Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Day 7. RDBMS And SQL

Smita B Kumar

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka
A distributed streaming platform

Apache Software Foundation

APACHE Spark

Java

hadoop

RDBMS And SQL - Topics Overview

- RDMBS Concepts
- Type of SQL statements
- DDL statements
- DML statements
- DQL statements
- TCL statements

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Oracle – RDBMS

What is RDBMS?

- RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.
- **What is a table?**
  - The data in an RDBMS is stored in database objects which are called as **tables**. This table is basically a collection of related data entries and it consists of numerous columns and rows.
- **What is a field?**
  - Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.
  - A field is a column in a table that is designed to maintain specific information about every record in the table.
- **What is a Record or a Row?**
  - A record is also called as a row of data is each individual entry that exists in a table.
- **What is a column?**
  - A column is a vertical entity in a table that contains all information associated with a specific field in a table.

SQL Constraints

- Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
- Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.
- Following are some of the most commonly used constraints available in SQL –
  - **NOT NULL** Constraint – Ensures that a column cannot have a NULL value.
  - **DEFAULT** Constraint – Provides a default value for a column when none is specified.
  - **UNIQUE** Constraint – Ensures that all the values in a column are different.
  - **PRIMARY Key** – Uniquely identifies each row/record in a database table.
  - **FOREIGN Key** – Uniquely identifies a row/record in any another database table.
  - **CHECK Constraint** – The CHECK constraint ensures that all values in a column satisfy certain conditions.
  - **INDEX** – Used to create and retrieve data from the database very quickly.

Data Integrity

The following categories of data integrity exist with each RDBMS –

- **Entity Integrity –** There are no duplicate rows in a table.
- **Domain Integrity –** Enforces valid entries for a given column by restricting the type, the format, or the range of values.
- **Referential integrity –** Rows cannot be deleted, which are used by other records.
- **User-Defined Integrity –** Enforces some specific business rules that do not fall into entity, domain or referential integrity.

Database Normalization

- Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process −
  - Eliminating redundant data, for example, storing the same data in more than one table.
  - Ensuring data dependencies make sense.
- Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure.
- Normalization guidelines are divided into normal forms; think of a form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database structure, so that it complies with the rules of first normal form, then second normal form and finally the third normal form.
- It is your choice to take it further and go to the fourth normal form, fifth normal form and so on, but in general, the third normal form is more than enough.
  1. First Normal Form (1NF)
  2. Second Normal Form (2NF)
  3. Third Normal Form (3NF)

Database Normalization- First Normal Form (1NF)

Rules for First Normal Form- The first normal form expects you to follow a few simple rules while designing your database, and they are:

Rule 1: Single Valued Attributes

Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now.

Rule 2: Attribute Domain should not change

This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

For example: If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows.

Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.

If one or more columns have same name, then the DBMS system will be left confused.

Rule 4: Order doesn't matters

This rule says that the order in which you store the data in your table doesn't matter.

Database Normalization- Second Normal Form (2NF)

- For a table to be in the Second Normal Form, it must satisfy two conditions:
  1. The table should be in the First Normal Form.
  2. There should be no Partial Dependency.
- **What is Dependency?**
  - Let's take an example of a Student table with columns student_id, name, reg_no(registration number), branch and address(student's home address).
- In this table, student_id is the primary key and will be unique for every row, hence we can use student_id to fetch any row of data from this table
- Even for a case, where student names are same, if we know the student_id we can easily fetch the correct record. we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.
- I can ask from branch name of student with student_id **10**, and I can get it. Similarly, if I ask for name of student with student_id **10** or **11**, I will get it. So all I need is student_id and every other column **depends** on it, or can be fetched using it.
- This is **Dependency** and we also call it **Functional Dependency**.

Database Normalization- Second Normal Form (2NF)

- **What is Partial Dependency?**
  - Now that we know what dependency is, we are in a better state to understand what partial dependency is.
  - For a simple table like Student, a single column like student_id can uniquely identfy all the records in a table.
  - But this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.
  - Let's create another table for **Subject**, which will have subject_id and subject_name fields and subject_id will be the primary key.
  - Now we have a **Student** table with student information and another table **Subject** for storing subject information.
  - Let's create another table **Score**, to store the **marks** obtained by students in the respective subjects. We will also be saving **name of the teacher** who teaches that subject along with marks.

Database Normalization- Second Normal Form (2NF)

- For a table to be in the Second Normal form, it should be in the First Normal form and it should not have Partial Dependency.
- Partial Dependency exists, when for a composite primary key, any attribute in the table depends only on a part of the primary key and not on the complete primary key.
- To remove Partial dependency, we can divide the table, remove the attribute which is causing partial dependency, and move it to some other table where it fits in well.

Database Normalization- Third Normal Form (3NF)

- A table is in a third normal form when the following conditions are met –
  - It is in second normal form.
  - All nonprimary fields are dependent on the primary key.
  - The dependency of these non-primary fields is between the data.

Student Table

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat
12	Bkon	09-WY	IT	Rajasthan

Subject Table

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

Score Table

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11	1	80

Database Normalization- Third Normal Form (3NF)

- **What is Transitive Dependency?**
- With exam_name and total_marks added to our Score table, it saves more data now. Primary key for our Score table is a composite key, which means it's made up of two attributes or columns → **student_id + subject_id**.
- Our new column exam_name depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Prctical exams and for some you don't. So we can say that exam_name is dependent on both student_id and subject_id.
- And what about our second new column total_marks? Does it depend on our Score table's primary key?
- Well, the column total_marks depends on exam_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.
- But, exam_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total_marks depends on it.
- This is **Transitive Dependency**. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

Database Normalization- Third Normal Form (3NF)

How to remove Transitive Dependency?

Again the solution is very simple. Take out the columns exam_name and total_marks from Score table and put them in an **Exam** table and use the exam_id wherever required.

The new Exam table

exam_id	exam_name	total_marks
1	Workshop	200
2	Mains	70
3	Practicals	30

Advantage of removing Transitive Dependency

- Amount of data duplication is reduced.
- Data integrity achieved.

Score Table: In 3rd Normal Form

score_id	student_id	subject_id	marks	exam_id

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Oracle –SQL 10g

History of SQL

- SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems
- SQL statements are used to retrieve and update data in a database
- SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.
- The first version of SQL was developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s
- This version, initially called SEQUEL, was designed to manipulate and retrieve data

Features of SQL

- SQL stands for Structured Query Language
- Is a Non-procedural query language
- Sometimes referred to as a Database Gateway Language as ANSI (American National Standards Institute) made it a standard language for all DBMS
- Used for creating, managing, manipulating and querying the database objects such as tables, views etc.

SQL Commands

- SQL commands are divided into following categories:
  - Data Definition Language
    - CREATE, ALTER, DROP,TRUNCATE
  - Data Manipulation Language
    - INSERT, UPDATE, DELETE
  - Data Query Language
    - SELECT
  - Data Control Language
    - GRANT, REVOKE
  - Transaction control Language
    - COMMIT, ROLLBACK

Data Definition Language (DDL)

- DDL commands are used to
  - Create database objects such as tables, indexes, views etc. in the database
  - Alter/Modify the structure of existing database objects
  - Drop the existing database objects from the database
    - CREATE
    - ALTER
    - DROP
    - TRUNCATE
- DDL commands are implicitly committed.

Creating a Table

```
CREATE TABLE tablename
(columnname1 data_type (size) [constraints],
(columnname2  data_type (size) [constraints],
…)
```

Defining Database Tables

- To create a table, you must specify:
  - Table name
  - Column names
  - Column data types
  - Column sizes
  - Constraints: restrictions on the data values that a column can store

Names and Properties: Conventions

- Series of rules Oracle Corporation established for naming all database objects:
  1. From 1 to 30 characters
  2. Only alphanumeric characters, and special characters ($, _, #)
  3. Must begin with a letter and can not contain blank spaces or hyphens

- Are the following names valid? Why?
  1. customer order
  2. customer-order
  3. #order

Data Types

- Data type
  - Specifies kind of data that column stores
  - Provides means for error checking
  - Enables DBMS to use storage space more efficiently by internally storing different types of data in different ways
- Basic types
  - Character
  - Number
  - Date/time
  - Large object

Basic Built-in Data Types

- Character
  - VARCHAR2
  - CHAR
  - NVARCHAR2 / NCHAR
- Numeric
  - NUMBER
- DATE
- OTHERS
  - CLOB, BLOB, LONG, RAW, LONG RAW

Constraints

- A rule that restricts the values that can be inserted into a column
- A mechanism used to protect
  - the relationship between data within an Oracle table, or
  - the correspondence between data in two different tables

Types of Constraints

- Integrity constraints: define primary and foreign keys
- Value constraints: define specific data values or data ranges that must be inserted into columns and whether values must be unique or not NULL. CHECK, NOT NULL, UNIQUE constraints

---

- Table constraint: restricts the data value with respect to other values in the table
- Column constraint: limits the value that can be placed in a specific column, irrespective of values that exist in other table records

Integrity Constraints

- Define Primary Key columns
- Specify Foreign Keys and their corresponding table and column references
- Specify Composite Keys

Default Clause

- Default: specifies a default value that is inserted automatically

- Syntax:

```
CREATE TABLE <tablename>(column name datatype DEFAULT
<value>);
```

- Example:

```
CREATE TABLE employee
(empno NUMBER(6) PRIMARY KEY,
ename VARCHAR2(30) NOT NULL,
grade CHAR(3) DEFAULT 'TG0',
dob DATE, deptno number(2));
```

Primary Key Constraints

- Column-Level: the constraint will be given along with the column definition. This is required when only one column is to be made as a primary key
- Table-level: this type is used when a compound primary key is to be created i.e. super key

- Syntax:

```
CREATE TABLE <table name>
(<Colname> <datatype>[(<size>)] [ [CONSTRAINT constraint_name] PRIMARY
 KEY ],
 <colname> <datatype>[(<size>)],
  . . .);
```

Column Level Primary Key

```
CREATE TABLE employee
   (empno NUMBER(6) constraint emp_empno_pk PRIMARY KEY,
    ename VARCHAR2(30),
    grade_class CHAR(2),
    email varchar2(15),
    dob DATE);
```

Table Level Primary Key Constraint

```
CREATE TABLE employee
      (empno NUMBER(6),
      ename VARCHAR2(30),
       grade CHAR(2),
      dob DATE, deptno number(3)
      CONSTRAINT emp_id_pk PRIMARY KEY (deptno,empno));
```

- Table level constraints are used for compound keys.

Oracle Constraint Naming Convention

- tablename_ columnname_constraintID

Constraint Type	Constraint Type Identifier
PRIMARY KEY	P
FOREIGN KEY	R
CHECK CONDITION	C
NOT NULL	C
UNIQUE	U

Foreign Key Constraints

- Can only be defined after column is defined as a primary key in the parent table

- Syntax:
  - Column level

```
CREATE TABLE <tablename>(Column name datatype     CONSTRAINT
 constraint_name
  REFERENCES primary_key_table_name (column_name));
```

  - Table level

```
 CREATE TABLE table_name
    (column1 datatype null/not null, column2 datatype
null/not null, …,CONSTRAINT fk_column
  FOREIGN KEY (column1, column2, ... column_n)
  REFERENCES parent_table (column1, column2, ...   column_n)
   );
```

Foreign Key Constraints - Column Level

```
CREATE TABLE dept
(deptno number(2) constraint dept_deptno_pk PRIMARY KEY,
  dname varchar2(20) NOT NULL,
  loc varchar2(10));

CREATE TABLE employee
  (empno NUMBER(6) constraint emp_empno_pk PRIMARY KEY,
   ename VARCHAR2(30),
    grade CHAR(2),
    dob DATE,
   deptno number(2) REFERENCES dept(deptno)
 ON DELETE CASCADE);
```

- ON DELETE CASCADE option permits deletions of referenced key values in the parent table and automatically deletes dependent rows in the child table to maintain referential integrity.

Foreign Key Constraints - Table Level

```
CREATE TABLE dept
( deptno number(2) constraint dept_deptno_pk PRIMARY KEY,
  dname varchar2(20) NOT NULL,
  loc varchar2(10));

CREATE TABLE employee
  (empno NUMBER(6) constraint emp_empno_pk  PRIMARY KEY,
   ename VARCHAR2(30),
    grade CHAR(2),
    dob DATE,
   deptno number(2), FOREIGN KEY (deptno)      REFERENCES
  dept(deptno)
);
```

Types of Value Constraints

- Check condition: restricts to specific values
- Syntax:

```
CREATE TABLE <table name>(Column name datatype, CHECK (<condition>);
```

- Example: column level

```
CREATE TABLE employee
   (empno NUMBER(6) PRIMARY KEY
CHECK (empno > 2999), ename VARCHAR2(30),
    grade CHAR(2),dob DATE, deptno number(2) REFERENCES dept(deptno)
);
```

Restrictions on CHECK Constraint

A check constraint cannot refer to a column of another table. It also cannot refer to special keywords, such as user, sysdate, currval, nextval, rownum and rowid.

create table checktab1(c1 date check(c1>='01-Jan-1976'));
The above will work, but the following will give an error:

```
SQL> create table checktab1(c1 date check(c1>='01-Jan-76'));
create table checktab1(c1 date check(c1>='01-Jan-76'))
                                     *
ERROR at line 1:
ORA-02436: date or system variable wrongly specified in
CHECK constraint
```

Check Constraint Example – Table Level

```
CREATE TABLE employee
  (empno NUMBER(6) PRIMARY KEY,
   ename VARCHAR2(30),
   grade CHAR(2),
   dob DATE,
   deptno number(2) REFERENCES dept(deptno),
   CHECK (empno > 2999 OR  dob >= '01-JAN-1988'  ))
```

Types of Value Constraints

- Not NULL: specifies that a column cannot be empty

- Syntax:

```
CREATE TABLE <tablename>(column name datatype NOT NULL);
```

- Example:

```
CREATE TABLE employee
 (empno NUMBER(6) PRIMARY KEY,
  ename VARCHAR2(30) NOT NULL,
  grade CHAR(2),
   dob DATE, deptno number(2)  REFERENCES
dept(deptno),CHECK (empno > 2999));
```

Types of Value Constraints

- Unique: specifies that a non-primary key column must have a unique value

- Syntax:

```
CREATE TABLE <table name>(column name datatype CONSTRAINT
term_term_desc_uk UNIQUE (term_desc));
```

- Example:

```
CREATE TABLE employee
        (empno NUMBER(6) PRIMARY KEY,
         ename VARCHAR2(30) NOT NULL,
         grade CHAR(3) DEFAULT 'TG0',
  email varchar2(20) constraint u_em_email UNIQUE,
  dob DATE, deptno number(2)  REFERENCES
dept(deptno), CHECK (empno > 2999));
```

Create Table from Another Table

- Syntax:

```
CREATE TABLE <tablename>
  AS ( Select query);
```

Example:

```
CREATE TABLE empcopy as SELECT * FROM emp WHERE empno>1010;
```

This is a fast way of creating a table from another table or a way to create a copy of a table. The table created will have the columns which are mentioned in the SELECT query along with the data , which satisfies the WHERE clause of the SELECT query.

Modifying and Deleting Database Tables

- Modify existing database tables by
  - Changing the name of a table
  - Adding new columns
  - Deleting columns that are no longer needed
  - Changing the data type or maximum size of an existing column
- Unrestricted action: some specifications can always be modified
- Restricted action: specifications modified only in certain situations

Deleting and Truncating Tables

- To delete:

```
Drop table <tablename> [CASCADE CONSTRAINTS];
```

  - Use with caution

- To delete foreign key constraints, add “cascade constraints”

```
DROP TABLE emp;
```

- Truncate Table <tablename> ;
  - removes all rows from a table
  - high-speed data deletion statement

- You cannot roll back a TRUNCATE statement
- Example

```
TRUNCATE TABLE dept;
```

Renaming and Adding Columns

- To change the name of a table use RENAME
  - Syntax

    ```
    Rename old_tablename to new_tablename;
    ```
  - Example

    ```
    RENAME employee TO EMP;
    ```

- To add a column
  - Syntax

    ```
    ALTER TABLE tablename ADD(<columnname> data_declaration [constraints],
    <columnname> data_declaration [constraints]…);
    ```
  - Example

    ```
    ALTER TABLE emp  ADD (Hire_date DATE);
    ```

Modifying Existing Column Data Definitions

- Can only change data type to compatible data type (i.e. varchar2 to char)

```
ALTER tablename MODIFY(columnname new_data_declaration);
```

- Examples
- Changing the column width of grade from 2 to 4:

```
ALTER TABLE employee
MODIFY (grade CHAR(4));
```

```
ALTER TABLE employee RENAME COLUMN
email TO email_address;
```

Deleting a Column

- Can be used to remove a column from a table

```
ALTER TABLE tablename DROP COLUMN columnname;
```

- Examples

```
ALTER TABLE employee
DROP COLUMN grade;
```

Adding and Deleting Constraints

- Add a constraint Syntax

```
ALTER TABLE tablename ADD CONSTRAINT
constraint name constraint definition;
```

- Remove a constraint Syntax

```
ALTER TABLE tablename DROP CONSTRAINT
constraint name;
```

- Examples

```
ALTER TABLE Emp
ADD CONSTRAINT emp_ename_uk UNIQUE (ename);

ALTER TABLE emp
DROP CONSTRAINT emp_ename_uk;
```

Enabling and Disabling Constraints

- When modifying a database it can be useful to disable constraints
- Constraints are enabled by default
- To disable a constraint: `ALTER TABLE tablename DISABLE CONSTRAINT constraint_name;`
- To enable a constraint: `ALTER TABLE tablename ENABLE CONSTRAINT constraint_name;`

Examples

```
ALTER TABLE emp
DISABLE CONSTRAINT emp_ename_uk;

ALTER TABLE emp
ENABLE CONSTRAINT emp_ename_uk;
```

Sequence

- It is a database object from which users may generate unique integers
- It is used to automatically generate primary key values
- When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back
- Syntax

```
CREATE SEQUENCE sequence_name
  START WITH integer
   INCREMENT BY integer
   MAXVALUE integer
   MINVALUE integer
   CYCLE|NOCYLE
   CACHE|NOCACHE;
```

Sequence

- Example

```
CREATE SEQUENCE emp_seq
START WITH 10
INCREMENT BY 1
MAXVALUE  1000
NOCACHE
NOCYCLE;
```

DML - Data Manipulation Language

- DML commands work with the *content* of the tables
- Used to add, delete and modify the contents of the tables:
  - Insert
  - Update
  - Delete
- DML commands are not committed as soon as they are issued, an explicit commit is required
- Auto commit happens under following conditions:
  - Issue of a DDL command
  - Closing the session

DML Insert

- Examples:

```
INSERT INTO emp Values(101, 'SUNIL', 'CLERK', '7902', '01-
 JAN-08',10000,100,10);

INSERT INTO emp (empno,ename,job,deptno) Values
(102, 'RAJESH', 'SALESMAN',10);

INSERT INTO emp VALUES (&empno, &ename, &job);

INSERT INTO empcopy (SELECT * FROM emp);
```

DML Update

```
UPDATE emp SET ename = 'KUMAR' WHERE empno = 101;

UPDATE emp SET job = 'PRESIDENT', ename = 'SUNIL'
WHERE empno = 101;

UPDATE emp SET sal=(SELECT sal FROM emp WHERE empno=7788)
WHERE empno = 7656;
```

DML Delete

```
DELETE FROM emp;

DELETE FROM emp WHERE deptno = 20;

DELETE FROM emp WHERE empno = 101 AND ename LIKE 'S%';
```

TCL: Transaction Control Statements

- COMMIT: to explicitly save the changes in the database. Execution of DDL statements does commit the changes automatically.

```
SQL> COMMIT;
```

- SAVEPOINT <savepoint name>: is a transaction marker.

```
SQL> SAVEPOINT A;
```

- ROLLBACK [TO savepoint name]: to undo the changes till the last commit or till the last savepoint.

```
SQL> ROLLBACK;
SQL> ROLLBACK TO SAVEPOINT A;
```

Schema used for Queries

1. EMP
2. DEPT
3. SALGRADE

To see the structure of these tables in SQL*PLUS:

```
SQL> DESC emp
SQL> DESC dept
SQL> DESC salgrade
```

Data Query Language (DQL)

- *SELECT* is used for retrieving data from database
- Syntax: **`SELECT <col1>,<col2>… or <*> FROM <tablename>;`**
- Example: displaying the records in *Department* table

```
SELECT * FROM dept;
```

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

SELECT Query

- ALIAS is renaming default column name

```
Select <Column Name1> <Alias Name1>, <Column Name2> <Alias
Name2> From <Table Name>
```

- Example: List the Deptno & Dname from Department

```
SELECT deptno "Department Number", dname FROM Dept;
```

Department Number	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

WHERE Clause

- Specifies criteria
- Example: List all employees who belong to department number 20

```
SQL> SELECT empno, ename, deptno FROM emp
          WHERE deptno=20;
```

EMPNO	ENAME	DEPTNO
7369	SMITH	20
7566	JONES	20
7788	SCOTT	20
7876	ADAMS	20
7902	FORD	20

DISTINCT Clause

- Allows to remove duplicates from the result set
- Syntax: **`SELECT DISTINCT columns FROM tables`**
- Example: List the number of jobs in *emp* table, displaying jobs only once, even if duplicate values exist

```
SELECT DISTINCT job FROM emp;
```

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

ORDER BY clause

- Sorts records using specified criteria in ascending or descending order
- Syntax:

```
ORDER BY <col>|<expression> [ASC | DESC] [, expression [ASC | DESC] ...]
```

- Examples:

```
SELECT * FROM Emp
ORDER BY ename ASC;
```

SQL Operators

- Arithmetic
  + , - , * , /
- Relational
  = , < , > , <= , >=
  < >, != , ^=
- Boolean
  AND, OR, NOT
- Set Operators
  UNION, UNION ALL, INTERSECT, MINUS
- Others
  IN, BETWEEN
  LIKE , IS NULL

Relational Operator

Example: List the employee name whose employee number is 7900

```
SELECT ename FROM Emp
      WHERE empno = 7900;
```

```
ENAME
----------
JAMES
```

Relational Operators

Example: List the employees whose hire date is before 28-SEP-81

```
SELECT empno, ename, hiredate, deptno
     FROM emp WHERE hiredate < = '28-SEP-81';
```

EMPNO	ENAME	HIREDATE	DEPTNO
7369	SMITH	17-DEC-80	20
7499	ALLEN	20-FEB-81	30
7521	WARD	22-FEB-81	30
7566	JONES	02-APR-81	20
7654	MARTIN	28-SEP-81	30
7698	BLAKE	01-MAY-81	30
7782	CLARK	09-JUN-81	10
7844	TURNER	08-SEP-81	30

Logical Operators

- Example: List the employees who get salary in the range of 1500 and 3000

```
SELECT empno, ename, sal
    FROM emp WHERE sal >= 1500  AND  sal <= 3000;
```

- Example: List the employee number & names of department 10 & 20

```
SELECT empno, ename, sal
    FROM emp WHERE deptno=10 OR deptno=20;
```

Other Operators

```
SELECT empno, ename, deptno
  FROM emp WHERE sal BETWEEN 1500 AND 2500;

SELECT empno, ename, sal
  FROM emp WHERE deptno IN(10,20);

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

SELECT empno, ename,comm
  FROM emp WHERE comm IS NULL;
```

Recap

```
SELECT * from emp;

SELECT ename, empno FROM emp;

SELECT ename, deptno FROM emp WHERE deptno = 10;

SELECT ename, sal,job FROM emp
  WHERE job = 'MANAGER';

SELECT DISTINCT job FROM emp;

SELECT empno, ename FROM emp WHERE comm IS NULL;
```

- **NULL* is absence of information

ORDER BY Clause

```
SELECT empno,ename,sal FROM emp
  ORDER BY ename;

SELECT ename FROM emp
  ORDER BY ename DESC;

SELECT job,ename FROM emp
  ORDER BY job,ename;

SELECT job,ename FROM emp
  ORDER BY job, ename DESC;

SELECT ename,job FROM emp
  ORDER BY 1 DESC;
```

Single Row Functions

- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- May modify the data type
- Can be *nested*

Single Row Functions (Contd...)

- Numeric
- Character
- Date
- Conversion
- Aggregate Functions
- Regular Expressions

Single Row Functions: Numeric

- NVL: Converts NULL values to required values

```
SELECT sal + NVL(comm,0) FROM emp;
```

- ROUND (m,n): Rounds values to the specified decimal

```
SELECT ROUND(52.5) FROM dual; o/p: 53
```

- TRUNC (m,n): Truncates values to the specified decimal

```
SELECT TRUNC(56.223,1) FROM dual; o/p : 56.2
```

Single Row Functions: Character

- CONCAT (col1, col2)

```
SELECT CONCAT (ename,job) FROM emp;
```

- UPPER, LOWER, INITCAP

```
SELECT UPPER(ename), LOWER(ename),
INITCAP(ename)  FROM emp;
```

- TRIM

```
SELECT TRIM(ename) FROM emp;
```

- LTRIM, RTRIM

```
SELECT LTRIM(ename), RTRIM(ename) FROM emp;
```

Single Row Functions: Character (Contd...)

- LENGTH (STRING)

```
SELECT LENGTH (ename) FROM emp;
```

- INSTR(STRING, 'search strng', start position)

```
SELECT INSTR(ename, 'S',1) FROM emp;
```

- SUBSTR(STRING, 'start pos', 'no. of chs');

```
SELECT SUBSTR(ename,1,3) FROM emp;
```

Single Row Functions: Date

- The *SYSDATE* function returns the current date & time as a DATETIME value
- TO_CHAR: Converts date into a required character format

```
SELECT TO_CHAR(hiredate, 'DD-MON-YY') FROM EMP;
```

- ADD_MONTHS: Adds months to a date

```
SELECT ADD_MONTHS(hiredate,11) FROM emp;
```

- MONTHS_BETWEEN: Finds the number of months between two dates

```
SELECT MONTHS_BETWEEN(sysdate,hiredate) FROM emp;
```

Conversion Functions

- TO_CHAR(X): Converts the value of X to a *character* or a *date* to a *character*

```
SELECT TO_CHAR(1981) FROM dual;
```

- TO_NUMBER(X): Converts a *nonnumeric* value X to a *number*

```
SELECT TO_NUMBER('1221') FROM dual;
```

- TO_DATE(X,[Y]): Converts a *non-date* value X to a *date* using the format specified by Y

```
SELECT TO_DATE('12-FEB-2007') FROM dual;
```

Aggregate Functions

- SUM: Returns the sum of the column values provided

```
SELECT SUM(sal) "Total Sal" FROM emp WHERE deptno = 20;
```

- AVG(n): Returns average value of n

```
SELECT AVG(Sal) "Average" FROM Emp ;
```

- COUNT: Returns the number of rows for the specified column
  - count(*) – Slow `SELECT count(*) "Tot_row" FROM Emp;`
  - count(empno) – Fast `SELECT count(ename) "Tot_row" FROM Emp;`

- MIN: Returns the minimum value of an expression

```
SELECT MIN(Sal) "Minimum" FROM emp;
```

- MAX: Returns the maximum value of an expression

```
SELECT MAX(Sal) "Maximum" FROM emp;
```

- *Aggregate* functions ignore *NULL* values by default

GROUP BY … HAVING Clause

- Syntax:

```
SELECT column1, column2, ... column_n, aggregate_function
      (expression)  FROM tables  WHERE predicates
          GROUP BY column1, column2, ... column_n
                HAVING condition1 ... condition_n;
```

- GROUP BY in a SELECT statement collects data from across multiple records & groups the results by one or more columns

- HAVING is very similar to WHERE, except that the statements within it are of an aggregate nature

```
SELECT deptno, count(*)  FROM emp  GROUP BY deptno;
```

```
SELECT deptno, count(*)  FROM emp  GROUP BY deptno
                                    HAVING COUNT(*) > 2;
```

Miscellaneous Functions

- Decode(): Facilitates conditional inquiries by doing the work of a *CASE* or *IF-THEN-ELSE* statement
- Syntax:

```
DECODE(col|expression, search1, result1
                    [, search2, result2,...,]
                    [, default])
```

  - *Decode()* compares <exp/col> with the search
  - If *exp_x* matches, it returns *result_x*
  - If not, it returns *default*, or, if *default* is left out, *NULL*

-
```
SQL> SELECT  empno, ename,
       DECODE(deptno,10,'Ten',20,'Twenty','Other')
         FROM emp;
```

-
```
SQL> SELECT empno, ename,
       DECODE(sign(sal-2000), 1, 'Above target',
         -1, 'below target', 'On Target')  FROM emp;
```

CASE Statement

- A *CASE* statement within an SQL statement has the functionality of an *IF-THEN-ELSE* statement

- Syntax:

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

CASE Statement (Contd...)

- Example:

```
SELECT empno, ename,
CASE
  WHEN deptno = 10 THEN 'NOIDA'
  WHEN deptno = 20 THEN 'PUNE'
  WHEN deptno = 30 THEN `MUMBAI'
ELSE 'BANGALORE'
END
FROM emp;
```

Joins

- EQUI
- NON-EQUI
- OUTER
- SELF

JOINS

- Used when we require data from more than one tables
- The columns compared by the '=' operator are called *join columns* and the join operation is called an *EQUI JOIN*
- ANSI / ISO Syntax for *Equi Join*:

```
SELECT ename, deptno, dname
FROM emp NATURAL JOIN dept;
```

- Oracle Syntax for *Equi Join*:

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

NON EQUI JOIN

- *Join* uses an unequal operation (<>, <, >, !=, BETWEEN) to match rows from the different tables

  Example:

- Oracle Syntax for *Non Equi Join*:

```
SELECT e.ename , e.sal,s.grade
FROM emp e , salgrade s
WHERE e.sal BETWEEN s.losal AND s.hisal;
```

- ANSI Syntax for *Non Equi Join*:

```
SELECT e.ename , e.sal, s.grade
FROM emp e INNER JOIN salgrade s
ON  e.sal BETWEEN s.losal AND s.hisal
```

SELF JOIN

- Done using table labels *(aliases)*, as if they were two separate tables
- Allows rows in a table to be joined to rows in the same table
- Example: List the employees working in the same department as that of employee 7900

```
SELECT b.ename from emp a, emp b
WHERE a.empno=7900 and a.deptno=b.deptno;
```

OUTER JOIN

- A row which does not satisfy a *join* condition, does not appear in the query result
- The missing row(s) are returned if an *outer join* operator is used in the join condition
- The operator is plus sign enclosed in parentheses (+)
- Placed on the side of the join(table) deficient in information
- ANSI / ISO Syntax for *Left Outer Join*:

```
SELECT e. ename, d.dname FROM emp e
LEFT OUTER JOIN dept d ON d.deptno=e.deptno;
```

- Oracle Syntax for *Left Outer Join*:

```
SELECT e. ename, d.dname FROM emp e, dept d
WHERE d.deptno(+)=e.deptno ORDER BY e.ename
```

OUTER JOIN

- ANSI / ISO Syntax for *Right Outer Join*:

```
SELECT e. ename, d.dname FROM
emp e RIGHT OUTER JOIN dept d ON d.deptno = e.deptno;
```

- Oracle Syntax for *Right Outer Join*:

```
SELECT e. ename, d.dname FROM emp e, dept d
WHERE d.deptno=e.deptno(+)
```

- Full Outer Join:

```
SELECT e. ename, d.dname
FROM emp e FULL OUTER JOIN dept d ON d.deptno = e.deptno;
```

Sub-queries

- A form of an SQL statement that appears inside another SQL statement, also termed as *Nested Query*
- The statement containing a sub-query is called a parent statement
- The parent statement uses the rows returned by the sub-query

Types of Sub-queries

- Single-row Sub-queries:

e.g.
```
SELECT ename FROM emp
        WHERE deptno = (SELECT deptno FROM dept
            WHERE dname = 'ACCOUNTING');
```

- Multi-row Sub-queries:

e.g.
```
SELECT ename FROM emp
        WHERE deptno IN (SELECT deptno FROM dept);
```

Types of Sub-queries using ANY & ALL

- ANY: The condition becomes *true*, if there exists at least one row selected by the sub-query for which the comparison holds
- Example:

```
SELECT * FROM emp WHERE sal >= ANY
  (SELECT sal FROM emp WHERE deptno = 30)
       AND deptno = 10;
```

- All: The condition becomes true, if all the rows are selected by the sub-query for which the comparison holds
- Example:

```
SELECT * FROM Emp WHERE sal >= ALL
   (SELECT sal FROM emp WHERE deptno = 30)
       AND deptno <> 30;
```

Multiple-column Sub-queries

- When we want to compare more than one columns in a sub-query

e.g.
```
SELECT ename, job FROM emp
      WHERE (deptno, sal) IN  (SELECT deptno,
max(sal)
        FROM emp GROUP BY deptno);
```

Inline Views

- A sub-query in the FROM clause of a SELECT statement
- Not a schema object
- Defines a data source that can be referenced in the main query

```
SELECT main_emp.ename, main_emp.sal, main_emp.deptno,
inline_emp. Maxsal
  FROM emp main_emp, (SELECT deptno, max(salary) maxsal
    FROM emp GROUP BY deptno) inline_emp
      WHERE main_emp.deptno = inline_emp.deptno AND
main_emp.sal < inline_emp.maxsal;
```

Application of Inline Views

- Top-n Analysis
  - Special queries for scenarios where it is asked to display only the *n top-most* or the *n bottommost* records from a table based on a condition
  - E.g. The top three earners in the company

- General Structure:

```
SELECT column1,column2….columnn,ROWNUM
 FROM (SELECT column1,column2….columnn FROM table
  ORDER BY Top-N_column) WHERE ROWNUM <= N;

SELECT ROWNUM as RANK, ename, sal
 FROM (SELECT ename,sal FROM emp ORDER BY sal DESC)
  WHERE ROWNUM <= 3;
```

Co-related Sub-queries

- A *co-related query* is a form of query used in Select, Update or Delete commands
- It forces the DBMS to evaluate the query once per row of the parent query rather than once for the entire query
- A *co-related query* answers questions whose answers depend on the values in each row of the parent query

Co-related Sub-queries (Contd…)

- EXISTS: This condition is considered "to be met" if the sub-query returns at least one row

- Examples:

```
SELECT deptno, dname FROM dept a
WHERE EXISTS (SELECT empno FROM emp e
                    WHERE a.deptno = e.deptno);

SELECT deptno, dname FROM dept a
WHERE NOT EXISTS (SELECT empno FROM emp e
                        WHERE a.deptno = e.deptno);
```

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

PL/SQL Basics

Need for PL/SQL

- SQL being a 4GL and non-procedural, can allow only operations related to data, like data design, data manipulation, data retrieval and data control. SQL has no provision for implementing application specific functionalities or business logic or procedural logic.
- Hence, a procedural language was introduced by Oracle to enable users to write application-specific SQL oriented programs.
- PLSQL(Procedural Language/Structured Query Language), extends SQL by adding constructs found in other procedural languages, such as:
  - Variables and other Identifier declarations
  - Control structures such as IF-THEN-ELSE statement and Loops
  - Procedures and Functions
  - Object types and Methods
- It combines the power and flexibility of SQL with the programming constructs available in 3GLs

Need for PL/SQL (Contd.)

- PL/SQL improves the performance when multiple SQL statements are to be executed.
  - Without PL/SQL, Oracle must process SQL statements one at a time.
  - Application or Front-end Programs that issue many SQL statements require multiple calls to the database, resulting in significant network and performance overhead.
  - With PL/SQL, an entire block of statements can be sent to Oracle at one time. This can drastically reduce network traffic between the database and an application.

PL/SQL Block

- PL/SQL is a block structured language
- It is composed of one or more blocks
- Types of Blocks
  - Anonymous Blocks: constructed dynamically and executed only once
  - Named Blocks: Subprograms, Triggers, etc.
    - Subprograms: are named PL/SQL blocks that are stored in the database and can be invoked explicitly as and when required; e.g. Stored Procedures, Stored Functions and Packages
    - Triggers: are named blocks that are also stored in the database; Invoked implicitly whenever the triggering event occurs; e.g. Database Triggers

PL/SQL Block Structure

- A PL/SQL block has the following structure

```
DECLARE (Optional Declaration Section)
        Variables, constants, types, cursors,
        user-defined exceptions, etc.
BEGIN (Mandatory Executable/Business Logic section)
    SQL and PL/SQL statements

EXCEPTION (Optional Exception-handling section)
    Trapping Errors occurred in executable  section
END;
/
```

Note: BEGIN & END are compulsory statements

PL/SQL Block Structure

- PL/SQL Block consists of three sections

  - Declarative: Contains declarations of variables, constants, cursors, user-defined exceptions and types (Optional)
  - Executable: Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block
  - Exception Handling: Specifies the actions to perform when errors and abnormal conditions arise in the executable section (Optional)

PL/SQL Data Types

- The PL/SQL data types are divided into
  - Scalar Types: used to hold a single value; e.g. DATE, VARCHAR2, NUMBER etc.
  - Composite Types: used to hold one or more items of the same type (collections) or dissimilar types(PL/SQL Records)

PL/SQL Variables

- Declaration Syntax

```
identifier [CONSTANT] datatype [NOT NULL] [:= expr | DEFAULT expr]
```

Note: Square brace indicates optional

- Valid variable declarations

```
DECLARE
      v_hiredate DATE;
      v_deptno NUMBER(2) NOT NULL := 10;
      v_location VARCHAR2(13) := 'Atlanta';
      c_comm CONSTANT NUMBER := 1400;
      v_NoOfSeats  NUMBER DEFAULT 45;
      v_FirstName VARCHAR2(20) DEFAULT 'SCOTT'
```

- Invalid variable Declarations

```
v_deptno number(2) NOT NULL;
v_name varchar2 DEFAULT 'Sachin';
```

- Inheriting data type

You can declare variables to inherit the data type of a database column or other variable

```
v_empno emp.empno%TYPE;
```

PL/SQL Variables (Contd.)

- Constants: using the CONSTANT keyword we can declare a constant

```
c_max_size CONSTANT NUMBER := 100;
```

- Bind Variable: a variable declared in a host environment and used in many blocks by referencing it with ":" prefix

- Row Type Variable: holds one record/row at a time

  e.g. `v_record emp%ROWTYPE;`

PL/SQL Variables (Contd.)

- PL/SQL Record type Variable: A record is a group of related data items stored in fields, each with its own name and data type
- A record containing a field for each item lets you treat the data as a logical unit
- The variable based on a PL/SQL record type is a composite data member having fields as defined in the corresponding record type
- Syntax:

```
TYPE type_name IS RECORD (field_declaration[,field_declaration]...);
     variable_name type_name;
```

- Example:

```
     -- Type declaration
    TYPE DeptRec IS RECORD (
            dept_id dept.deptno%TYPE,
            dept_name VARCHAR2(14),
            dept_loc VARCHAR2(13)
     );
       -- Record type variable declaration
      vDeptRec DeptRec;
```

Comments in PL/SQL

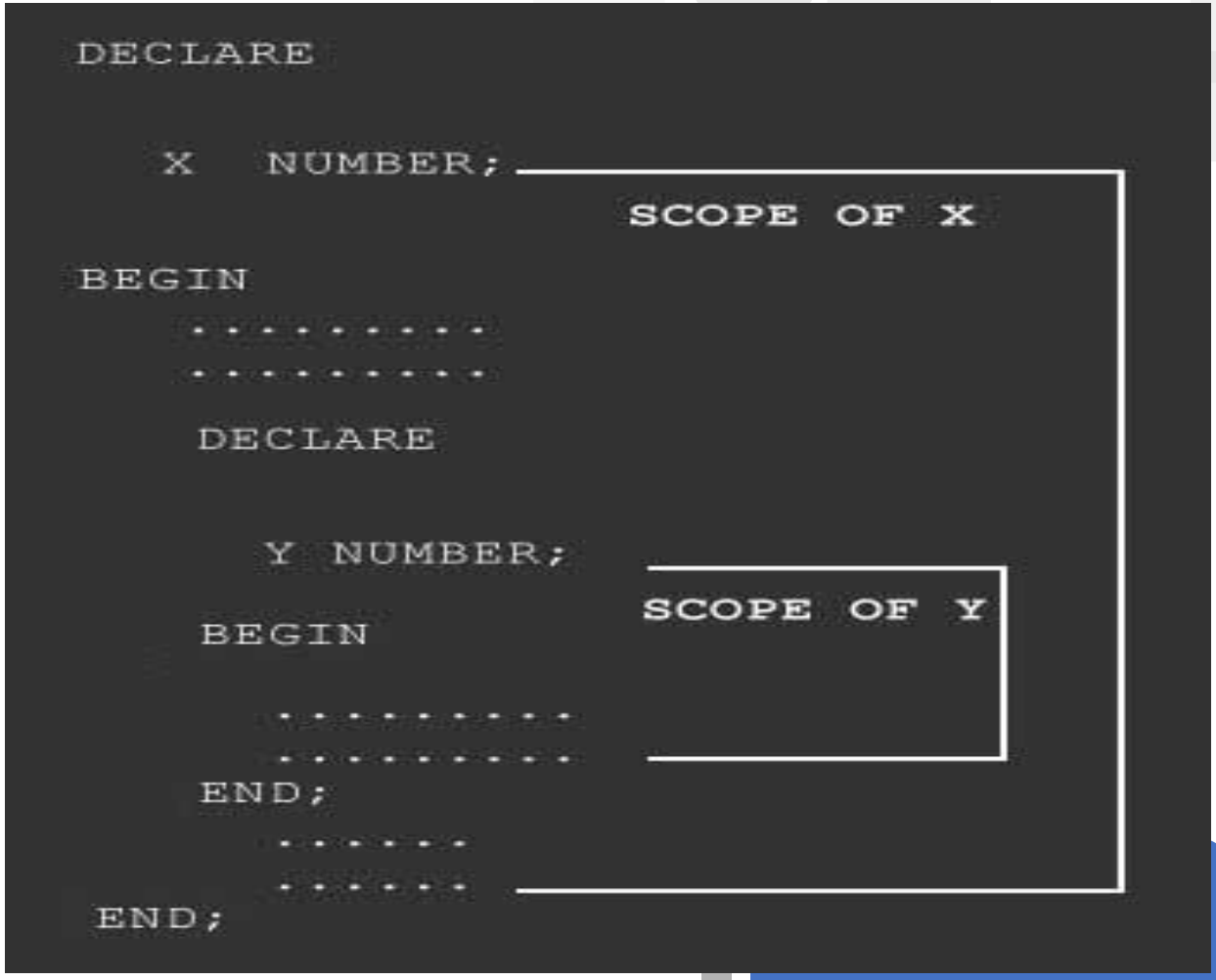
- To comment line or lines in a PL/SQL program use,
  - -- for single line comment
  - /* */ for multi-line comment

First PL/SQL Program

```
DECLARE
     X    NUMBER(3)  :=   10;
    Y    NUMBER(3)  :=   20;
BEGIN
     DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL Programming');
     DBMS_OUTPUT.PUT_LINE('The value of variable X is : ' ||  X);
     DBMS_OUTPUT.PUT_LINE('The value of variable Y is : ' ||  Y);
END;
/
```

Nested Blocks

- A block within a block called as a Nested Block



Example: Labeled Nested Block

```
-- Outer Block Starts here
<<l_outer>>   -- Label given to the outer block
DECLARE
        v_empname VARCHAR2(20) := 'Neeta';
        v_salary NUMBER(4) := 1000;
BEGIN
        DECLARE  -- Inner Block Starts Here
                v_empname VARCHAR2(30) :='Richa';
                v_address VARCHAR2(20) := 'Pune';
        BEGIN
                DBMS_OUTPUT.PUT_LINE(l_outer.v_empname);
                DBMS_OUTPUT.PUT_LINE (v_empname);
                DBMS_OUTPUT.PUT_LINE (v_salary);
        END; -- Inner block ends
        DBMS_OUTPUT.PUT_LINE (v_empname);
        DBMS_OUTPUT.PUT_LINE (v_salary);
END; -- Outer Block Ends
/
```

SQL Statements in PL/SQL

- Valid SQL statements inside PL/SQL Block
  - SELECT statement with INTO clause
  - All DML statements
  - Transaction statements like COMMIT & ROLLBACK
- SELECT statement inside PL/SQL Block
  - To use the SELECT statement, the INTO clause is mandatory
  - Select statements must return a single row
  - Returning no row or multiple rows, both, generate an error
  - Syntax: `SELECT column_list INTO variable/s FROM table_name WHERE condition;`

SELECT Statement inside PL/SQL Block

```
DECLARE
        v_empno emp.empno%TYPE := &empno;
        v_ename emp.ename%TYPE;
  BEGIN
         SELECT ename INTO v_ename
         FROM emp WHERE empno=v_empno;
         DBMS_OUTPUT.PUT_LINE(' The name is :  '||v_ename);
  END;
  /
```

%ROWTYPE Variable Example

```
DECLARE
      v_dno dept.deptno%TYPE := &dno;
      deptRec dept%ROWTYPE;
BEGIN
       SELECT * INTO deptRec
       FROM dept WHERE deptno=v_dno;
       DBMS_OUTPUT.PUT_LINE(deptRec.deptno);
       DBMS_OUTPUT.PUT_LINE(deptRec.dname);
       DBMS_OUTPUT.PUT_LINE(deptRec.loc);
END;
/
```

Bind Variable Example

```
VARIABLE result NUMBER



BEGIN
    SELECT (nvl(sal,0)*12)+(nvl(comm,0)*12) INTO :result FROM emp WHERE
  empno=7839;
   DBMS_OUTPUT.PUT_LINE('The total amount payable : '|| :result);
END;
/
-- To see the value set for the variable on SQL prompt use the SQL * plus
  command PRINT


PRINT result
```

The variable is declared at session level outside the block. To refer to this environment variable in the PL/SQL block we have to use colon sign

PL/SQL Record Type Variable

- %TYPE and %ROWTYPE work with single value and one complete record respectively
- How do we create our own composite data type, with our own specified number of values to hold?
  - Let us consider a table of about 20 columns
  - We need to work with only seven of those columns
  - If we use %ROWTYPE, we get all 20 values unnecessarily
  - If we use seven %TYPE declarations it will be bit clumsy
  - A better way to solve this problem is by defining our own data type, which can hold seven values

PL/SQL Record Type Variable Example

```
DECLARE
     TYPE myrec_type IS RECORD
     (eno emp.empno%TYPE,
      name emp.ename%TYPE,
           esal emp.sal%TYPE);
     emp_record myrec_type;
BEGIN
     SELECT empno,ename,sal INTO emp_record.eno,
      emp_record.name, emp_record.esal
       FROM emp WHERE empno=7839;
     DBMS_OUTPUT.PUT_LINE('Empno :'||emp_record.eno);
     DBMS_OUTPUT.PUT_LINE('Ename :'||emp_record.name);
     DBMS_OUTPUT.PUT_LINE('Salary :'||emp_record.esal);
END;
```

Manipulating Data in PL/SQL

- To manipulate data in the database use DML statements INSERT, UPDATE and DELETE in PL/SQL

- INSERT

```
DECLARE
      v_empno     emp.empno%TYPE :=   &empno;
      v_ename      emp.ename%TYPE :=   '&ename';
      v_salary       emp.sal%TYPE  :=   &sal;
BEGIN
      INSERT  INTO  emp(empno,ename,sal)
      VALUES(v_empno,v_ename,v_salary);
      COMMIT;
END;
```

Manipulating Data in PL/SQL

- UPDATE

```
DECLARE
  v_empno     emp.empno%TYPE;
  v_salary    emp.sal%TYPE :=&sal;

BEGIN
  UPDATE emp SET sal=v_salary where empno=1234;
   COMMIT;
END;
```

Manipulating Data in PL/SQL

- DELETE

```
DECLARE
v_empno emp.empno%TYPE :=&eno;
BEGIN
  DELETE emp WHERE empno=v_empno;
  COMMIT;
END;
```

PL/SQL Control Structures

- PL/SQL, like other 3GL has a variety of control structures which include
  - Conditional statements
  - Loops
- Conditional Statements

  There are three forms of IF statements
  - IF-THEN-END IF;
  - IF-THEN-ELSE-END IF;
  - IF-THEN-ELSIF-END IF;

Example: IF..ELSE..END IF

```
-- Block to demonstrate IF...ELSE...END IF
DECLARE
  vname emp.ename%TYPE;
  veno emp.empno%TYPE := &Emp_Num;
  vsal emp.sal%TYPE;
BEGIN
  SELECT ename, sal INTO vname, vsal FROM emp WHERE empno = veno;
                --Displays an appropriate message if salary is greater than 1500
  IF vsal > 1500 THEN
        DBMS_OUTPUT.PUT_LINE (vname|| ' earns a salary greater than 1500');
       --Else it shows the employee name whose sal is <1500
  ELSE
     DBMS_OUTPUT.PUT_LINE (vname|| ' earns a salary not greater than 1500');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('This line executes irrespective of the condition');
END;
/
```

Example: Nested IF

```
DECLARE
    firstNo NUMBER(5) := &firstNo;
    secondNo NUMBER(5) := &secondNo;
 BEGIN
   IF firstNo IS NULL OR secondNo IS NULL THEN
         DBMS_OUTPUT.PUT_LINE('Improper Input');
   ELSE
    IF firstNo = secondNo THEN
                DBMS_OUTPUT.PUT_LINE('The numbers are equal');
    ELSE
         IF firstNO > secondNo THEN
               DBMS_OUTPUT.PUT_LINE('The first no. is greater');
         ELSE
                DBMS_OUTPUT.PUT_LINE('The second no. is greater');
         END IF;
    END IF;
  END IF;
 END;
/
```

PL/SQL Control Structures

- LOOP Statements
  - Simple Loops
  - WHILE Loops
  - FOR Loops

LOOP Statements

- Simple Loops

```
LOOP
        Sequence_of_statements;
END LOOP;
```

Add EXIT statement to exit from the loop

- WHILE Loops

```
WHILE  condition
LOOP
        Statements;
END LOOP;
```

Condition is evaluated before each iteration of the loop

LOOP Statement: Example

```
DECLARE
        v_i NUMBER(2) := 1;
BEGIN
        LOOP
                DBMS_OUTPUT.PUT_LINE('Value : '|| v_i);
                EXIT WHEN v_i = 10;
                v_i:=v_i+1;
        END LOOP;
END;
/
```

WHILE Loop: Example

```
DECLARE
        v_i NUMBER(2) := 1;
BEGIN
        WHILE ( v_i <= 10 )
        LOOP
                DBMS_OUTPUT.PUT_LINE('Value : '|| v_i);
                v_i:=v_i+1;
        END LOOP;
END;
/
```

FOR Loops

- The number of iterations for simple loops and WHILE loops is not known in advance, it depends on the loop condition. Numeric FOR loops, on the other hand, have defined number of iterations.

```
FOR  counter  IN  [REVERSE]  low_bound  ..  high_bound
 LOOP
                  Statements;
 END LOOP;
```

  - Where:
    - counter: is an implicitly declared integer whose value automatically increases or decreases by 1 on each iteration
    - REVERSE: causes the counter to decrement from upper bound to lower bound
    - low_bound: specifies the lower bound for the range of counter values
    - high_bound: specifies the upper bound for the range of counter values

FOR Loop: Example

```
BEGIN
  FOR v_i IN 1..10
 /* The LOOP VARIABLE v_i of type BINARY_INTEGER is  declared automatically */
   LOOP
  DBMS_OUTPUT.PUT_LINE('Value : '|| v_i);
   END LOOP;
END;
/
```

For Loop with EXIT condition

```
DECLARE
        myNo NUMBER(5):= &myno;
        counter NUMBER(5):=1;
BEGIN
        FOR i IN 2..myNo-1
        LOOP
                counter:= counter+1;
                EXIT WHEN myNo mod i = 0;
        END LOOP;

        IF counter = myNo-1 THEN
                DBMS_OUTPUT.PUT_LINE( 'The given
                number is prime' );
        ELSE
                DBMS_OUTPUT.PUT_LINE('The given number is not
              a prime number' );
        END IF;
END;
/
```

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Q & A

Contact: smitakumar@synergetics-india.com

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You