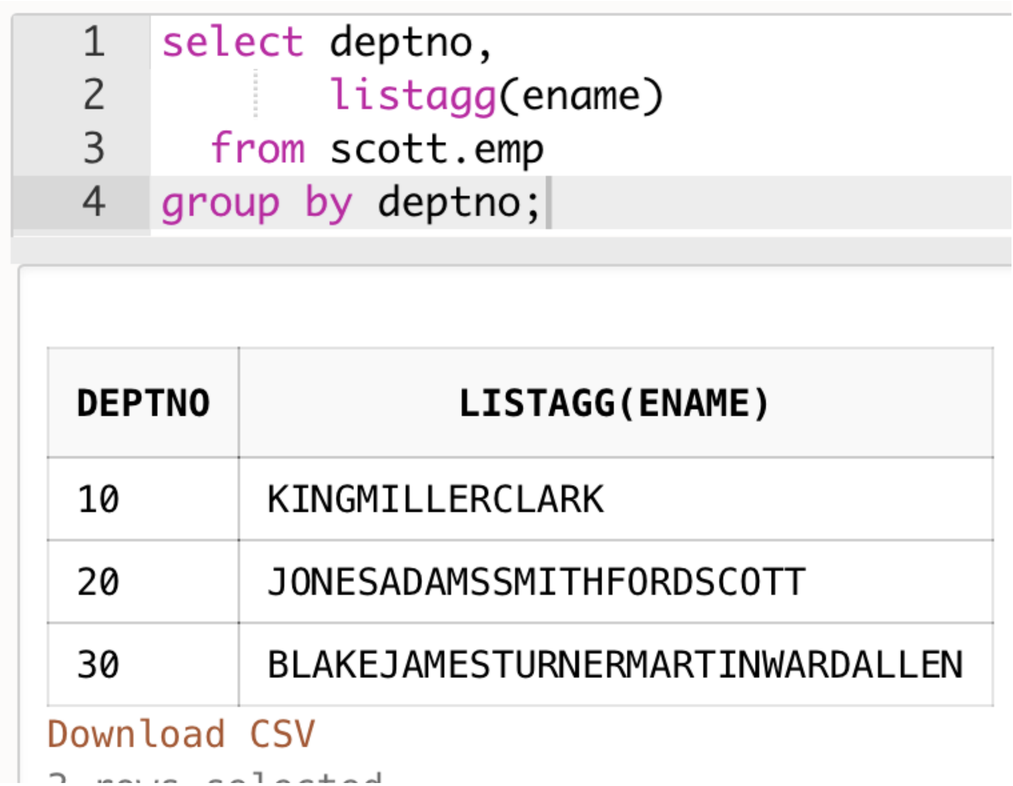
Data Management and Database Design

Week#9

Northeastern University

String aggregation LISTAGG function



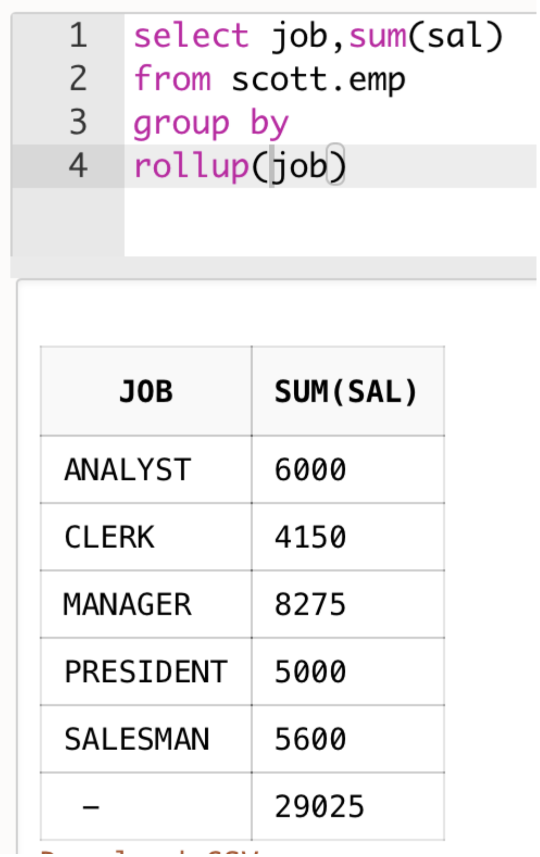
```sql
select deptno,
       listagg(ename)
  from scott.emp
group by deptno;
```

DEPTNO	LISTAGG(ENAME)
10	KINGMILLERCLARK
20	JONESADAMSSMITHFORDSCOTT
30	BLAKEJAMESTURNERMARTINWARDALLEN

3 rows selected

Rollup()

- if you want to generate total and subtotals, ROLLUP() function can be used
  - Usually better for date datatype

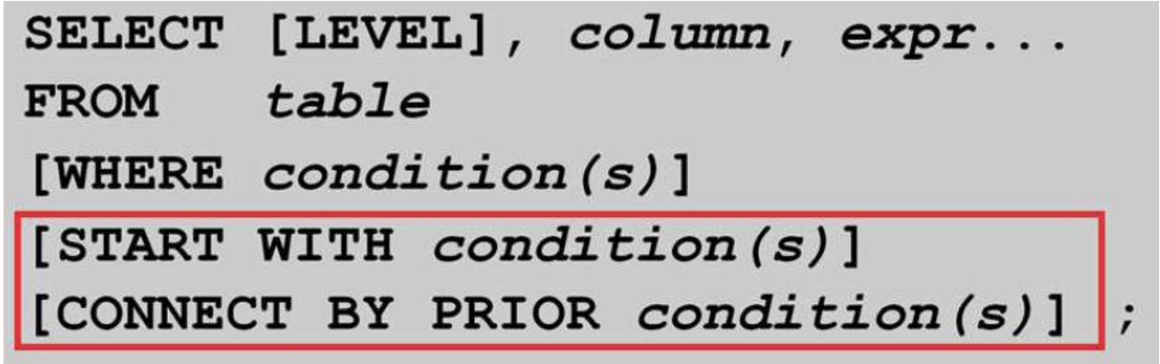


```
select job,sum(sal)
from scott.emp
group by
rollup(job)
```

JOB	SUM(SAL)
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
-	29025

Hierarchical Queries

- Hierarchical queries can be identified by the key words **CONNECT BY** and **START WITH** clauses
- **LEVEL** – For each row returned by a hierarchical query, the **LEVEL** pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
- **START WITH** Specifies the root rows of the hierarchy which means where to start. This clause is required for a hierarchical query
- **CONNECT BY** Specifies the columns in which the relationship between *parent* and *child* **PRIOR** rows exist. This clause is required for a hierarchical query.



```
SELECT [LEVEL], column, expr...
FROM   table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)] ;
```

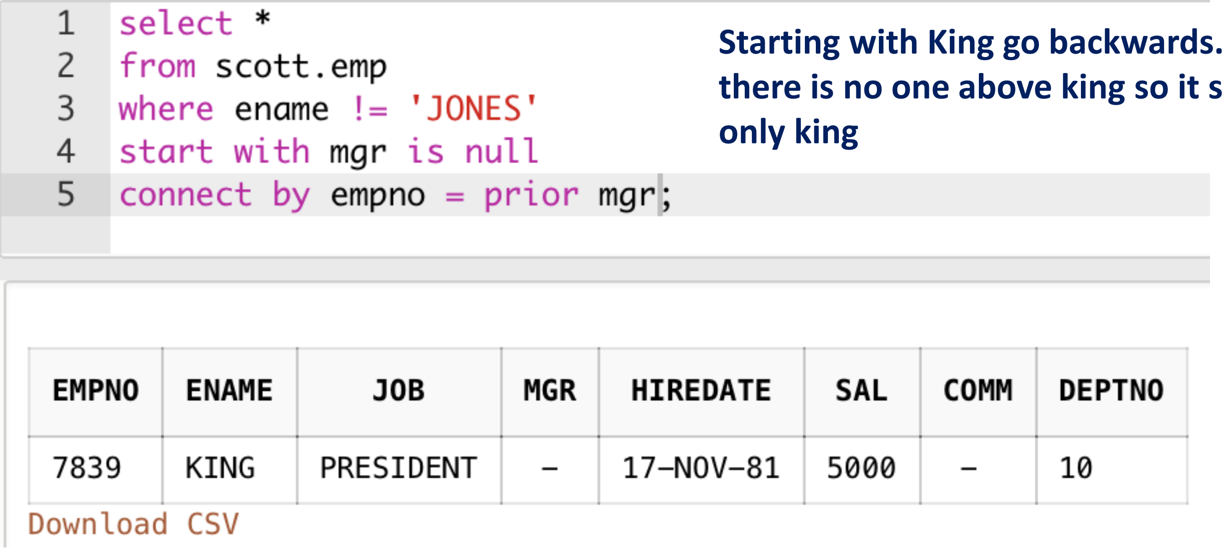
Hierarchical Queries – Tree walking

```
CONNECT BY PRIOR column1 = column2
```

Direction

Top down → **Column1 = Parent Key**
Column2 = Child Key

Bottom up → **Column1 = Child Key**
Column2 = Parent Key

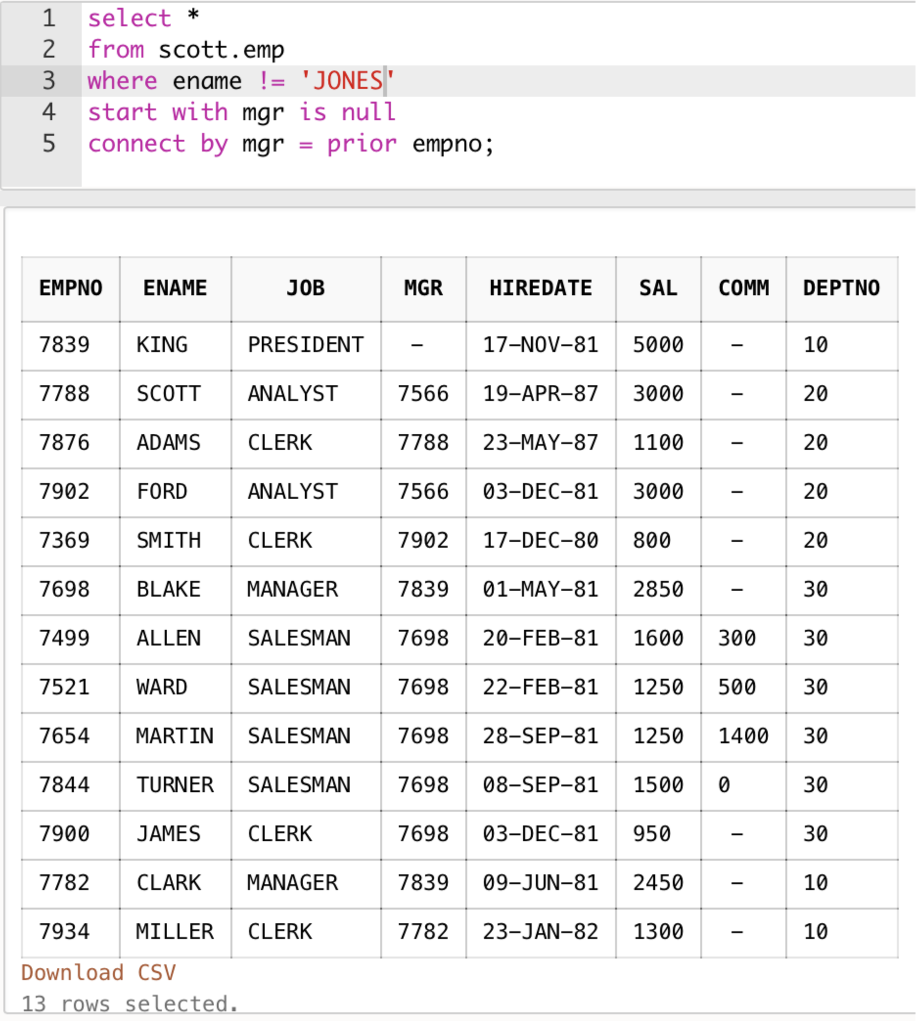


```
select *
from scott.emp
where ename != 'JONES'
start with mgr is null
connect by empno = prior mgr;
```

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

Download CSV

Starting with King go backwards. Since there is no one above king so it shows only king



```
select *
from scott.emp
where ename != 'JONES'
start with mgr is null
connect by mgr = prior empno;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17–NOV–81	5000	–	10
7788	SCOTT	ANALYST	7566	19–APR–87	3000	–	20
7876	ADAMS	CLERK	7788	23–MAY–87	1100	–	20
7902	FORD	ANALYST	7566	03–DEC–81	3000	–	20
7369	SMITH	CLERK	7902	17–DEC–80	800	–	20
7698	BLAKE	MANAGER	7839	01–MAY–81	2850	–	30
7499	ALLEN	SALESMAN	7698	20–FEB–81	1600	300	30
7521	WARD	SALESMAN	7698	22–FEB–81	1250	500	30
7654	MARTIN	SALESMAN	7698	28–SEP–81	1250	1400	30
7844	TURNER	SALESMAN	7698	08–SEP–81	1500	0	30
7900	JAMES	CLERK	7698	03–DEC–81	950	–	30
7782	CLARK	MANAGER	7839	09–JUN–81	2450	–	10
7934	MILLER	CLERK	7782	23–JAN–82	1300	–	10

Download CSV
13 rows selected.

PL/SQL
procedural language extension to SQL

PL/SQL
procedural language extension to SQL

- Objectives of this section

  - Understand block structure and architecture
  - Understand use of SQL with PL/SQL
  - Advantages
  - Cursors
  - PL/SQL Datatypes

PL/SQL
procedural language extension to SQL

- The purpose of PL/SQL is to combine database language and procedural programming language
- The basic unit in PL/SQL is called a **BLOCK** and is made up of three parts –

• Declaration	- DECLARE
• Executable	- BEGIN
• Exception	- EXCEPTION

- PL/SQL enables users to mix SQL statements with procedural constructs
- PL/SQL blocks are compiled once and stored in executable
    - This is called a stored procedure
    - Stored procedure that implicitly started when an DML statement is issued against an associated table is called **TRIGGER**

PL/SQL Vs SQL

SQL	PL/SQL
Single statement could be DML, DDL or DATA Retrieval	BLOCK of code contains more than one statement
Executes as single statement	Executes as block of statements
Used to manage data	We can extend to manage application
Direct interaction with DATABASE Server Objects	Can act as layer between DB objects and clients

PL/SQL

- PL/SQL blocks are defined by keywords
  - DECLARE
    - Used to define and initialize variables, constants
    - Uninitialized variable values will be NULL
  - BEGIN
    - Actual business logic code
  - EXCEPTION
    - Errors are captured here
  - END
    - End of programming block

PL/SQL

- PL/SQL block is of 2 types –
  - Anonymous block
    - Is a BLOCK without a NAME
    - Structure looks as below
    - DECLARE
      - <Declarations>
    - BEGIN
      - <Executable statements>
    - EXCEPTION
      - <Exception Handlers>
    - END

  - Sub Program
    - These are NAMED PL/SQL blocks
    - These can be declared as PROCEDURES, FUNCTIONS, PACKAGES

PL/SQL

- Declaration section contains (Optional) –
  - Variables
  - Constants
  - Cursors
  - User defined exceptions
- Executable section contains (Mandatory) –
  - SQL statements
  - DML statements
- Exception handling section contains (Optional) –
  - Specifies actions to perform when errors, abnormal conditions during execution of executable section

PL/SQL Fundamentals

- Character set
  - PL/SQL programs are written as lines of text
  - Not case-sensitive
- Lexical units
  - Identifiers, Operators, Literals separated by one or more spaces
- Delimiters
  - Simple compound symbol that has special meaning
  - Can be used to represent arithmetic operations

PL/SQL Fundamentals

- Simple symbols
  - +, -, *, /, =, <, >, %, etc
- Compound symbols
  - !=, <>, <=, >=, --, /*, */
- Identifiers
  - Can be used to name PL/SQL programs, objects include constants, variables, subprograms
  - Must start alphabetically
- Literals
  - Character and Date literals must be enclosed in single quotes
  - Numeric literals are of 2 kinds – Integers (89, -89) and Reals (9.88. -3.43)

PL/SQL Datatypes

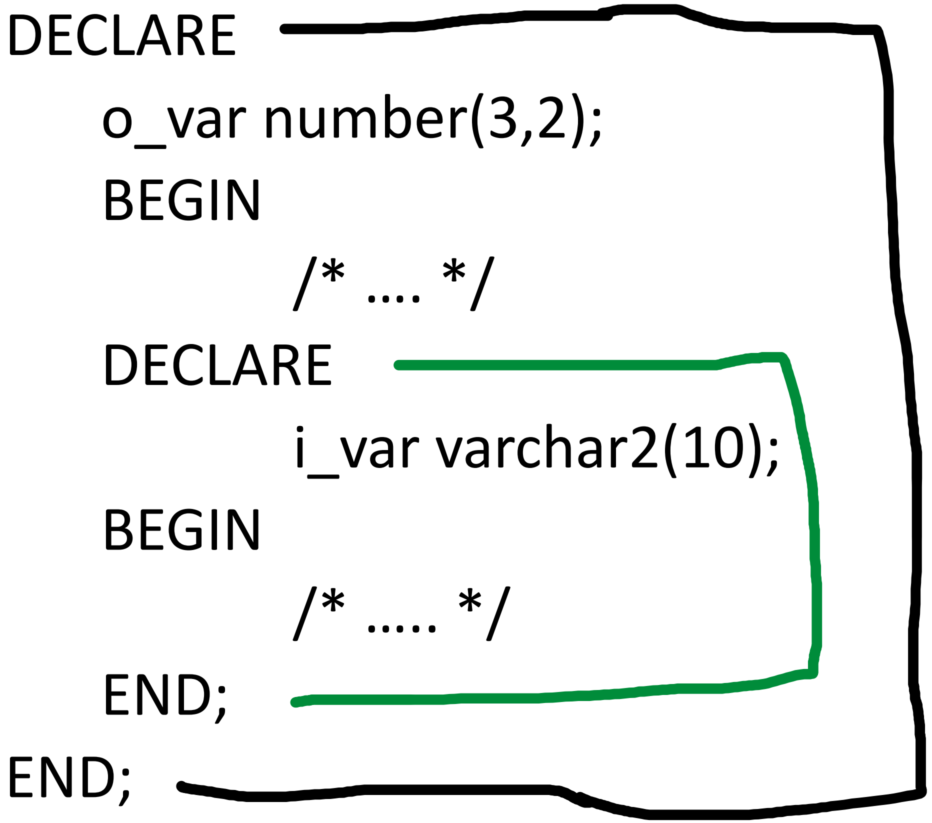
- Scalar types
  - INTEGER
  - FLOAT
  - NUMBER
  - CHAR
  - RAW
  - ROWID
  - VARCHAR2
  - DATE
- Composite types
  - TABLE
  - RECORD
  - VARRAY

PL/SQL
Variables and Constants

- Variables are used to store result of query or calculation
- Variables must be declared before its used
- DEFAULT reserve word is used to initialize variables and constants
- Variables can also be declared using the row attributes of a table
    - %ROWTYPE Record type
    - %TYPE Used to avoid type and size conflict between variable and column
- Declaration syntax: <Variable Name> <TYPE> [:=<VALUE>];
    - V_ENAME CHAR(30);
    - V_DEPTNO NUMBER(3) := 20;
    - V_MGR EMP.MGR%TYPE;
    - V_EMP_ROW EMP%ROWTYPE;

PL/SQL
Scope and Visibility of Variable

- Scope of variable is portion of program in which the variable can be accessed
- Visibility of variable is portion of program where variable can be accessed without having to qualify the reference



```
DECLARE
        o_var number(3,2);
        BEGIN
                /* …. */
        DECLARE
                i_var varchar2(10);
        BEGIN
                /* ….. */
        END;
END;
```

PL/SQL

- Error reporting functions – There are 2 functions to report error
    - SQLCODE
    - SQLERRM
- Conditional and Iterative control
    - IF – THEN – ELSE statement

```
IF <condition1> THEN
      <statement1>
ELSIF <condition2> THEN
      <statement2>
ELSE
      <statement3>
END IF;
```

```
IF t_job = 'CLERK' THEN
      update emp set sal=1000 where …
ELSE
      update emp set sal=500 where …
END IF;
----------------------------------------
IF sales > 1000 THEN
      bouns := 1500;
ELSIF sales > 2000 THEN
      bonus := 500;
ELSE
      bonus := 0;
END IF;
```

PL/SQL

- LOOP – END LOOP
  - FOR LOOP
  - WHILE LOOP
- Each time flow of execution reaches the END LOOP statement, control is returned to the corresponding LOOP statement. LOOP is endless without EXIT statement.

```
LOOP
   <statements>
END LOOP;
```

```
LOOP
        ctr := ctr + 1;
        IF ctr = 10 THEN
                EXIT;
        End IF;
END LOOP;
```

```
LOOP
        ctr := ctr + 1;
        EXIT WHEN ctr = 5;
END LOOP;
```

PL/SQL

- FOR LOOP
  - For loops iterate over specified range of integers.

```
FOR <variable> IN <lower> .. <upper>
LOOP
   <statements>
END LOOP;
```

```
Example –

FOR ctr IN 1 .. 10
LOOP
    INSERT INTO EMP(ID) VALUES(ctr);
    …..
    ….
END LOOP;
```

<variable> whose value will be incremented automatically on each iteration of the loop. It has certain properties as –

Datatype is NUMBER and doesn’t need to be declared

Scope is only with in the FOR LOOP

With in this LOOP, this index variable can be referenced but cannot be changed / modified.

PL/SQL

- When using **SQL** inside PL/SQL –
  - DDL statements are illegal in PL/SQL
  - SELECT statement which do not return a single ROW will cause exception to be raised
    - Exceptions are identifiers in PL/SQL which may be “RAISED” during execution of a BLOCK to terminate its MAIN BODY of actions.
  - DML statements can process multiple rows
- When using DML inside PL/SQL –
  - DML statements that affect ZERO rows will not cause errors
- INTO Clause
  - INTO clause is used with SELECT to store values from the table into VARIABLES
  - INTO clause occurs between SELECT and FROM clause
  - This clause specifies names of variables that will be populated by the items being selected in select clause. Separate variables for each field and order is important.

Writing PL/SQL code

- PL/SQL code is written using text editor
- PL/SQL program is compiled and executed using command @<filename>
- Inserting comments in PL/SQL program can be placed with –
  - “--” (2 minus symbols is a single line comment)
  - “/*… .*/” is a multiline comment
- DBMS_OUTPUT.PUT_LINE()
  - This procedure will produce the output on the screen
  - Accepts only one argument, Hence the different variables are concatenated with double pipe (||) symbol.
  - To enable the server output, the **SET SERVEROUTPUT ON** command must be given at SQL* Plus prompt prior to execution of this procedure.
    - `dbms_output.put_line(‘My name is ‘||ename||’ working in department number ‘||deptno);`

Finally, Writing PL/SQL code

- Update salary of employee number 7788 to $2800 if salary is less than $2800.

```
DECLARE
   x number(7,2);
   y number(7,2) constant := 2800;
BEGIN
   select sal into x from emp where empno = 7788;
   if x < y then
      update emp set sal = y where empno = 7788;
   end if;
END;
```

Predefined PL/SQL Exceptions

- Below are the list of predefined oracle exceptions.

```
NO_DATA_FOUND
CURSOR_ALREAD_OPEN
DUP_VAL_ON_INDEX
INVALID_NUMBER
TOO_MANY_ROWS
ZERO_DIVIDE
CASE_NOT_FOUND
```

Example user defined PL/SQL Exceptions

```
BEGIN
   DECLARE ---------- sub-block begins
      past_due EXCEPTION;
      due_date DATE := trunc(SYSDATE) - 1;
      todays_date DATE := trunc(SYSDATE);
   BEGIN
      IF due_date < todays_date THEN
         RAISE past_due;
      END IF;
   END; ------------- sub-block ends
EXCEPTION
   WHEN past_due THEN
   dbms_output.put_line(‘raied exception’);
END;
```

Cursors

- It is a pointer that points to result set of an executed query.
- 2 types
  - Implicit
  - Explicit

- Implicit cursors are automatically created by oracle when ever
  - Any SQL statement is executed
  - Developer doesn’t have any control on these cursors
  - In case of Insert – cursor stores the data that is being inserted
  - In case of Update OR Delete – cursor holds rows that are being affected

Explicit Cursors

- Explicit cursors are declared by Developer
- SQL result aka output can be pointed to explicit cursor
- How to **create**?
    - `CURSOR <cursor name> IS <select statement>;`
- How to **access**?
    - `OPEN <cursor name>;`
- How do **read** cursor data?
    - `FETCH <cursor name> into <variables>;`
    - `If you have more than one row to read, then loop through the cursor`
- How do I **release** pointer from storage area?
    - `CLOSE <cursor name>`

Cursor loop construct

- How to loop through cursor variable?
  - Few things to be considered Looping should have start and end or else its infinite loop
  - You should know how long to loop, basically number of rows may be?
  - What you should do if you are end of the loop

- Cursor processing attributes
  - <cursor name>%**ROWCOUNT** is used to get number of rows in a cursor
  - <cursor name>%**FOUND** tells that we have data to read
  - <cursor name>%**NOTFOUND** tells end of cursor
  - <cursor name>%**ISOPEN** tells if a cursor is already open or not

- We can read cursor using either LOOP of FOR LOOP construct

Cursor FOR loop construct

```
DECLARE
        CURSOR cur_emp_details is
                SELECT empno, ename, dname FROM emp,dept WHERE emp.deptno=dept.deptno;
BEGIN
        FOR rec IN c_cur_emp_details
        LOOP
                dbms_output.put_line(Employee|| ' ' ||rec.ename||’ works in '||rec.dname);
        END LOOP;
END;
/
```

NOTE: For single row details retrieval, always use SELECT INTO instead of CURSOR.

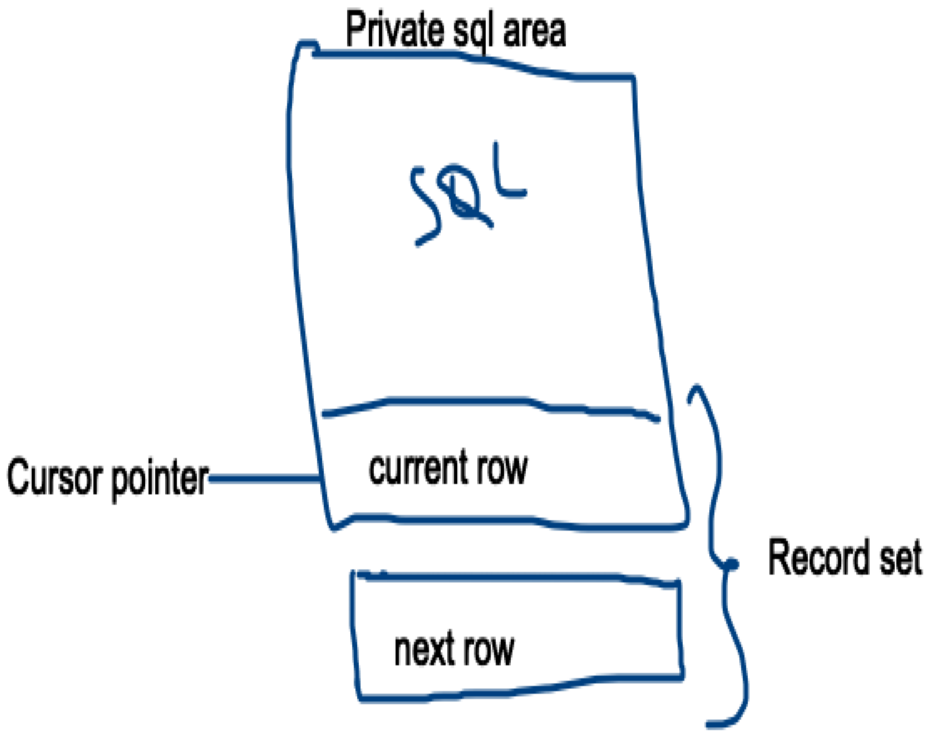
Cursor loop construct

```
DECLARE
        l_empno emp.empno%type;
        l_ename emp.ename%type;
        l_dname dept.dname%type;
        CURSOR cur_emp_details is
                SELECT empno, ename, dname FROM emp,dept WHERE emp.deptno=dept.deptno;
BEGIN
        OPEN c_cur_emp_details;
        LOOP
                FETCH cur_emp_details INTO l_empno, l_ename, l_dname;
                EXIT WHEN cur_emp_details%NOTFOUND;
                dbms_output.put_line(Employee || ' ' || l_ename || ‘ works in ' || l_dname);
        END LOOP;
        CLOSE c_cur_emp_details;
END;
/
```

This does read records one by one. If we want to process bulk rows at a time, then?

Cursor

```
DECLARE
   v_sal emp.sal%type;
   v_job emp.job%type;
   cursor cur1 is
      select sal, job from emp;
   cur1rec cur1%rowtype;
BEGIN
   open cur1;
   loop
      fetch cur1 into v_sal,v_job;
      exit when cur1%NOTFOUND;
EXCEPTION
   ….
END;
```



Stored procedures

```
CREATE PROCEDURE <procedure name>(param-1 datatype, …., param-N datatype)
AS
        LOCAL Variables declaration

BEGIN
        Executable code

EXCEPTION
        when <exception Name> then
                <Action>;
        when others then
                <Action>;

End <procedure name>;
/
```

Stored procedures
Positional Vs Named arguments

```
CREATE or REPLACE PROCEDURE update_sal(pi_empno number, pi_sal number)
AS
BEGIN
   <write your code….>;
END;
```

--Anonymous block to execute a procedure

```
BEGIN
        update_sal (7788, 2500); -- Parameters passing by position
        update_sal(pi_sal => 3000, pi_empno => 7760); -- Parameters passing by Name
END; /
```

Isolation levels

- Isolation means ability of a transaction to run without interference.
- ANSI/ISO SQL standard defines four levels of transaction isolation
  - Read Uncommitted
  - Read Committed
  - Repeatable Read
  - Serializable
- These levels are defined in terms of three phenomena/events or facts that are either permitted or not for a given isolation level
  - Dirty Read
  - Non-Repeatable Read
  - Phantom Read

Isolation levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	-	Allowed	Allowed
REPEATABLE READ	-	-	Allowed
SERIALIZABLE	-	-	-

Questions?