ActiveNET®

Oracle

SQL, PL/SQL

By Suryanarayana

#202, Manjeera Plaza, Opp: Aditya Park Inn,
Ameerpet, HYD-38

98 48 111 2 88 ***www.activenetinformatics.com***

netactive74@gmail.com
activesurya@gmail.com

SQL:

1. Evolution of RDBMS
2. RDBMS Introduction
3. RDBMS Installation & their client tools
4. CODD Rules
5. Normalization
6. SQL Datatypes
    a. NUMBER
    b. CHAR
    c. DATE
    d. BINARY
7. Database objects
    a. SQL- Table, View, Synonym, Cluster, Materialized View, Index, Sequence
    b. PL/SQL - Function, Procedure, Package, Trigger & Cursor
8. NUMBER Functions
9. CHAR Functions
10. DATE Functions
11. DDL (Data Definition Language)
    a. CREATE, ALTER & DROP
12. DML (Data Manipulation Language)
    a. INSERT, UPDATE & DELETE
13. DRL (Data Retrieval Language)
    a. SELECT
        i. various NUMBER/Aggregate, DATE & CHAR functions
        ii. SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY,
        iii. JOIN
            1. INNER, LEFT OUTER, RIGHT OUTER, FULL, CROSS, SELF
    b. Sub/Nested Queries
        i. Single-Row-Single-Column
        ii. Multi-Row-Single-Column
        iii. Single-Row-Multi-Column
        iv. Correlated Sub Queries
14. SET operations (UNION, INTERSECT, UNION ALL, MINUS)

PL/SQL:

1. PL/SQL Architecture
2. Anonymous/Unnamed PL/SQL blocks
3. Data types in PL/SQL
4. Variable declarations
    a. NUMBER, BOOLEAN, DATE & CHAR
5. Anchored data types

6. Differentiating local vars & table columns
7. Substitution Variables
8. := intialization
9. Control Structures in PL/SQL
    a. Conditions
        i. IF-THEN, IF-THEN-ELSE, IF-THEN-ELSIF-THEN-ELSE,
        ii. Nested IF Statements
        iii. Before CASE, Searched CASE Statements
        iv. CASE Expressions
    b. Loops
        i. LOOP
        ii. WHILE Loop
        iii. Premature Termination of Loop
        iv. FOR Loop, REVERSE FOR Loop
        v. Error in FOR Loop
        vi. Scope of FOR Loop
        vii. Exiting FOR Loop with IF Condition
        viii. Exiting FOR Loop with EXIT Condition
        ix. Nested FOR Loops
10. Exception Handling in PL/SQL
11. Many ways to iterate across Cursors using loops
    a. Implicit Cursors
    b. Explicit Cursors
12. Transactions in PL/SQL
13. Functions with IN,OUT, INOUT Parameters
14. Procedures with IN,OUT, INOUT Parameters
15. Accessing Functions & Procedures from Anonymous blocks
16. CREATE Package, CREATE PACKAGE BODY
17. Calling Functions & Procedure of Package
18. Triggers
    a. BEFORE INSERT | UPDATE | DELETE Trigger
    b. AFTER INSERT | UPDATE | DELETE Trigger
    c. For Each Row
    d. For Entire Table

SQL

Evolution of RDBMS

- Ledger Books ((Manual)
- Flat files/Text files/Comma Separated value files / & separated value files
- Spreadsheet files
- Database Management Systems
- Relational Database Management Systems

RDBMS Introduction

In business system the entity data (student, emp, course, batch, payslip) is stored in various data sources - these entities attributes/properties are called as data. The system in which we are storing business entity data is called as Database Management System.

http://www.digitivity.com/articles/2008/10/evolution-of-relational-database.html

Relational Database Management System or RDBMS was developed in 1970. It was introduced by Edgar F. Codd who was working with IBM. Basically, RDBMS is a developed form of some popular database systems like hierarchical system etc.

Though this database system was developed originally by IBM, many other stalwarts related to IT came forward with the versions of their own. This led to a competition, between various vendors, to make it more user-friendly and efficient. As a matter of fact, this competition only added to make it simpler and more advanced.

RDBMS is most widely used database system as far as data storage is concerned. Out of many RDBMS products available in the market, its most famous products are Oracle, D2B family by IBM, Microsoft's access etc. Some other famous products are Foxbase, Sybase, Informix etc.

The relational database management system uses Standard Query Language(SQL). The database system keeps all the information in a tabular form. This tabular form uses certain data values to define and describe the data. This database system also ensures that no information in a specific column in repeated.

RDBMS consists of one or many tables. These tables are made of interrelation between many columns and rows. The developers use set theory. This set theory puts these data in these columns and rows. These columns and rows, with data full in them, help to do certain data related operations.

RDBMS has been regarded by many as a very simple but advanced database system. Its simplicity makes its more user-friendly. Its easy-to-use features make it more productive. To make it simpler, simple tables are used by developers. Hence, the processing and understanding of data becomes easier. It also inspire easy communication of data and their transfers.

SQL has always remained too easy to learn by a database operator or user . As RDBMS uses SQL, it becomes lot easier for anybody to master this system at a great speed. Thus, this learning ease makes it immensely productive.

Though the relational database management system works nicely with SQL, it can't be operated using any other language. The developers tried a lot the use of languages like JavaScript and C++, but failed as RDBMS resisted to work with ease using these languages.

Further, it shows its great inability to store any information in form of an image or pictures. Moreover, a user can't store any video reel or any document in form of audio. In short, nothing sort of digital things could be loaded in this system.

However, the drawbacks and limitations are too less to be counted. These few shortcomings of RDBMS stand nowhere when we have a thorough look upon its vast and effective use and operations. No doubt, it was no less than a wonder when introduced almost forty years ago and interestingly, it is a wonder still.

Ledger Book - Entries are manually made in a book.

Text Files (comma separator is used to store the entity attributes/column values, field values)
Advantages:

- Less expertise is required
- Easy to manage
- Platform independent

Limitations of text files:

- In text file structure doesn't exist (we cannot specify these many columns a text file must contain for each row, we cannot apply data types, range, pattern, & custom restrictions, we cannot apply constraints like unique, not null etc).
- Column values of each row if contains space character they cannot be treated as a single column value because space is the separator between each column value.
- Proper data formatting doesn't takes place.
- We cannot perform computation/calculation on the same column values
- No security to data
- Data duplication/redundancy.

DBMS (Data Base Management System)

DOS	Windows
D Base III+	MS-Access
Lotus 123	MS-Excel
Fox Pro	MS-Visual Fox Pro
Word Star	MS-Word

In the DB management systems we can create new database file, in that we can create tables, insert, update, delete & select operations we can perform. But in DBMS there is a chance that same column may repeat more than once in a table, the same value may repeat in more than one row of a column. The data is often redundant/duplicate.

The redundant data has following drawbacks:
i) Repeated values consume more space.
ii) If a value want to be changed it must be changed in many places, if not changed so then inconsistent problem arises.

Then the redundant data must be separated from one table to many other tables and between those tables the PK and foreign key relationship must be established.

RDBMS Installation & their client tools

RDBMSs follow CODD rules (0-12)

Rule 0: The Foundation rule:
A relational database management system must manage its stored data using only its relational capabilities. The system must qualify as relational, as a database, and as a management system. For a system to qualify as a relational database management system (RDBMS), that system must use its relational facilities (exclusively) to manage the database.

Rule 1: The information rule:
All information in a relational database (including table and column names) is represented in only one way, namely as a value in a table.

Rule 2: The guaranteed access rule:
All data must be accessible. This rule is essentially a restatement of the fundamental requirement for primary keys. It says that every individual scalar value in the database must be logically addressable by specifying the name of the containing table, the name of the containing column and the primary key value of the containing row.

Rule 3: Systematic treatment of null values:
The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values (for example, "distinct from zero or any other number", in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

Rule 4: Active online catalog based on the relational model:
The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.

Rule 5: The comprehensive data sublanguage rule:
The system must support at least one relational language that

Has a linear syntax

Can be used both interactively and within application programs,

Supports data definition operations (including view definitions), data manipulation operations (update as well as retrieval), security and integrity constraints, and transaction management operations (begin, commit, and rollback).

Rule 6: The view updating rule:

All views that are theoretically updatable must be updatable by the system.

Rule 7: High-level insert, update, and delete:
The system must support set-at-a-time insert, update, and delete operators. This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

Rule 8: Physical data independence:
Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

Rule 9: Logical data independence:
Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure. Logical data independence is more difficult to achieve than physical data independence.

Rule 10: Integrity independence:
Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

Rule 11: Distribution independence:
The distribution of portions of the database to various locations should be invisible to users of the database. Existing applications should continue to operate successfully:

when a distributed version of the DBMS is first introduced; and
when existing distributed data are redistributed around the system.

Rule 12: The nonsubversion rule:
If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

Normalization

Developer who design tables in RDBMS must follows 5 Normalization Forms:
i) 1 NF - Same column must not be repeated more than once in a table. Separate that column to another table and maintain main table PK as FK in the separated table. More than father or mother emailids in a student table.

ii) 2 NF - Multiple rows must not contain same column values. Separate them into one more table that separated table PK must be used as fk in main table. More than one student qualification may be same.

iii) 3 NF - All columns in a table must be fully functionally dependent on PK column in a row. address entity attributes/fields/column/properties are not fully dependent on student record. Hence they must be separated into one more table and maintain that table pk as a fk in student table address_id column.

iv) 3.5 NF (Three and Half) (BCNF Boyce-Codd NF) - Candidate key associates to a PK to identify a row. Composite key generates here. For instance to one course many batches exist, each faculty teaches many batches, hence courseid and facultyid are maintained as fk in batch table. Each student attends more than one batch, to each batch many students will attend. Hence batchid and studentid must be maintained in one more table. The combination of batch&studentid must occur only once hence to batchid, studentid becomes a candidate key to form a composite pk.

v) 4 NF - A row of one table contains relation with many rows of other table, and vice-versa. Then there exist multi-valued dependency between table records. Hence such multi-valued dependencies must be avoided by writing one more table and maintain both table PKs as fks in that join table whose occurrence is only once.

SQL Data types

- NUMBER
- CHAR, VARCHAR, VARCHAR2, NCHAR, NVARCHAR2
- DATE
- BINARY

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.

column_name NUMBER, Ex: sname NUMBER

column_name NUMBER(precision, scale) Ex: salary NUMBER(6,2)

precision (total number of digits) and **scale** (number of digits to the right of the decimal point)

CHAR The `CHAR` datatype stores fixed-length character strings. When you create a table with a CHAR column, you must specify a string length (in bytes or characters) between 1 and 2000 bytes for the CHAR column width. The default is 1 byte

Column_name CHAR Ex: joining_status CHAR(1)

VARCHAR2 and VARCHAR The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for theVARCHAR2 column. For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle Database returns an error. UsingVARCHAR2 and VARCHAR saves on space used by the table.

For example, assume you declare a column VARCHAR2 with a maximum size of 50 characters. In a single-byte character set, if only 10 characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece stores only the 10 characters (10 bytes), not 50.

The VARCHAR data type is synonym to VARCHAR2 datatype. Always use the VARCHAR2 datatype to store variable-length character strings

NCHAR and NVARCHAR2 NCHAR and NVARCHAR2 are Unicode datatypes that store Unicode character data. The character set of NCHAR and NVARCHAR2 datatypes can only be either AL16UTF16 or UTF8 and is specified at database creation time as the national character set. AL16UTF16 and UTF8 are both Unicode encoding.

The maximum length of an NCHAR column is 2000 bytes. The maximum length of an NVARCHAR2 column is 4000 bytes.

DATE The DATE datatype stores point-in-time values (dates and times) in a table. The DATE datatype stores the year (including the century), the month, the day, the hours, the minutes, and the seconds (after midnight).

Oracle Database can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 9999 CE (Common Era, or 'AD'). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

Oracle Database uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

For input and output of dates, the standard Oracle date format is DD-MON-YY, as follows:

```
'13-NOV-92
```

To later Date Format use ALTER SESSION SET NLS_DATE_FORMAT = ‘dd-mm-yyyy’. This date format exist as long as SQL prompt is opened.

```
To change in SQL statement TO_DATE ('November 13, 2017', 'MONTH DD, YYYY')
```

BINARY/LOB Data types The LOB datatypes BLOB, CLOB, NCLOB, and BFILE enable you to store and manipulate large blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) in binary or character format. They provide efficient, random, piece-wise access to the data. Oracle recommends that you always use LOB datatypes over LONG datatypes. You can perform parallel queries (but not parallel DML or DDL) on LOB columns.

BLOB The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to 128 terabytes of binary data.

CLOB & NCLOB datatypes The CLOB and NCLOB datatypes store up to 128 terabytes of character data in the database. CLOBs store database character set data, and NCLOBs store Unicode national character set data.

BFILE datatype The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILEcolumn or attribute stores a file locator that points to an external file containing the data. The amount of BFILEdata that can be stored is limited by the operating system.

Database objects

SQL DB objects:

Table:

RDBMS is a permanent persistence store in which we want to store many entities data. Entities such as student, employee, course, batch, attendance. To store such entities in DB, in DB one physical structure must be defined. The phy struct is table. The table contains whichever attributes of the entities we want to to store they will be defined as columns. While specifying the columns we will specify column name, data type, size, range, pattern restrictions. User whoever insert records into table must obey these restrictions.

View:

Different perspective from which we see the table data is called as View. View is also called as Projection. Selected columns of a table with or without condition want to be executed frequently instead of executing query frequently we can store the query into one view. When we query on view in-turn the stored query will execute.

Views can be updated and deleted, the result will be effected on matched records. While creating view we should not use key columns in select statement.

Synonym:

Synonym can be used on a table, view, another synonym, cluster, materialized view, sequence, procedure, function, package etc.

Synonym purposes in two ways:

- To keep the actual database object name confidential may be security reason the synonyms are used.
- If the database object names larger in size synonym can be used to shorten their names.

But synonym created on any database object doesn't change the privileges of user on the DB objects.
We can create PRIVATE and PUBLIC synonyms. PRIVATE synonyms can be used by the same user. Whereas PUBLIC synonyms can be accessed from other uses also.

Sequence:

Sequences are used to generate PK values for a table while record insertion. For example while inserting record into emp table

```
insert into emp values (1, 'ABC', 'Kumar', 'Sr Soft Engg', 25000.00);
insert into emp values (2, 'XYZ', 'Kumar', 'Soft Engg', 20000.00);

CREATE SEQUENCE emp_seq
        START WITH 1
        INCREMENT BY 1
        MAXVALUE n|NOMAXVALUE
        CYCLE|NOCYCLE
        CACHE 20|NOCACHE;

INSERT INTO emp VALUES (emp_seq.nextval, 'LMN', 'Kumar',...);
```

INSERT INTO emp VALUES (emp_seq.nextval, 'PQR', 'Kumar',...);

Index:
index is also an DB object. But unlike table (separate structure is created in the DB to store records). index is created on table itself. Whenever the table is created with PK by default one index will be created on PK field. In addition to that if developer wants to fetch data from the same table by requesting different conditions on different columns, then on such columns developer need to create indexes.

Index in SQL is created on existing tables to retrieve the rows quickly. When there are thousands of records in a table, retrieving information will take a long time. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly. Indexes can be created on a single column or a group of columns. When an index is created, it first sorts the data and then it assigns a ROWID for each row.

CREATE INDEX index_name ON table_name (column_name1, column_name2, ….);

There are two types of indexes. One is implicit index and the other one is explicit index.

Implicit indexes created by default on PRIMARY KEY and FOREIGN KEY columns while creating table.

Explicit indexes are created on table using CREATE INDEX SQL command.

Note on Indexes:

1) Even though sql indexes are created to access the rows in the table quickly, they slow down DML operations like INSERT, UPDATE, DELETE on the table, because the indexes and tables both are updated along when a DML operation is performed. So use indexes only on columns which are used to search the table frequently.
2) It is not required to create indexes on table which have less data.
3) In oracle database you can define up to sixteen (16) columns in an INDEX.

Types of indexes:
a) B-Tree index (Binary Tree)
b) Bitmap index
c) Function based index
d) Application domain index

When table column contains limited set of values such as 'Male', 'Female' or 'National', 'Foreigner' then on such columns we can create bitmap index. Bitmap index maintains 2 dimensional table in which the column values are written in a row manner and column PKs are maintained as column headers. Wherever the value matches in first row / second row for every record that value be updated as 1 otherwise 0 will be updated.

When the column value ranges are large in amount like Java, .net, PHP, Oracle etc then tree nodes are created with the column values, all rows matched with each value their corresponding ROWIDs are placed in one-one node. This is b-tree index.

Oracle In-Built Functions

There are two types of functions in Oracle:

1) Single Row Functions: Single row or Scalar functions return a value for every row that is processed in a query.

2) Group Functions: These functions group the rows of data based on the values returned by the query. This is discussed in SQL GROUP Functions. The group functions are used to calculate aggregate values like total or average, which return just one total or one average value after processing a group of rows.

There are four types of single row functions. They are:

1) Numeric Functions: These are functions that accept numeric input and return numeric values.

2) Character or Text Functions: These are functions that accept character input and can return both character and number values.

3) Date Functions: These are functions that take values that are of datatype DATE as input and return values of datatype DATE, except for the MONTHS_BETWEEN function, which returns a number.

4) Conversion Functions: These are functions that help us to convert a value in one form to another form. For Example: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE etc.

You can combine more than one function together in an expression. This is known as nesting of functions.

What is a DUAL Table in Oracle?

This is a single row and single column dummy table provided by oracle. This is used to perform mathematical calculations without using a table.

SQL: SELECT * FROM dual;

Output:

```
DUMMY
-------
X
```

SQL: SELECT 777*888 FROM dual;

Output:

```
777 * 888
---------------------------------------------------------------
689976
```

NUMERIC Functions

Numeric functions are used to perform operations on numbers. They accept numeric values as input and return numeric values as output.

ABS(), ACOS(), ASIN(), ATAN(), ATAN2(), BIT_AND(), BIT_COUNT(), BIT_OR(), CEIL(), CEILING(), CONV(), COS(), COT(), DEGREES(), EXP(), FLOOR(), FORMAT(), GREATEST(), INTERVAL(), LEAST(), LOG(), LOG10(), MOD(), OCT(), PI(), POW(), POWER(), RADIANS(), ROUND(), SIN(), SQRT(), STD(), STDDEV(), TAN(), TRUNCATE()

Function Name	Examples	Return Value
ABS (x)	ABS (1)	1
	ABS (-1)	-1
CEIL (x)	CEIL (2.83)	3
	CEIL (2.49)	3
	CEIL (-1.6)	-1
FLOOR (x)	FLOOR (2.83)	2
	FLOOR (2.49)	2
	FLOOR (-1.6)	-2
TRUNC (x, y)	ROUND (125.456, 1)	125.4
	ROUND (125.456, 0)	125
	ROUND (124.456, -1)	120
ROUND (x, y)	TRUNC (140.234, 2)	140.23
	TRUNC (-54, 1)	54
	TRUNC (5.7)	5
	TRUNC (142, -1)	140

Character OR Text Functions

Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character and number values as output.

ASCII(), BIN(), BIT_LENGTH(), CHAR_LENGTH(), CHAR(), CHARACTER_LENGTH(), CONCAT_WS(), CONCAT(), CONV(), ELT(), EXPORT_SET(), FIELD(), FIND_IN_SET(), FORMAT(),HEX(), INSERT(), INSTR(), LCASE(), LEFT(), LENGTH(), LOAD_FILE(), LOCATE(), LOWER(), LPAD(), LTRIM(), MAKE_SET(), MOD(), OCT(), OCTET_LENGTH(), ORD(), POSITION(), QUOTE(), REGEXP(), REPEAT(), REPLACE(), REVERSE(), RIGHT(), RPAD(), RTRIM(), SOUNDEX(), SOUNDLIKE(), SPACE(), STRCMP(), SUBSTRING_INDEX(), SUBSTRING(), SUBSTR(), TRIM(), UCASE(), UNHEX(), UPPER().

Function Name	Examples	Return Value
LOWER(string_value)	LOWER('Good Morning')	good morning

UPPER(string_value)	UPPER('Good Morning')	GOOD MORNING
INITCAP(string_value)	INITCAP('GOOD MORNING')	Good Morning
LTRIM(string_value, trim_text)	LTRIM ('Good Morning', 'Good)	Morning
RTRIM (string_value, trim_text)	RTRIM ('Good Morning', ' Morning')	Good
TRIM (trim_text FROM string_value)	TRIM ('o' FROM 'Good Morning')	Gd Mrning
SUBSTR (string_value, m, n)	SUBSTR ('Good Morning', 6, 7)	Morning
LENGTH (string_value)	LENGTH ('Good Morning')	12
LPAD (string_value, n, pad_value)	LPAD ('Good', 6, '*')	**Good
RPAD (string_value, n, pad_value)	RPAD ('Good', 6, '*')	Good**

DATE Functions

These are functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS_BETWEEN function, which returns a number as output.

ADDDATE(), ADDTIME(), CONVERT_TZ(), CURDATE(), CURRENT_DATE(), CURRENT_DATE, CURRENT_TIME(), CURRENT_TIME, CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP, CURTIME(), DATE_ADD(), DATE_FORMAT(), DATE_SUB(), DATE(), DATEDIFF(), DAY(), DAYNAME(), DAYOFMONTH(), DAYOFWEEK(), DAYOFYEAR(), EXTRACT(), FROM_DAYS(), FROM_UNIXTIME(), HOUR(), LAST_DAY(), LOCALTIME(), LOCALTIME, LOCALTIMESTAMP(), LOCALTIMESTAMP, MAKEDATE(), MAKETIME, MICROSECOND(), MINITE(), MONTH(), MONTHNAME(), NOW(), PERIOD_ADD(), PERIOD_DIFF(), QUARTER(), SEC_TO_TIME(), SECOND(), STR_TO_DATE(), SUBDATE(), SUBTIME(), SYSDATE(), TIMEFORMAT(), TIME_TO_SEC(), TIME(), TIMEDIFF(), TIMESTAMP(), TIMESTAMPADD(), TIMESTAMPDIFF(), TO_DAYS(), UNIX_TIMESTAMP(), UTC_DATE(), UTC_TIME(), UTC_TIMESTAMP(), WEEK(), WEEKDAY(), WEEKOFYEAR(), YEAR(), YEARWEEK().

Function Name	Examples	Return Value
ADD_MONTHS ( )	ADD_MONTHS ('16-Sep-81', 3)	16-Dec-81
MONTHS_BETWEEN( )	MONTHS_BETWEEN ('16-Sep-81', '16-Dec-81')	3
NEXT_DAY( )	NEXT_DAY ('01-Jun-08', 'Wednesday')	04-JUN-08
LAST_DAY( )	LAST_DAY ('01-Jun-08')	30-Jun-08
NEW_TIME( )	NEW_TIME ('01-Jun-08', 'IST', 'EST')	31-May-08

Conversion Functions

These are functions that help us to convert a value in one form to another form. For Ex: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE.

Function Name	Examples	Return Value

TO_CHAR ()	TO_CHAR (3000, '$9999') TO_CHAR (SYSDATE, 'Day, Month YYYY')	$3000 Monday, June 2008
TO_DATE ()	TO_DATE ('01-Jun-08')	01-Jun-08
NVL (x,y)	If x value is null, then replaces x with y. x and y must be of same date type. NVL(salary,20000.00)	20000.00

DDL (Data Definition Language)

- **CREATE TABLE**
- **ALTER TABLE**
- **DROP TABLE**
- **RENAME TABLE**
- **CREATE VIEW**
- **ALTER VIEW**
- **DROP VIEW**
- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**
- **CREATE SEQUENCE**
- **ALTER SEQUENCE**
- **DROP SEQUENCE**

CREATE TABLE

The CREATE TABLE statement is used to create a new table in a database.

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

```
sqlplus system/manager

create user am36 identified by am36;
grant connect, resource to am36;

connect am36/am36;
create table dept(deptno number,
            deptname varchar2(50),
            loc varchar2(100),
            CONSTRAINT dept_deptno_pk PRIMARY KEY(deptno));
```

```
ALTER TABLE dept ADD country VARCHAR2(50);
ALTER TABLE dept MODIFY country VARCHAR2(100);
ALTER TABLE dept DROP COLUMN country;
ALTER TABLE dept DROP CONSTRAINT dept_deptno_pk;
ALTER TABLE dept ADD CONSTRAINT dept_deptno_pk PRIMARY KEY(deptno);

CREATE TABLE employee (
                eid NUMBER,
                first_name VARCHAR2(30) NOT NULL,
                last_name VARCHAR2(30) NOT NULL,
                salary NUMBER,
                hire_date DATE,
                job_id VARCHAR2(20),
                mgr_id NUMBER,
                deptno NUMBER,
CONSTRAINT employee_eid_pk PRIMARY KEY(eid),
CONSTRAINT employee_salary_chk CHECK(salary <= 100000 AND salary >= 5000),
CONSTRAINT employee_job_id_chk CHECK (job_id IN ('Director','OB','Counsellor','Office
Admin','Faculty','PD','LC')),
CONSTRAINT employee_mgr_id_fk FOREIGN KEY (mgr_id) REFERENCES employee(eid),
CONSTRAINT employee_deptno_fk FOREIGN KEY (deptno) REFERENCES dept(deptno));

ALTER TABLE employee DROP CONSTRAINT employee_job_id_chk;

ALTER TABLE employee ADD CONSTRAINT employee_job_id_chk CHECK (job_id IN
('Director','OB','Counsellor','Office Admin','Faculty','PD','LC','Accountant'));
```

DML (Data Manipulation Language)

```
SET DEFINE OFF;
INSERT INTO dept VALUES (1,'Admin','Ameerpet');
INSERT INTO dept VALUES (2,'Accounts','Ameerpet');
INSERT INTO dept VALUES (3,'Training','Ameerpet');
INSERT INTO dept VALUES (4,'Development','Gayatri Nagar');
INSERT INTO dept VALUES (5,'R&D','SRNagar');
INSERT INTO dept VALUES (6,'Marketing','Gayatri Nagar');
COMMIT;

INSERT INTO employee (eid,first_name,last_name,salary,hire_date,job_id,deptno) VALUES
(1,'ABC','Kumar',50000.00,'01-Jan-2000','Director',1);
```

```
INSERT INTO employee (eid,first_name,last_name,salary,hire_date,job_id,deptno) VALUES
(2,'LMN','Kumar',50000.00,'01-Jan-2000','Director',3);

CREATE SEQUENCE employee_seq
                        START WITH 1
                        INCREMENT BY 1
                        NOMAXVALUE
                        NOCYCLE
                        NOCACHE;

SELECT employee_seq.nextval FROM dual;
SELECT employee_seq.currval FROM dual;
SELECT employee_seq.nextval FROM dual;

INSERT INTO employee VALUES (3,'XYZ','Rao',30000.00,'1-JAN-2001','Office Admin',1,1);

ALTER TABLE employee DROP CONSTRAINT employee_job_id_chk;

ALTER TABLE employee ADD CONSTRAINT employee_job_id_chk CHECK (job_id IN
('Director','OB','Counsellor','Office Admin','Faculty','PD','LC','Accountant','ME'));

INSERT INTO employee VALUES (4,'PQR','Rao',20000.00,to_date('1-1-01','DD-MM-YY'),'Accountant',1,2);

INSERT INTO employee VALUES (5,'DEF','Rao',25000.00,'1-Jan-10','Faculty',2,3);
INSERT INTO employee VALUES (6,'MNO','Rao',20000.00,'1-Jan-10','LC',2,3);
INSERT INTO employee VALUES (7,'UVW','Kumar',25000.00,'1-Jan-10','PD',2,4);
INSERT INTO employee VALUES (8,'IJK','Kumar',20000.00,'1-Jan-10','ME',1,6);

SELECT to_char(hire_date,'dd/mm/yyyy')
        FROM employee;

INSERT INTO employee VALUES (employee_seq.nextval, ‘&first_name’, ‘&last_name’, &sal,
to_date(‘&date’,'DD-MM-YY'), ‘&job_id’, &mgr_id, &deptno);

INSERT INTO employee VALUES (employee_seq.nextval, '&first_name', '&last_name', &sal,
to_date('&dateddmmyy','DD-MM-YY'), '&job_id', &mgr_id, &deptno);

INSERT INTO employee VALUES (employee_seq.nextval, ‘&first_name’, ‘&last_name’, &sal,
to_date(‘&dateddmmyy’,'DD-MM-YY'), ‘&job_id’, &mgr_id, &deptno);

COMMIT;
```

CREATE OR REPLACE VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

CREATE INDEX

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes; they are just used to speed up searches/queries.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

DRL (Data Retrieval Language)

SELECT statement:
SELECT statement contains 10 clauses:

SELECT list_of_columns|*
FROM table_name alias
JOIN another_table_name
WHERE conditions as column operator value AND|OR
GROUP BY column_name
HAVING condition
ORDER BY column_name asc|desc
UNION
INTERSECT
MINUS

SET LINES 256;
SET TRIMOUT ON;
SET SPACE 1;
SET TAB OFF;

```
SET FEEDBACK ON;
COLUMN first_name FORMAT a5;
COLUMN last_name FORMAT a7;
COLUMN salary FORMAT 99999;
COLUMN job_id FORMAT a12;
SET LINESIZE 100 PAGESIZE 50;
SELECT eid, first_name, salary, job_id FROM employee;
SELECT * FROM employee;
```

Simple condition:

```
SELECT * FROM employee WHERE salary>5000.00;
```

Logical Conditions:

A logical condition combines the results of two component conditions to produce a single result based on them to retrieve the result of a single condition.

```
SELECT * FROM employee WHERE mgr_id IS NULL;
SELECT * FROM employee WHERE NOT (mgr_id IS NULL);
SELECT * FROM employee WHERE first_name LIKE 'A_';
SELECT * FROM employee WHERE first_name LIKE 'A__';
SELECT * FROM employee WHERE first_name LIKE 'A%';
SELECT * FROM employee WHERE last_name LIKE '%a%';
SELECT * FROM employee WHERE last_name LIKE '%o%';
SELECT * FROM employee WHERE last_name LIKE '%o%' AND salary >= 25000;
SELECT * FROM employee WHERE last_name LIKE '%o%' OR salary >= 25000;
```

Membership Conditions:

Tests for whether record is in membership of a given list of values or subquery.

```
SELECT * FROM employee WHERE deptno IN (1,3,5);
SELECT * FROM employee WHERE deptno IN (SELECT deptno FROM dept WHERE loc='Ameerpet');
SELECT * FROM employee WHERE deptno NOT IN (SELECT deptno FROM dept WHERE loc='Ameerpet');
```

NULL Conditions:

```
SELECT * FROM employee WHERE mgr_id IS NULL;
SELECT * FROM employee WHERE mgr_id IS NOT NULL;
```

EXISTS Condition:

An EXISTS condition tests for existence of row in a subquery.

```
SELECT * FROM dept d WHERE EXISTS (SELECT * FROM employee e WHERE d.deptno = e.deptno);
```

Aggregate Functions

```
SELECT DISTINCT job_id FROM employee;
SELECT DISTINCT(job_id) FROM employee;

SELECT ALL job_id FROM employee;
SELECT ALL(job_id) FROM employee;

SELECT count(salary), SUM(salary), AVG(salary), MIN(salary), MAX(salary) FROM employee;

SELECT ROUND(12.255,2) from dual;// 12.25
SELECT ROUND(12.244,2) from dual; // 12.24
SELECT ROUND(12.245,2) from dual; // 12.25
SELECT ROUND(12.246,2) from dual; // 12.25
SELECT ROUND(12.246,1) from dual; // 12.2
SELECT ROUND(12.256,1) from dual; // 12.3
select ROUND( 1234.345, -1) from dual; //1230
select ROUND( 1235.345, -1) from dual; // 1240
select ROUND( 1234.345, -2) from dual; // 1200
select ROUND( 1235.345, -2) from dual; // 1200
select ROUND( 1250.345, -2) from dual; // 1300
select ROUND( 1250.345, -3) from dual; // 1000
select ROUND( 1450.345, -3) from dual; // 1000
select ROUND( 1550.345, -3) from dual; // 2000

select TRUNC(12.255,2) FROM dual; // 12.25
select TRUNC(12.254,2) FROM dual; // 12.25
select TRUNC(12.245,2) FROM dual; // 12.24
select TRUNC(12.246,2) FROM dual; // 12.24
select TRUNC(12.246,-1) FROM dual; // 10
SELECT TRUNC( 1234.345, 2) from dual; // 1234.34
SELECT TRUNC( 1234.345, 0) from dual; // 1234
SELECT TRUNC( 1234.345, -1) from dual; // 1230
SELECT TRUNC( 1234.345, -2) from dual; // 1200
SELECT TRUNC( 1234.345, -3) from dual; // 1000
SELECT TRUNC( 1234.345, -4) from dual; // 0
SELECT TRUNC( 1234.345, -5) from dual; // 0
```

Which employee salary is highest?

```
select * from employee where salary=(select max(salary) from employee);
```

Which employee salary is lowest?

```
select * from employee where salary=(select min(salary) from employee);
```

Which employees whose salary is below the highest salary?

```
select * from employee where salary<(select max(salary) from employee);
```

Find second highest salary employee?

```
select max(salary) from employee where salary<(select max(salary) from employee);
select * from employee where salary=(select max(salary) from employee where salary<(select max(salary) from employee));
```

Find third highest salary employee?

```
select max(salary) from employee where salary<(select max(salary) from employee where salary<(select max(salary) from employee));
```

Findout nth highest salary employee?

```
SELECT *
FROM Employee Emp1
WHERE (4-1) = (
SELECT COUNT(DISTINCT(Emp2.Salary))
FROM Employee Emp2
WHERE Emp2.Salary > Emp1.Salary);

SELECT *
FROM Employee Emp1
WHERE (4) = (
SELECT COUNT(DISTINCT(Emp2.Salary))
FROM Employee Emp2
WHERE Emp2.Salary >= Emp1.Salary);

select * from (select eid,first_name,salary ,dense_rank()over(order by salary) emp_rank from employee) samp_emp where samp_emp.emp_rank=5;

SELECT SUM(salary) from employee;

select  abs(-123),  abs(0),  abs(456), sign(-123), sign(0), sign(456) from   dual;

select ceil(12.234) from dual;
select floor(12.234) from dual;
select floor(12.534) from dual;
select exp(4) "e to the 4th power" from DUAL;
select remainder(3,2) from dual;
select remainder(10,2) from dual;
SELECT ROUND(15.193,1) "Round" FROM DUAL;
SELECT ROUND(15.193,-1) "Round" FROM DUAL;
SELECT ROUND(1.5), ROUND(2.5) FROM DUAL;
SELECT ROUND(1.5f), ROUND(2.5f) FROM DUAL;
SELECT SQRT(26) "Square root" FROM DUAL;
SELECT SQRT(36) "Square root" FROM DUAL;
SELECT SQRT(37) "Square root" FROM DUAL;
SELECT SQRT(48) "Square root" FROM DUAL;
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;
SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;
SELECT CONCAT(CONCAT(last_name, ' "s job category is '), job_id) "Job" FROM employee
WHERE eid = 4;

SELECT INITCAP('the soap') "Capitals" FROM DUAL;
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase" FROM DUAL;
select lpad(first_name,15,'*.') FROM employee;
SELECT RTRIM('BROWNING: ./=./=./=./=./=.=','/=.') "RTRIM example" FROM DUAL;

create table names_ (id number, name  varchar2(40));
insert into names_ values (1,   'Hofmann' );
insert into names_ values (2,   'Hoffmann');
insert into names_ values (3,   'Meyer'   );
insert into names_ values (4,   'Maier'   );
insert into names_ values (5,   'Meier'   );
insert into names_ values (6,   'Saffran' );
insert into names_ values (7,   'Saphran' );
insert into names_ values (8,   'A. Basler');
commit;

select * from names_ where soundex(name) = soundex('hofmann');
select * from names_ where soundex(name) = soundex('meier');
select * from names_ where soundex(name) = soundex('safran');
select * from names_ where soundex(name) = soundex('a basler');

insert into names_ values (9, 'Surya');
insert into names_ values (10, 'Suria');

insert into names_ values (11, 'Narayana');
insert into names_ values (12, 'Naraina');

insert into names_ values (13, 'Lakshmi');
insert into names_ values (14, 'Laxmi');

insert into names_ values (15, 'Sri');
insert into names_ values (16, 'Sree');
```

```
commit;

select * from names_ where soundex(name)=soundex('Surya');
select * from names_ where soundex(name)=soundex('Narayana');
select * from names_ where soundex(name)=soundex('Sri');

SELECT SUBSTR('ABCDEFG',3,4) "Substring" FROM DUAL;
SELECT SUBSTR('ABCDEFG',-5,4) "Substring" FROM DUAL;
SELECT SUBSTRB('ABCDEFGHIJ',5,4.2) "Substring with bytes" FROM DUAL;

select trim(TRAILING ' ' FROM first_name), trim(TRAILING ' ' FROM last_name) from employee;
select trim(LEADING ' ' FROM first_name), trim(LEADING ' ' FROM last_name) from employee;

select trim(TRAILING ' ' FROM ' Anand   '), trim(TRAILING ' ' FROM '   Bujji   ') from dual;
select trim(LEADING ' ' FROM ' Anand   '), trim(LEADING ' ' FROM '   Bujji   ') from dual;
select trim(BOTH ' ' FROM ' Anand   '), trim(BOTH ' ' FROM '   Bujji   ') from dual;

select trim(BOTH ' ' FROM first_name), trim(BOTH ' ' FROM last_name) from employee;
select trim(first_name), trim(last_name) from employee;
select trim(TRAILING 'C' FROM first_name), trim(TRAILING ' ' FROM last_name) from employee;

select months_between(to_date('2012/01/04', 'yyyy/mm/dd'), to_date('2012/05/04','yyyy/mm/dd'))
from dual;
select months_between(to_date('2012/05/04','yyyy/mm/dd'),to_date('2012/01/04', 'yyyy/mm/dd'))
from dual;
select next_day(sysdate, 'Saturday') from dual;

select numtodsinterval(150, 'HOUR') from dual;
```

SET operations in SQL
SET operation combine and fetches records commonly available in more than one table.

UNION - Fetches distinct (if the row is found in both the tables, it will fetch only one row) rows from both the tables.
UNION ALL - Fetches distinct and duplicate rows from both the tables.
INTERSECT - Fetches common rows
MINUS - Retrieves uncommon rows

```
CREATE TABLE oldemployee (eid NUMBER PRIMARY KEY, first_name VARCHAR2(30), last_name
VARCHAR2(30), salary number, tdate DATE);
INSERT INTO oldemployee (eid, first_name, last_name, tdate) VALUES (4, 'XYZ', 'Rao', '1-Jan-2009');
INSERT INTO oldemployee (eid, first_name, last_name, tdate) VALUES (6, 'PQR', 'Rao', '1-Jan-2010');
COMMIT;

SELECT eid, first_name, last_name , salary, null as "Transfer Date" FROM employee
UNION
SELECT eid, first_name, last_name , null as "Salary", tdate FROM oldemployee;

SELECT eid, first_name, last_name , salary, null as "Transfer Date" FROM employee
UNION ALL
SELECT eid, first_name, last_name , null as "Salary", tdate FROM oldemployee;

SELECT eid, first_name, last_name , salary, null as "Transfer Date" FROM employee
INTERSECT
SELECT eid, first_name, last_name , null as "Salary", tdate FROM oldemployee;

SELECT eid, first_name, last_name , salary, null as "Transfer Date" FROM employee
MINUS
SELECT eid, first_name, last_name , null as "Salary", tdate FROM oldemployee;
```

JOINS

Joins are used to combinedly fetch data from two related tables. To apply joins the relational tables must have a common column. In one-to-many relationships one table PK may act as foreign key to many rows in other table. In one-to-one relationship one table PK acts as a FK in other table and the same key also acts as PK. In absence of joins from the PK existing table we need to fetch PK and that should be passed as argument in FK exsiting table to fetch child table records. So that number of queries executed on DB are increases to as many PK exist.

This problem can be solved with joins:

join with table alias:

equi-joins:

```
select e.eid, e.first_name, d.deptno, d.deptname
                              FROM employee e, dept d
                              WHERE e.deptno=d.deptno;

select *
          FROM employee e, dept d
          WHERE e.deptno=d.deptno;
```

non equi-joins:

```
create table salgrade (grade CHAR(1),
                              losal number,
                              hisal number);
```

```
insert into salgrade values ('A', 0.0, 10000.00);
insert into salgrade values ('B', 10001.0, 20000.00);
insert into salgrade values ('C', 20001.0, 30000.00);
insert into salgrade values ('D', 30001.0, 40000.00);
insert into salgrade values ('E', 40001.0, 50000.00);
commit;
cle scr

select e.eid, e.first_name, e.salary, s.grade
                    from employee e, salgrade s
                    where e.salary between s.losal and s.hisal;
```

self-joins:

```
select e.eid, e.first_name, m.eid, m.first_name
                              from employee e, employee m
                              where e.mgr_id=m.eid;
```

ansi joins:

inner: fetches common records from both the tables.

```
select * from employee e, dept d WHERE e.deptno=d.deptno;
```

left outer: all LHS records and matched records from RHS table

```
insert into employee (eid, first_name, last_name) values (9, 'EFG', 'Kumar');
select * from employee e, dept d where e.deptno=d.deptno(+);
```

right outer: all RHS records and matched records from LHS table

```
select * from employee e, dept d where e.deptno(+)=d.deptno;
```

cross join: For every row of left side table, all records of right hand side table are displayed.

```
select * from employee e, dept d;
```

SQL joins:

inner join:

```
select * from employee e INNER JOIN dept d ON e.deptno=d.deptno;
```

left outer:

```
select * from employee e LEFT OUTER JOIN dept d ON e.deptno=d.deptno;
```

right outer:

```
select * from employee e RIGHT OUTER JOIN dept d ON e.deptno=d.deptno;
```

full join:

```
select * from employee e FULL JOIN dept d ON e.deptno=d.deptno;
```

and cross joins:

```
select * from employee e CROSS JOIN dept d;
```

GROUP BY

```
SELECT deptno, avg(salary) FROM employee GROUP BY deptno;
SELECT deptno, avg(salary) FROM employee GROUP BY deptno;
SELECT deptno, min(salary) FROM employee GROUP BY deptno;
SELECT deptno, max(salary) FROM employee GROUP BY deptno;
SELECT deptno, count(*) FROM employee GROUP BY deptno;
SELECT SUM(salary), deptno FROM employee GROUP BY deptno HAVING sum(salary) >= 5000;

SELECT deptno, COUNT(*) FROM employee GROUP BY deptno HAVING COUNT(*) >= 2;

SELECT (select deptname from dept d where d.deptno=e.deptno), COUNT(*) FROM employee e GROUP
BY deptno HAVING COUNT(*) >= 2;

select d.deptname, avg(e.salary)
from employee e inner join dept d
on e.deptno=d.deptno
group by d.deptname;

SELECT d.loc, count(e.eid) FROM employee e INNER JOIN dept d ON e.deptno = d.deptno GROUP BY
d.loc;

SELECT d.loc, count(*) FROM employee e INNER JOIN dept d ON e.deptno = d.deptno GROUP BY d.loc;

SELECT eid, first_name, salary, CASE deptno
WHEN 1 THEN 'Admin'
WHEN 2 THEN 'Accounts'
WHEN 3 THEN 'RandD'
ELSE 'Unknown' END
FROM employee;

SELECT
AVG(CASE
        WHEN e.salary>2000
        THEN e.salary
        ELSE 2000
        END) "Average Salary" FROM employee e;
```

```
SELECT
AVG(CASE
        WHEN e.salary>50000
        THEN e.salary
        ELSE 30000
        END) "Average Salary" FROM employee e;
```

Simple Queries

```
SELECT first_name, salary, job_id FROM employee WHERE salary >
                        (SELECT salary FROM employee WHERE eid = 8);

SELECT Ename, Sal, Job FROM Emp WHERE Job =
                        (SELECT Job FROM Emp WHERE Ename = UPPER('smith')) ORDER BY Sal;


SELECT Empno, Ename, Hiredate, Sal FROM Emp WHERE Hiredate >
                        (SELECT Hiredate FROM Emp WHERE Ename = 'TURNER') ORDER BY Sal;

SELECT Empno, Ename, Sal, Job FROM Emp WHERE Deptno =
                        (SELECT Deptno FROM Dept WHERE Dname = ‘SALES’);

SELECT Empno, Ename, Sal, Comm, Sal + NVL( Comm, 0 ) FROM Emp WHERE Deptno =
                        (SELECT Deptno FROM Dept WHERE Loc = 'DALLAS');
```

Having clause

```
SELECT deptno, min(salary) FROM employee GROUP BY deptno HAVING  min(salary) > (SELECT
min(salary) FROM employee WHERE deptno = 2);

SELECT job_id, avg(salary), TO_CHAR(AVG(salary),'L99,999.99') FROM employee GROUP BY job_id
HAVING  AVG(salary) <
                        (SELECT MAX(AVG(salary)) FROM employee GROUP BY deptno);
```

GROUP Functions:

```
SELECT STDDEV(ALL salary) "Deviation" FROM employee;
SELECT first_name, job_id, salary FROM employee WHERE salary < (SELECT STDDEV(salary) FROM
employee);
```

Inline

```
SELECT e.first_name, e.salary, e.deptno, e1.SalAvg FROM employee e, (SELECT deptno, AVG(salary)
SalAvg FROM employee GROUP BY deptno) e1 WHERE e.deptno = e1.deptno AND e.salary > e1.SalAvg;
```

SELECT E.first_name, E.salary, E.deptno, ROUND(E1.SalAvg, 2) DeptAvgSal, ROUND(E.salary - E1.SalAvg) DiffSalAvg
FROM employee E, (SELECT deptno, AVG(salary) SalAvg FROM employee GROUP BY deptno) E1 WHERE E.deptno = E1.deptno
ORDER BY deptno;

SELECT T1.deptno, T1.deptname, T2.Staff FROM Dept T1,(SELECT deptno, COUNT(*) AS Staff FROM employee GROUP BY deptno) T2 WHERE T1.deptno = T2.deptno AND T2.Staff >= 1;

Multi-column sub query

SELECT first_name, deptno, salary FROM employee WHERE (deptno, salary) IN (SELECT deptno, MAX(salary) FROM employee GROUP BY deptno);

SELECT first_name, deptno, salary FROM employee WHERE deptno IN (SELECT deptno
FROM employee GROUP BY deptno) AND salary IN (SELECT MAX(salary) FROM employee GROUP BY deptno);

Multi-row, Single-column

SELECT first_name, salary, deptno FROM employee WHERE salary IN(select MIN(salary) FROM employee GROUP BY deptno);

SELECT first_name, salary, deptno FROM employee WHERE salary IN(select MAX(salary) FROM employee GROUP BY deptno);

SELECT eid, first_name, job_id, salary, deptno FROM employee WHERE salary IN (select salary FROM employee WHERE hire_date IN(SELECT hire_date FROM employee WHERE deptno = 3));

SubSelect in SELECT:

SELECT first_name, salary, (SELECT AVG(salary) FROM employee) "Organization Average" FROM employee;

select 1000/4 from dual;

SELECT first_name, salary, (SELECT SUM(salary) FROM employee)/(SELECT count(*) FROM employee) "Organization Average" FROM employee;

Correlated sub query:

In a SQL database query, a **correlated subquery** (also known as a synchronized **subquery**) is a **subquery** (a query nested inside another query) that uses values from the outer query. Because the **subquery** may be evaluated once for each row processed by the outer query, it can be inefficient.

SELECT eid, first_name, E.deptno, salary, mgr_id FROM employee E WHERE E.salary > ANY (SELECT M.salary FROM employee M WHERE M.eid = E.mgr_id);

SELECT eid, first_name, E.deptno, salary, mgr_id FROM employee E WHERE E.salary < ANY (SELECT M.salary FROM employee M WHERE M.eid = E.mgr_id);

SELECT Empno, Ename, E.Deptno, Sal, MGR FROM Emp E WHERE E.Sal <ALL (SELECT M.Sal FROM Emp M WHERE M.Empno = E.MGR)

SELECT Empno, Ename, E.Deptno, Sal, MGR FROM Emp E WHERE E.Sal >ALL (SELECT M.Sal FROM Emp M WHERE M.Empno = E.MGR);

SELECT Empno, Ename, E.Deptno, Sal, MGR FROM Emp E WHERE E.Sal IN (SELECT M.Sal FROM Emp M WHERE M.Empno = E.MGR);

PL/SQL

PL/SQL Architecture:

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.

When a problem can be solved using SQL, you can issue SQL statements from your PL/SQL programs, without learning new APIs.

Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap run-time errors.

You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

PL/SQL has these advantages:

- Tight Integration with SQL
- High Performance
- High Productivity
- Full Portability
- Tight Security
- Access to Predefined Packages
- Support for Object-Oriented Programming
- Support for Developing Web Applications and Server Pages

Tight Integration with SQL

SQL has become the standard database language because it is flexible, powerful, and easy to learn. A few English-like statements such as SELECT, INSERT, UPDATE, and DELETE make it easy to manipulate the data stored in a relational database.

PL/SQL is tightly integrated with SQL. With PL/SQL, you can use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudocolumns.

PL/SQL fully supports SQL data types. You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a database column of the SQL type VARCHAR2, it can store that value in a PL/SQL variable of the type VARCHAR2. Special PL/SQL language features let you work with table columns and rows without specifying the data types, saving on maintenance work when the table definitions change.

Running a SQL query and processing the result set is as easy in PL/SQL as opening a text file and processing each line in popular scripting languages. Using PL/SQL to access metadata about database objects and handle database error conditions, you can write utility programs for database administration that are reliable and produce readable output about the success of each operation. Many database features, such as triggers and object types, use PL/SQL. You can write the bodies of triggers and methods for object types in PL/SQL.

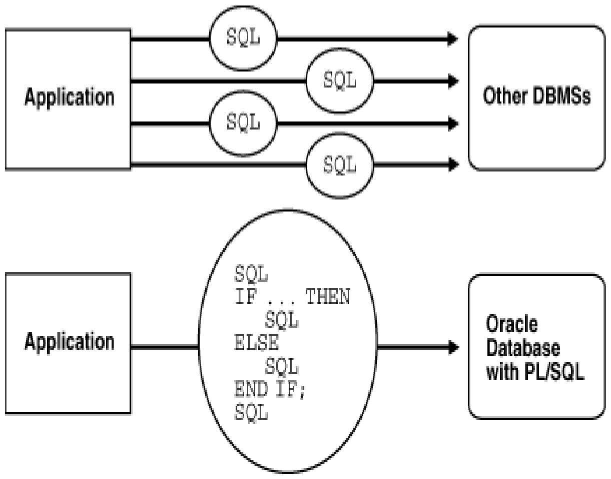
PL/SQL supports both static and dynamic SQL. **Static SQL** is SQL whose full text is known at compilation time.**Dynamic SQL** is SQL whose full text is not known until run time. Dynamic SQL enables you to make your applications more flexible and versatile.

High Performance

With PL/SQL, an entire block of statements can be sent to the database at one time. This can drastically reduce network traffic between the database and an application. As below figure shows, you can use PL/SQL blocks and subprograms (procedures and functions) to group SQL statements before sending them to the database for execution. PL/SQL also has language features to further speed up SQL statements that are issued inside a loop.

PL/SQL stored subprograms are compiled once and stored in executable form, so subprogram calls are efficient. Because stored subprograms execute in the database server, a single call over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and call overhead.

PL/SQL Boosts Performance



High Productivity

PL/SQL lets you write very compact code for manipulating data. In the same way that scripting languages such as PERL can read, transform, and write data from files, PL/SQL can query, transform, and update data in a database. PL/SQL saves time on design and debugging by offering a full range of software-engineering features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

Full Portability

Applications written in PL/SQL can run on any operating system and platform where the database runs. With PL/SQL, you can write portable program libraries and reuse them in different environments.

Tight Security

PL/SQL stored subprograms move application code from the client to the server, where you can protect it from tampering, hide the internal details, and restrict who has access. For example, you can grant users access to a subprogram that updates a table, but not grant them access to the table itself or to the text of the UPDATEstatement. Triggers written in PL/SQL can control or record changes to data, making sure that all changes obey your business rules.

Access to Predefined Packages

Oracle provides product-specific packages that define APIs you can invoke from PL/SQL to perform many useful tasks. These packages include DBMS_ALERT for using triggers, DBMS_FILE for reading and writing operating system text files, UTL_HTTP for making hypertext transfer protocol (HTTP) callouts, DBMS_OUTPUT for display output from PL/SQL blocks and subprograms, and DBMS_PIPE for communicating over named pipes.

Support for Object-Oriented Programming

Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides enabling you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

By encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods. Also, object types hide implementation details, so that you can change the details without affecting client programs.

Support for Developing Web Applications and Server Pages

You can use PL/SQL to develop Web applications and Server Pages (PSPs).

Anonymous/Unnamed PL/SQL blocks

```
SET SERVEROUTPUT ON;
BEGIN
   SELECT first_name, last_name
     INTO v_first_name, v_last_name
     FROM employee
    WHERE empid = 123;

   DBMS_OUTPUT.PUT_LINE ('Employee name: '||v_first_name||
      ' '||v_last_name);
END;
/

SET SERVEROUTPUT ON;
DECLARE
   v_first_name VARCHAR2(35);
   v_last_name VARCHAR2(35);
   v_salary NUMBER := 0;
BEGIN
   SELECT first_name, last_name
     INTO v_first_name, v_last_name
     FROM employee
    WHERE eid = 1;

   DBMS_OUTPUT.PUT_LINE ('Employee name: '||v_first_name||
      ' '||v_last_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no employee with '||
         'employee id 123');
END;
/

SET SERVEROUTPUT ON;
DECLARE
   v_empid NUMBER := &sv_emp_id;
   v_first_name VARCHAR2(35);
   v_last_name VARCHAR2(35);
```

```
BEGIN
   SELECT first_name, last_name
     INTO v_first_name, v_last_name
     FROM employee
    WHERE eid = v_empid;

   DBMS_OUTPUT.PUT_LINE ('Employee name: '||v_first_name||
      ' '||v_last_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no such employee');
END;
/

SET SERVEROUTPUT ON;
DECLARE
   exception VARCHAR2(15);
BEGIN
   exception := 'This is a test';
   DBMS_OUTPUT.PUT_LINE(exception);
END;
/

SET SERVEROUTPUT ON;
DECLARE
   v_var1 NUMBER(2) := 123;
   v_var2 NUMBER(3) := 123;
   v_var3 NUMBER(5,3) := 123456.123;
BEGIN
   DBMS_OUTPUT.PUT_LINE('v_var1: '||v_var1);
   DBMS_OUTPUT.PUT_LINE('v_var2: '||v_var2);
   DBMS_OUTPUT.PUT_LINE('v_var3: '||v_var3);
END;
/

SET SERVEROUTPUT ON
DECLARE
   v_name employee.first_name%TYPE:='ABC';
   v_salary employee.salary%TYPE:=50000.00;
BEGIN
   DBMS_OUTPUT.PUT_LINE(NVL(v_name, 'No Name ')||' has salary of '||NVL(v_salary, 0));
END;
```

```
/

SET SERVEROUTPUT ON
DECLARE
  v_cookies_amt NUMBER := 2;
  v_calories_per_cookie CONSTANT NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('I ate ' || v_cookies_amt ||' cookies with ' ||  v_cookies_amt *
v_calories_per_cookie || ' calories.');

  v_cookies_amt := 3;
  DBMS_OUTPUT.PUT_LINE('I really ate ' ||v_cookies_amt|| ' cookies with ' || v_cookies_amt *
v_calories_per_cookie || ' calories.');

  v_cookies_amt := v_cookies_amt + 5;
  DBMS_OUTPUT.PUT_LINE('The truth is, I actually ate '|| v_cookies_amt || ' cookies with '
||v_cookies_amt * v_calories_per_cookie|| ' calories.');
END;
/

set serveroutput on
<<find_stu_num>>
BEGIN
  DBMS_OUTPUT.PUT_LINE('The procedure find_stu_num has been executed.');
  END find_stu_num;
/

<< outer_block >>
DECLARE
  v_test NUMBER := 123;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Outer Block, v_test: '||v_test);
  << inner_block >>
  DECLARE
    v_test NUMBER := 456;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inner Block, v_test: '||v_test);
    DBMS_OUTPUT.PUT_LINE('Inner Block, outer_block.v_test: '||outer_block.v_test);
  END inner_block;
END outer_block;
/
```

```
SET SERVEROUTPUT ON
DECLARE
   e_show_exception_scope EXCEPTION;
   v_student_id NUMBER := 123;
BEGIN
  DBMS_OUTPUT.PUT_LINE('outer student id is '||v_student_id);
   DECLARE
     v_student_id    VARCHAR2(8) := 125;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('inner student id is '||v_student_id);
      RAISE e_show_exception_scope;
      DBMS_OUTPUT.PUT_LINE('Statement after exception raise');
   END;
EXCEPTION
   WHEN e_show_exception_scope
   THEN
      DBMS_OUTPUT.PUT_LINE('When am I displayed?');
      DBMS_OUTPUT.PUT_LINE('outer student id is '||v_student_id);
END;
/

SET SERVEROUTPUT ONE;
DECLARE
-- fname VARCHAR2(50);
v_eid employee.eid%TYPE:=&eid;
fname employee.first_name%TYPE;
BEGIN
SELECT first_name INTO fname FROM employee WHERE eid=v_eid;
DBMS_OUTPUT.PUT_LINE('Employee '||v_eid||' first name is '||fname);
END;
/

SET SERVEROUTPUT ON
DECLARE
   v_average_salary VARCHAR2(13);
BEGIN
   SELECT TO_CHAR(AVG(salary),'$9,99,999.99') INTO v_average_salary FROM employee;
   DBMS_OUTPUT.PUT_LINE('The average salary of an '||'employee in our organization is
'||v_average_salary);
END;
/
```

```
CREATE TABLE student (student_id NUMBER PRIMARY KEY,
                      last_name varchar2(20),
                      zip number,
                      created_by varchar2(20),
                      created_date date,
                      modified_by varchar2(20),
                      modified_date date,
                      registration_date date);
```

insert record into table with max incremented value

```
DECLARE
  v_max_id number:=0;
  counter number:=0;
BEGIN
  SELECT count(*) INTO counter FROM student;
  IF counter = 0 THEN
    v_max_id:=0;
  ELSE
    SELECT MAX(student_id) INTO v_max_id FROM student;
  END IF;
  v_max_id:=v_max_id+1;
  -- DBMS_OUTPUT.PUT_LINE(v_max_id||' Hello ');
  INSERT into student(student_id, last_name, zip, created_by, created_date, modified_by,
modified_date, registration_date) VALUES (v_max_id, 'Rosenzweig', 11238, 'BROSENZ', '01-JAN-99',
'BROSENZ', '01-JAN-99', '01-JAN-99');
END;
/
```

(or)

```
DECLARE
  v_max_id number:=0;
  counter number:=0;
BEGIN
  /* SELECT count(*) INTO counter FROM student;
  IF counter = 0 THEN
    v_max_id:=0;
  ELSE
    SELECT MAX(student_id) INTO v_max_id FROM student;
  END IF;
  v_max_id:=v_max_id+1;
  -- DBMS_OUTPUT.PUT_LINE(v_max_id||' Hello ');
  */
  SELECT NVL(MAX(student_id),0) INTO v_max_id FROM student;
  v_max_id:=v_max_id+1;
  INSERT into student(student_id, last_name, zip, created_by, created_date, modified_by,
modified_date, registration_date) VALUES (v_max_id, 'Rosenzweig', 11238, 'BROSENZ', '01-JAN-99',
'BROSENZ', '01-JAN-99', '01-JAN-99');
END;
/
```

<u>insert record into table with sequence generated value</u>

```
drop sequence student_seq;
create sequence student_seq
                    start with 1
                    increment by 1
                    NOCYCLE
                    NOCACHE
                    NOMAXVALUE;

select student_seq.nextval from dual;

set serveroutput on;
DECLARE
  v_user student.created_by%TYPE;
  v_date student.created_date%TYPE;
BEGIN
  SELECT USER, sysdate INTO  v_user, v_date FROM dual;
 INSERT INTO student
 (student_id, last_name, zip, created_by, created_date, modified_by, modified_date, registration_date)
 VALUES (student_seq.nextval, 'Smith', 11238, v_user, v_date, v_user, v_date,       v_date);
END;
/
```

<u>Example on Savepoint</u>

```
SET SERVEROUTPUT ON
DECLARE
 sumsalary number;
BEGIN
UPDATE employee
  SET salary = 95000
  WHERE last_name = 'Kumar';

  SAVEPOINT justsmith;
  UPDATE employee
  SET salary = 100000;

  SAVEPOINT everyone;

  SELECT SUM(salary) INTO sumsalary FROM employee;

  DBMS_OUTPUT.PUT_LINE(sumsalary);

  ROLLBACK TO SAVEPOINT justsmith;

  COMMIT;
END;
/
```

Savepoint will act like pointer or mark. Once after savepoint is created whatever the transaction we do will be stored under that savepoint, when we say rollback to savepoint justsmith, whatever the TX we did after savepoint to till rollback statement are all rollbacked, after that if we say commit, the statements/updation performed before savepoint creation are committed into DB.

Examples IF Condition

```
SET SERVEROUTPUT ON
DECLARE
  v_num1 NUMBER := 5;
  v_num2 NUMBER := 3;
  v_temp NUMBER;
BEGIN
  -- if v_num1 is greater than v_num2 rearrange their values
  IF v_num1 > v_num2 THEN
    v_temp := v_num1;
    v_num1 := v_num2;
    v_num2 := v_temp;
  END IF;

  -- display the values of v_num1 and v_num2
  DBMS_OUTPUT.PUT_LINE ('v_num1 = '||v_num1);
  DBMS_OUTPUT.PUT_LINE ('v_num2 = '||v_num2);
END;
/

SET SERVEROUTPUT ON
DECLARE
```

```
  v_num NUMBER := &sv_user_num;
  yesno varchar2(1);
BEGIN
  -- test if the number provided by the user is even
  IF MOD(v_num,2) = 0  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END IF;
  yesno:='&yorn';
  DBMS_OUTPUT.PUT_LINE (yesno);
  DBMS_OUTPUT.PUT_LINE ('Done');
END;
/

SET SERVEROUTPUT ON
DECLARE
  v_num NUMBER := &sv_num;
BEGIN
  IF v_num < 0 THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
  ELSIF v_num = 0 THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is equal to zero');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
 END IF;
END;
/

SET SERVEROUTPUT ON
DECLARE
  v_num1 NUMBER := &sv_num1;
  v_num2 NUMBER := &sv_num2;
  v_total NUMBER;
BEGIN
  IF v_num1 > v_num2 THEN
    DBMS_OUTPUT.PUT_LINE ('IF part of the outer IF');
    v_total := v_num1 - v_num2;
  ELSE
    DBMS_OUTPUT.PUT_LINE ('ELSE part of the outer IF');
    v_total := v_num1 + v_num2;
    IF v_total < 0 THEN
      DBMS_OUTPUT.PUT_LINE ('Inner IF');
      v_total := v_total * (-1);
    END IF;

  END IF;
  DBMS_OUTPUT.PUT_LINE ('v_total = '||v_total);
END;
/

SET SERVEROUTPUT ON
DECLARE
  v_letter CHAR(1) := '&sv_letter';
BEGIN
  IF (v_letter >= 'A' AND v_letter <= 'Z') OR
    (v_letter >= 'a' AND v_letter <= 'z')
  THEN
    DBMS_OUTPUT.PUT_LINE ('This is a letter');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('This is not a letter');

    IF v_letter BETWEEN '0' and '9' THEN
      DBMS_OUTPUT.PUT_LINE ('This is a number');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('This is not a number');
    END IF;

  END IF;
END;
/
```

Before CASE and Searched CASE statement

```
SET SERVEROUTPUT ON
DECLARE
  v_num NUMBER := &sv_user_num;
BEGIN
  -- test if the number provided by the user is even
  IF MOD(v_num,2) = 0  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('Done');
END;
/
```

CASE Statement

```
SET SERVEROUTPUT ON
DECLARE
  v_num NUMBER := &sv_user_num;
  v_num_flag NUMBER;
BEGIN
  v_num_flag:=MOD(v_num,2);
  CASE v_num_flag
     WHEN 0 THEN
          DBMS_OUTPUT.PUT_LINE(v_num||' is a even number');
         ELSE
          DBMS_OUTPUT.PUT_LINE(v_num||' is a odd number');
  END CASE;
  DBMS_OUTPUT.PUT_LINE(' Done ');
END;
/
```

Searched CASE Statement

```
SET SERVEROUTPUT ON
DECLARE
  v_num NUMBER := &sv_user_num;
BEGIN
  CASE
     WHEN MOD(v_num,2) = 0 THEN
          DBMS_OUTPUT.PUT_LINE(v_num||' is a even number');
         ELSE
          DBMS_OUTPUT.PUT_LINE(v_num||' is a odd number');
  END CASE;
  DBMS_OUTPUT.PUT_LINE(' Done ');
END;
/
```

CASE Expression

```
SET SERVEROUTPUT ON;
DECLARE
  v_num NUMBER := &sv_user_num;
  v_num_flag NUMBER;
  v_result VARCHAR2(30);
```

```
BEGIN
  v_num_flag := MOD(v_num,2);

  v_result :=
    CASE v_num_flag
      WHEN 0 THEN v_num||' is even number'
      ELSE v_num||' is odd number'
    END;
  DBMS_OUTPUT.PUT_LINE (v_result);
  DBMS_OUTPUT.PUT_LINE ('Done');
END;
/
```

<u>Exception Handling:</u>
<u>Handling Compilation error</u>

```
SET SERVEROUTPUT ON
DECLARE
  v_num1 INTEGER := &sv_num1;
  v_num2 INTEGER := &sv_num2;
  v_result NUMBER;
BEGIN
  v_result = v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
END;
/

SET SERVEROUTPUT ON
DECLARE
  v_num1 INTEGER := &sv_num1;
  v_num2 INTEGER := &sv_num2;
  v_result NUMBER;
BEGIN
  v_result := v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
END;
/

SET SERVEROUTPUT ON
DECLARE
  v_num1 integer := &sv_num1;
  v_num2 integer := &sv_num2;
  v_result number;
```

```
BEGIN
  v_result := v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE
      ('A number cannot be divided by zero.');
END;
/

SET SERVEROUTPUT ON;
DECLARE
  v_employee_name VARCHAR2(50);
BEGIN
  SELECT first_name||' '||last_name
    INTO v_employee_name
    FROM employee
   WHERE eid = 10;

  DBMS_OUTPUT.PUT_LINE ('Employee name is '||v_employee_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such employee');
END;
/
```

Iterative Controls

LOOP:

```
SET SERVEROUTPUT ON
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
   IF v_counter > 5 THEN
     EXIT;
   END IF;
   DBMS_OUTPUT.PUT_LINE(v_counter);
   v_counter:=v_counter+1;
  END LOOP;
END;
/
```

WHILE Loop:

```
SET SERVEROUTPUT ON
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 5 LOOP
     DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
     v_counter := v_counter + 1;
     -- v_counter+=1;
  END LOOP;
END;
/
```

Premature termination of WHILE Loop:

```
SET SERVEROUTPUT ON
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 5 LOOP
     DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    IF v_counter = 2 THEN
      EXIT;
    END IF;

    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

FOR LOOP

```
SET SERVEROUTPUT ON
BEGIN
  FOR v_counter IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
END;
/
```

REVERSE FOR LOOP

```
SET SERVEROUTPUT ON
BEGIN
```

```
  FOR v_counter IN REVERSE 1..5 LOOP
     DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
END;
/
```

<u>ERROR in FOR LOOP</u>

```
SET SERVEROUTPUT ON
BEGIN
  FOR v_counter IN 1..5 LOOP
     v_counter := v_counter + 1;
     DBMS_OUTPUT.PUT_LINE ('v_counter = '|| v_counter);
  END LOOP;
END;
/
```

<u>Scope of the FOR LOOP Variable</u>

```
SET SERVEROUTPUT ON
BEGIN
  FOR v_counter IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Counter outside the loop is '||
    v_counter);
END;
/
```

<u>Exiting FOR Loop with IF condition</u>

```
SET SERVEROUTPUT ON
BEGIN
  FOR v_counter IN 1..5 LOOP
     DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
     IF v_counter = 3 THEN
       EXIT;
     END IF;
  END LOOP;
END;
/
```

<u>Exiting FOR Loop with EXIT condition</u>

```
SET SERVEROUTPUT ON
BEGIN
```

```
  FOR v_counter IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    EXIT WHEN v_counter = 3;
  END LOOP;
END;
/
```

Nested FOR LOOP

```
SET SERVEROUTPUT ON
DECLARE
  v_counter1 INTEGER := 0;
  v_counter2 INTEGER;
BEGIN
  WHILE v_counter1 < 3 LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter1: '||v_counter1);
    v_counter2 := 0;
    LOOP
      DBMS_OUTPUT.PUT_LINE ('v_counter2: '||v_counter2);
      v_counter2 := v_counter2 + 1;
      EXIT WHEN v_counter2 >= 2;
    END LOOP;
    v_counter1 := v_counter1 + 1;
  END LOOP;
END;
/
```

Introduction to Cursors

Cursor is a pointer that points to the memory location in DB into which the records are fetched from DB during SELECT statement.

> Cursors are memory areas that allow you to allocate an area of memory and access the information retrieved from a SQL statement.

A cursor is a handle, or pointer, to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. Two important features about the cursor are as follows:

1. Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
2. A cursor is named so that it can be referenced.

```
SET SERVEROUTPUT ON
DECLARE
BEGIN
```

```
  SELECT * FROM employee;
END;
/
```

error: an INTO clause is expected in this SELECT statement

```
SET SERVEROUTPUT ON
DECLARE
  v_eid employee.eid%TYPE;
  v_fn employee.first_name%TYPE;
  v_ln employee.last_name%TYPE;
BEGIN
  SELECT eid, first_name, last_name INTO v_eid, v_fn, v_ln FROM employee;
END;
/
```

error: exact fetch returns more than requested number of rows

Implicit cursor

```
SET SERVEROUTPUT ON
DECLARE
  emp_rec employee%ROWTYPE;
BEGIN
   SELECT * INTO emp_rec
      FROM employee WHERE eid=1;
   DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END;
/
```

Explicit Cursor

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR emp_cur IS
                    SELECT * FROM employee;
  emp_rec emp_cur%ROWTYPE;
  -- emp_rec employee%ROWTYPE;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO emp_rec;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_rec.first_name||' '||emp_rec.last_name||' '||emp_rec.salary);
  END LOOP;
  CLOSE emp_cur;
END;
/

SET SERVEROUTPUT ON
DECLARE
   v_salary employee.salary%TYPE;
BEGIN
   SELECT salary
     INTO v_salary
     FROM employee
    WHERE eid = 1;
   IF SQL%ROWCOUNT = 1
   THEN
     DBMS_OUTPUT.PUT_LINE(v_salary ||' has a '||'eid of 1');
   ELSIF SQL%ROWCOUNT = 0
   THEN
      DBMS_OUTPUT.PUT_LINE('The eid 1 is not in the database');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Stop harassing me');
   END IF;
   EXCEPTION WHEN
              NO_DATA_FOUND THEN
          DBMS_OUTPUT.PUT_LINE('Stop harassing me');
END;
/

create table employee_log
       (description VARCHAR2(250));

SET SERVEROUTPUT ON
DECLARE
   CURSOR c_employee IS
      SELECT eid, last_name, first_name
        FROM employee
       WHERE eid < 5;
BEGIN
   FOR r_employee IN c_employee
   LOOP
      INSERT INTO employee_log
         VALUES(r_employee.last_name);
   END LOOP;
COMMIT;
END;
/
```

So far we wrote unnamed blocks / anonymous blocks which are used to execute logic instantly for a results. But if we want to write reusable PL/SQL blocks we need to write them as functions and procedures.

Procedure doesn't return anything. Function has a return type.

PROCEDURE with only IN parameter

```
SET SERVEROUTPUT ON
CREATE or REPLACE PROCEDURE emp_raise_salary (perc IN NUMBER) AS
CURSOR emp_cur IS SELECT * FROM employee;
emp_rec emp_cur%ROWTYPE;
emp_sal employee.salary%TYPE;
emp_eid employee.eid%TYPE;
emp_count NUMBER;
BEGIN
        OPEN emp_cur;
        LOOP
                FETCH emp_cur INTO emp_rec;
                EXIT WHEN emp_cur%NOTFOUND;
                emp_sal:=emp_rec.salary;
                emp_sal:=emp_sal+emp_sal*perc;
                emp_eid:=emp_rec.eid;
                UPDATE employee SET salary=emp_sal WHERE eid=emp_eid;
                emp_count:=emp_count+1;
                DBMS_OUTPUT.PUT_LINE(emp_eid||' employee salary is '||emp_sal);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE(emp_count||' number of employees updated ');
        COMMIT;
END;
/

SQL>
DECLARE
  perc NUMBER:=&perc;
BEGIN
  emp_raise_salary (perc);
  DBMS_OUTPUT.PUT_LINE('emp sals updated');
```

```
END;
/
```

SQL> cal emp_raise_salary(0.01);
Procedures cannot be called in SELECT command because if procedure name is used in SELECT clause it will expect some return value from the procedure which it cannot return.

PROCEDURE with IN and OUT parameters

```
SET SERVEROUTPUT ON
CREATE or REPLACE procedure emp_raise_salary_proc (perc IN NUMBER, count OUT NUMBER) AS
CURSOR emp_cur IS SELECT * FROM employee;
emp_rec emp_cur%ROWTYPE;
emp_sal employee.salary%TYPE;
emp_eid employee.eid%TYPE;
emp_count NUMBER;
BEGIN
        OPEN emp_cur;
        LOOP
                FETCH emp_cur INTO emp_rec;
                EXIT WHEN emp_cur%NOTFOUND;
                emp_sal:=emp_rec.salary;
                emp_sal:=emp_sal+emp_sal*perc;
                emp_eid:=emp_rec.eid;
                UPDATE employee SET salary=emp_sal WHERE eid=emp_eid;
                emp_count:=emp_count+1;
         count:=emp_count;
                -- count:=count+1; -- wrong
                DBMS_OUTPUT.PUT_LINE(emp_eid||' employee salary is '||emp_sal);
        END LOOP;
    DBMS_OUTPUT.PUT_LINE(emp_count||' number of employees updated ');
        COMMIT;
END;
/

show errors;

SQL>
DECLARE
  perc NUMBER:=&perc;
  count1 NUMBER;
BEGIN
  emp_raise_salary_proc (perc, count1);
  DBMS_OUTPUT.PUT_LINE(count1||' number of emp sals updated');
END;
/
```

Creating Function

```
CREATE OR REPLACE FUNCTION emp_raise_salary_func (perc IN NUMBER) RETURN NUMBER AS
CURSOR emp_cur IS SELECT * FROM employee;
emp_rec employee%ROWTYPE;
v_eid employee.eid%TYPE;
v_sal employee.salary%TYPE;
counter NUMBER;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO emp_rec;
    EXIT WHEN emp_cur%NOTFOUND;
    v_eid:=emp_rec.eid;
    v_sal:=emp_rec.salary;
    v_sal:=v_sal+v_sal*perc;
    UPDATE employee SET salary=v_sal WHERE eid=v_eid;
    counter:=counter+1;
    DBMS_OUTPUT.PUT_LINE(v_eid||' salary is '||v_sal);
  END LOOP;
  COMMIT;
  RETURN counter;
END;
/
```

Accessing FUNCTION in anonymous block

```
SET SERVEROUTPUT ON;
DECLARE
 counter NUMBER;
BEGIN
  counter:=emp_raise_salary_func(0.01);
  DBMS_OUTPUT.PUT_LINE(counter||' number of employee salaries raised');
END;
/
```

Creating Package with a Procedure and Function

```
CREATE OR REPLACE PACKAGE manage_employees
AS
```

```
PROCEDURE find_first_name (i_employee_id IN employee.eid%TYPE,
o_first_name OUT employee.first_name%TYPE,
o_last_name OUT employee.last_name%TYPE);

FUNCTION id_is_good(i_employee_id IN employee.eid%TYPE) RETURN BOOLEAN;

END manage_employees;
/

CREATE OR REPLACE PACKAGE BODY manage_employees
AS

 PROCEDURE find_first_name (
            i_employee_id IN employee.eid%TYPE,
o_first_name OUT employee.first_name%TYPE,
o_last_name OUT employee.last_name%TYPE)
IS
 v_employee_id employee.eid%TYPE;
 BEGIN
  SELECT first_name, last_name INTO o_first_name, o_last_name FROM employee WHERE
eid=i_employee_id;
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error in finding employee '||i_employee_id);
 END find_first_name;

FUNCTION id_is_good(i_employee_id IN employee.eid%TYPE)
RETURN BOOLEAN
IS
            v_id_cnt number;
            BEGIN
                        SELECT COUNT(*)
                                    INTO v_id_cnt
                                    FROM employee
                                    WHERE eid = i_employee_id;
                        RETURN 1 = v_id_cnt;
                        EXCEPTION
                                    WHEN OTHERS THEN
                                 RETURN FALSE;
END id_is_good;

END manage_employees;
```

```
/
```

Calling Function & Procedure of a package

```
DECLARE
  v_first_name employee.first_name%TYPE;
  v_last_name employee.last_name%TYPE;
  v_eid NUMBER:=&v_eid;
BEGIN
  IF manage_employees.id_is_good(v_eid)
  THEN
    manage_employees. find_first_name(v_eid, v_first_name, v_last_name);
  DBMS_OUTPUT.PUT_LINE('Emp ID. '||v_eid||' is '||v_last_name||', '||v_first_name);
ELSE
   DBMS_OUTPUT.PUT_LINE
   ('Employee ID: '||v_eid||' is not in the database.');
END IF;
END;
/
```

Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. **Triggers** are, in fact, written to be executed in response to any of the following events − A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).

```
CREATE TABLE student1 (student_id NUMBER PRIMARY KEY,
                        first_ame VARCHAR2(25),
                        last_ame VARCHAR2(25),
                        registration_date NUMBER,
                        created_by VARCHAR2(25),
                        created_date DATE,
                        modified_by VARCHAR2(25),
                        modified_date DATE);

CREATE SEQUENCE STUDENT_ID_SEQ
                START WITH 1
                INCREMENT BY 1
                NOCYCLE
                NOCACHE
                NOMAXVALUE;

CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON student1
```

```
FOR EACH ROW
DECLARE
  v_student_id STUDENT.STUDENT_ID%TYPE;
BEGIN
  SELECT STUDENT_ID_SEQ.NEXTVAL
    INTO v_student_id
    FROM dual;
  :NEW.student_id := v_student_id;
  :NEW.created_by := USER;
  :NEW.created_date := SYSDATE;
  :NEW.modified_by := USER;
  :NEW.modified_date := SYSDATE;
END;
/

INSERT INTO student1 (first_ame, last_ame) VALUES ('Anna', 'Hazare');
INSERT INTO student1 (first_ame, last_ame) VALUES ('Arvind', 'Kejriwal');
```

ଓଓଓ Good Day, Best of Luck ஐஐஐ