Function Categories:

1. Single row functions
2. Aggregate functions
3. Analytic Functions
4. Object reference functions
5. Model functions
6. User defined functions

*single_row_function*s are sub classified into

1. Numeric functions
2. Character functions
3. Data mining functions
4. Date time functions
5. Conversion functions
6. Collection functions
7. XML functions
8. Miscellaneous functions

Single-Row Functions

Numeric Functions

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

Character Functions Returning Character Values

CHR	CONCAT	INITCAP	LOWER
LPAD	LTRIM	NLS_INITCAP	NLS_LOWER
NLSSORT	NLS_UPPER	REGEXP_REPLACE	REGEXP_SUBSTR
REPLACE	RPAD	RTRIM	SOUNDEX
SUBSTR	TRANSLATE	TREAT	TRIM
UPPER			

NLS Character Functions

NLS_CHARSET_DECL_LEN	NLS_CHARSET_ID	NLS_CHARSET_NAME

Character Functions Returning Number Values

ASCII	INSTR	LENGTH	REGEXP_INSTR

Datetime Functions

ADD_MONTHS	CURRENT_DATE	CURRENT_TIMESTAMP
DBTIMEZONE	EXTRACT (datetime)	FROM_TZ
LAST_DAY	LOCALTIMESTAMP	MONTHS_BETWEEN
NEW_TIME	NEXT_DAY	NUMTODSINTERVAL
NUMTOYMINTERVAL	ROUND (date)	SESSIONTIMEZONE
SYS_EXTRACT_UTC	SYSDATE	SYSTIMESTAMP
TO_CHAR (datetime)	TO_TIMESTAMP	TO_TIMESTAMP_TZ
TO_DSINTERVAL	TO_YMINTERVAL	TRUNC(date)
TZ_OFFSET		

General Comparison Functions

GREATEST	LEAST

Conversion Functions

ASCIISTR	BIN_TO_NUM	CAST
CHARTOROWID	COMPOSE	CONVERT
DECOMPOSE	HEXTORAW	NUMTODSINTERVAL
NUMTOYMINTERVAL	RAWTOHEX	RAWTONHEX

ROWIDTOCHAR	ROWIDTONCHAR	SCN_TO_TIMESTAMP
TIMESTAMP_TO_SCN	TO_BINARY_DOUBLE	TO_BINARY_FLOAT
TO_CHAR (character)	TO_CHAR (datetime)	TO_CHAR(number)
TO_CLOB	TO_DATE	TO_DSINTERVAL
TO_LOB	TO_MULTI_BYTE	TO_NCHAR(character)
TO_NCHAR (datetime)	TO_NCHAR (number)	TO_NCLOB
TO_NUMBER	TO_DSINTERVAL	TO_SINGLE_BYTE
TO_TIMESTAMP	TO_TIMESTAMP_TZ	TO_YMINTERVAL
TO_YMINTERVAL	TRANSLATE ... USING	UNISTR

Large Object Functions

BFILENAME	EMPTY_BLOB	EMPTY_CLOB

Collection Functions

CARDINALITY	COLLECT	POWERMULTISET
POWERMULTISET_BY_CARDINALITY		SET

Hierarchical Function

SYS_CONNECT_BY_PATH

Data Mining Functions

CLUSTER_ID	CLUSTER_PROBABILITY	CLUSTER_SET
FEATURE_ID	FEATURE_SET	FEATURE_VALUE
PREDICTION	PREDICTION_COST	PREDICTION_DETAILS
PREDICTION_PROBABILITY		PREDICTION_SET

XML Functions

APPENDCHILDXML	DELETEXML	DEPTH
EXTRACT (XML)	EXISTSNODE	EXTRACTVALUE
INSERTCHILDXML	INSERTXMLBEFORE	PATH
SYS_DBURIGEN	SYS_XMLAGG	SYS_XMLGEN
UPDATEXML	XMLAGG	XMLCDATA
XMLCOLATTVAL	XMLCOMMENT	XMLCONCAT
XMLFOREST	XMLPARSE	XMLPI

XMLQUERY	XMLROOT	XMLSEQUENCE
XMLSERIALIZE	XMLTABLE	XMLTRANSFORM

Encoding and Decoding Functions

DECODE	DUMP	ORA_HASH	VSIZE

NULL-Related Functions

COALESCE	LNNVL	NULLIF	NVL	NVL2

Environment and Identifier Functions

SYS_CONTEXT	SYS_GUID	SYS_TYPEID	UID
USER	USERENV		

Aggregate Functions

AVG	COLLECT	CORR CORR_*	
COUNT	COVAR_POP	COVAR_SAMP	CUME_DIST
DENSE_RANK	FIRST	GROUP_ID	GROUPING
GROUPING_ID	LAST	MAX	MEDIAN
MIN	PERCENTILE_CONT	PERCENTILE_DISC	
PERCENT_RANK	RANK	REGR_(Linear Regression)Fun.	
STATS_BINOMIAL_TEST	STATS_CROSSTAB	STATS_F_TEST	
STATS_KS_TEST	STATS_MODE	STATS_MW_TEST	
STATS_ONE_WAY_ANOVA		STATS_T_TEST_*	
STATS_WSR_TEST	STDDEV	STDDEV_POP	
STDDEV_SAMP	SUM	VAR_POP	
VAR_SAMP	VARIANCE		

Analytic Functions

AVG *	CORR *	COVAR_POP *	COVAR_SAMP *
COUNT *	CUME_DIST	DENSE_RANK	FIRST
FIRST_VALUE *	LAG	LAST	LAST_VALUE *
LEAD	MAX *	MIN *	NTILE
PERCENT_RANK	PERCENTILE_CONT	PERCENTILE_DISC	RANK
RATIO_TO_REPORT		REGR_ (Linear Regression) Functions *	
ROW_NUMBER	STDDEV *	STDDEV_POP *	STDDEV_SAMP *
SUM*	VAR_POP *	VAR_SAMP *	VARIANCE *

Object Reference Functions

DEREF	MAKE_REF	REF	REFTOHEX
VALUE			

Model Functions

CV	ITERATION_NUMBER	PRESENTNNV
PRESENTV	PREVIOUS	