Oracle SQL, PL/SQL

Module 1. Getting Started with Oracle

- Overview
  - Introduction to Databases
  - Introducing SQL
  - Main Components of Oracle
  - Starting SQL * Plus
  - Exiting SQL * Plus

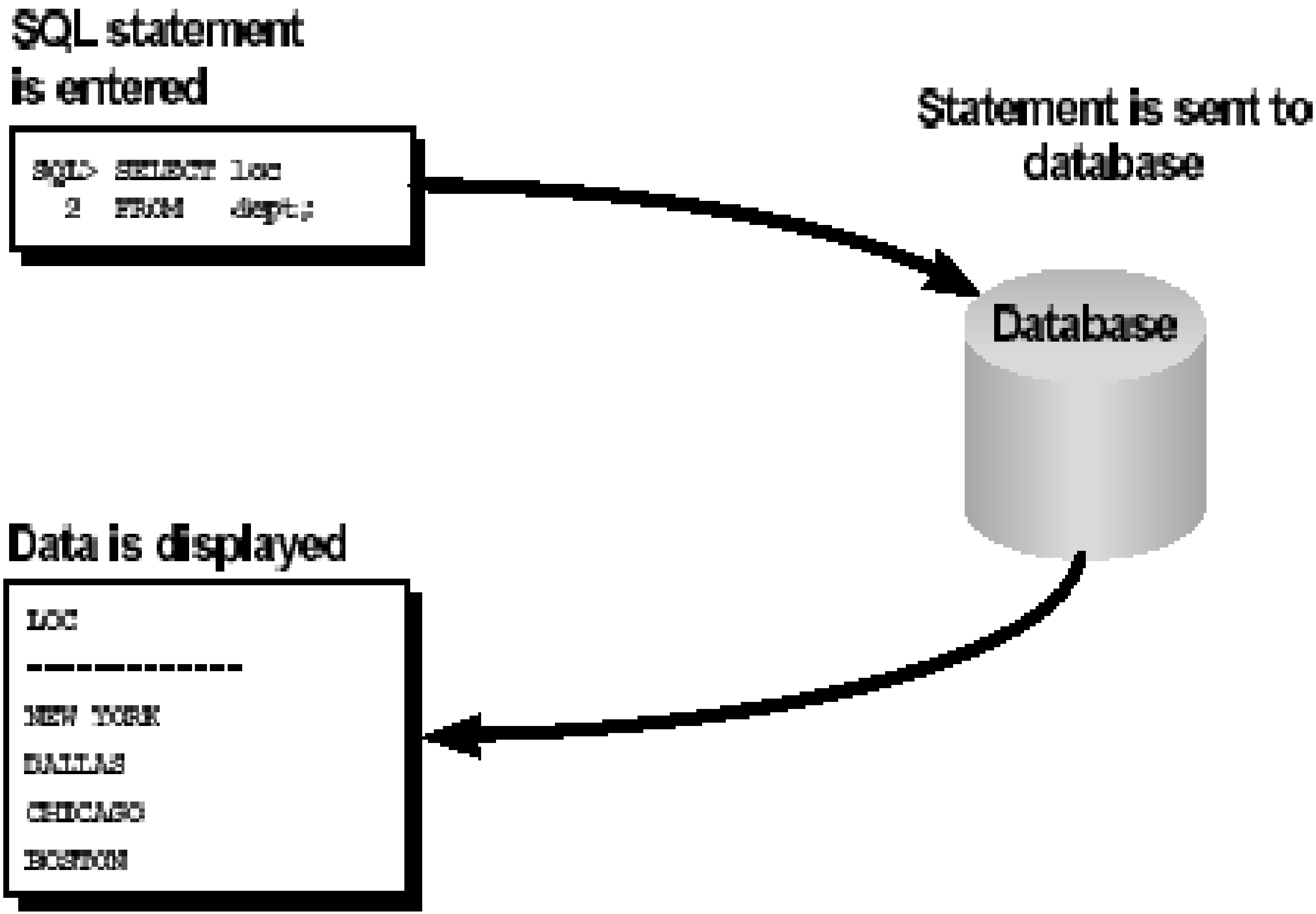
Introduction to Databases

Computerized record-keeping system.

EMPLOYEE

EMPNO	ENAME	JOB	MANAGER	HIREDATE	SALARY	COMMISSION	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-1980	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-1981	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-1981	1250	500	30
7566	JONES	MANAGER	7839	02-APR-1981	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-1981	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-1981	2850		30
7782	CLARK	MANAGER	7839	09-JUN-1981	2450	0	10
7788	SCOTT	ANALYST	7566	19-APR-1987	3000		20

Introducing SQL



Main Components of Oracle

SQL * Plus

Oracle SQL*Plus

File Edit Search Options Help

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Feb 26 11:37:12 2008

Copyright (c) 1982, 2005, Oracle.  All rights reserved.


Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> show user
USER is "SCOTT"
SQL> select * from tab;

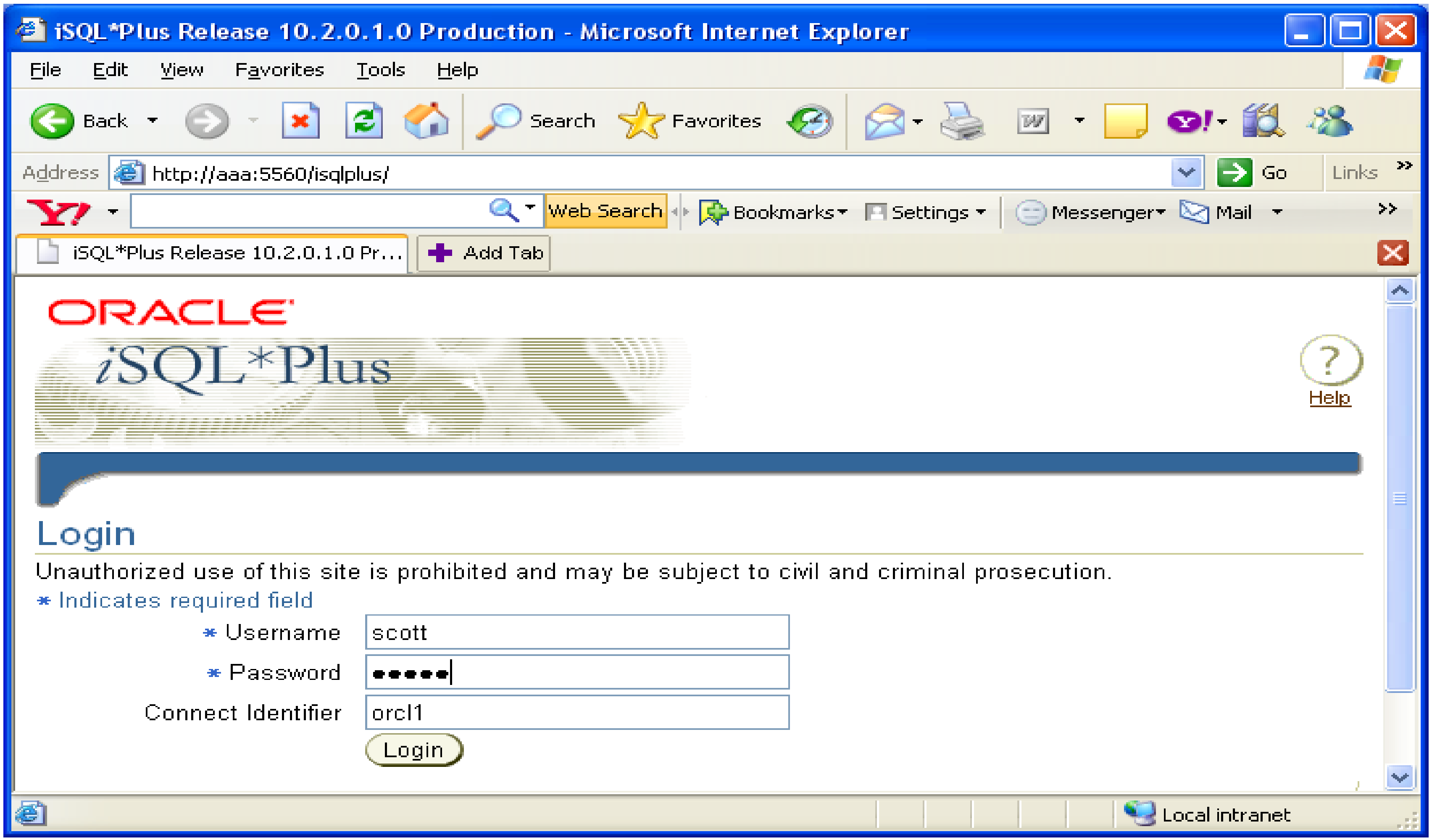
TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
DEPT                           TABLE
EMP                            TABLE
BONUS                          TABLE
SALGRADE                       TABLE
COPYSALGRADE                   TABLE
BIN$HBjqIyn7SAe3MxbPeAS/kw==$0 TABLE

6 rows selected.

SQL>
```

Main Components of Oracle

- iSQL * Plus



Main Components of Oracle

- PL/SQL

Oracle SQL*Plus

File Edit Search Options Help

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Feb 26 11:40:30 2008

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

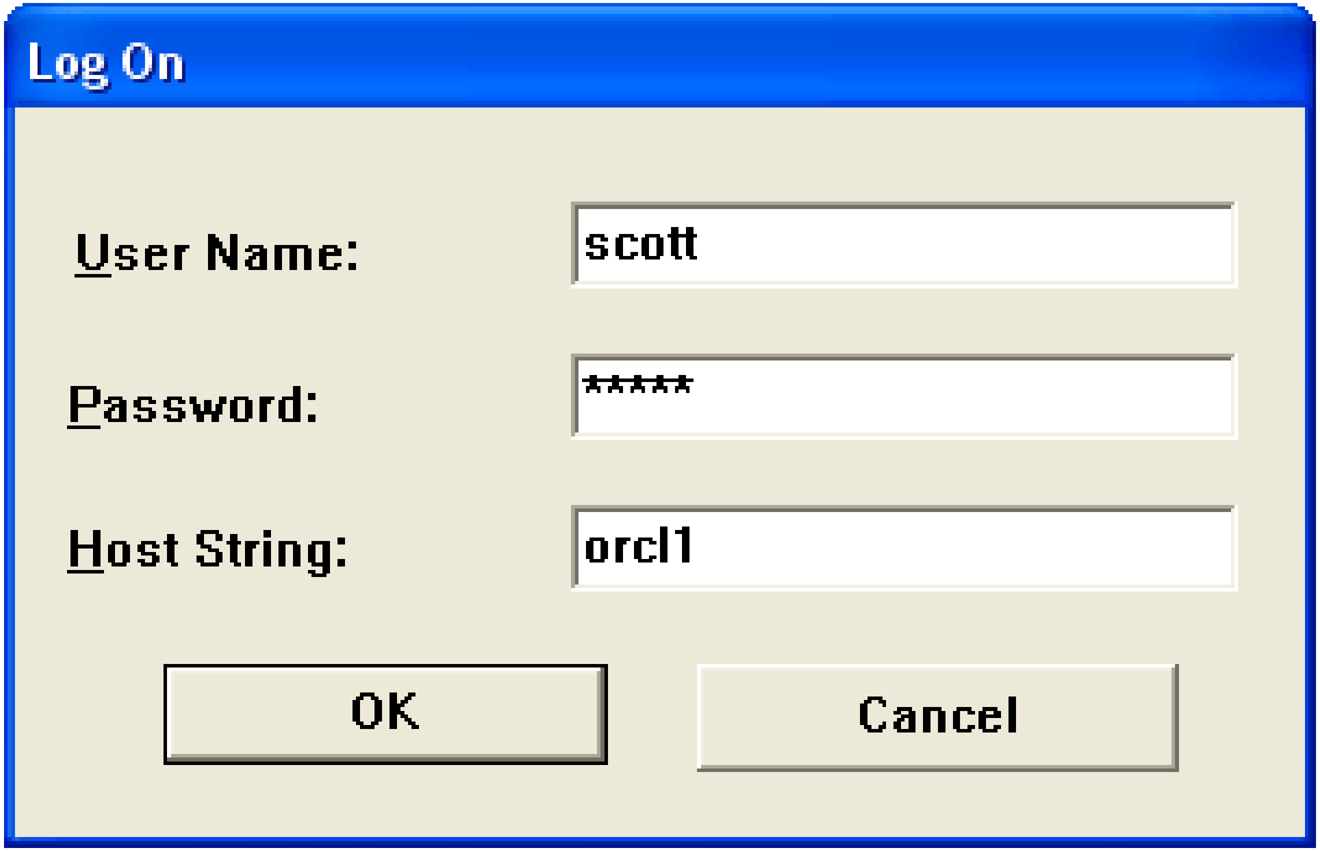

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> set serveroutput on
SQL> DECLARE
  2          nsal number(6);
  3       BEGIN
  4           select sal into nsal from emp where    empno=7900;
  5           DBMS_OUTPUT.PUT_LINE('Salary of    empno 7900 is :'||nsal);
  6       END;
  7  /
Salary of    empno 7900 is :950

PL/SQL procedure successfully completed.

SQL> |
```

Starting SQL*Plus



Starting SQL * Plus from Command Prompt

```
D:\oracle\product\10.2.0\db_1\bin\SQLPLUS.exe

SQL*Plus: Release 10.2.0.1.0 - Production on Tue Feb 26 13:26:14 2008

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

Enter user-name: SCOTT
Enter password:

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> SELECT * FROM EMP;

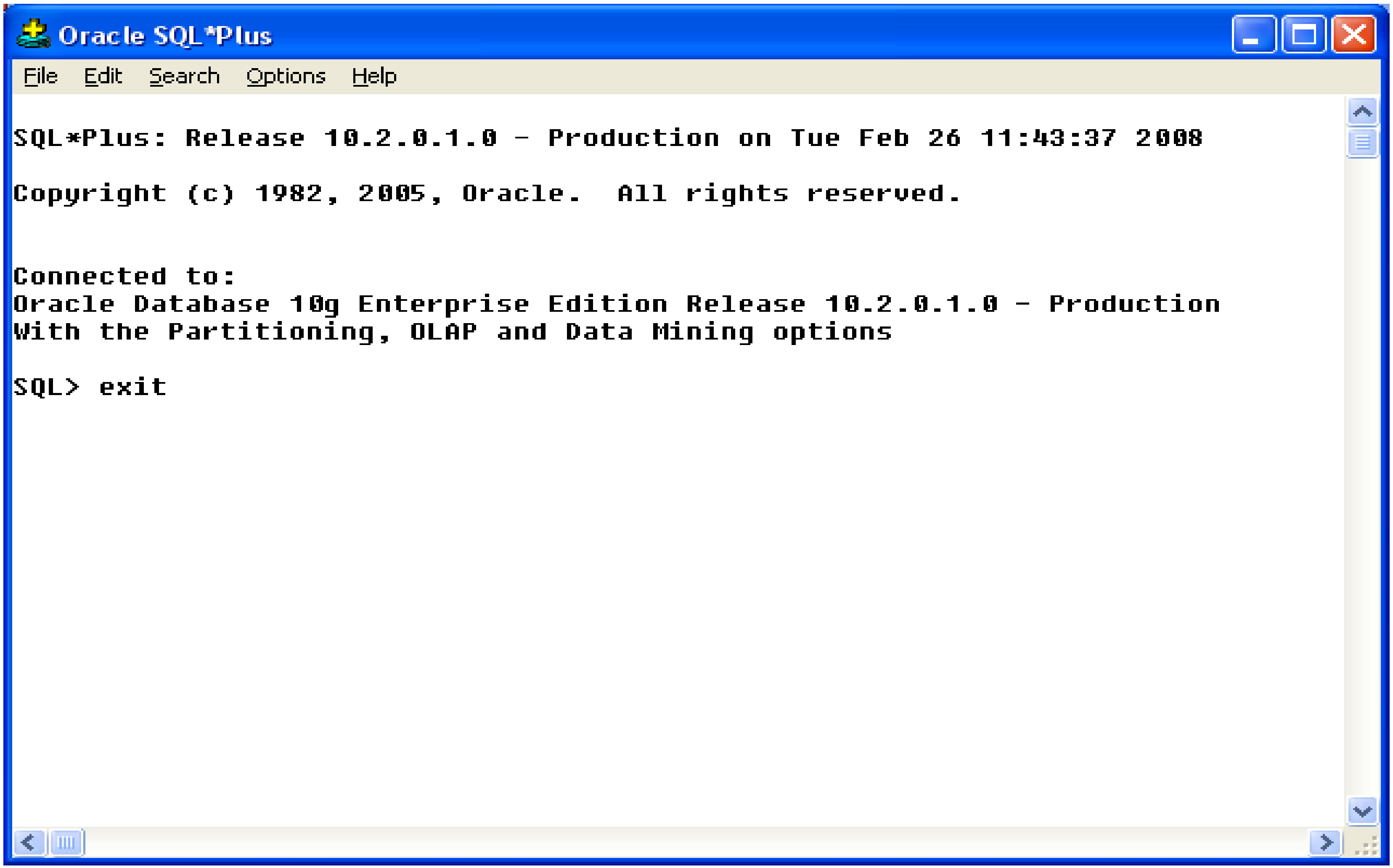
     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM
---------- ---------- --------- ---------- --------- ---------- ----------
    DEPTNO
----------
       111 AA         CLERK                                1240


      7369 SMITH      CLERK           7902 17-DEC-80        800
        20

      7499 ALLEN      SALESMAN        7698 20-FEB-81       1600        300
```

Exiting SQL*Plus

SQL> EXIT



Module 2. Editing SQL Commands

- Overview
  - ➢ Entering SQL commands
  - ➢ Editing SQL commands
  - ➢ Managing SQL files

Entering and Editing SQL Commands

- Terminating SQL statement:
  - a semicolon at the end of a line,
  - a semicolon on a line by itself,
  - a slash ("/") on a line by itself.

- Viewing contents of SQL buffer:
  - SQL> prompt:
  - SQL> list;

- Viewing specific line:
  - SQL> 2

Managing SQL files

SQL> get training.sql

SQL> @ training.sql

SQL> run training.sql

SQL> clear buffer

SQL> spool C:\training

SQL> spool off

SQL> spool out

Module 3. Data Retrieval & Ordering Output

- Overview
  - Simple Data Retrieval
  - Describing Table Structure
  - Conditional Retrieval using Arithmetic, Relational, Logical and Special Operators
  - The ORDER BY clause.
  - Aggregate functions
  - The GROUP BY and HAVING clause

Data Retrieval

```
SQL> SELECT * FROM tab;

SQL> DESC dept
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 DEPTNO                                             NUMBER(2)
 DNAME                                              VARCHAR2(14)
 LOC                                                VARCHAR2(13)
 BUDGET                                             NUMBER
```

Data Retrieval

SQL> **SELECT * FROM** employee;

SQL> **SELECT** emp_code, emp_name **FROM** employee;

SQL> **SELECT** distinct dept_code **FROM** employee;

Conditional Retrieval

SQL> SELECT * FROM employee **WHERE** salary > 3500;

SQL> SELECT emp_code, emp_name FROM employee **WHERE** dept_code = 'MKTG';

Relational Operators

=	equal to
!=	not equal to
^=	not equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

SQL> SELECT * FROM employee WHERE salary **>** 3000;

SQL> SELECT emp_name FROM employee WHERE dept_code **!=** ‘MKTG’;

SQL> SELECT * FROM employee WHERE emp_name **=** ‘Vijay Gupta’;

Logical Operators

The **AND** Operator

SQL> SELECT * FROM employee
WHERE dept_code = 'MKTG' **AND** sex = 'F';

SQL> SELECT * FROM employee
WHERE salary >= 3000 **AND** salary <= 4000;

The **OR** Operator

SQL> SELECT * FROM employee
WHERE dept_code = 'MKTG' **OR** dept_code = 'FIN';

The **NOT** Operator

SQL> SELECT * FROM employee WHERE **NOT** dept_code =
'MKTG‘;

Special Operators

The **BETWEEN** operator

SQL> SELECT * FROM employee WHERE salary **BETWEEN** 3000 and 4000;

SQL> SELECT * FROM employee WHERE date_join **BETWEEN** '01-JAN-80' and '31-DEC-89‘

The **IN** operator

SQL> SELECT * FROM employee WHERE dept_code **IN** ('MKTG', 'FIN');

SQL> SELECT * FROM employee
WHERE dept_code **NOT IN** ('MKTG', 'FIN');

Special Operators

The **LIKE** operator

SQL> SELECT * FROM employee WHERE emp_name **LIKE** 'P%';

SQL> SELECT * FROM employee WHERE emp_name **LIKE** '%Gupta';

SQL> SELECT * FROM employee WHERE emp_name **LIKE** '%Gupta%';

SQL> SELECT * FROM employee WHERE grade LIKE '_1';

The **IS NULL** operator

SQL> SELECT * FROM employee WHERE reports_to **IS NULL**;

SQL> SELECT * FROM employee WHERE reports_to **IS NOT NULL**;

Arithmetic Operators

+ addition
- subtraction
* multiplication
/ division

```
SQL> SELECT  * FROM  product
     WHERE direct_sales + indirect_sales > target;

SQL> SELECT  prod_code, prod_name, direct_sales + indirect_sales
     FROM product;

SQL> SELECT  prod_code,
     direct_sales + indirect_sales "Total sales"
     FROM product;
```

Ordering the SELECT Query Output

Ordering on single column

SQL> SELECT * FROM employee **ORDER BY** emp_code;

SQL> SELECT * FROM employee WHERE sex = 'M' **ORDER BY** emp_name;

SQL> SELECT * FROM employee **ORDER BY** age DESC;

Ordering on multiple columns

SQL> SELECT * FROM employee **ORDER BY** dept_code, emp_name;

SQL> SELECT * FROM employee **ORDER BY** dept_code, age DESC;

Aggregate Functions

SQL> SELECT **COUNT** (*) FROM employee;

SQL> SELECT **SUM** (salary) FROM employee;

SQL> SELECT **AVG** (age) FROM employee;

SQL> SELECT **MAX** (salary) FROM employee;

SQL> SELECT **MIN** (salary) FROM employee;

The GROUP BY clause

```
SQL> SELECT  dept_code, sum (salary)
     FROM employee
     GROUP BY dept_code;
```

The HAVING Clause

```
SQL> SELECT  dept_code, sum (salary) FROM employee
      GROUP BY dept_code
      HAVING sum (salary) > 10000;

SQL>  SELECT  dept_code, sum (salary)
      FROM employee
      WHERE age > 30
      GROUP BY dept_code
      HAVING sum (salary) > 10000
      ORDER BY sum (salary) desc;
```

Module 4. Creating Tables

- Overview
  - Creating a Table
  - Data Types

Creating Tables

```
CREATE TABLE tablename (
            column-name data-type [other clauses]... );
```

```
SQL> CREATE TABLE dept (
            dept_code varchar2 (4),
            dept_name varchar2 (20) );
```

Creating Tables

```
SQL> CREATE TABLE dept (
                    dept_code varchar2 (4),
                    dept_name varchar2 (20) );

SQL>create table emp_priti(
empno number(4),
ename varchar2(20),
hiredate date default sysdate,
sal number(8,2),
deptno number(2))
```

Data Types

Data Type	Description	Example
char (n)	Fixed-length character data. Max 2000 bytes	dept_code char (4)
varchar (n)	Variable-length character data. Max 4000 bytes	dept_code varchar2 (4)
varchar2 (n)	Variable-length character data. Max size is 8000 bytes	dept_code varchar (4)
number (p, s)	Numeric data, ‘p’ is the total length and “s” is the number of decimal places.	reading number (5, 2). Maximum value: 99.99
Date	Date and time. Range is 01/01/4712 BC to 31/12/4712 AD.	date_join date

Data Types

Data Type	Description	Example
long	Variable-length character data. Max 2 GB	remarks long
raw (n)	Binary format data. Max size 2000 bytes	esc_seqraw (15)
Long raw	Same as raw, but maximum size is 2 GB.	picture long raw
BLOB	Stores binary large objects up to 4GB	
CLOB	Stores character large objects up to 4GB.	
BFILE	Enables access to binary file LOBs that are stored in the file system outside the Oracle database. Maximum file size up to 4GB.	

Module 5. Inserting, Modifying & Deleting Data

- Overview
  - ➢ Inserting Data into a Table
  - ➢ Inserting Data into a Table using Sub query
  - ➢ Modifying Data in a Table
  - ➢ Deleting Data from a Table

Inserting Data into a Table

SQL> desc dept;

Name	Null?	Type
DEPT_CODE	NOT NULL	VARCHAR2(4)
DEPT_NAME	NOT NULL	VARCHAR2(20)

```
INSERT INTO table-name VALUES (value1, value2, ...);
```

SQL> **INSERT INTO** dept **VALUES** ('MKTG', 'Marketing');

SQL> **INSERT INTO** dept **VALUES** ('FIN', 'Finance');

SQL> **INSERT INTO** dept **VALUES** ('TRNG', 'Training');

Inserting Data into Table

SQL> **INSERT INTO** employee (emp_code, age, emp_name)
VALUES(101, 33, 'Sunil');

SQL> **INSERT INTO** senior
SELECT * FROM employee WHERE age > 50;

To prompt the values

- insert into emp_priti values (&empno, '&ename','&hiredate',&sal,&deptno)
- insert into emp_priti values (1, ‘satyen’,default,10000,10)
- insert into emp_priti values (2,’krishna’,default,10000,null)
- insert into emp_priti (empno, ename, hiredate, sal, deptno)

values(2,'trupti','10-may-2010',20000,20)

You can change the order of the columns.

Modifying and Deleting Data

SQL> **UPDATE** employee **SET** salary = salary + 100;

SQL> **UPDATE** employee **SET** salary = salary + 200 WHERE sex = 'F';

SQL> **DELETE** FROM employee;

SQL> **DELETE** FROM employee WHERE dept_code = 'MKTG';

Module 6. Modifying Table Structure

- Overview
  - Altering Table structure
  - Dropping Column from a Table
  - Dropping a Table

Modifying a Table Structure

SQL> **ALTER** table employee
ADD (age number (2));

SQL> **ALTER** table employee
MODIFY (age number (3));

SQL> **ALTER** table employee **DROP** column sex;

SQL> **ALTER** table employee **DROP** (age,married);

Dropping a Table

```
DROP TABLE table-name;
```

SQL> **DROP** table dept;

Module 7. Integrity Constraints

- Overview
  - Understanding Table and Column Constraints
  - Creating, Modifying and Dropping Column level constraints
  - Creating, Modifying and Dropping Table level constraints
  - Adding Constraints to Columns of an existing table
  - Enabling and Disabling Constraints
  - Dropping Columns and Tables having constraints

Integrity Constraints

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

Column Constraints

```
SQL> CREATE TABLE employee (
        emp_code number (5) NOT NULL,
        emp_name varchar2 (25) NOT NULL,
        dept_code varchar2 (4) );
```

Column Constraints

```
SQL> CREATE TABLE employee (
          emp_code number (5)
          CONSTRAINT employee_uq UNIQUE,
          emp_name varchar2 (25)
          CONSTRAINT employee_null NOT NULL);

SQL> SELECT  constraint_name FROM USER_CONSTRAINTS
     WHERE table_name = ‘EMPLOYEE’;
```

The UNIQUE Constraint

```
SQL> CREATE TABLE supplier (
        supp_code number (4)
        CONSTRAINT  code_pk PRIMARY KEY,
        supp_name varchar2 (30)
        CONSTRAINT name_uq UNIQUE);
```

```
SQL> CREATE TABLE supplier (
        supp_code number (4)
        CONSTRAINT  code_pk PRIMARY KEY,
        supp_name varchar2 (30)
        CONSTRAINT  name_uq UNIQUE
        CONSTRAINT  name_null NOT NULL);
```

The CHECK Constraint

```
SQL> CREATE TABLE employee (
        emp_code number (5) PRIMARY KEY,
        emp_name varchar2 (25) NOT NULL,
        dept_code varchar2 (4) CONSTRAINT code_check
        CHECK (dept_code = upper (dept_code) ));
```

The PRIMARY KEY Constraint

```
SQL> CREATE TABLE employee (
        emp_code number (5)
        CONSTRAINT  code_pk PRIMARY KEY,
        emp_name varchar2 (25)
        CONSTRAINT  name_null NOT NULL,
        dept_code varchar2 (4));
```

The REFERENCES Constraint

```
SQL> CREATE TABLE employee (
        emp_code number (5)
        CONSTRAINT  code_pk PRIMARY KEY,
        emp_name varchar2 (25)
        CONSTRAINT  name_null NOT NULL,
        dept_code varchar2 (4)
        CONSTRAINT  code_ref
        REFERENCES  dept (dept_code));
```

The REFERENCES Constraint

SQL> CREATE TABLE employee (
emp_code number (5) primary key,
emp_name varchar2 (25) not null,
dept_code varchar2 (4) **CONSTRAINT** code_ref
REFERENCES dept(dept_code)
ON DELETE CASCADE);

The REFERENCES Constraint

```
SQL> CREATE TABLE employee (
        emp_code number (5) primary key,
        emp_name varchar2 (25) not null,
        dept_code varchar2 (4) CONSTRAINT code_ref
        REFERENCES dept(dept_code)
        ON DELETE CASCADE);
```

Table Constraints

```
SQL> CREATE TABLE employee(
            emp_code number(4)    not null,
            emp_name varchar2(40)not null,
            dept_code varchar2(4),
            CONSTRAINT emp_uq UNIQUE (emp_code,emp_name)
     );
```

The PRIMARY KEY and CHECK Constraint

```
SQL> CREATE TABLE orders (
        order_year number (4),
        order_number number (5),
        order_date date,
        CONSTRAINT order_pk PRIMARY KEY (order_year,
            order_number));

SQL> CREATE TABLE employee (emp_code number (4),
        emp_name varchar2 (20),date_birth date,date_join date,
        CONSTRAINT employee_dates CHECK (date_join >
            date_birth));
```

The FOREIGN KEY Constraint

```
SQL> CREATE TABLE ship (
        ship_code varchar2 (5),
        ship_name varchar2 (20) not null,
        CONSTRAINT ship_pk PRIMARY KEY (ship_code) );

SQL> CREATE TABLE voyage (
        ship_code varchar2 (5),
        voyage_number number (3),
        date_arrival date,
        CONSTRAINT voyage_pk PRIMARY KEY
        (ship_code,voyage_number),
        CONSTRAINT voyage_fk FOREIGN KEY (ship_code)
        REFERENCES ship (ship_code) );
```

The FOREIGN KEY Constraint

SQL> CREATE TABLE docket (
docket_number number (5),
docket_date date,
ship_code varchar2 (5),
voyage_number number (3),
CONSTRAINT docket_pk PRIMARY KEY (docket_number),
CONSTRAINT docket_fk **FOREIGN KEY** (ship_code, voyage_number)
REFERENCES voyage (ship_code, voyage_number)
ON DELETE CASCADE);

Adding Constraints to Columns of an existing Table

```
SQL> ALTER TABLE employee MODIFY
        (date_join constraint emp_dt_join not null);

SQL> ALTER TABLE dept
        ADD CONSTRAINT cd_pk PRIMARY KEY (dept_code);

SQL> ALTER TABLE employee ADD
        CONSTRAINT cd_fk FOREIGN KEY(dept_code)
        REFERENCES dept (dept_code);
```

Adding Constraints to Columns of an existing Table

SQL> **ALTER TABLE** employee **ADD**
CONSTRAINT emp_dates CHECK (date_join > date_birth);

Enabling and Disabling Constraints

```
SQL> alter table dept
       DISABLE CONSTRAINT cd_pk;

SQL> alter table dept
       ENABLE CONSTRAINT cd_pk;

SQL> alter table dept
       DISABLE CONSTRAINT cd_pk CASCADE CONSTRAINTS;
```

Dropping a Constraint

```
SQL> alter table employee
        DROP CONSTRAINT emp_date;

SQL> alter table employee
        DROP CONSTRAINT emp_date CASCADE;
```

Dropping a Constraint

```
SQL> alter table dept
        DROP COLUMN dept_code CASCADE CONSTRAINTS;

SQL> drop table dept;
```

Module 8. Built-In Functions

- Overview
  - Numeric functions
  - Character functions
  - Date functions
  - Special formats with Date data types
  - Conversion functions

Functions on Numeric data types

Function	Returns	Example	Result
ceil (n)	Nearest whole integer greater than or equal to n.	SELECT ceil (9.86) FROM dual;	10
floor (n)	Largest integer equal to or less than n.	SELECT floor (9.86) FROM dual;	9
mod (m, n)	Remainder of m divided by n. If n = 0, then m is returned.	SELECT mod (11, 4) FROM dual;	3
power (m, n)	Number m raised to the power of n.	SELECT power (5, 2) FROM dual;	25
round (n, m)	Number n rounded off to m decimal places.	SELECT round (9.86, 1) FROM dual;	9.9
sign (n)	If n = 0, returns 0. If n > 0, returns 1. If n < 0, returns -1.	SELECT sign (9.86) FROM dual;	1
sqrt (n)	Square root of n.	SELECT sqrt (25) FROM dual;	5

Functions on Character data type

Function	Returns	Example	Result
initcap (x)	Changes the first character of each word to capital letters.	SELECT initcap ('inder kumar gujral') FROM dual;	Inder Kumar Gujral
lower (x)	Converts the entire string to lowercase.	SELECT lower ('Inder Kumar Gujral') FROM dual;	inder kumar gujral
upper (x)	Converts the entire string to uppercase.	SELECT upper ('Inder Kumar Gujral') FROM dual;	INDER KUMAR GUJRAL
replace (char, str1, str2)	Every occurrence of str1 in char is replaced with str2.	SELECT replace(‘Cap' , 'C', 'M') FROM dual;	Map
soundex (x)	Every word that has a similar phonetic sound, even if it is spelled differently.	SELECT emp_name FROM employee WHERE soundex (emp_name) = soundex ('Sivananda')	Shivanand Joshi
substr (char, m, n)	Part of char, starting FROM position m and taking characters.	SELECT substr ('Computer', 1, 4) FROM dual;	Comp
length (char)	Length of char.	SELECT length ('Oracle') FROM dual;	6

Functions on Date data types

Function	**Returns**	**Example**	**Result**
sysdate	Current date and time.	SELECT sysdate FROM dual;	25-NOV-97
last_day (date)	Last day of the month for the given date.	SELECT last_day (sysdate) FROM dual;	30-NOV-97
add_months (date, n)	Adds n months to the given date.	SELECT add_months (sysdate, 2) FROM dual;	25-JAN-98
months_between (date1, date2)	Difference in months between date1 and date2.	SELECT months_between (sysdate, '01-JAN-99') FROM dual;	-13.20232
next_day (date, day)	Date is the specified day of the week after the given date.	SELECT next_day (sysdate, 'sunday') FROM dual;	30-NOV-97

Formats with Date data types

Format	Returns	Example	Result
Y	Last digit of the year.	SELECT to_char (sysdate, 'Y') FROM dual;	7
YY	Last 2 digits of the year.	SELECT to_char (sysdate, 'YY') FROM dual;	97
YYY	Last 3 digits of the year	SELECT to_char (sysdate, 'YYY') FROM dual;	997
YYYY	All 4 digits of the year	SELECT to_char (sysdate, 'YYYY') FROM dual;	1997
year	Year spelled out.	SELECT to_char (sysdate, 'year') FROM dual;	Nineteen ninety-seven
Q	Quarter of the year (Jan through Feb is 1).	SELECT to_char (sysdate, 'q') FROM dual;	4

Formats with Date data types

MM	Month of the year (01-12).	SELECT to_char (sysdate, 'mm') FROM dual;	11
RM	Roman numeral for month.	SELECT to_char (sysdate, 'rm') FROM dual;	XI
month	Name of the month as a nine-character long string.	SELECT to_char (sysdate, 'month') FROM dual;	novemb er
WW	Week of the year	SELECT to_char (sysdate, 'ww') FROM dual;	48
W	Week of the month	SELECT to_char (sysdate, 'w') FROM dual;	4

Format with Date data types

Format	**Returns**	**Example**	**Result**
DDD	Day of the year; January 01 is 001; December 31 is 365 or 366.	SELECT to_char (sysdate, 'ddd') FROM dual;	329
DD	Day of the month.	SELECT to_char (sysdate, 'dd') FROM dual;	25
D	Day of the week. Sunday = 1; Saturday = 7.	SELECT to_char (sysdate, 'd') FROM dual;	3
DY	Abbreviated name of the day.	SELECT to_char (sysdate, 'dy') FROM dual;	tue
HH or HH12	Hour of the day (01-12).	SELECT to_char (sysdate, 'hh') FROM dual;	04
HH24	Hour of the day in 24-hour clock.	SELECT to_char (sysdate, 'hh24') FROM dual;	16
MI	Minutes (00-59)	SELECT to_char (sysdate, 'mi') FROM dual;	20
SS	Seconds (00-59)	SELECT to_char (sysdate, 'ss') FROM dual;	22

Conversion Functions

- The conversion functions are:
  - to_char()
  - to_number()
  - to_date()

Module 9. Oracle Architecture

- Overview
  - Physical Structure
  - Logical Structure
  - Memory Structure

Physical Structure

- Consists of various files on the disk (and more):
  - Data files
  - Redo log files
  - Control files
  - Parameter file
  - And more…

Data Files

- Store data, such as tables, indexes, etc.
- At least one data file is required in a database. More than one data file may be created in a database.
- While creating a database, at least one data file needs to be specified, along with the size of the file.
- The size of the file can be increased later, if required.
- The name and location of the data files is transparent to the users.

Redo Log Files

- Keep a record of all changes to the database (such as record inserts, index updates, etc.)
- Used for recovery of a database.
- Two types:
  - Online redo log files (mandatory)
  - Offline / archived redo log files (optional; useful in production environments)

Online Redo Log Files

- Used by Oracle cyclically to record changes to the database.
- Minimum two online redo log files; more may be used.
- Need to specify at least two online redo log files while creating a database.
- Name, location are transparent to users.
- Fixed in size; never grow in size.

Offline Redo Log Files

- Used by Oracle only in the ARCHIVELOG mode. Useful for production environments.
- In this mode, the online redo log files are backed up to offline redo log files before being overwritten.
- Unlimited number of redo log files can get created, whenever an online redo log file switch takes place.

Control Files

- Minimum one control file. Needs to be specified while creating a database.
- If there are multiple control files, they are identical in content.
- Contains some important information about the database, such as the database name, the names and locations of data files, online redo log files, archived log files, etc.
- Data dictionary view:
  - SELECT * FROM v$controlfile;

Parameter File

- Text file, containing parameters for the database.
- First file used by Oracle when starting a database.
- File contains various parameters, such as:
  - Database name
  - Control file name and location
  - Many other parameters that affect the memory size and performance of the database.

Logical Structure

- The logical structure of the Oracle architecture dictates how the physical space of a database is to be used.

- A hierarchy exists in this structure that consists of tablespaces, segments, extents, and blocks.

Segments

- A database object that uses storage is called a *segment*. The different types of segments include:
  - Data segments (tables, partitions and clusters)
  - Index segments
  - Temporary segments

Tablespaces

- Tablespace = A logical storage unit, consisting of one or more datafiles.
- While creating a segment, a tablespace can be specified for it. Example:

```
CREATE TABLE customer (
    cust_code number (6),
    cust_name char (30)
) tablespace ts_user;
```

- SYSTEM tablespace is automatically created, and is mandatory. Meant for data dictionary tables, etc.
- SYSAUX tablespace is also a mandatory tablespace (Oracle 10g). Meant to act as an auxiliary tablespace for SYSTEM tablespace.

Oracle Data Blocks

- Unit of space allocation for segments within a data file.
- Also the unit of any I/O performed on the data files.
- Minimum block size: 2KB, maximum block size: 32KB.
- Choice of block size can impact database performance. Depends on size of typical records in the database.

Memory Structure

- System Global Area
  - Is a part of the instance area of the database which resides in memory.
  - Used to store data and control structures.
  - Is shared by all users and processes of the database instance.
  - DBA decides the size of SGA after considering various parameters.
  - Components of SGA:
    - Data buffer cache
    - Redo buffer cache
    - Shared pool area
    - Some other optional components

Components of SGA

- Database Buffer Cache:
  - Holds recently accessed Oracle blocks containing user data.
  - All modifications (insert / update / delete) to the database are done through the database buffer cache.
  - Writing of blocks to the data files done by DBWR background process.

Components of SGA

- Redo Buffer Cache:
  - Keeps a log of all changes made to the database.
  - Commands of all transactions are entered here first.
  - They are written to the online redo log files only when the transaction is committed or when the buffer becomes full.
  - LGWR background process responsible for writing to online redo log files.

Components of SGA

- Shared Pool Area:
  - Consists of three sub-caches:
    - Library Cache
    - Dictionary Cache
    - Control Structures

Components of SGA

- Library Cache
  - Stores the build plan for all SQL queries in Shared SQL area.
  - Session-specific information stored in a separate area known as Private SQL area.
  - Stores PL/SQL procedures and packages

- Dictionary Cache
  - Contains data FROM the Oracle data dictionary.
  - Data dictionary is a collection of database tables and views containing reference information about the database, its structures, and its users.

- Control Structures
  - Consists of locks and library cache handles.

Components of SGA

- Some other optional components:
  - Large pool
  - Java pool
  - Streams pool (10g)

Program Global Area

- Contains session-specific data and control structures.
- A PGA is assigned to each session.
- Not a shared component.

Background Processes

- Background processes provide various services to the user processes.
- Many background processes can be there for an Oracle instance but, the common ones are:
  - DBWR
  - LGWR
  - SMON
  - PMON
  - ARCH
  - CKPT
- Out of the above, DBWR, LGWR, SMON and PMON are mandatory.

Module 10. Sequences & Synonyms

- Overview
  - Creating, altering, dropping and using Sequences
  - Creating, dropping and using Synonyms
  - Querying the data dictionary

Sequences

- Is a database object which is used to generate automatic unique integer values.

```
CREATE SEQUENCE sequence_name
    [INCREMENT BY n1]
    [START WITH n2]
    [MAXVALUE n3]
    [MINVALUE n4]
    [CYCLE | NOCYCLE];
```

Sequences

SQL> **CREATE SEQUENCE** Emp_Number
Increment By 2
Start With 3;

Sequences

SQL> insert into employee (emp_code, emp_name)
values (**EMP_NUMBER.NEXTVAL**, 'Satish');

SQL> SELECT **EMP_NUMBER.CURRVAL** FROM dual;

Sequences

SQL> **ALTER SEQUENCE** emp_number
maxvalue 250;

SQL> **DROP SEQUENCE** emp_number;

Synonyms

- Is an alternative name for another object, which may be a table, view, index or a sequence.

```
CREATE SYNONYM synonym_name FOR table_name
```

Synonyms

SQL> **CREATE SYNONYM** myemp for employee;

SQL> SELECT * FROM myemp;

SQL> **DROP SYNONYM** myemp;

Querying the Data Dictionary

- For Sequences
  - desc **USER_SEQUENCES**
  - SELECT * FROM **USER_SEQUENCES**;

- For Synonyms
  - desc **USER_SYNONYMS**
  - SELECT * FROM **USER_SYNONYMS**;

Module 11. Indexes

- Overview

  - Understanding Indexes
  - Unique and Non-unique Indexes
  - Creating and dropping Indexes
  - Querying the Data Dictionary

Indexes

- Are database objects used to improve the performance of the database.

- Uses the ROWID for search operations.

- There are two types of index:
  - UNIQUE
  - NON UNIQUE

```
CREATE [UNIQUE] INDEX index_name ON  table_name
(column_name);
```

SQL> **CREATE UNIQUE INDEX** code_idx **ON** employee(dept_code);

Indexes

SQL> **CREATE INDEX** cd_idx **ON** employee (dept_code);

SQL> **CREATE INDEX** emp_idx **ON** employee (dept_code, emp_code);

SQL> **DROP INDEX** cd_idx;

Querying the Data Dictionary

SQL> desc **USER_INDEXES**

SQL> SELECT index_name, uniqueness FROM **USER_INDEXES**
WHERE table_name = 'EMPLOYEE';

Module 12. Views

- Overview
  - Understanding Views
  - Creating views
  - Altering & dropping views
  - Manipulating data using views
  - Querying the Data Dictionary

Views

SQL> **CREATE VIEW** fin_emp **AS**
SELECT * FROM employee WHERE dept_code = 'FIN';

SQL> SELECT * FROM fin_emp;

SQL> **DELETE** fin_emp;

Views

```
SQL> insert into fin_emp (emp_code, emp_name, dept_code)
     values (111, 'Sunil', 'FIN');

SQL> CREATE OR REPLACE  VIEW fin_emp AS
     SELECT  * FROM  employee;
```

Views

```
SQL> CREATE OR REPLACE  view fin_emp AS
      SELECT  * FROM  employee
      with read only;

SQL> DROP VIEW fin_emp;
```

Querying the Data Dictionary

SQL> desc **USER_VIEWS**

SQL> SELECT view_name, text FROM **USER_VIEWS**;

Module 13. Advanced Queries

- Overview
  - Table joins
  - Sub queries
  - Set operators
  - Multi-table insert and delete
  - MERGE statement

JOINS

Equi Join

SQL> SELECT emp_name, dept_name
FROM EMPLOYEE, DEPT
WHERE dept.dept_code = employee.dept_code;

JOINS

Self Join

SQL> SELECT a.emp_name, b.emp_name
FROM employee A, employee B
WHERE A.reports_to = B.emp_code;

JOINS

Outer Join

```
SQL> SELECT  A.emp_name, B.emp_name
       FROM employee A, employee B
       WHERE A.reports_to = B.emp_code (+);
```

JOINS

```
SQL> SELECT  emp_code, dept.dept_code FROM employee CROSS
        JOIN  dept;

SQL> SELECT  emp_code,dept_code,dept_name
        FROM employee NATURAL JOIN dept ;

SQL> SELECT  emp_code, dept_name
        FROM employee JOIN dept USING (dept_code);

SQL> SELECT  emp_code, dept_name
        FROM employee a JOIN dept b
        ON (a.dept_code=b.dept_code
        And a.emp_code<20);
```

JOINS

```
SQL> SELECT  a.emp_code, a.dept_code, b.dept_name
      FROM employee a RIGHT OUTER JOIN dept b
      ON(a.dept_code=b.dept_code);

SQL> SELECT  a.emp_code,a.dept_code,b.dept_name
      FROM employee a LEFT OUTER JOIN dept b
      ON (a.dept_code=b.dept_code);

SQL> SELECT  a.emp_code, a.dept_code, b.dept_name
      FROM employee a FULL OUTER JOIN dept b
      ON (a.dept_code=b.dept_code);
```

SUBQUERIES

SQL> SELECT * FROM orders
WHERE cust_code **IN** (SELECT cust_code FROM customer
WHERE city_code = 'PUNE');

SQL> SELECT * FROM dept WHERE **EXISTS** (
SELECT * FROM employee WHERE employee.dept_code =
dept.dept_code);

SQL> SELECT * FROM dept WHERE **NOT EXISTS** (
SELECT * FROM employee WHERE employee.dept_code =
dept.dept_code);

SET Operators

```
SQL> SELECT  prod_code, prod_name FROM product
     UNION
     SELECT  prod_code, prod_name FROM old_products;

SQL> SELECT  * FROM  product
     INTERSECT
     SELECT  * FROM  import_product;

SQL>  SELECT  * FROM  product
      MINUS
      SELECT  * FROM  import_product;
```

Multi-Table Insert and Delete

- Multi-table inserts allow a single INSERT INTO .. SELECT statement to conditionally, or non-conditionally, insert into multiple tables.

- It reduces table scans and PL/SQL code necessary for performing multiple conditional inserts compared to previous versions.

- Prior to Oracle 9i, the only option available was to run separate **insert** statements, which was a costly option.

- There are four kinds of multi-table insert.
  - Unconditional
  - Pivoting
  - Conditional
  - Insert First

Multi-Table Insert and Delete

- Unconditional Insert

  - ➢ Inserts the given data into multiple tables without restrictions.

  - ➢ **INSERT ALL**

```
INTO retail_revenue
values(transaction_number,date_key, product_key, store_key,
 promotion_key, sale_quantity, sale_value)
INTO retail_cost
values (transaction_number,date_key, product_key, store_key,
 promotion_key, sale_quantity, cost_value)
SELECT * from retail_sale;
```

Multi-Table Insert and Delete

- Pivoting Insert

  - Is used to insert into the same table multiple times.

  - **INSERT ALL**

```
INTO all_paycheck VALUES (emp_id,'JAN',net_pay_jan)
INTO all_paycheck VALUES (emp_id,'FEB',net_pay_feb)
INTO all_paycheck VALUES (emp_id,'MAR',net_pay_mar)
INTO all_paycheck VALUES (emp_id,'APR',net_pay_apr)
INTO all_paycheck VALUES (emp_id,'MAY',net_pay_may)
INTO all_paycheck VALUES (emp_id,'JUN',net_pay_jun)
SELECT  emp_id,net_pay_jan, net_pay_feb, net_pay_mar, net_pay_apr,
net_pay_may, net_pay_jun
FROM  monthwise_paycheck_report;
```

Multi-Table Insert and Delete

- Conditional Insert

  - Used for conditional control of each insert based on established specific criteria.

  - **INSERT ALL**

```
WHEN  grade like 'M%' then
INTO manager_pay VALUES (employee_id, gross_pay, net_pay)
WHEN dept_code like ‘GEN' then
INTO worker_pay VALUES   (employee_id, gross_pay, net_pay)
ELSE
INTO all_pay VALUES (employee_id, gross_pay, net_pay)
SELECT employee_id, gross_pay, net_pay,grade,dept_code
   FROM extern_pay_amount,employee
where extern_pay_amount.employee_id=employee.emp_code
```

Multi-Table Insert and Delete

- Insert First
  - Allows for conditional execution of an insert statement based on a series of WHEN clause.
  - **INSERT FIRST**

```
WHEN  grade like 'M%' then
INTO manager_pay VALUES (employee_id, gross_pay, net_pay)
WHEN  grade like 'E%' then
INTO worker_pay VALUES   (employee_id, gross_pay, net_pay)
else
INTO all_pay VALUES (employee_id, gross_pay, net_pay)
   SELECT employee_id, gross_pay, net_pay,grade
   FROM extern_pay_amount,employee
where extern_pay_amount.employee_id=employee.emp_code;
```

MERGE statement

- Meaning

  - MERGE statement allows you to insert a record into a table if it doesn't exist, and it allows you to update an existing record in a table during the execution of the statement.

  - You can specify conditions to determine whether to update or insert into the target table or view.

  - Is a convenient way to combine multiple operations.

  - It lets you avoid multiple INSERT, UPDATE, and DELETE DML statements.

MERGE statement

- Example

    - **MERGE INTO** all_pay a

```
USING manager_pay b
on (a.emp_id=b.emp_id)
when matched then
update set a.gross_pay=b.gross_pay,a.net_pay=b.net_pay
when not matched then
insert values (b.emp_id,b.gross_pay,b.net_pay);
```

Transaction Processing

- COMMIT
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- SAVEPOINT
- LOCK TABLE

Transaction Processing

SQL> SAVEPOINT savepointname;

SQL> SAVEPOINT stage1;

SQL> ROLLBACK TO savepointname;

SQL> ROLLBACK TO stage1;

Transaction Processing

lock table table_reference_list in lock_mode [nowait]

```
SQL> LOCK table emp IN ROW EXCLUSIVE MODE;

SQL>  LOCK table emp,dept IN SHARE MODE NOWAIT;

SQL>  LOCK table scott.emp@new_york IN SHARE UPDATE MODE;

SQL> update employee
        set salary =
        (SELECT  salary FROM employee
                WHERE emp_name = 'Manorama Gupta'
        )
        WHERE emp_name = 'Neerja Girdhar';
```

Module 15. Formatting the Output

- Overview
  - Setting page layout
  - Formatting column output and spooling
  - Computing column values at breaks in SQL * Plus

Setting Page Layout

```
SQL> set LINESIZE 60;

SQL> set PAGESIZE 20;

SQL> TTITLE 'List of Employees';

SQL> TTITLE 'Pragati Software|List of Employees';

SQL> BTITLE 'Sample Report';
```

Formatting Column Output

```
SQL> SPOOL c:\mydata\output.txt;

SQL> COLUMN emp_name FORMAT A10 HEADING 'Employee|Name';

SQL> COLUMN salary FORMAT 9,99,999;

SQL> show all;
```

Formatting the Output

```
SQL> SELECT  dept_name,
        emp_name, salary
        FROM employee, dept
        WHERE employee.dept_code
        = dept.dept_code
        ORDER BY dept_name;
```

Oracle SQL*Plus

File Edit Search Options Help

```
DEPT_NAME            EMP_NAME                     SALARY
-------------------- ------------------------- ---------
Commercial           Madhusoodan Adiga              6000
Finance              Anuradha Gadkari              10000
Finance              Santosh Khot                   3800
Finance              Ragini Mazumdar                3600
General Management   Rajiv Jain                    16000
Human Resources Mgmt Shivanand Joshi               11500
Human Resources Mgmt Sanjeev Achar                  2400
Human Resources Mgmt Neerja Girdhar                 4200
Marketing            Sudha Ganesan                 12500
Marketing            R Srinivasan                   4900
Marketing            S Kannan                       6400
Marketing            Udai Kotecha                   6200
Marketing            Manorama Gupta                 3100
Marketing            Neeta Gopinath                 5400
Secretarial          Bimal Mehta                    4800
Software             Ashit Kothari                  6000
Software             Deepak Deo                     8000
Software             V V Nath                       8000
Software             Jayesh Shah                    5500
Software             Rama Kini                      4600
```

```
SQL> BREAK ON dept_name;

SQL> SELECT  dept_name,
        emp_name, salary
        FROM employee, dept
        WHERE employee.dept_cod
        = dept.dept_code
        ORDER BY dept_name;
```

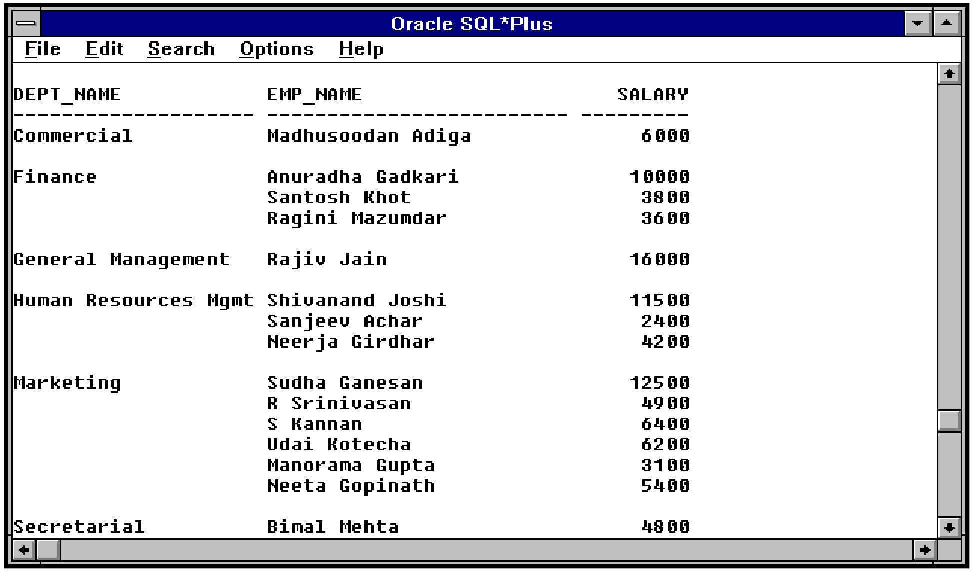
Oracle SQL*Plus

File Edit Search Options Help

```
DEPT_NAME            EMP_NAME                     SALARY
-------------------- ------------------------- ---------
Commercial           Madhusoodan Adiga              6000
Finance              Anuradha Gadkari              10000
                     Santosh Khot                   3800
                     Ragini Mazumdar                3600
General Management   Rajiv Jain                    16000
Human Resources Mgmt Shivanand Joshi               11500
                     Sanjeev Achar                  2400
                     Neerja Girdhar                 4200
Marketing            Sudha Ganesan                 12500
                     R Srinivasan                   4900
                     S Kannan                       6400
                     Udai Kotecha                   6200
                     Manorama Gupta                 3100
                     Neeta Gopinath                 5400
Secretarial          Bimal Mehta                    4800
Software             Ashit Kothari                  6000
                     Deepak Deo                     8000
                     V V Nath                       8000
                     Jayesh Shah                    5500
                     Rama Kini                      4600
```

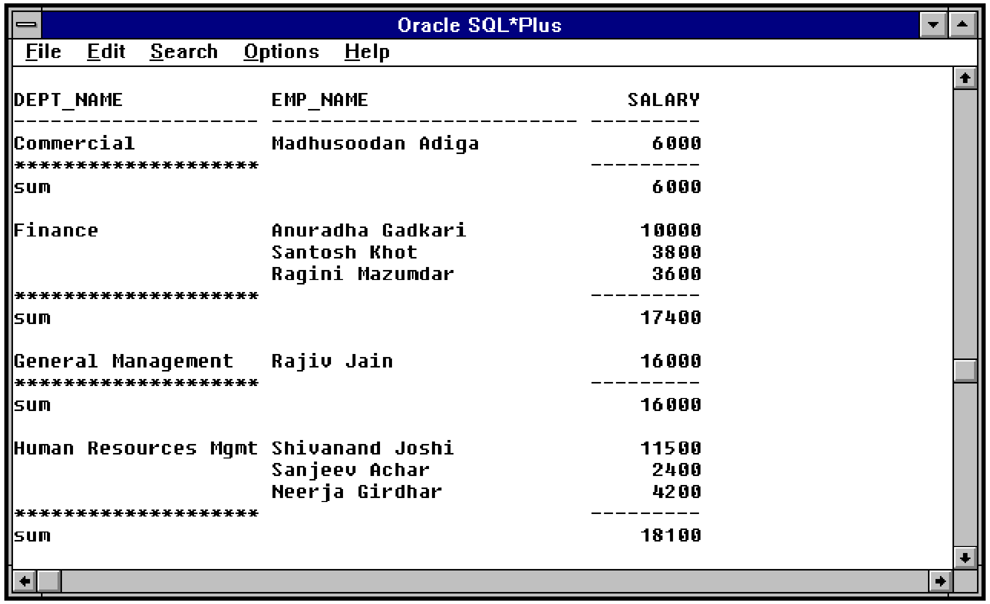
Formatting the Output

SQL> break on dept_name SKIP 1;



SQL> COMPUTE SUM OF salary

on dept_name;



Formatting the Output

SQL> break on report skip on dept_name skip 2;

```
Oracle SQL*Plus
File  Edit  Search  Options  Help

Secretarial          Bimal Mehta                   4800
********************                          ---------
sum                                                4800


Software             Ashit Kothari                 6000
                     Deepak Deo                    8000
                     V V Nath                      8000

DEPT_NAME            EMP_NAME                    SALARY
-------------------- ------------------------- ---------
Software             Jayesh Shah                   5500
                     Rama Kini                     4600
                     Anay Kamat                    5200
                     Gangadhar Kini                8200
********************                          ---------
sum                                               45500


                                               ---------
sum                                              146300
```

SQL> show all;

Module 16: Other Data Types

- Overview
  - DATETIME, INTERVAL, DATE data-types
  - Various TIMESTAMP data-types
  - INTERVAL YEAR TO MONTH data-type
  - INTERVAL DAY TO SECOND data-type
  - Floating-Point and Conversion functions

Other Data Types

The datetime data-types are:

- DATE
  - Stores centuary,year,month,day,hour,minutes and seconds
- TIMESTAMP
  - Like DATE but also provides subsecond times upto 9 digits(default 6)
- TIMESTAMP WITH TIME ZONE, and
  - Like Timestamp but includes local timstamp relative to UTC
- TIMESTAMP WITH LOCAL TIME ZONE.
  - Like Timestamp but returns time corresponding to the location of client

The interval data-types are:

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

  Both types provide the difference between 2 dates but do so in years /months or days/seconds

Other Data Types

```
SQL> insert into my_table values (1, SYSDATE);

SQL> insert into my_table values (2, TRUNC(SYSDATE));

SQL> SELECT  * FROM  my_table;

ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02
```

Other Data Types

```
SQL> SELECT  * FROM  my_table
  WHERE datecol = TO_DATE('03-OCT-02','DD-MON-YY');
  ROW_NUM DATECOL
  ---------- ---------
        2 03-OCT-02

SQL> SELECT  * FROM  my_table
  WHERE datecol > TO_DATE('02-OCT-02', 'DD-MON-YY');
  ROW_NUM DATECOL
        ---------- ---------
        1 03-OCT-02
        2 03-OCT-02

SQL> SELECT  * FROM  my_table WHERE TRUNC(datecol)
=to_date('20-MAR-05','dd-mon-yy');
```

Other Data Types

```
SQL> insert into my_table values
        (3, TO_DATE('3-OCT-2002','DD-MON-YYYY'));

SQL> insert into my_table values (4, '03-OCT-02');

SQL> insert into my_table values (5, TRUNC(SYSDATE));
```

TIMESTAMP [(fractional_seconds_precision)]

Other Data Types

TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE

TIMESTAMP ‘1997-01-31 09:26:56.66 +02:00’

TIMESTAMP ‘1999-04-15 8:00:00 -8:00’

TIMESTAMP ‘1999-04-15 11:00:00 -5:00’

Other Data Types

```
SQL> CREATE TABLE T(
        C TIMESTAMP(5) WITH TIME ZONE);
Table created.

SQL> insert into T values(CURRENT_TIMESTAMP);

SQL> SELECT  * FROM  T;
        C
        ------------------------------------------------------------
        20-MAR-05 11.24.43.12900 AM +05:30
```

Other Data Types

TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE

```
SQL> CREATE TABLE T(
        C TIMESTAMP WITH LOCAL TIME ZONE);
Table created.

SQL> insert into T values(CURRENT_TIMESTAMP);
1 row created.

SQL> SELECT  * FROM  T;
        C
        -----------------------------------------------------------------------
        20-MAR-05 11.32.59.426000 AM
```

Other Data Types

INTERVAL YEAR [(year_precision)] TO MONTH

```
SQL> CREATE TABLE t(c interval year(3) to month);

SQL>  insert into t values(interval '123-2' year(3) to month);
SQL>  insert into t values(interval '4' year);
SQL>  insert into t values(interval '300' month);
SQL>  insert into t values(interval '300' month(3));

SQL> SELECT  * FROM  t;
        C
        ------------------------------------------
        +123-02
        +004-00
        +025-00
        +025-00
```

Other Data Types

INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]

SQL> CREATE TABLE t (c interval day(3) to second);

SQL> insert into t values(INTERVAL '4 5:12:10.222' DAY TO SECOND(3));

SQL> insert into t values(INTERVAL '4 5:12' DAY TO MINUTE);

SQL> insert into t values(INTERVAL '400 5' DAY(3) TO HOUR);

SQL> insert into t values(INTERVAL '11:12:10.2222222' HOUR TO SECOND(7));

SQL> insert into t values(INTERVAL '11:20' HOUR TO MINUTE);

SQL> SELECT * FROM t;

Other Data Types

SQL> SELECT sysdate FROM dual;

SQL> SELECT sysdate,current_date, sessiontimezone FROM dual;

SQL> SELECT current_timestamp, localtimestamp FROM dual;

SQL> SELECT current_timestamp FROM dual;

SQL> SELECT sysdate, current_timestamp, sessiontimezone FROM dual;

Other Data Types

SQL> SELECT dbtimezone, sessiontimezone FROM dual;

SQL> SELECT localtimestamp ts1, FROM_TZ(localtimestamp,'-07:00') ts2 FROM dual;

- Returns timestamp with timezone.

SQL> SELECT sysdate chicago,NEW_TIME(sysdate,'CDT','PDT') los_angeles FROM dual;

- Returns date in the second timezone for date in first timezone

Other Data Types

```
SQL> SELECT  current_timestamp local,
        SYS_EXTRACT_UTC(current_timestamp) as gmt FROM dual;
```

- Returns UTC for the timestamp with timezone (ts-offset)

```
SQL> SELECT  tz_offset(dbtimezone) chicago,
                TZ_OFFSET( 'US/EASTERN') NEW_YORK ,
                TZ_OFFSET( 'EUROPE/LONDON') LONDON ,
                TZ_OFFSET( 'ASIA/SINGAPORE')  SINGAPORE
        FROM   dual;
```

- Returns numeric timezone offset

```
SQL> SELECT  sydate, extract(year FROM sysdate) year,
                EXTRACT(month FROM systimestamp) month,
                EXTRACT(time_zone_hour FROM systimestamp) tzh
        FROM   dual;
```

- Returns the specified component of the date/time or interval expression

Other Data Types

NUMBER	**Binary floating-point numbers**
stored using decimal precision. (the digits 0 through 9).	stored using binary precision (the digits 0 and 1).
All literals are stored exactly as **NUMBER** because literals are expressed using decimal precision	Binary storage cannot represent all values using decimal precision exactly. The error that occurs when converting a value FROM decimal to binary precision is undone when the value is converted back FROM binary to decimal precision. The literal 0.1 is such an example.
In a NUMBER column, floating point numbers have decimal precision.	In a BINARY_FLOAT or BINARY_DOUBLE column, floating-point numbers have binary precision.
	The binary floating-point numbers support the special values infinity and **NaN** (not a number)

Floating Point Number Limits

Value	**Binary-Double**	**Binary-Float**
Maximum finite value	1.79e308	3.4e38
Minimum finite value	-1.79e308	-3.4e38
Smallest positive value	2.3e-308	1.2e-38
Smallest negative value	-2.3e-308	-1.2e-38

Other Data Types

Valid NUMBER literals:	**Valid floating-point number literals:**
25 +6.34 0.5 25e-03 -1	25f +6.34F 0.5d -1D

Literal	**Meaning**	**Example**
binary_float_nan	A value of type BINARY_FLOAT for which the condition IS NAN is true	SELECT COUNT(*) FROM employees WHERE TO_BINARY_FLOAT(commission_pct) IS NOT NAN;
binary_float_infinity	Single-precision positive infinity	SELECT COUNT(*) FROM employees WHERE salary < BINARY_FLOAT_INFINITY;
binary_double_nan	A value of type BINARY_DOUBLE for which the condition IS NAN is true	SELECT COUNT(*) FROM employees WHERE TO_BINARY_DOUBLE(commission_pct) IS NOT NAN;
binary_double_infinity	Double-precision positive infinity	SELECT COUNT(*) FROM employees WHERE salary < BINARY_FLOAT_INFINITY;

Other Data Types

- Conversion functions
  - TO_BINARY_DOUBLE(expr)
  - TO_BINARY_DOUBLE(expr)

Other Data Types

```
SQL> CREATE TABLE float_point_demo
     (dec_num NUMBER(10,2), bin_doubl BINARY_DOUBLE,
     bin_float BINARY_FLOAT);
SQL> insert into float_point_demo
     values (1234.56,1234.56,1234.56);
SQL> SELECT  * FROM  float_point_demo;
     DEC_NUM BIN_DOUBLE  BIN_FLOAT
     ---------- ---------- ----------
     1234.56 1.235E+003 1.235E+003
SQL> SELECT  dec_num, TO_BINARY_DOUBLE(dec_num)
     FROM float_point_demo;
     DEC_NUM TO_BINARY_DOUBLE(DEC_NUM)
     ---------- ------------------------
     1234.56              1.235E+003
```

Other Data Types

```
SQL> SELECT  DUMP(dec_num) "Decimal",
     DUMP(bin_double) "Double"
     FROM float_point_demo;

     Decimal                            Double
     -------------------------- ---------- --------------------------------
     Typ=2 Len=4: 194,13,35,57          Typ=101 Len=8:
                                        192,147,74,61,112,163,215,10
```

Dump function returns a varchar2 value that includes the datatype code, the length in bytes and the internal representation of the expression.

Other Data Types

```
SQL> SELECT  dec_num, TO_BINARY_FLOAT(dec_num)
     FROM float_point_demo;

     DEC_NUM TO_BINARY_FLOAT(DEC_NUM)
     ---------- ------------------------
     1234.56              1.235E+003
```

Module 17. Introduction to PL/SQL

- Overview
  - Understanding PL/SQL
  - PL/SQL data types
  - Declaring variables
  - Looping and Conditional constructs
  - Anchored Data types

PL/SQL

- Facilities provided by PL/SQL
  - Row-level processing
  - Conditional statements (IF...ELSE...END IF, CASE)
  - Looping statements (FOR...NEXT, WHILE...LOOP...END LOOP, etc.)
  - Variable declaration and manipulation
  - Exception handling
  - User-defined procedures which may be called FROM anyWHERE

```
SQL> update employee
        set salary = salary + 111
        WHERE dept_code = 'MKTG';
```

PL/SQL

```
SQL> begin
            update employee
            set salary = salary + 111
            where dept_code = 'MKTG';
     end;


SQL> @pl_sql_filename;

     or

SQL> run pl_sql_filename;
```

PL/SQL

```
SQL> begin
                update employee
                set salary = salary + 111 WHERE dept_code =
                &deptval;
        end;
SQL> declare
                nSalary number (6);
        begin
                SELECT  salary into nSalary
                        FROM employee WHERE emp_code = 11;
                if nSalary < 8000 then
                        update employee
                        set salary = salary + 101 WHERE emp_code = 11;
                end if;
        end;
```

PL/SQL Data Types

- Scalar
- Composite
- Reference
- LOB

Scalar Types

- Scalar type is a data type that holds a single value. Scalar type includes the following categories:
  - Character / String
  - Number
  - Boolean
  - Date / Time

Scalar Types

- Character / String
  - ➢ Allows PL/SQL character or string types.
  - ➢ Up to 32 K in size.
- Number
  - ➢ Allows integer data types.
- Boolean
  - ➢ Allows TRUE, FALSE or NULL values.

Scalar Types

- Date / Time
  - Include DATE and TIMESTAMP datatypes.
  - DATE type is used to store date.
  - TIMESTAMP is an extension of DATE type. Includes year, month, day, hour, minute, seconds and fraction of second.

Scalar Types

- Timestamp
  - Provides date and time with fraction of seconds up to 9 places.
  - declare

```
   v_date timestamp(9) := systimestamp;
begin
   dbms_output.put_line(v_date);
end;
```

Scalar Types

- Timestamp with time zone
  - Is an extension of TIMESTAMP type in that it stores the local timestamp relative to UTC.
  - ```
    declare
       v_date timestamp(3) with time zone := systimestamp;
    begin
       dbms_output.put_line(v_date);
    end;
    ```
    ```

Scalar Types

- Timestamp with local time zone
  - Returns time corresponding to the location of the client accessing the database server.

  - declare

```
declare
   v_date timestamp(5) with local time zone := systimestamp;
begin
   dbms_output.put_line(v_date);
end;
```

Composite Types

- Contain internal components
- Are reusable
- Are of two types:
  - PL/SQL Records
  - PL/SQL Collections
    - Index By tables
    - Varrays
    - Nested tables
    - Object types

Reference Types and LOB Types

- Reference types provide memory structures
- They can point to different storage locations through out the program
- Are of two types:
  - Ref Cursor
  - Ref
- LOB types are used to work with data types up to 4 GB in size.

PL/SQL IF-ELSE Statements

```
if nValue > 40 then
      nCount := nCount + 1;
end if;
```

```
if nValue between 40 and 50 then
            nCount := nCount + 1;
end if;
```

```
if nValue > 40 then
      if nValue < 50 then
         nCount := nCount + 1;
     end if;
end if;
```

PL/SQL IF-ELSE Statements

```
if nValue > 40 then
        nCount1 := nCount1 + 1;
else
        nCount2 := nCount2 + 1;
end if;
```

```
if nSalary < 2000 then
                nProfTax := 0;
elsif nSalary < 3500 then
                nProfTax := 15;
elsif nSalary < 5000 then
                nProfTax := 30;
else
                nProfTax := 50;
end if;
```

PL/SQL

```
SQL> declare
            nSalary number (6);
      begin
            SELECT  salary into nSalary
                  FROM employee WHERE emp_code <= 11;
            if nSalary < 8000 then
                  update employee
                  set salary = salary + 101
                        WHERE emp_code = 11;
            end if;
      end;
```

PL/SQL

```
SQL> SELECT  ename,
        (           case deptno
                    when 10 then 'ACCOUNTS'
                    when 20 then 'RESEARCH'
                    when 30 then 'SALES'
                    when 40 then 'OPERATIONS'
                    else 'UNASSIGNED'
                    end
        ) as Department
        FROM emp;
```

PL/SQL

```
SQL> SELECT  ename,sal,deptno,
       Case
             when sal<=500 then 0
             when sal>500 and sal<1500 then 100
             when sal>=1500 and sal <2500 and deptno=10 then
             200
             when sal >1500 and sal <2500 and deptno=20 then
             500
             when sal>=2500 then 300
       Else 0
       End "bonus"
       FROM emp;
```

PL/SQL LOOPS

```
SQL> declare
          nCode number (5);
     begin
          nCode := 101;
          loop
               insert into employee (emp_code, emp_name)
                    values (nCode, 'Somebody');
               nCode := nCode + 1;
               if nCode > 110 then
                    exit;
               end if;
          end loop;
     end;
```

PL/SQL LOOPS

```
SQL> declare
        nCode number (5);
    begin
        nCode := 101;
        while nCode <= 110
        loop
            insert into employee (emp_code, emp_name)
                values (nCode, 'Somebody');
            nCode := nCode + 1;
        end loop;
    end;
```

```
WHILE condition
LOOP
        statements;
END LOOP;
```

PL/SQL LOOPS

```
SQL> begin
        for ncode in 101..110
        loop
            insert into employee (emp_code, emp_name)
                values (ncode, 'somebody');
        end loop;
    end;
```

```
FOR variable in [REVERSE] lowerbound .. Upperbound
LOOP
    statements;
END LOOP;
```

PL/SQL

```
SQL> declare
          nSalary employee.salary%type;
     begin
          SELECT  salary into nsalary
                 FROM employee
                 WHERE emp_code = 11;

          if nsalary < 8000 then
                 update employee
                        set salary = salary + 101
                        WHERE emp_code = 11;
          end if;
     end;
```

PL/SQL

```
SQL> declare
          nRecord employee%rowtype;
     begin
          SELECT  * into nRecord
               FROM employee
               WHERE emp_code = 11;
          if nRecord.salary < 8000 then
               update employee
                    set salary = salary + 101
                    WHERE emp_code = 11;
          end if;
     end;
```

recorddatatype.columnname
nRecord.Salary

PL/SQL

```
SQL> declare
        nRecord employee%rowtype;
        nFactor constant number (4, 2) := 1.11;
    begin
        SELECT  * into nRecord
        FROM employee
        WHERE emp_code = 11;
        if nRecord.salary < 8000 then
            update employee
                set salary = salary * nFactor
                WHERE emp_code = 11;
        end if;
    end;
```

constantname CONST datatype := value;

Module 18. Cursors

- Overview
  - Understanding Cursors
  - Types of cursors
  - Cursor operations
  - Cursor attributes
  - Parameterized Cursors
  - Cursor FOR loop
  - REF Cursors

Cursors

- Provide a subset of data, defined by a query, retrieved into memory when opened, and stored in memory until the cursor is closed.
- Point to a memory region in the Process Global Area (PGA) called the context area.
- Context area holds:
  - Reference to the rows returned by the query
  - Number of rows processed by the query
  - A pointer to the parsed query in the Shared Pool
- Pointer is to memory, not to the data directly.
- We are guaranteed a consistent view of data throughout the transaction.

Types of Cursors

- Implicit Cursors
  - Are implicitly created by PL/SQL, whenever any DML or SELECT....INTO statement is executed in a PL/SQL block.
- Explicit Cursors
  - Are declared and named by the programmer.
  - The programmer directly controls almost all operations of the cursor.
  - CURSOR employee_cur IS select * from employee;
  - OPEN employee_cur;
  - FETCH employee_cur INTO employee_rec;
  - CLOSE employee_cur;

Implicit Cursors

```
SQL> declare
        var1 varchar2(30);
     begin
        SELECT  emp_name INTO var1 FROM employee
        WHERE emp_code=34;
        dbms_output.put_line('Value of var1 is : '||var1);
     end;
```

Implicit Cursor Attributes

- SQL%FOUND
- SQL%NOTFOUND
- SQL%ROWCOUNT
- SQL%ISOPEN

Implicit Cursors

```
SQL> begin
        update employee set salary = salary + 100
        where dept_code='TRNG';
        dbms_output.put_line(SQL%ROWCOUNT || 'rows updated');
        if SQL%NOTFOUND
        then
            dbms_output.put_line ('Unable to update department
            TRNG');
        end if;
    end;
```

Explicit Cursors

- To use an explicit cursor, we have to perform 4 different operations:
  - Declare the cursor
  - Open the cursor
  - Fetch the cursor
  - Close the cursor

Explicit Cursors

```
SQL> declare
         cursor sales is
         SELECT
         a.salesman_id,a.salesman_name, b.target, c.sales
         FROM salesman a, target b, sales c
         WHERE a.salesman_id=b.salesman_id
         and b.tmonth=c.smonth;
     begin
         .............
     end;
```

Explicit Cursors

```
SQL> declare
        /* explicit declaration of a cursor*/
                cursor emptyp_cur is SELECT  emptyp.type
                from employee emp, employee_type emptyp
                WHERE emp.type_code= emptyp.type_code;
     begin
        /* check to see if cursor is already open. If not, open it.
         checked using %isopen. It is a cursor attribute explained later */
          if not emptyp_cur%isopen then
                open emptyp_cur;
          end if;
        /* fetch row FROM cursor directly into an oracle forms item*/
                fetch emptyp_cur into :emp.type_desc;
        /* close the cursor*/
                close emptyp_cur;
     end;
```

Explicit Cursor Operations

- Cursor Operations
  - Parse
  - Bind
  - Open
  - Execute
  - Fetch
  - Close

Explicit Cursor Attributes

Name	Description
%FOUND	Returns TRUE if record was fetched successfully, FALSE otherwise.
%NOTFOUND	Returns TRUE if record was not fetched successfully, FALSE otherwise.
%ROWCOUNT	Returns number of records fetched FROM the cursor at that point in time.
%ISOPEN	Returns TRUE if cursor is open, FALSE otherwise

```
CURSOR caller_cur IS
        SELECT  caller_id, company_id FROM caller;
```

Explicit Cursors

```
SQL> declare
        cursor caller_cur is SELECT  caller_id, company_id FROM caller;
        caller_rec caller_cur%rowtype;
     begin
        /*open the cursor if it is not open*/
        if not caller_cur%isopen then open caller_cur;
        end if;
        fetch caller_cur into caller_rec;
        /* keep fetching until no more records are found */
        while caller_cur%found
        loop
                dbms_output.put_line (‘Just fetched record number’ || to_char
                (caller_cur%rowcount));
                fetch caller_cur into caller_rec;
        end loop;
        close caller_cur;
     end;
```

Explicit Cursors

```
SQL> declare
        cursor emp_cur is SELECT  * FROM  employee;
        emp_rec emp_cur%rowtype;
        v_rowcount number :=0;
     begin
        if not emp_cur%isopen then
            open emp_cur;
        end if;
        loop
            fetch emp_cur into emp_rec;
            dbms_output.put_line(emp_rec.emp_name);
            v_rowcount := emp_cur%rowcount;
            exit when emp_cur%notfound;
        end loop;
```

Explicit Cursors

```
 dbms_output.put_line('total rows retrieved are : ' ||
 v_rowcount);
 close emp_cur;
if emp_cur%isopen =false then
        dbms_output.put_line('cursor closed');
else
        dbms_output.put_line('the cursor is still open');
end if;
end;
```

Explicit Cursors

Explicit Cursors

```
SQL>  open caller_cur;
      loop
            fetch caller_cur into caller_rec;
            exit when not caller_cur%found;
            update caller set caller_id=caller_rec.caller_id;
            WHERE call_timestamp< sysdate;
      end loop;
      close caller_cur;
```

Cursors

- **Differences Between Implicit and Explicit Cursor Attributes**

  - If the RDBMS has not opened an SQL cursor in the session, SQL%ROWCOUNT attribute returns NULL. References to other attributes (ISOPEN, FOUND, NOTFOUND) all return FALSE .
  - The %ISOPEN attribute will always return FALSE – before and after the SQL statement.
  - You must reference the SQL% attribute immediately after you execute the SQL statement.
  - The %FOUND attribute returns TRUE if an UPDATE, DELETE or INSERT affected at least one record. It will return FALSE if those statements failed to affect any records.
  - When an implicit SELECT statement does not return any rows, PL/SQL raises NO_DATA_FOUND exception and if more than one row are returned, PL/SQL raises the TOO_MANY_ROWS exception.

Parameterized Cursors

- Provide a way to pass information into and out of a module.
- Makes a cursor more reusable.
- Makes the cursor more generalized.

Parameterized Cursors

```
SQL> declare
        CURSOR employee_cur(dept_desc varchar2)
        IS
        SELECT  emp_name,salary,dept_code FROM employee
        WHERE dept_code= upper(dept_desc);
        emp_rec employee_cur%rowtype;
     begin
        open employee_cur('&dept_desc');
        fetch employee_cur into emp_rec;
        DBMS_OUTPUT.PUT_LINE(emp_rec.emp_name|| '
        ' ||emp_rec.salary|| ' ' ||emp_rec.dept_code);
     end;
```

Cursor FOR Loop

- Is associated with an explicit cursor or a SELECT statement embedded directly within a loop.
- Does not require an explicit OPEN, FETCH or, CLOSE.
- PL/SQL handles its processing.
- The variable declared in the FOR loop need not be declared.

Cursor FOR Loop

```
SQL> declare
        CURSOR c1 IS SELECT emp_name, dept_code FROM
        employee  WHERE dept_code LIKE '%FI%';
    begin
        FOR item IN c1
        loop
            dbms_output.put_line('Name = ' || item.emp_name || ',
            Department code = ' || item.dept_code);
        end loop;
    end;
```

REF Cursors

- REF Cursors offer a dynamic and persistent cursor alternative to the static explicit cursors.
- Are evaluated at run time instead of compile time.
- Can be opened for multiple SELECT statements in the same block.
- That is, a single cursor variable can be used to fetch from different result sets.
- REF Cursors can be either ***strongly typed*** *or* ***weakly typed.***

REF Cursors

- Declaring REF CURSOR types and Cursor Variables:

```
TYPE cursor_type_name is REF CURSOR [return return_type];
```

- TYPE company_curtype is REF CURSOR RETURN company%rowtype;

- TYPE generic_curtype is REF CURSOR ;

REF Cursors

```
SQL> declare
        TYPE emp_type is REF CURSOR return employee%rowtype;
        emp_curv emp_type;
        v_emp employee%rowtype;
    begin
        OPEN emp_curv FOR
        SELECT * FROM employee WHERE dept_code='FIN';
        dbms_output.put_line('-----------------');
        dbms_output.put_line('Opened the cursor');
        dbms_output.put_line('-----------------');
        loop
            FETCH emp_curv INTO v_emp;
            EXIT WHEN emp_curv%NOTFOUND;
            dbms_output.put_line(v_emp.emp_name);
        end loop;
        CLOSE emp_curv;
```

REF Cursors

```
        OPEN emp_curv FOR
        select * from employee where dept_code='TRNG';
        dbms_output.put_line('----------------------');
        dbms_output.put_line('Opened the cursor again');
        dbms_output.put_line('----------------------');
        loop
                FETCH emp_curv INTO v_emp;
                EXIT WHEN emp_curv%NOTFOUND;
                dbms_output.put_line(v_emp.emp_name);
        end loop;
        CLOSE emp_curv;
END;
```

REF Cursors

REF Cursors

```
SQL> declare
        TYPE emp_type is REF CURSOR;
        emp_curv emp_type;
        v_emp employee%rowtype;
        v_dept dept%rowtype;
    begin
        OPEN emp_curv FOR
        SELECT * FROM employee WHERE dept_code='MKTG';
        dbms_output.put_line('------------------------------');
        dbms_output.put_line('Opened the cursor for EMP table');
        dbms_output.put_line('------------------------------');
        loop
            FETCH emp_curv INTO v_emp;
            EXIT WHEN emp_curv%NOTFOUND;
            dbms_output.put_line(v_emp.emp_name);
        end loop;
        close emp_curv;
```

REF Cursors

```
        OPEN emp_curv FOR
        select * from dept;
        dbms_output.put_line('--------------------------------');
        dbms_output.put_line('Opened the cursor for DEPT table');
        dbms_output.put_line('--------------------------------');
        loop
                FETCH emp_curv INTO v_dept;
                EXIT WHEN emp_curv%NOTFOUND;
                dbms_output.put_line(v_dept.dept_name);
        end loop;
        CLOSE emp_curv;
end;
```

REF Cursors

Module 19. Exception Handlers

- Overview
  - Understanding exceptions
  - Named system exceptions
  - Unnamed system exceptions
  - Named programmer-defined exceptions
  - Unnamed programmer-defined exceptions

Exception Handlers

- Exceptions are errors in a PL/SQL block that are raised during execution of the block.
- Exceptions can be raised implicitly by the oracle server or explicitly by the application program.
- An exception handler may be specified to handle the raised exception.
- Exceptions are raised either at compile time or run time:

Error Type	Reported By	How Handled
Compile-time	PL/SQL Compiler	Interactively **:** compiler reports errors, and you have to correct them.
Run-time	PL/SQL run-time engine	Programmatically **:** exceptions are raised and caught by exception handlers.

Exception Handlers

- Types of exceptions:
  - Named System exceptions
  - Named Programmer-defined exceptions
  - Unnamed System exceptions
  - Unnamed Programmer-defined exceptions

Exception Handlers

The Exception Section	**An English Like Translation**
EXCEPTION WHEN NO_DATA_FOUND THEN executable_statements1;	If the NO_DATA_FOUND exception was raised, then execute the first set of statements.
WHEN payment_overdue THEN executable_statements2;	If the payment is overdue, then execute the second set of statements
WHEN OTHERS THEN executable_statements3;	If any other exception is encountered, then execute the third set of statements

Exception Handlers

```
declare
...... declarations ......
begin
        ...... executable statements......
[exception
        ...... exception handlers......]
end;
```

exception
    when exception_name
    then
        <executable statements>
end;

Exception Handlers

- Named System Exceptions
  - Are raised implicitly when its associated Oracle error occurs.
  - Are declared in the STANDARD package.

```
SQL>  declare
          v_emp emp%rowtype;
      begin
          SELECT  * into v_emp FROM emp;
      end;
```

Exception Handlers

Name of Exception Oracle Error/SQLCODE	Description
CURSOR_ALREADY_OPEN ORA-6511 SQLCODE=-6511	You tried to OPEN a cursor that was already OPEN. You must CLOSE a cursor before you try to OPEN or re-OPEN it.
DUP_VAL_ON_INDEX ORA-00001 SQLCODE=-1	Your INSERT or UPDATE statement attempted to store duplicate values in a column or columns in a row, which is restricted by a unique index.
INVALID_CURSOR ORA-01001 SQLCODE=-1001	You made reference to a cursor that did not exist. This usually happens when you try to FETCH FROM a cursor or CLOSE a cursor before that cursor is OPENed.
INVALID_NUMBER ORA-01722 SQLCODE=-1722	PL/SQL executes a SQL statement that cannot convert a character string successfully to a number. This exception is different FROM the VALUE_ERROR exception, as it is raised only FROM within a SQL statement.
LOGIN_DENIED ORA-01017 SQLCODE=-1017	Your program tried to log onto the Oracle RDBMS with an invalid username-password combination. This exception is usually encountered when you embed PL/SQL in 3GL language.
NO_DATA_FOUND ORA-01403 SQLCODE=+100	This exception is raised in three different scenarios: (1) You executed a SELECT INTO statement(implicit cursor) that returned no rows. (2) You referenced an uninitialized row in a local PL/SQL table. (3) You read past end of file with UTL_FILE package.

Exception Handlers

NOT_LOGGED_ON ORA-01012 SQLCODE=-1012	Your program tried to execute a call to the database (usually with a DML statement) before it had logged into the Oracle RDBMS.
PROGRAM_ERROR ORA-06501 SQLCODE=-6501	PL/SQL encounters an internal problem. The message text usually also tells you to “Contact Oracle Support”.
STORAGE_ERROR ORA-06500 SQLCODE=-6500	Your program ran out of memory or memory in some was corrupted.
TIMEOUT_ON_RESOURCE ORA-00051 SQLCODE=-51	A timeout occurred in the RDBMS while waiting for a resource.

Exception Handlers

Name of Exception Oracle Error/SQLCODE	Description
TRANSACTION_BACKED_OUT ORA-00061 SQLCODE=-61	The remote part of a transaction is rolled back, either with an explicit ROLLBACK command or as a result of some other action
VALUE_ERROR ORA-06502 SQLCODE=-6502	PL/SQL raises a VALUE_ERROR whenever it encounters an error having to do with the conversion, truncation or invalid constraining of numeric and character data. This is a very general and common exception. If this same type of error is encountered in a SQL DML statement within a PL/SQL block, then the INVALID_NUMBER exception is raised.
ZERO_DIVIDE ORA-01476 SQLCODE=-1476	Your program tried to divide by zero.

Exception Handlers

```
SQL> declare
        name varchar2(30);
     begin
        SELECT  emp_name into name FROM employee WHERE
                emp_code = &emp_code;
        dbms_output.put_line(name);
     exception
         when no_data_found then
            dbms_output.put_line ('Such employee does not exists');
         when others then
             dbms_output.put_line (SQLERRM || '     '||SQLCODE);
             dbms_output.put_line ('Some other error');
     end;
```

Exception Handlers

- Named Programmer-Defined Exceptions

```
SQL> declare
        acbalance number (6);
        neg_bal exception;
    begin
        SELECT  balance into acbalance FROM accounts
         WHERE balance <= 250;
         if acbalance < 0 then
                raise neg_bal;
         end if;
    exception
         when neg_bal then
                update accounts
                set fine = fine * 5 WHERE ac_type = 'cur';
   end;
```

Exception Handlers

```
SQL> declare
        current_sal number;
        emp_id number;
        sal_null exception;
     begin
        emp_id:=&empno;
        SELECT  salary into current_sal FROM employee
        WHERE emp_code=emp_id;
        if current_sal is null then
           raise sal_null;
        else
           update employee set salary = current_sal+1000
           WHERE emp_code = emp_id;
        end if;
```

Exception Handlers

```
exception
    when sal_null then
    dbms_output.put_line('Salary is missing');
end;
```

Exception Handlers

- Unnamed System Exceptions
  - ➢ Are standard Oracle server errors but, are not defined.
  - ➢ Also known as non-predefined Oracle errors.
  - ➢ Can be trapped by explicitly declaring it first with the PRAGMA EXCEPTION_INIT keyword.

Exception Handlers

```
declare
        exception_name exception;
        pragma exception_init
        (exception_name,error_code_literal);
begin
        ........... executable statements ...........
        raise exception_name;
exception
        when exception_name then
                ........... executable statements ...........
end;
```

Exception Handlers

```
SQL> declare
        e_missing exception;
        pragma exception_init (e_missing,-1400);
     begin
        insert into new1(empno) values (null);
     exception
        when e_missing then
        dbms_output.put_line('ora-1400 occurred');
     end;
```

Exception Handlers

- Unnamed Programmer-Defined Exceptions

  •The final type of exception is the unnamed programmer-defined exception.
  •This kind of exception occurs when you need to raise an application specific error FROM within the server and communicate this error back to the client application process.
  •The special procedure RAISE_APPLICATION_ERROR lets us issue user-defined error messages FROM stored subprograms

```
raise_application_error(error_number, message[, {true | false}]);
```

Exception Handlers

```
SQL> declare
        current_sal number;
        emp_id number;
     begin
        emp_id:=&empno;
        SELECT  salary into current_sal FROM employee WHERE
        emp_code=emp_id;
        if current_sal is null then
                raise_application_error(-20102,'salary is missing');
        else
                update employee set salary=current_sal+1000
                WHERE emp_code=emp_id;
        end if;
     end;
```

Module 20. Stored Procedures and Functions

- Overview
  - Understanding Stored Procedures
  - Creating Stored Procedures
  - Parameter modes
  - Understanding Stored Functions
  - Creating Stored Functions

Stored Procedures

- Stored Procedures

  - When a procedure is created, the Oracle engine automatically performs the following steps
    - Compiles the procedure or function.
    - Stores the procedure or function in the database.

  - The Oracle engine performs the following steps to execute a procedure or function
    - Verifies the user access.
    - Verifies procedure or function validity.
    - Executes the procedure or function.

Stored Procedures

```
CREATE OR REPLACE PROCEDURE [ schema]  procedurename
                (argument {IN, OUT, INOUT} datatype,….)
                {IS ,AS}
                variable declarations;
                constant declarations;
 BEGIN
                PL/SQL subprogram body;
 EXCEPTION
                Exception pl/sql block;
 END;
```

Stored Procedures

- Procedure parameter modes

  - The **IN** parameter mode is used to pass values to the procedure when invoked.
  - The **OUT** parameter mode is used to return a values to the caller of the procedure.
  - The **IN OUT** parameter is used to pass initial values to the procedure when invoked and it also returns updated values to the caller.

Stored Procedures

```
SQL> create or replace procedure get_emp_data (
            eno in number,
            ename out varchar2,
            esal out number)
      as
            cursor mycursor (eno number) is
                  SELECT  emp_name, salary FROM employee
                  WHERE emp_code = eno;
      begin
            open mycursor (eno);
            fetch mycursor into ename, esal;
            close mycursor;
      end;
```

Stored Procedures

```
SQL> create or replace procedure raise_salary (emp_id number,
                                               increase  number)
     as
          current_salary number;
     begin
          SELECT  salary into current_salary FROM employee
          WHERE emp_code = emp_id;
          if current_salary is null then
          raise_application_error(-20101, 'salary is missing');
          else
          update employee set salary = current_salary +increase
          WHERE emp_code = emp_id;
          end if;
     end raise_salary;
```

Stored Procedures

```
SQL> create or replace procedure update_dept
        (deptno in varchar2,
         dname in out varchar2)
     as
     begin
        update dept set dept_name=dname where
        dept_code=deptno;
        dbms_output.put_line(deptno ||' ' ||dname);
     end update_dept;
```

Stored Functions

- Stored Functions
  - Is similar to stored procedures except that a function returns only a single values.

```
SQL> create or replace function insert_dept (
            dcode varchar2, dname varchar2)
            return boolean
     as
     begin
            insert into dept values (dcode, dname);
            return true;
     exception
            when others then
                   return false;
     end;
```

Stored Functions

```
SQL> declare
          dummy boolean;
     begin
          dummy := insert_dept ('XYZ', 'Dept XYZ');
     end;
```

Stored Functions

```
SQL> create or replace function compute_tax
     (empno number,
      tax in out number)
      return number
     is
     begin
          select salary into tax from employee where
          emp_code=empno;
          tax := tax * .3;
          return tax;
     end;
```

Stored Functions

```
SQL> declare
        var1 number;
        var2 number;
     begin
        var1 := compute_tax(40,var2);
        dbms_output.put_line('Tax value is: '||var2);
     end;
```

Module 21 : Packages

- Overview
  - Introduction to Packages
  - Package Specification and Body
  - Creating Packages
  - Package Overloading
  - Altering and Dropping Packages
  - Advantages of Packages

Introduction to Packages

- Packages
  - ➢ Is a named PL/SQL block that groups logically related PL/SQL constructs such as:
    - Procedures
    - Functions
    - Cursors
    - Variables and constants
    - Exception definitions
    - PL/SQL Types
  - ➢ Packages cannot be called, passed parameters, or nested.

Package Specification and Body

```
PACKAGE name IS  -- specification (visible part)
        -- public type and object declarations
        -- subprogram specifications
END [name];
```

```
PACKAGE BODY name IS  -- body (hidden part)
        -- private type and object declarations
        -- subprogram bodies
[BEGIN
        -- initialization statements]
END [name];
```

Creating Packages

```
SQL> create package emp_actions
     as
          procedure hire_employee
          (empid number,
          ename varchar2,
          dept varchar2,
          grade varchar2,
          sal number
          );
          procedure fire_employee (empid number);
     end emp_actions;
```

Creating Packages

SQL> **create or replace package body emp_actions**

as

```
procedure hire_employee
(empid number,
ename varchar2,
dept varchar2,
grade varchar2,
sal number
)
is
begin
        insert into employee(emp_code, emp_name,
        dept_code, grade, salary)
        values(empid,ename,dept,grade,sal);
end hire_employee;
```

Creating Packages

```
        procedure fire_employee(empid number)
        is
        begin
                delete from employee where emp_code = empid;
        end fire_employee;
end emp_actions;
```

Creating Packages

```
create or replace package emp_data as
    procedure open_cv (generic_cv in out sys_refcursor, choice
                 in number);
end emp_data;

create or replace package body emp_data as
  procedure open_cv (generic_cv in out sys_refcursor, choice
                 in number)  is
  begin
        if choice = 1 then
        open generic_cv for SELECT  * FROM  employee;
```

Creating Packages

```
        elsif choice = 2 then
                open generic_cv for SELECT  * FROM  dept;
        elsif choice = 3 then
                open generic_cv for SELECT  * FROM  salgrade;
        end if;
  end open_cv;
end emp_data;
/

variable x refcursor;
exec emp_data.open_cv(:x,1);
Print x;
```

Creating Packages

- Package which has functions for random number generation.

```
SQL> create or replace package randomnumbers
     is
             procedure srand( new_seed in number );
             function rand(range in number) return number;
     end randomnumbers;
```

Creating Packages

```
SQL> create or replace package body randomnumbers
     is
     multiplier      constant number         := 22695477;
             increment    constant number         := 1;
             "2^32"          constant number          := 2 ** 32;
     "2^16"          constant number         := 2 ** 16;
             Seed            number := 1;
     -- Procedure Srand is used to pass a new_seed number
     procedure srand( new_seed in number )
     is
      begin
```

Creating Packages

```
            Seed := new_seed; --seed is a global variable
            dbms_output.put_line(seed);
      end srand;
  -- Function rand is used to generate a random number
       function rand(range in number) return number
       is
       begin
             seed := mod( multiplier * seed + increment, "2^32" );
             return bitand( seed/"2^16", range);
         end rand;
   end randomnumbers;

  Select randomnumbers.rand(1234) from dual;
```

Package Overloading

- Within a package, procedure and functions can be overloaded.
- This means that there is more than one procedure or function with the same name, but with different parameters.
- It allows the same function to be applied to objects of different types.
- This option is useful when you want a subprogram to accept similar sets of parameters that have different datatypes.
- Makes code maintenance easier.

Package Overloading

- SQL> **create or replace package** journal_entries

```
as
...
    procedure journalize (amount real, trans_date varchar2);
    procedure journalize (amount real, trans_date int);
end journal_entries;
```

Package Overloading

- SQL> **create or replace package body** journal_entries

```
as
...
    procedure journalize
    (amount real, trans_date varchar2)
    is
    begin
        insert into journal values
        (amount,to_date(trans_date, 'dd-mon-yyyy'));
    end journalize;
```

Package Overloading

```
    procedure journalize
    (amount real, trans_date integer)
    is
    begin
        insert into journal values
        (amount, to_date(trans_date, 'j'));
    end journalize;
end journal_entries;
```

Altering and Dropping Packages

- Altering Packages
  - Just like procedures and functions, packages should be recompiled if their referenced constructs are changed for any reason.
  - ALTER PACKAGE package_name COMPILE;
  - ALTER PACKAGE package_name COMPILE BODY;

- Dropping Packages
  - DROP PACKAGE BODY package_name;
  - DROP PACKAGE package_name;

Advantages of Packages

- Advantages of Packages
  - Modularity
  - Easier Application Design
  - Information Hiding
  - Added Functionality
  - Better Performance

Advantages of Packages

Module 22. Triggers

- Overview
  - ➢ Understanding Triggers
  - ➢ Keywords and Parameters
  - ➢ Applying Triggers
  - ➢ Types of Triggers
  - ➢ Expressions in Triggers
  - ➢ Conditional Predicates
  - ➢ Recompiling and Dropping Triggers

Database Triggers

- A trigger defines an action the database should take when some database related event occurs.
- It may be used to supplement declarative referential integrity, to enforce complex business rules or to audit changes of data.
- Uses of Triggers are:
    - Audit Data Modifications
    - Log Events Transparently
    - Enforce Complex Business Rules
    - Derive Column Values Automatically
    - Implement Complex Security Authorizations
    - Maintain Replicate Tables

Database Triggers

```
CREATE OR REPLACE TRIGGER [schema] triggername
        {BEFORE, AFTER}
        {DELETE, INSERT, UPDATE [OF column,.....]}
ON [schema.]tablename
        [REFERENCING {OLD AS old, NEW AS new}]
        [FOR EACH ROW [WHEN condition]]

DECLARE
                Variable declarations;
                Constant declarations;
BEGIN
        PL/SQL subprogram body;
EXCEPTION
                Exception PL/SQL block;
END;
```

Database Triggers

OR REPLACE	Recreates the trigger if it already exists. This option can be used to change the definition of an existing trigger without first dropping it.
Schema	Is the schema to contain the trigger. If the schema is omitted, the Oracle engine creates the trigger in the users own schema.
Triggername	Is the name of the trigger to be created
BEFORE	Oracle engine fires the trigger before executing the triggering statement.
AFTER	Oracle engine fires the trigger after executing the triggering statement.
DELETE	Oracle engine fires the trigger whenever a DELETE statement removes a row FROM the table.
INSERT	Indicates that the Oracle engine fires the trigger whenever a INSERT statement adds a row to table.
UPDATE	Indicates that the Oracle engine fires the trigger whenever an UPDATE statement changes a value in one of the columns specified in the OF clause. If the OF clause is omitted, the Oracle engine fires the trigger whenever a UPDATE statement changes a value in any column of the table.

Database Triggers

ON	Specifies the schema and name of the table, which the trigger is to be created. If schema is omitted, the Oracle engine assumes the table is in the user's own schema. **A trigger cannot be created on a table in the schema SYS.**
REFERENCI-NG	Specifies correlation names. Correlation names can be used in the PL/SQL block and WHEN clause of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. **If the row trigger is associated with a table named OLD or NEW, this clause can be used to specify different correlation names to avoid confusion between table name and he correlation name**.
FOR EACH ROW	Designates the trigger to be a row trigger. The Oracle engine fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the when clause. If this clause is omitted the trigger is a statement trigger
WHEN	Specifies the trigger restriction. The trigger restriction contains a SQL condition that must be satisfied for the Oracle engine to fir the trigger. This condition must contain correlation names and cannot contain a query. **Trigger restriction can be specified only for the row triggers.** The Oracle engine evaluates this condition for each row affected by the triggering statement.
PL/SQL Block	is the PL/SQL block that the Oracle engine executes when the trigger is fired. **The PL/SQL block cannot contain transaction control SQL statements (COMMIT, ROLLBACK AND SAVEPOINT**)

Database Triggers

- Applying Triggers
  - Triggering Event
    - It can be Insert, Update or Delete statement for a table
  - Trigger Constraint (Optional)
    - A boolean expression for each row trigger specified using a WHEN clause
  - Trigger Action
    - PL/SQL code to be executed when a triggering statement is encountered

Database Triggers

- Types of Triggers
  - The 'time' when the trigger fires
    - BEFORE trigger (before the triggering action).
    - AFTER trigger (after the triggering action)
    - INSTEAD OF trigger (for Views)
  - The 'item' the trigger fires on
    - Row trigger: once for each row affected by the triggering statement
    - Statement trigger: once for the triggering statement, regardless of the number rows affected

Database Triggers

BEFORE triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete.

By using a BEFORE trigger, you can eliminate unnecessary processing of the triggering statement.

BEORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

Database Triggers

- Expressions in Triggers
  - Help in referring to values in row triggers.
  - Need to use :OLD and :NEW prefixes.

```
If :NEW.column_name < :OLD.column_name……
```

Database Triggers

- Conditional Predicates

  - Useful when the trigger fires more than one type of DML operation.
  - Need to use the INSERTING, UPDATING or DELETING clause.
  - These are pre – defined PL/SQL Boolean type variables which evaluate to either true or false.

```
IF DELETING ('column_name') THEN……
```

Database Triggers

```
SQL> create trigger reorder
            /* triggering event */
            after update of qty_on_hand on inventory  -- table
            for each row
            /* trigger constraint */
            when (new.reorderable = 't')
    begin
            /* trigger action */
            if :new.qty_on_hand < :new.reorder_point then
                    insert into pending_orders
                    values (:new.part_no, :new.reorder_qty, sysdate);
            end if;
    end;
```

Database Triggers

```
SQL> create table sal_raise(emp_code NUMBER(5),
            old_sal  NUMBER(6),new_sal NUMBER(6),
            Change_time date);

     create or replace trigger sal_record
     after update on employee
     for each row
     begin
            dbms_output.put_line('Trigger fired...');
            insert into sal_raise
            values(:old.emp_code, :old.salary, :new.salary,
            sysdate);
     end;
```

Database Triggers

•Mutating Trigger Example:-

```
SQL> create or replace trigger total_salary
        after delete or insert or update
        of dept_code, salary on employee
        for each row
    begin
         --assume dept_code, salary are non-null
        if (deleting) or (updating and
        :old.dept_code <> :new.dept_code) then
            update employee
            set salary = salary -  :old.salary
            WHERE dept_code= :old.dept_code;
        end if;
```

Database Triggers

```
        if (inserting) or (updating and
        :old.dept_code <> :new.dept_code) then
                update employee set salary = salary + :new.salary
                 WHERE dept_code = :new.dept_code;
        end if;
        if (updating) and (:old.dept_code = :new.dept_code)
        and (:old.salary <> :new.salary) then
                update employee
                set salary = salary + (:new.salary - :old.salary)
                WHERE dept_code = :old.dept_code;
        end if;
    end;
```

Recompiling and Dropping Triggers

- Recompiling a trigger
  - Just like procedures, functions and packages, triggers can be recompiled.
  - ALTER TRIGGER trigger_name COMPLIE;

- Dropping a trigger
  - DROP TRIGGER trigger_name

Module 23. Records and Tables

- Overview
  - Introduction to PL/SQL Records
  - Defining and Declaring PL/SQL Records
  - Initializing, referencing, assigning and comparing Records
  - PL/SQL table of records

PL/SQL Records and Tables

- PL / SQL Records
  - A PL/SQL Record provides the means of defining a programming a structure which is a set of variable types.
  - Record types map to a stored definition of the structure.
  - A record type is a programming structure that mirrors a single row in a table.
  - The attribute %ROWTYPE lets you declare a record that represents a row in a database table.

type type_name **is record** (field_declaration[, field_declaration]...);

field_name **field_type** [[NOT NULL] {:= | DEFAULT} expression]

PL/SQL Records and Tables

```
SQL> declare
        type deptrec is record
          ( dept_id   dept.deptno%type,
            dept_name varchar2(15),
            dept_loc  varchar2(15)
          );
```

PL/SQL Records and Tables

```
SQL> declare
        type timerec is record
          ( seconds smallint,
            minutes smallint,
            hours   smallint);

        type flightrec is record
          ( flight_no   integer,
            plane_id     varchar2(10),
            captain      employee,  -- declare object
            passengers   passengerlist,  -- declare varray
            depart_time  timerec,  -- declare nested record
            airport_code varchar2(10));
```

PL/SQL Records and Tables

```
SQL> declare
        type emprec is record
            ( emp_id    integer
             last_name varchar2(15),
             dept_num  integer(2),
             job_title varchar2(15),
             salary real(7,2));
```

FUNCTION nth_highest_salary (n INTEGER) **RETURN EmpRec** IS ...

```
SQL> declare
        type stockitem is record
           ( item_no     integer(3),
            description varchar2(50),
            quantity    integer,
            price       real(7,2));
            item_info stockitem;  -- declare record
```

PL/SQL Records and Tables

```
SQL> declare
        type emprec is record
           ( emp_id    emp.empno%type,
             last_name varchar2(10),
             job_title varchar2(15),
             salary    number(7,2));
                ...
        procedure raise_salary (emp_info emprec);

SQL> declare
        type timerec is record
           ( seconds smallint := 0,
             minutes smallint := 0,
             hours   smallint := 0);
```

PL/SQL Records and Tables

```
SQL> declare
        type stockitem is record
            ( item_no integer(3) not null := 999,
              description varchar2(50),
              quantity integer,
               price real(7,2));
```

record_name.field_name

emp_info.hire_date ...

- To refer to fields in the record that is returned by a function:

    function_name(parameters).field_name

PL/SQL Records and Tables

```
SQL> declare
        type emprec is record(emp_id    number(4),job_title char(14),salary
        real);
         middle emprec;
        middle_sal real;
        function nth_highest_sal (n integer) return emprec      is
                emp_info emprec;
                begin
                        emp_info.emp_id:=10;
                        emp_info.job_title:='Working';
                        emp_info.salary:=1000;
                        return emp_info;  -- return record
                end;
        begin
                middle := nth_highest_sal(10);--call function
                middle_sal :=nth_highest_sal(10).SALARY; --call function
                dbms_out.put_line(middle_sal);
        end ;
```

PL/SQL Records and Tables

```
SQL> declare
        type timerec is record (
                minutes smallint,
                hours   smallint);
        type agendaitem is record (
                priority integer,
                 subject  varchar2(100),
                duration timerec);
        function item (n integer) return agendaitem is
                 item_info agendaitem;
        begin
        ...
        return item_info;  -- return record
        end;
    begin
        ...
        if item(3).duration.minutes > 30 then ...  -- call function
```

PL/SQL Records and Tables

```
SQL> declare
            type flightrec is record (
                  flight_no   integer,
                  plane_id     varchar2(10),
                  captain      employee,  -- declare object
                  passengers   passengerlist,  -- declare varray
                  depart_time  timerec,  -- declare nested record
                  airport_code varchar2(10));
      flight flightrec;
      begin
   .    ..
       if flight.captain.name = 'H Raowlings' then ...
```

PL/SQL Records and Tables

```
record_name.field_name := expression;

emp_info.ename := UPPER(emp_info.ename);
```

PL/SQL Records and Tables

```
SQL>declare
                type deptrec is record (
                        dept_num  number(2),
                        dept_name char(14),
                        location  char(13));
                 type deptitem is record (
                        dept_num  number(2),
                        dept_name char(14),
                        location  char(13));
        dept1_info deptrec;
        dept2_info deptitem;
     begin
      ...
        dept2_info := dept1_info;  -- illegal; different datatypes
```

PL/SQL Records and Tables

```
SQL> declare
        type deptrec is record (
            dept_num  number(2),
            dept_name char(14),
            location  char(13));
            dept_info deptrec;
   begin
        SELECT  deptno, dname, loc into dept_info FROM dept
WHERE                          ...
```

```
insert into dept values (dept_info);  -- illegal
record_name := (value1, value2, value3, ...);  -- illegal
```

PL/SQL Records and Tables

```
SQL> Declare
        type timerec is record (minutes smallint, hours smallint);
        type meetingrec is record (
                day     date,
                time    timerec,  -- nested record
                room_no integer(4));
        type partyrec is record (
                day   date,
                time  timerec,  -- nested record
                place varchar2(25));
        seminar meetingrec;
        party   partyrec;
     Begin
      ...        party.time := seminar.time;/*Allowed*/
```

PL/SQL Records and Tables

- Records cannot be tested for nullity, equality, or inequality.

```
SQL> begin
        ...
        if emp_info is null then ...  -- illegal
         if dept2_info > dept1_info then ...  -- illegal
```

PL/SQL Records and Tables

- Collect accounting figures FROM database tables assets and liabilities
- Then use ratio analysis to compare the performance of two subsidiary companies

```
SQL> declare
        type figuresrec is record (cash real, notes real, ...);
        sub1_figs figuresrec;
        sub2_figs figuresrec;
        ...
        function acid_test (figs figuresrec) return real is ...
    begin
        SELECT  cash, notes, ... into sub1_figs FROM assets, liabilities
        WHERE assets.sub = 1 and liabilities.sub = 1;
        SELECT  cash, notes, ... into sub2_figs FROM assets, liabilities
         WHERE assets.sub = 2 and liabilities.sub = 2;
        if acid_test(sub1_figs) > acid_test(sub2_figs) then ...
        ...
    end;
```

PL/SQL Records and Tables

```
SQL> create package emp_actions
        as  -- specification
        type emprectyp is record (empid number, sal number);
        cursor desc_salary return emprectyp;
        procedure hire_employee
        (           empid number,
                    ename varchar2,
                    dept varchar2,
                    grade varchar2,
                    sal number
        );
        procedure fire_employee (empid number);
        end emp_actions;
```

PL/SQL Records and Tables

```
SQL> create package body emp_actions
        as  -- body
        cursor desc_salary return emprectyp
        is
        select  empno, sal from emp order by sal desc;
        procedure hire_employee
        (           empid number,
                    ename varchar2,
                    dept varchar2,
                    grade varchar2,
                    sal number
        )
        is
```

PL/SQL Records and Tables

```
        begin
                insert into employee(emp_code, emp_name,
                        dept_code, grade, salary)
                values(empid,ename,dept,grade,sal);

        end hire_employee;
        procedure fire_employee (empid number) is
        begin
                delete from employee where emp_code = empid;
        end fire_employee;
    end emp_actions;
```

PL/SQL Records and Tables

- PL / SQL Tables
  - A PL/SQL table is a one-dimensional, unbounded collection of homogeneous elements, indexed by integers. It is like an array.

```
TYPE <table_name> IS TABLE OF <datatype> [NOT NULL]
        INDEX BY BINARY INTEGER;

type company_keys_tabtype is table of company.company_id%type not
null index by binary integer;

type reports_requested_tabtype is table of varchar2 (100)
index by binary integer;
```

PL/SQL Records and Tables

<table_name> <table_type>

SQL> declare

```
        type countdown_tests_tabtype is table of varchar2 (20)
        index by binary_integer;
        countdown_tests_list     countdown_tests_tabtype;

    begin
        countdown_tests_list (1) := 'all systems go';
        countdown_tests_list (2) := 'internal pressure';
        countdown_tests_list (3) := 'engine inflow';
    end;
```

PL/SQL Records and Tables

```
type  local_emp_table is table of employee%rowtype
          index by binary_integer;
```

```
cursor emp_cur is SELECT  * FROM  employee;

type cursor_emp_table is table of emp_cur%rowtype
          index by binary_integer;
```

```
type emp_rectype
is
record (employee_id integer, emp_name varchar2(60)) ;

type emp_table
is
table of emp_rectype index by binary_integer;
```

PL/SQL Records and Tables

```
<table_name>(<index_ expression).<field_name>
emp_tab(375).emp_name := 'SALIMBA';
```

Operator	**Description**
COUNT	Returns the number of elements currently contained in the PL/ SQL table.
DELETE	Deletes one or more elements FROM the PL /SQL table.
EXISTS	Returns FALSE if a reference to an element at the specified index would raise the no_data_found exception.
FIRST	Returns the smallest index of the PL/SQL table for which an element is defined.
LAST	Returns the greatest index of the PL/SQL table for which an element is defined.
NEXT	Returns the smallest index of the PL/SQL table containing an element which is greater than the specified index.
PRIOR	Returns the greatest index of the PL/SQL table containing an element which is less than the specified index.

PL/SQL Records and Tables

- An operation which takes no arguments

  <table name>.<operation>
- An operation which takes a row index for an argument.

  <table name>.<operation>(<index number> [, <index number])

- Total_rows := emp_table.COUNT;
- Names_tab.DELETE;
- IF seuss_characters_table.EXISTS(1) THEN……
- First_entry_row := employee_table.FIRST;
- Last_entry_row := employee_table.LAST;
- Next_index := employee_table.NEXT (curr_index);
- Prev_index := employee_table.PRIOR (curr_index);
- Next_index := employee_table.NEXT (curr_index);

Module 24. ORDBMS

- Overview
  - Features of object-oriented programming
  - Advantages of object orientation
  - Creating abstract data types
  - Creating methods
  - Retrieving information about objects
  - Creating tables using an abstract data types
  - Inserting records into tables
  - Constructor methods
  - selecting columns, object attributes, and methods FROM object tables
  - Comparing objects with map and order methods
  - Inserting data using constructor methods

ORDBMS

- Features of Object-Oriented programming
  - Encapsulation
  - Inheritance
  - Polymorphism

- An object type is a user-defined composite data type.

- Object types are database objects in Oracle

ORDBMS

- Advantages of Object Orientations
  - Object Reuse
  - Standard Adherence
  - Defined Access Path

ORDBMS

- Creating Abstract Data types:

SQL> **create or replace type** address_type **as object**

```
	(	street varchar2(50),
		city varchar2(25),
		state varchar2(2),
		zip number
	);
	/
```

ORDBMS

- Using an abstract data type within another abstract data type :

```
SQL> create or replace type person_type as object
        (       name varchar2(25),
                address address_type
        );
        /
```

ORDBMS

- Creating a table using abstract data types :

```
SQL> CREATE TABLE customer
            (customer_id number,
            person person_type
            );

SQL > desc  customer

NAME                    NULL ?          TYPE
------------------------------------------------------------
CUSTOMER_ID                             NUMBER
PERSON                              PERSON_TYPE
```

ORDBMS

- Inserting records into table based on abstract data types :

SQL> insert into customer values

(100,

person_type('jocksports',

address_type('345 viewridge', 'belmont', 'ca', 96711)));

ORDBMS

SQL> insert into customer value
(101,
person_ty(‘tkb sport shop’,
address_ty(‘490 boli rd.’, ‘redwood city’, ‘ca’, 94061)))

ORDBMS

- Selecting records FROM tables using abstract data types :

```
SQL> SELECT  customer_id FROM customer;

        CUSTOMER_ID
        -------------
        100
        101
```

ORDBMS

SQL> SELECT * FROM customer;

CUSTOMER_ID

\-------------

PERSON(NAME, ADDRESS(STREET,CITY,STATE, ZIP))

\---

100

PERSON_TY('JOCKSPORTS', ADDRESS_TY('345 VIEWRIDGE', 'BELMONT', 'CA',96711))

101

PERSON_TY('TKB SPORT SHOP', ADDRESS_TY('490 BOLI RD.', 'REDWOOD CITY', 'CA',94061))

SQL> SELECT name FROM customer;

ORDBMS

SQL>SELECT **a.person.name** FROM customer **a**;

```
PERSON.NAME
-------------------------------------------
JOCKSPORTS
TKB SPORT SHOP
```

SQL> SELECT **a.person.address.street** FROM customer **a**;

```
PERSON.ADDRESS.STREET
--------------------------------------------
345 VIEWRIDGE
490 BOLI RD.
```

ORDBMS

- Updating values in a table using abstract data types :

```
SQL> update customer a
        set a.person.address.city='chicago'
        WHERE a.person.address.city like 'b%';
```

ORDBMS

- Deleting records FROM tables using abstract data types :

SQL> delete FROM customer **a**
WHERE **a.person.name=**'JOCKSPORTS';

ORDBMS

- Dropping Object types :

SQL> **drop type** PERSON_TYPE;
drop type PERSON_TY
*****
ERROR at line 1”
ORA-02303:cannot drop or replace a type with type or table dependents

SQL> **drop type** person_ty **force**;
type dropped.

ORDBMS

- Implementing Object Views :
  - Refers to the ability to define object oriented objects by using the existing relational tables.
  - It allows the reuse of relational tables by using object oriented features.

ORDBMS

```
SQL> desc employee

 Name                                      Null?    Type
 ----------------------------------------- -------- ---------------
 EMP_CODE                                           NUMBER(5)
 EMP_NAME                                  NOT NULL VARCHAR2(25)
 DEPT_CODE                                          VARCHAR2(4)
 GRADE                                              VARCHAR2(2)
 AGE                                                NUMBER(2)
 DATE_JOIN                                          DATE
 SEX                                                VARCHAR2(1)
 SALARY                                             NUMBER(6)
 MARRIED                                            VARCHAR2(1)
 REPORTS_TO                                         NUMBER(5)
```

ORDBMS

- Creating an abstract data type :

SQL> **create or replace type** other_ty **as object**
(dept_code varchar2(4),
grade varchar2(2),
age number(2),
date_join date,
sex varchar2(1),
salary number(6),
married varchar2(1),
reports_to number(5));
SQL> **create or replace type** name_ty **as object**
(emp_name varchar2(25),
other other_ty);

ORDBMS

- Creating a final abstract data type :

```
SQL> create or replace type emp_ty as object
                    (emp_code number(5),
                    emp_name name_ty);
```

ORDBMS

- Creating Object Views :

SQL> **create or replace view** emp_ov(emp_code,emp_name) **as**
(SELECT emp_code, name_ty(emp_name,
other_ty(dept_code,grade,age,date_join,sex,salary,
married,reports_to))
FROM employee);

SQL> **create or replace view** emp_ov(emp_code,emp_name) **as**
(SELECT emp_code, name_ty(emp_name,
other_ty(dept_code,grade,age,date_join,sex,salary,
married,reports_to))
FROM employee
WHERE dept_code='FIN');

ORDBMS

- Benefits of Object Views:
  - ➢ Reuse of existing relational table
  - ➢ Allow data manipulation in two different ways
    - •Relational table
    - •Object table.

ORDBMS

- Manipulating data via object views :

SQL> **insert into employee**
values
(36,'Arun Nair','TRNG','M3',27,'10-MAR-08','F',8000,'Y',25);

SQL> **insert into emp_ov**
values
(100,
name_ty('Ritika Chauhan',
other_ty('FIN','M2',45,'10-MAR-08','F',12000,'Y',16)
));

ORDBMS

- Methods :

```
SQL> Create or replace type ADDRESS as object
        (street1 varchar2(20),
         Street2 varchar2(20),
        City varchar2(20),
        State varchar2(2),
        Zip_code varchar2(5),
        Phone varchar2(10));

SQL>  Function address (streeet1 in varchar2,
                        Street2 in varchar2,
                        City in varchar2,
                        State in varchar2,
                        Zip_code varchar2,
                        Phone in varchar2)  Return address
```

ORDBMS

```
CREATE TYPE type_name {IS | AS} OBJECT (
        attribute_name datatype [, attribute_name data type]
        [{MAP|ORDER} MEMBER function function_specification,]
        [MEMBER function function_specification,]
        [MEMBER procedure procedure_specification]
        restrict_references_pragma);
```

```
CREATE TYPE BODY type_name {IS | AS}
      [MAP | ORDER] MEMBER function_body
      [MEMBER] function function_body
      [MEMBER] procedure procedure_body
END;
```

ORDBMS

```
SQL> Create or Replace type ADDRESS
as object
(street1 varchar2(20),
street2 varchar2(20),
city varchar2(20),
state varchar2(2),
zip_code varchar2(5),
phone varchar2(10),
Member procedure changeadd(st1 in varchar2,
        st2 in varchar2,
        ct in varchar2,
        stat in varchar2,
        zip in varchar2),
Member function getstreet(line_no in number) return varchar2,
Member function getcity return varchar2,
Member function getstat return varchar2,
Member function getphone return varchar2,
Member procedure setphone (newphone in varchar2));
```

ORDBMS

```
SQL> Create or Replace Type Body ADDRESS as
Member Procedure Changeadd(st1 varchar2,
   st2 varchar2,
   ct varchar2,
   stat varchar2,
   zip varchar2) is
Begin
If (st1 is null)  or  (st2 is null) or (ct is null) or (stat is null) or (zip is null) or  (upper(stat) not in
('US','UK','CA')) or
zip <> LTRIM(TO_CHAR(TO_NUMBER(ZIP),'09999'))
Then
RAISE_APPLICATION_ERROR(-20001,'INVALID DATA');
Else
        street1 := st1;
        street2 := st2;
        city := ct;
        state := UPPER(stat);
        zip_code := zip;
        End If;
End;
```

ORDBMS

```
-------Function GetStreet
Member Function GetStreet (line_no number) return varchar2 is
Begin
If line_no =1 then
                    Return street1;
Elsif line_no = 2 then
                    Return street2;
Else----If there is no street in the database then return nothing.
                    Return ' ' ;
End If;
End;
-----------Function GetCity
Member Function Getcity return varchar2 is
Begin
          Return city;
End;
----Function GetStat
Member Function GetStat return varchar2 is
Begin
          Return state;
End;
```

ORDBMS

-----------Function GetPhone
Member Function GetPhone return varchar2 is
Begin
    Return phone;
End;
-------Procedure Setphone
Member Procedure SetPhone(Newphone varchar2) is
Begin
    Phone := Newphone;
End;
End;
/
Type body created.

ORDBMS

- Modifying object type by adding new member function :

```
SQL> ALTER TYPE <TYPE NAME> REPLACE AS OBJECT
        (MEMBER FUNCTION <FUNCTION NAME> RETURN CHAR);

SQL> ALTER TYPE <TYPE NAME> COMPILE;

SQL> ALTER TYPE <TYPE NAME> COMPILE BODY;
```

ORDBMS

- The ORDER method :

Return value	Meaning
-1	Self is less than the argument
0	Self is equal to the argument.
1	Self is greater than the argument.

ORDBMS

SQL> Create or replace TYPE dept_ty as object
(deptno number(2),
Dname varchar2(15),
Loc varchar2(20),
ORDER MEMBER FUNCTION order_dept(d dept_ty)
Return number);

ORDBMS

```
SQL> Create or replace type body dept_ty as
        Order member function order_dept (d dept_ty) return number is
                Retval number(2) :=1;
                Begin
                if self.deptno <d.deptno Then
                        retval:= -1;
                Elsif self.deptno=d.deptno Then
                        retval:= 0;
                Elsif self.deptno>d.deptno Then
                        retval:= 1;
                End if;
                return retval;
                End order_dept;
        End;
/
```

```
SQL> CREATE TABLE emp16
        (empno number(4),
        Ename varchar2(15),
        Job varchar2(20),
        Hiredate date,
        Sal number(10,2),
        Comm number(7,2),
        Dept dept_ty);
```

ORDBMS

```
insert into emp16  values
(1,'SMITH','PHYSICIAN','20-SEP-90',115000,NULL,
DEPT_TY(1,'PEDIATRICS', 'ROCHSTER')
) ;
```

```
SELECT  * FROM  emp16;
EMPNO  ENAME   JOB     HIREDATE     SAL   COMM
-----  ------ -------  --------    ----- -------
DEPT(DEPTNO, DANAME, LOC)
-------------------------- ------------------
1          SMITH    PHYSICIAN 20-SEP-90  115000
DEPT_TY(1,'PEDIATRICS', ROCHSTER'
```

ORDBMS

```
SQL> declare
        obj1 dept_ty;
        obj2 dept_ty;
    begin
        SELECT  a.dept into obj1 FROM emp16 a WHERE a.empno = 1;
        SELECT  a.dept into obj2 FROM emp16 a WHERE a.empno = 1;
        if obj1 < obj2 then
            dbms_output.put_line(‘obj1 < obj2’);
        end if;
        if obj1 = obj2 then
            dbms_output.put_line(‘obj1 = obj2’);
        end if;
        if obj1 > obj2 then
            dbms_output.put_line(‘obj1 > obj2’);
        end if;
    end;
    /
OBJ1=OBJ2
PL/SQL procedure successfully completed.
```

ORDBMS

- **Can declare a map method or an order method but not both**
- An object type can contain only one map method
- Must be a parameter less function with one of the following scalar return types : DATE, NUMBER, VARCHAR2, ANSI SQL type such as CHARACTER OR REAL.

```
SQL> Create or Replace type DEPT_TY as object
        (deptno number(2),
        dname varchar2(15),
        loc varchar2(20),
        map member function MAP_DEPT return number
         );
```

ORDBMS

- MAP and ORDER methods will be used by ORACLE in ORDER BY clause of SELECT statement.
- If order or map method is not present then ORDER BY ABSTRACT_DATA_COLUMN is not possible.

```
SQL> create or replace type body dept_ty as
    map member function map_dept return number
    is
      begin
              return self.deptno;-- return a scalar datatype.
      end map_dept;
    end;
```

ORDBMS

- To call a package function or methods from a SQL statement, the PRAGMA RESTRICT_REFERENCES must follow the function declaration in the package specification or in the method specification.
- This is used to assure that the function or method does not modify data
- PRAGMA RESTRICT_REFERENCES

```
PRAGMA RESTRICT_REFERENCES({DEFAULT | method name},
{RNDS | WNDS | RNPS | WNPS}[, RNDS | WNDS | RNPS | WNPS}]…);
```

WNDS	Write no database state.
RNDS	Read no database state.
WNPS	Write no package state.
RNPS	Read no package state

ORDBMS

- Creating emp_ty data type :

SQL> create or replace type emp_ty as object

```
(ename varchar2(10),
job varchar2(20),
sal number(7,2),
comm number(7,2),
member function tot_sal return number,
pragma restrict_references (tot_sal, wnds)
);
```

- Creating emp_ty body :

SQL> create or replace type body emp_ty

```
as
member function tot_sal return number is
begin
        return (nvl(sal,0) + nvl(comm,0));
end;
end;
```

ORDBMS

- Creating an emp table and assigning emp_ty to a column :

```
SQL> CREATE TABLE emp01
(empno number(4),
emp_det emp_ty
);
```

- Inserting and selecting rows of emp table :

```
SQL> insert into emp01 values(7839,emp_ty('king','president',5000,null));
SQL> insert into emp01 values (7900, emp_ty ('smith', 'salesman', 3500,
         100));
SQL> SELECT  a.emp_det.tot_sal(  ) FROM emp01 a;
         a.emp_det.tot_sal( )
         ---------------------
                    5000
                    3600
```

ORDBMS

PRAGMA RESTRICT_REFERENCES (DEFAULT, WNDS, WNPS)

- The pragma applies to all the member functions including the system defined constructor.

- A non-default pragma overrides the default pragma and can apply to only one method

- Among overloaded methods, the pragma always applies to the nearest method

Module 25. Varying Arrays & Nested Tables

- Overview
  - ➢ Introduction to Collections.
  - ➢ Introduction to Varrays.
  - ➢ Creating Varrays
  - ➢ Introduction to Nested tables
  - ➢ Creating nested tables
  - ➢ Manipulating data using Varrays and nested tables
  - ➢ Altering Varrays and nested tables

Varying Arrays & Nested Tables

- What are Collections?
  - Are group of elements of the same type.
  - Are similar to conventional arrays.
  - Help you to manipulate data.
  - Collections are of following types:
    - Varying Arrays (Varrays)
    - Nested Tables
    - PL/SQL Tables (Associative Arrays)

Varying Arrays & Nested Tables

- Introduction to Varrays
  - Are collection of homogenous elements.
  - Help to store repeating attributes of a record in a single row.
  - Referencing individual elements done through subscripts.

- Creating Varrays :

```
SQL> create or replace type DESG_TY as object
        (desg_name varchar2(10)
        );
```

SQL > **Create or replace type DESG_VA as varray(5) of DESG_TY**;

Varying Arrays & Nested Tables

- Creating a table that uses the varray desg_va :

```
SQL > CREATE TABLE dep_des
          ( dname varchar2(25) primary key,
          desg desg_va
          );

SQL > desc dep_des;
Name          Null?               Type
-------       -------             ------
DNAME         NOT NULL            VARCHAR2(25)
DESGS                             DESGS_VA

SQL> desc user_types;
```

Varying Arrays & Nested Tables

```
SQL> SELECT  coll_type,elem_type_owner,elem_type_name,upper_bound,length
         FROM user_coll_types WHERE type_name = 'DESG_VA';

COLL_TYPE        ELEM_TYPE_OWNER              ELEM_TYPE_NAME
-----------      ---------------------------  ---------------------
UPPER_BOUND      LENGTH
-----------------  ----
VARRYING ARRAY SCOTT                                   DESG_TY
         5
SQL> SELECT  * FROM  user_type_attrs WHERE type_name = 'DESG_TY';
TYPE_NAME                 ATTR_NAME          ATTR_TY  ATTR_TYPE_OWNER
------------------        ------------------
ATTR_TYPE_NAME  LENGTH PRECISION             SCALE
-------------------------------     -----
CHARACTER_SET_NAME
------------------
DESG_TY DESG_NAME
VARCHAR2                  10
CHAR_CS
```

Varying Arrays & Nested Tables

- Inserting values into Varray :

SQL> insert into **dep_des** values

```
        ('production', desg_va(desg_ty('manager'),
                               desg_ty('asst.mgr.'),
                               desg_ty('sr. engr.'),
                               desg_ty('jr. engr.'),
                               desg_ty(null)));
```

Varying Arrays & Nested Tables

- Selecting data from Varray :

```
SQL> declare
            cursor c1 is
            SELECT  * FROM  dep_des;
      begin
            for x in c1
            loop
                  dbms_output.put_line('department : '|| x.dname);
                  for i in 1..x.desg.count
                  loop
                  dbms_output.put_line(x.desg(i).desg_name);
                  end loop;
            end loop;
      end;
```

Varying Arrays & Nested Tables

- The output of the previous PL/SQL block is as follows :

```
Department : PRODUCTION
MANAGER
ASST. MGR
SR. ENGR.
JR. ENGR
PL/SQL procedure successfully completed.
```

Varying Arrays & Nested Tables

- Updating Varrays :

```
SQL> declare
        desigs desg_va := desg_va(desg_ty('manager'),
                                  desg_ty('ast. mgr'),
                                  desg_ty('sr. engr'),
                                  desg_ty('jr. engr'),
                                  desg_ty('trainee'));
     begin
             update dep_des
             set desg = desigs
             WHERE dname ='PRODUCTION';
     end;
```

Varying Arrays & Nested Tables

Varying Arrays & Nested Tables

- Introduction to Nested Tables
  - Is a table within another table
  - It is represented as a column within another table.
  - It is unbounded, unlike Varrays.
  - Known as “out-of-line” storage.
  - We can have multiple rows in nested table for each row in the main table.

Varying Arrays & Nested Tables

- Creating Nested Tables

```
SQL> CREATE TYPE emp01_ty as object
        (Empno number(4),
        Ename varchar2(30));
```

SQL > **CREATE TYPE emps_nt as table of emp01_ty;**

```
SQL > CREATE TABLE depts
        (deptno number(2),dname varchar2(15),
        emps emps_nt)
        nested table emps store as emps_nt_tab;
```

Varying Arrays & Nested Tables

- Inserting records in Nested Tables :

SQL > insert into depts values
(10,'research', **emps_nt(emp01_ty**(1000,'ARJUN'),
emp01_ty(1001,'KRISHNA'),
emp01_ty(1002,'MOHINI')));

- To view the structure of the table :

SQL> desc user_tab_columns;

Varying Arrays & Nested Tables

- Steps to select records from nested table:
  - It is essential to know the structure of the table
  - In order to select columns FROM the nested table, you first have to flatten the table
  - The THE function is used for this purpose

- SELECT the nested table column from the main table.

  SQL> SELECT **emps** FROM depts;

- Enclose this query within the THE function.

  THE (SELECT emps FROM depts)

- Make use of the above query enclosed in the THE function as though it were a table.

  SQL > **SELECT NT.empno, NT.ename FROM**

  The (SELECT emps FROM depts) NT;

Varying Arrays & Nested Tables

- The **THE** function

  - To perform inserts and updates directly against the nested table, use the **THE** function.

```
SQL > insert into
        the (SELECT  emps FROM depts
              WHERE deptno = 10) values(emp01_ty(1003,'RADHA'));

SQL > update the (SELECT  emps FROM depts WHERE deptno = 10)
        Set ename = 'MEERA'
        WHERE ename = 'RADHA';
```

Varying Arrays & Nested Tables

- Performing inserts based on queries :

To insert a record in your main table using the existing portion of the nested table,

- Use Cast allows to model the result of a query as a nested table
- Use multiset allows the cast query to contain multiple records.

SQL > insert into depts values
(20,'EDP',
cast (multiset (SELECT * FROM
the(SELECT emps FROM depts WHERE deptno = 10) NT
WHERE NT.ename = 'KRISHNA') as emps_NT));--this data type is of the nested table in which the record is to be inserted.

Varying Arrays & Nested Tables

SQL > SELECT * FROM

The (SELECT emps FROM depts WHERE deptno = 20) NT;

```
EMPNO      ENAME
---------  --------------------
10001      KRISHNA
```

Varying Arrays & Nested Tables

Collection Methods	Description
COUNT	Returns the number of elements that a collection currently contains. For varrays count always equals last. For nested tables, if elements are deleted count becomes smaller than last.
EXISTS(n)	Returns FALSE if a reference to an element at the specified index would raise the no_data_found exception.
FIRST	Returns the smallest index number in a collection. for which an element is defined. If collection is empty returns null.For varrays always 1. For nested tables 1 if elements are not deleted from the beginning
LAST	Returns the greatest index number in a collection for which an element is defined. If collection is empty returns null. null.For varrays always equals count. For nested tables equals count if elements are not deleted.

Varying Arrays & Nested Tables

Collection Methods	Description
PRIOR	Prior (n) returns the index number that precedes index n in a collection If n has no predecessor, prior (n) returns null.
NEXT	Next (n) returns the index number that succeeds index n. if n has no successor, next (n) returns null.
EXTEND	To increase the size of a collection use extend. Extend appends one null element to a collection. Extend (n) appends n null elements to a collection. Extend n, i appends n copies of the ith element to a collection.
TRIM	Trim removes one element FROM the end of a collection. Trim (n) removes n elements FROM the end of a collection.
DELETE	Delete removes all elements FROM a collection. Delete (n) removes the nth element FROM a nested table. If n is null, delete (n) does nothing.

Varying Arrays & Nested Tables

- Use varrays in case of data set with limited number of entries

- Use nested tables if the number of entries is unlimited

- As the size of collectors increases performance problems are faced as they cannot be indexed

- In such cases it is always better to use separate relational table.

Module 26 : More on Records and Collections

- Overview
  - Inserting PL/SQL records into the database
  - Updating the database with PL/SQL record values
  - Restrictions on record inserts / updates
  - Querying data into collection of records
  - Associative arrays

More on Records and Collections

- Inserting PL/SQL records into the database :

  - PL/SQL allows the use of %ROWTYPE to insert records into the database.

More on Records and Collections

```
SQL> declare
        dept_info department%rowtype;
     begin
     -- dept_code, dept_name are the table columns.
     -- the record picks up these names FROM the %rowtype.
        dept_info.dept_code := 'new';
        dept_info.dept_name := 'newdept';
     -- using the %rowtype means we can leave out the column list
      -- (dept_code, dept_name) FROM the insert statement.
        insert into department values dept_info;
     end;
```

More on Records and Collections

- Updating the database using PL/SQL record values :
  - The keyword ROW is allowed only on the left side of a SET clause.
  - The argument to SET ROW must be a real PL/SQL record, not a subquery that returns a single row.
  - The record can also contain collections or objects.

```
SQL>  declare
            dept_info department%rowtype;
      begin
            dept_info.dept_code := 'new';
             dept_info.dept_name := 'newdepartment';
      -- The row will have values for the filled-in columns, and null
      -- for any other columns.
      update department set row = dept_info WHERE dept_code =
            'new';
      end;
```

More on Records and Collections

- Using the RETURNING clause with a record :
  - Returns column values FROM the affected row into a PL/SQL record

```
SQL>  declare
        type emprec is record (emp_name employee.emp_name%type,
                               salary employee.salary%type);
        emp_info emprec;
        emp_id number := 10;
      begin
        update employee set salary = salary * 1.1
        WHERE emp_code = emp_id
        returning emp_name, salary into emp_info;
        dbms_output.put_line('just gave a raise to ' ||
        emp_info.emp_name ||  ', who now makes ' || emp_info.salary);
        rollback;
      end;
```

More on Records and Collections

- Restrictions on record Inserts / Updates :

  - Record variables are allowed only in the following places:
    - On the right side of the SET clause in an UPDATE statement
    - In the VALUES clause of an INSERT statement
    - In the INTO subclause of a RETURNING clause

  - Record variables are not allowed in a SELECT list, WHERE clause, GROUP BY clause, or ORDER BY clause.

  - The following are not supported:
    - Nested record types
    - Functions that return a record
    - Record inserts/updates using the EXECUTE IMMEDIATE statement.

More on Records and Collections

- Querying data into collection of records :
  - Use the BULK COLLECT clause with a SELECT INTO or FETCH statement to retrieve a set of rows into a collection of records

```
SQL> declare
        type employeeset is table of employee%rowtype;
        underpaid employeeset;-- holds rows FROM employee table.
        cursor c1 is SELECT  emp_code, emp_name FROM employee;
        type nameset is table of c1%rowtype;
        some_names nameset; --holds partial rows FROM employee
                                             table.
     begin
     -- with one query, we bring all the relevant data into the collection of
        records.
        SELECT  * bulk collect into underpaid FROM employee
        WHERE salary < 2500 ORDER BY salary desc;
```

More on Records and Collections

```
-- Now we can process the data by examining the collection, or
   passing it to
-- a separate procedure, instead of writing a loop to FETCH each
   row.
    dbms_output.put_line(underpaid.count || ' people make less
                                   than 2500.');
     for i in underpaid.first .. underpaid.last
     loop
         dbms_output.put_line(underpaid(i).emp_name || ' makes '
                                       || underpaid(i).salary);
      end loop;
```

More on Records and Collections

```
-- We can also bring in just some of the table columns.
-- Here we get the first and last names of 10 arbitrary employees.
      SELECT  emp_code, emp_name bulk collect into
some_names           FROM employee WHERE rownum < 11;
      for i in some_names.first .. some_names.last
      loop
            dbms_output.put_line('employee = ' ||
            some_names(i).emp_code || ' ' ||
            some_names(i).emp_name);
      end loop;
end;
```

More on Records and Collections

- Associative Arrays are sets of key-value pairs
  - WHERE each key is unique and is used to locate a corresponding value in the array.
  - The key can be an integer or a string.

```
SQL> declare
        type population_type is table of number index by
                                            varchar2(64);
        country_population population_type;
        continent_population population_type;
        howmany number;
        which varchar2(64);
    begin
        country_population('greenland') := 100000; --creates new entry
        country_population('iceland') := 750000;   -- creates new entry
```

More on Records and Collections

```
    -- looks up value associated with a string
howmany := country_population('greenland');
continent_population('australia') := 30000000;
continent_population('antarctica') := 1000; -- creates new entry
continent_population('antarctica') := 1001; -- replaces previous value
    -- returns 'antarctica' as that comes first alphabetically.
which := continent_population.first;
dbms_output.put_line(which);
    -- returns 'australia' as that comes last alphabetically.
 which := continent_population.last;
 dbms_output.put_line(which);
    -- returns the value corresponding to the last key
      howmany := continent_population(continent_population.last);
 end;
```

More on Records and Collections

Choosing Which PL/SQL Collection Types to Use

•Arrays in other languages become varrays in PL/SQL.

•Sets and bags in other languages become nested tables in PL/SQL.

•Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

Choosing Between Nested Tables and Associative Arrays

•Nested tables can be stored in a database column, but associative arrays cannot.

•Nested tables can simplify SQL operations WHERE you would normally join a single-column table with a larger table.

•Associative arrays are appropriate for relatively small lookup tables WHERE the collection can be constructed in memory each time a procedure is called or a package is initialized.

•Their index values are more flexible, because associative array subscripts can be negative, can be nonsequential, and can use string values instead of numbers.

More on Records and Collections

Choosing Between Nested Tables and Varrays

Varrays are a good choice when:

- The number of elements is known in advance.
- The elements are usually all accessed in sequence.

When stored in the database, varrays keep their ordering and subscripts.

Each varray is stored as a single object, either inside the table of which it is a column (if the varray is less than 4KB) or outside the table but still in the same tablespace (if the varray is greater than 4KB).

Nested tables are a good choice when:

- The index values are not consecutive.
- There is no predefined upper bound for index values.
- You need to delete or update some elements, but not all the elements at once.
- You would usually create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

- Nested tables can be sparse

More on Records and Collections

- Associative array :

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
 INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(size_limit)];
 INDEX BY key_type;
```

- An initialization clause is not allowed.
- There is no constructor notation for associative arrays.

PLS_INTEGER

- To store signed integers. Its magnitude range is -2**31 .. 2**31.
- Require less storage than NUMBER values.
- PLS_INTEGER operations use machine arithmetic, so they are faster than NUMBER and BINARY_INTEGER operations
- PLS_INTEGER and BINARY_INTEGER are not fully compatible.
- When a PLS_INTEGER calculation overflows, an exception is raised.
- When a BINARY_INTEGER calculation overflows, no exception is raised if the result is assigned to a NUMBER variable.

More on Records and Collections

- Using an Associative array :

```
SQL>  declare
          type emptabtyp is table of employee%rowtype
                                   index by pls_integer;
          emp_tab emptabtyp;
      begin
          /* retrieve employee record. */
          SELECT  * into emp_tab(10) FROM employee WHERE
          emp_code = 10;
      end;
```

More on Records and Collections

- Assigning Collections :

```
collection_name(subscript) := expression;
```

More on Records and Collections

- Data type compatibility for collection assignment :

  Collections must have the same data type for an assignment to work

```
SQL> declare
          type last_name_typ is varray(3) of varchar2(64);
          type surname_typ is varray(3) of varchar2(64);
     -- these first two variables have the same data type.
          group1 last_name_typ := last_name_typ('jones','wong','marceau');
          group2 last_name_typ := last_name_typ('klein','patsos','singh');
     -- this third variable has a similar declaration, but is not the same type.
          group3 surname_typ := surname_typ('trevisi','macleod','marquez');
     begin
     -- allowed because they have the same data type
          group1 := group2;
     -- not allowed because they have different data typess
       --   group3 := group2;
     end;
```

More on Records and Collections

- Assigning a null value to a nested table :
  - Assigning an **automically null (unintialised)**nested table or varray to a second nested table or varray
  - Assigning the value NULL to acollection.

```
SQL> declare
        type colors is table of varchar2(64);
    -- this nested table has some values.
        crayons colors := colors('silver','gold');
    -- this nested table is not initialized ("atomically null").
        empty_set colors;
    begin
    -- at first, the initialized variable is not null.
        if crayons is not null then
            dbms_output.put_line('ok, at first crayons is not null.');
        end if;
```

More on Records and Collections

```
--Then we assign a null nested table to it.
        crayons := empty_set;
        crayons := null;
-- now it is null.
        if crayons is null then
         dbms_output.put_line('ok, now crayons has become null.');
        end if;
-- we must use another constructor to give it some values.
        crayons := colors('yellow','green','blue');
end;
```

Assigning a value to a collection element can cause various exceptions:

- Subscript is null or is not convertible to the right data type, VALUE_ERROR.
- Subscript refers to an uninitialized element, SUBSCRIPT_BEYOND_COUNT.
- Collection is automically null, COLLECTION_IS_NULL.

Module 27. Bulk Binds

- Overview
  - Introduction to Bulk Binds
  - Improving performance using bulk binds
  - Querying data into collections of records
  - Using DML on collections with deleted elements
  - Using DML on selected elements in collections
  - Effects of rollback on FORALL

Bulk Binds

- The switch from PL/SQL engine to SQL engine is called context switch.
- Context switch degrades the performance of the PL/SQL block or subprogram.
- In order to avoid a context switch, bulk binding is used.
- The example on the next slide illustrates a context switch.

Bulk Binds

- To illustrate a context switch :

```
declare
        type numlist is varray(20) of number;
        depts numlist := numlist(10, 30, 70, ...); -- department numbers
begin
                ..
                for i in depts.first..depts.last loop
                        delete FROM emp WHERE depto = depts(i);
                end loop;
        end;
```

- In the example above the context switch occurs for every iteration of the loop.
- To avoid it and improve the performance of block Bulk Binding is used.
- Assigning of values to PL/SQL variables in SQL statements is called Binding.
- Binding of entire collection at once is called Bulk Binding.

Bulk Binds

- Improving the performance using bulk bind :

To bulk-bind input collections, use the FORALL statement.

```
SQL> declare
        type numlist is table of number;
        mgrs numlist := numlist(7566, 7782, ...) -- manager
                                                  numbers
     begin
         ...
        forall i in mgrs.first..mgrs.last
            delete FROM emp WHERE mgr = mgrs(i);
     end;
```

```
FORALL index IN lower_bound..upper_bound
     sql_statement;
```

Bulk Binds

Although FORALL statement contains an iteration scheme ,it is not a FOR loop.

The index can be referenced only within the FORALL statement and only as a collection subscript.

It can only repeat a single DML statement

The DML statement **can reference more than one collection**, but FORALL only improves performance WHERE the index value is used as a subscript.

All collection elements in the specified range must exist. If an element is missing or was deleted, you get an error.

The FORALL statement iterates over the index values specified by the elements of this collection

Bulk Binds

- Issuing DELETE statement in a loop :

```
CREATE TABLE employees2 as SELECT  * FROM  emp;

declare
     type numlist is varray(20) of number;
    depts numlist := numlist(10, 30, 70);  -- department codes
begin
     forall i in depts.first..depts.last
           delete FROM employees2 WHERE deptno = depts(i);
            commit;
           end;
select * from emp minus
select * from employees2;
```

Bulk Binds

Drop Table Employees2

- Issuing INSERT statements in a loop :

SQL> CREATE TABLE parts1 (pnum integer, pname varchar2(15));

SQL> CREATE TABLE parts2 (pnum integer, pname varchar2(15));

Bulk Binds

```
SQL> declare
        type numtab is table of parts1.pnum%type index by
        pls_integer;
        type nametab is table of parts1.pname%type index by
        pls_integer;
        pnums  numtab;
        pnames nametab;
        iterations constant pls_integer := 500;
        t1 integer;  t2 integer;  t3 integer;
   begin
        for j in 1..iterations loop  -- load index-by tables
                pnums(j) := j;
                pnames(j) := 'part no. ' || to_char(j);
        end loop;
```

Bulk Binds

```
        t1 := dbms_utility.get_time;
        for i in 1..iterations loop  -- use for loop
                insert into parts1 values (pnums(i), pnames(i));
        end loop;
        t2 := dbms_utility.get_time;
        forall i in 1..iterations  -- use forall statement
                insert into parts2 values (pnums(i), pnames(i));
         t3 := dbms_utility.get_time;
        dbms_output.put_line('execution time (secs)');
        dbms_output.put_line('---------------------');
        dbms_output.put_line('for loop: ' || to_char((t2 - t1)/100));
        dbms_output.put_line('forall:   ' || to_char((t3 - t2)/100));
        commit;
end;
```

Bulk Binds

SQL> DROP TABLE parts1;

SQL> DROP TABLE parts2;

Bulk Binds

- Using FORALL with part of a collection :

```
SQL> CREATE TABLE employees2 as SELECT  * FROM  emp;
SQL> declare
            type numlist is varray(10) of number;
            depts numlist :=
            numlist(5,10,20,30,50,55,57,60,70,75);
      begin
            forall j in 4..7  -- use only part of varray
              delete from employees2
              where deptno = depts(j);
              commit;
      end;
SQL> drop table employees2;
```

Bulk Binds

- How FORALL affects rollback :
  - If any execution of the SQL statement raises an unhandled exception, all database changes made during previous executions are rolled back.
  - If a raised exception is caught and handled, changes are rolled back to an implicit savepoint marked before each execution of the SQL statement.

```
SQL> CREATE TABLE emp2 (deptno number(2), job varchar2(18));
SQL> declare
          type numlist is table of number;
          depts numlist := numlist(10, 20, 30);
     begin
          insert into emp2 values(10, 'clerk');
          insert into emp2 values(20, 'bookkeeper');  -- lengthening
                                this job title causes an exception.
          insert into emp2 values(30, 'analyst');
          commit;
```

Bulk Binds

```
        forall j in depts.first..depts.last -- run 3 update statements.
                update emp2 set job = job || ' (senior)' WHERE deptno
                = depts(j);
                 -- raises a "value too large" exception
    exception
        when others then
                dbms_output.put_line('problem in the forall
                                        statement.');
                commit; -- commit results of successful updates.
    end;

SQL> drop table emp2;
```

Bulk Binds

To bulk-bind output collections, use the BULK COLLECT clause

BULK COLLECT INTO collection_name [,collection _name]

```
SQL> declare
        type numtab is table of emp.empno%type;
        type nametab is table of emp.ename%type;
        enums numtab; -- no need to initialize
        names nametab;
     begin
        SELECT  empno, ename bulk collect into enums,
                names  FROM emp;
        ...
     end;
```

Bulk Binds

- Using with FETCH INTO statement

```
SQL> declare
        type nametab is table of emp.ename%type;
        type saltab is table of emp.sal%type;
        names nametab;
        sals saltab;
        cursor c1 is SELECT  ename, sal FROM emp
        WHERE sal > 1000;
    begin
        open c1;
        fetch c1 bulk collect into names, sals;
        ...
    end;
```

Bulk Binds

- Using with RETURNING INTO caluse

```
SQL>  declare
            ………..
      begin
            …………
            forall j in depts.first..depts.last
                  delete FROM emp WHERE empno = depts(j)
                  returning empno bulk collect into enums;
            ………….
      end ;
```

Module 28.Flashback Table,DBMS_FLASHBACK

- Overview
  - Introduction
  - Privileges required
  - Flashing back dropped tables
  - Purge
  - Overview of DBMS_FLASHBACK
  - Subprograms of DBMS_FLASHBACK

Flashback Table

- Introduction
  - Flashback table is a new feature introduced in Oracle 10g.
  - Allows us to restore to an earlier state of a table.
  - Reads as per a specific SCN or timestamp.
- Privileges Required
  - FLASHBACK object privilege on the table or the FLASHBACK ANY TABLE system privilege.
  - SELECT , INSERT, DELETE, and ALTER object privileges on the table.
  - To flash back a table to before a DROP TABLE operation, you need only the privileges necessary to drop the table.(i.e. you should be the owner or have DROP ANY TABLE).
  - Row movement must be enabled for all tables in the Flashback list.

Flashback Table

SQL> CREATE TABLE employees_demo
enable row movement
as SELECT * FROM employee;

```
FLASHBACK TABLE
   [ schema. ]table
     [, [ schema. ]table ]...
TO { { SCN | TIMESTAMP } expr
       [ { ENABLE | DISABLE } TRIGGERS ]
     | BEFORE DROP [ RENAME TO table ]
     } ;
```

Flashback Table

Flashback Table

Flashback Table

SQL> SELECT * FROM **RECYCLEBIN;**

SQL> SELECT * FROM **USER_RECYCLEBIN;**

Flashback Table

```
SQL> CREATE TABLE employees_demo
        enable row movement
        as SELECT  * FROM  employee;
```

Flashback Table

```
SQL> SELECT  salary
         FROM employees_demo
          WHERE salary < 2500;
   SALARY
   ----------
    2400
    2200
    2100
    2400

SQL> update employees_demo
         set salary = salary * 1.1
         WHERE salary < 2500;
5 rows updated.

SQL> COMMIT;
```

Flashback Table

```
SQL> SELECT  salary
        FROM employees_demo WHERE salary < 2500;
  SALARY
  ----------
   2420
   2310
   2420

SQL> flashback table employees_demo
       to timestamp (systimestamp - interval '1' minute);

SQL> SELECT  salary FROM employees_demo WHERE salary < 2500;
   SALARY
   ----------
   2400
   2200
   2100
   2400
```

Flashback Table

SQL> **flashback table** employee **to before drop;**

SQL> **flashback table** employee **to before drop rename to employees_old;**

SQL> SELECT object_name, droptime FROM **user_recyclebin** WHERE original_name = 'employee';

```
OBJECT_NAME                    DROPTIME
------------------------------ -------------------
RB$$45703$TABLE$0              2003-06-03:15:26:39
RB$$45704$TABLE$0              2003-06-12:12:27:27
RB$$45705$TABLE$0              2003-07-08:09:28:01
```

Flashback Table

SQL> SELECT **ora_rowscn**, last_name FROM employee WHERE emp_code = 35;

SQL> SELECT **scn_to_timestamp(ora_rowscn)**, last_name FROM employee WHERE emp_code = 35;

Flashback Table

SQL> SELECT * FROM **recyclebin**;

SQL> SELECT * FROM **user_recyclebin**;

```
PURGE TABLE table_name|
          INDEX index_name|
          RECYCLEBIN |
          DBA_RECYCLEBIN|
          TABLESPACE tablespace_name [USER user_name];
```

Flashback Table

SQL> **purge table** test;

SQL> **purge table** rb$$33750$table$0;

SQL> **purge** recyclebin;

DBMS_FLASHBACK

- Using DBMS_FLASHBACK, you can flashback to a version of the database at a specified wall-clock time or a specified SCN.
- You require the EXECUTE privilege to use DBMS_FLASHBACK

DBMS_FLASHBACK

Error	Description
ORA-08180	Time specified is too old.
ORA-08181	Invalid system change number specified.
ORA-08182	User cannot begin read-only or serializable transactions in Flashback mode.
ORA-08183	User cannot enable Flashback within an uncommitted transaction.
ORA-08184	User cannot enable Flashback within another Flashback session.
ORA-08185	SYS cannot enable Flashback mode.

DBMS_FLASHBACK

```
SQL> drop table employee;

SQL> drop table keep_scn;
```

DBMS_FLASHBACK

```
SQL> CREATE TABLE keep_scn (scn number);

SQL> SELECT  lpad(' ', 2*(level-1)) || emp_name name
        FROM employee
        connect by prior emp_code = reports_to
        start with emp_code = 1
        ORDER BY level;
```

DBMS_FLASHBACK

```
SQL> declare
         i number;
     begin
         i := dbms_flashback.get_system_change_number;
         insert into keep_scn values (i);
         commit;
     end;
SQL> delete FROM employee WHERE emp_name = ‘Nimesh Shah';

SQL> commit;
```

DBMS_FLASHBACK

```
SQL> SELECT  lpad(' ', 2*(level-1)) || emp_name name
       FROM employee
       connect by prior emp_code = reports_to
       start with emp_code = 1
       ORDER BY level;

SQL>  declare
            restore_scn number;
       begin
            SELECT   scn into restore_scn FROM keep_scn;
            dbms_flashback.enable_at_system_change_number
            (restore_scn);
       end;
```

DBMS_FLASHBACK

```
SQL> SELECT  lpad(' ', 2*(level-1)) || emp_name name
        FROM employee
        CONNECT BY PRIOR emp_code = reports_to
        START WITH emp_code = (SELECT  emp_code FROM employee
        WHERE emp_name = ‘Nimesh Shah')
        ORDER BY level;

SQL> declare
            CURSOR c1 IS
            SELECT  emp_code, emp_name, reports_to, salary,
            date_join FROM employee
            CONNECT BY PRIOR emp_code = reports_to
            START WITH emp_code = (SELECT  emp_code FROM employee
            WHERE emp_name = ‘Nimesh Shah');
            c1_rec c1 % rowtype;
```

DBMS_FLASHBACK

```
begin
        open c1;
        /* disable flashback */
        dbms_flashback.disable;
        loop
                fetch c1 into c1_rec;
                 exit when c1%notfound;
        /*
        note that all the dml operations inside the loop are performed
        with flashback disabled
        */
        insert into employee(emp_code,emp_name, reports_to, salary,
date_join) values (c1_rec. emp_code,
        c1_rec. emp_name, c1_rec. reports_to ,
        c1_rec.salary, c1_rec. date_join);
```

DBMS_FLASHBACK

```
            end loop;
    close c1;
    commit;
    end;

SQL> SELECT  lpad(' ', 2*(level-1)) || emp_name name
    FROM employee
    connect by prior emp_code = reports_to
    start with emp_code = 1
    ORDER BY level;
```

DBMS_FLASHBACK

Subprogram	Description
DISABLE Procedure	Disables the Flashback mode for the entire session
ENABLE_AT_SYSTEM_CHANGE_NUMBER Procedure	Enables Flashback for the entire session. Takes an SCN as an Oracle number and sets the session snapshot to the specified number. Inside the Flashback mode, all queries will return data consistent as of the specified wall-clock time or SCN
ENABLE_AT_TIME Procedure	Enables Flashback for the entire session. The snapshot time is set to the SCN that most closely matches the time specified in query_time
GET_SYSTEM_CHANGE_NUMBER Function	Returns the current SCN as an Oracle number. You can use the SCN to store specific snapshots
SCN_TO_TIMESTAMP Function	Takes the current SCN as an Oracle number data type and returns a TIMESTAMP.
TIMESTAMP_TO_SCN Function	Takes a TIMESTAMP as input and returns the current SCN as an Oracle number data type

DBMS_FLASHBACK

SQL> **dbms_flashback.disable;**

SQL> **execute dbms_flashback.enable_at_time('30-aug-2000');**

SQL> SELECT salary FROM employee WHERE emp_name = ‘Vijay Gupta‘;

SQL> **execute dbms_flashback.disable;**

DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER (query_scn IN NUMBER);

DBMS_FLASHBACK

Parameter	Description
query_time	This is an input parameter of type TIMESTAMP. A time stamp can be specified in the following ways: □Using the TIMESTAMP constructor: Example: execute **`dbms_flashback.enable_at_time(TIMESTAMP'2001-01-09 12:31:00').`** □Using the TO_TIMESTAMP function: Example: execute `dbms_flashback.enable_at_time(TO_TIMESTAMP('12-02-2001 4:35:00', 'DD-MM-YYYY HH24:MI:SS')).` □If the time is omitted FROM query time, it defaults to the beginning of the day, that is, 12:00 A.M. □Note that if the query time contains a time zone, the time zone information is truncated.

DBMS_FLASHBACK

```
DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER
RETURN NUMBER;


DBMS_FLASHBACK.SCN_TO_TIMESTAMP( query_scn IN
NUMBER) RETURN TIMESTAMP;
```

DBMS_FLASHBACK

Parameter	Description
query_time	This is an input parameter of type TIMESTAMP. A time stamp can be specified in the following ways: □Using the TIMESTAMP constructor: Example: execute DBMS_FLASHBACK.ENABLE_AT_TIME(TIMESTAMP '2001-01-09 12:31:00'). Use the Globalization Support (NLS) format and supply a string. The format depends on the Globalization Support settings. □Using the TO_TIMESTAMP function: Example: execute dbms_flashback.enable_at_time(TO_TIMESTAMP('12-02-2001 14:35:00', 'DD-MM-YYYY HH24:MI:SS')). You provide the format you want to use. This example shows the TO_TIMESTAMP function for February 12, 2001, 2:35 PM. □If the time is omitted FROM query time, it defaults to the beginning of the day, that is, 12:00 A.M. □Note that if the query time contains a time zone, the time zone information is truncated.

DBMS_FLASHBACK

DBMS_FLASHBACK

```
SQL> SELECT  ora_rowscn, emp_name, salary FROM employee
         WHERE emp_code = 26;

ORA_ROWSCN    EMP_NAME    SALARY
__________    ________    ______
202553        Amit Sharma  8000
```

DBMS_FLASHBACK

SQL> UPDATE employee set salary = salary + 100
WHERE emp_code = 26 and **ora_rowscn** = 202553;

0 rows updated.

DBMS_FLASHBACK

```
SQL> UPDATE employee set salary = salary + 100
       WHERE emp_code = 26 and ora_rowscn = 415639;

1 row updated.

SQL> commit;

SQL> SELECT  ora_rowscn, emp_name, salary FROM employee
        WHERE emp_code =  26;

ORA_ROWSCN    EMP_NAME    SALARY
___________   _________   _______
465461        Amit Sharma   8100
```