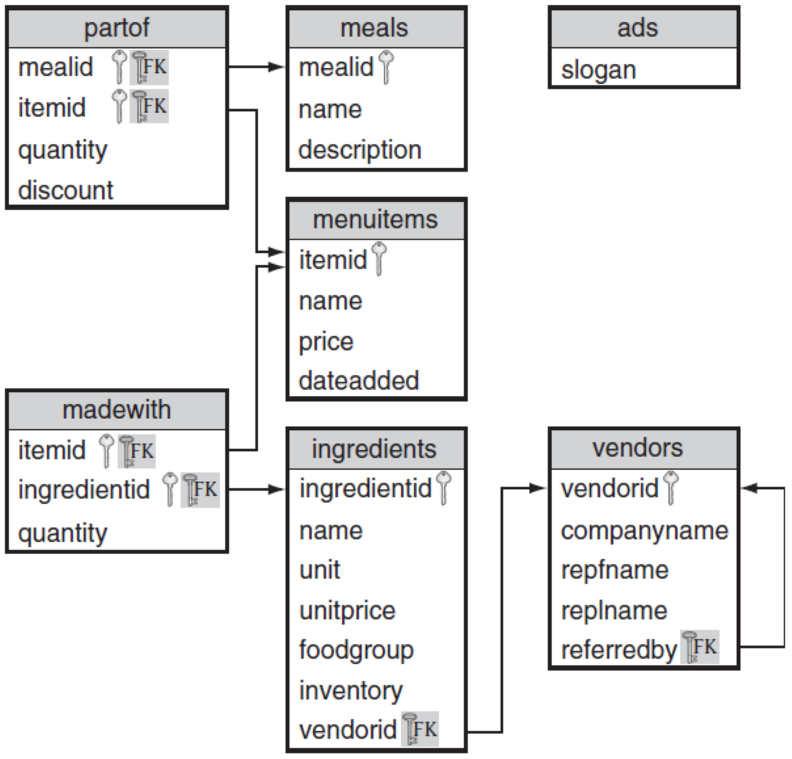
Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #6

Today's Topics:

- ❖ views
- ❖ transactions
- ❖ PL/SQL

Views:

- ❖ we will learn about views, transactions and practice them on a schema of a restaurant chain.
- ❖ The table schema is given below.



ABOUT VIEWS

- ❖ A view is a virtual table defined by a query. It provides a mechanism to create alternate ways of working with the data in a database. A view acts much like a table. We can query it with a SELECT, and some views even allow INSERT, UPDATE, and DELETE. There are several uses for a view.
- ❖ **Usability**—We can use a view as a wrapper around very complex SELECT statements to make our system more usable.
- ❖ **Security**—If we need to restrict the data a user can access, we can create a view containing only the permitted data. The user is given access to the view instead of the base table(s).

- **Reduced Dependency**—The database schema evolves over time as our enterprise changes. Such changes can break existing applications that expect a certain set of tables with certain columns. We can fix this by having our applications access views rather than base tables. When the base tables change, existing applications still work as long as the views are correct.

CREATING VIEWS

```
CREATE VIEW <view name> [(<column list>)] AS <SELECT statement>
```

- This creates a view named <view name>. The column names/types and data for the view are determined by the result table derived by executing <SELECT statement>. Optionally, we can specify the column names of the view in <column list>. The number of columns in <column list> must match the number of columns in the <SELECT statement>.

Now let us see an example, try out the following query which creates a view called vrs showing the ingredients (ingredient ID,name, inventory, and inventory value) supplied to us by Veggies_R_Us.

```
CREATE VIEW vrs AS
SELECT ingredientid, name, inventory, inventory * unitprice AS value
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Veggies_R_Us';
```

SELECT FROM VIEWS:

- Selecting from views is just similar to selecting from a table. View is just a virtual table.

```
SELECT * from vrs;
```

- The above query selects all rows from the view vrs. We can also give WHERE conditions as in tables.

Find all ingredients provided by Veggies_R_Us with an inventory of more than 100?

```
SELECT name
FROM vrs
WHERE inventory > 100;
```

- Notice that the above query gives the same result when we use the original tables in the query with out using the view.

```
SELECT name
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Veggies_R_Us'
AND inventory > 100;
```

- ❖ Because the view is just a virtual table, any changes to the base tables are instantly reflected in the view data.
- ❖ See the following query updates the inventory and these changes are reflected the view vrs.

```
UPDATE ingredients
SET inventory = inventory * 2
WHERE ingredientid = 'TOMTO';
```

- ❖ The inventory is updated in the view

```
SELECT * from vrs;
```

- ❖ Now, lets see another example which shows how to create view on a complex query .

Let's create a new view called menuitems that lists all of the items we have for sale, including meals and items, and how much they cost.

```
CREATE VIEW menuitems (menuitemid, name, price) AS
(SELECT m.mealid, m.name, CAST(SUM(price * (1-discount)) AS
NUMERIC(5,2))
FROM meals m LEFT OUTER JOIN partof p ON m.mealid = p.mealid
LEFT OUTER JOIN items i ON p.itemid = i.itemid
GROUP BY m.mealid, m.name)
UNION
(SELECT itemid, name, price
FROM items);
```

- ❖ After creating the view , we can easily list our menu items.

Find all menu items

```
Select * from menuitems;
```

Find the most expensive menu item

```
SELECT name
FROM menuitems
WHERE price =
(SELECT MAX(price)
FROM menuitems);
```

Find the priceless menu items

```
SELECT COUNT(*)
FROM menuitems
WHERE price IS NULL;
```

- ❖ Notice that these queries would have been more complex without creating views

<u>Restrictions on DML operations for views</u>

- ❖ Restrictions on DML operations for views use the following criteria in the order listed:
  - o If a view is defined by a query that contains `SET` or `DISTINCT` operators, a `GROUP BY` clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
  - o If a view is defined with `WITH CHECK OPTION`, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
  - o If a `NOT NULL` column that does not have a `DEFAULT` clause is omitted from the view, then a row cannot be inserted into the base table using the view.
  - o If the view was created by using an expression, such as `DECODE(deptno, 10, "SALES", ...)`, then rows cannot be inserted into or updated in the base table using the view.

<u>Updating a Join View</u>

- ❖ An updatable join view (also referred to as a modifiable join view) is a view that contains more than one table in the top-level `FROM` clause of the `SELECT` statement, and is not restricted by the `WITH READ ONLY` clause.
- ❖ The rules for updatable join views are shown in the following table. Views that meet these criteria are said to be inherently updatable.

Rule	**Description**
General Rule	Any `INSERT`, `UPDATE`, or `DELETE` operation on a join view can modify only one underlying base table at a time.
`UPDATE` Rule	All updatable columns of a join view must map to columns of a key-preserved table*. If the view is defined with the `WITH CHECK OPTION` clause, then all join columns and all columns of repeated tables are not updatable.
`DELETE` Rule	Rows from a join view can be deleted as long as there is exactly one key-preserved table in the join. The key preserved table can be repeated in the `FROM` clause. If the view is defined with the `WITH CHECK OPTION` clause and the key preserved table is repeated, then the rows cannot be deleted from the view.
`INSERT` Rule	An `INSERT` statement must not explicitly or implicitly refer to the columns of a non-key-preserved table. If the join view is defined with the `WITH CHECK OPTION` clause, `INSERT` statements are not permitted.

* The concept of a key-preserved table is fundamental to understanding the restrictions on modifying join views. A table is key-preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.

For details, see : http://docs.oracle.com/cd/B28359_01/server.111/b28310/views001.htm#ADMIN11783

- ❖ Finally, even if a view is updatable, not all columns within the view may be updatable. For example, derived columns such as *value* in the *vrs* cannot be updated. Thus, our *menuitems* view cannot be directly updated because it contains attributes from the *items, meals,* and *partof* tables. The *vrs* view can be updated as if it were a base table

```
UPDATE vrs SET inventory = inventory * 2;
SELECT * FROM vrs;
```

- ❖ It is important to note that updates through views can have unexpected consequences, depending on the behavior of a DBMS. For example, a DBMS might allow an INSERT on vrs, such as the following insert. The underlying ingredients table would be updated with the provided values, but the vendorid would be set to the default value (in our case, NULL).

```
INSERT INTO vrs(ingredientid, name, inventory) VALUES
'NEWIN','New ingredient',100);


SELECT * FROM vrs;
```

- ❖ Observe the result. Is there anything wrong it? If it is Why? Because the query specification for vrs requires the vendorid to be VGRUS, our new row does NOT appear in the view.

- ❖ In general, updates through views work best when the view is defined as a subset of a table and all attributes that determine if a row is in a view are updatable.

- ❖ **Note:** the update operations supported on views are generally specific to the SQL vendor.

ALTER A VIEW:

- ❖ Altering view can also be done by dropping it and recreating it. but that would drop the associated granted permissions on that view. By using alter clause, the permissions are preserved.

```
ALTER VIEW [ schema_name . ] view_name [ ( column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ] [ ; ]
<view_attribute> ::=
{
    [ ENCRYPTION ]
    [ SCHEMABINDING ]
    [ VIEW_METADATA ]
}
```

To alter the view vrs to display the ingredients supplied by 'Spring Water Supply'.

```
CREATE OR REPLACE VIEW vrs AS
```

```
SELECT ingredientid, name, inventory, inventory * unitprice AS value
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Spring Water Supply';
```

DROP A VIEW:

```
DROP VIEW <view name> [CASCADE | RESTRICT];
```

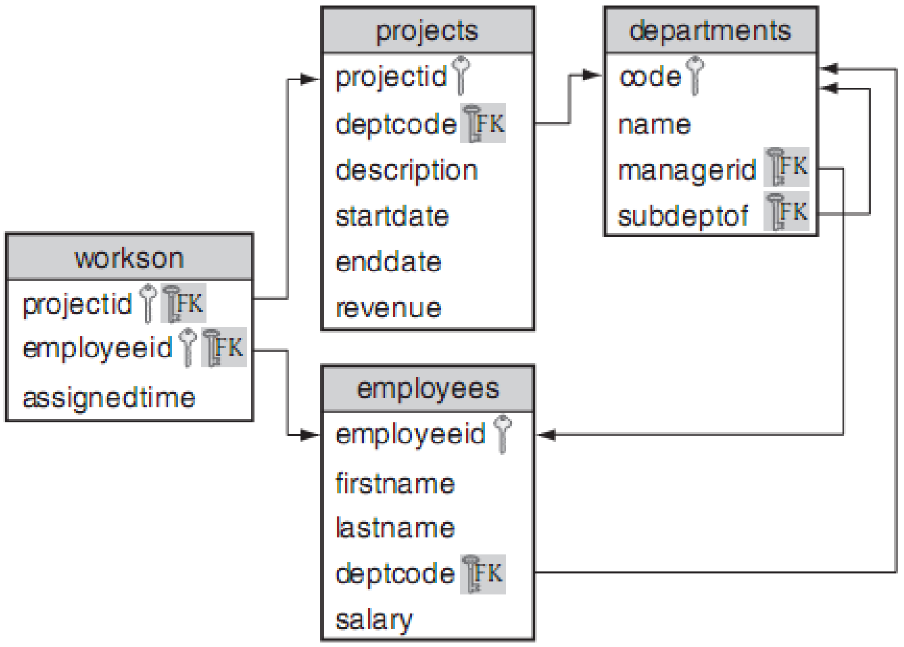
- ❖ This removes the specified view; however, it does not change any of the data in the database. SQL does not allow a view to be dropped if view is contained in the SELECT statement of another view. This is default behavior and is equivalent to the effect of using RESTRICT Option. If we want to drop a view which is being used in a select command we must use the CASCADE option.

```
drop VIEW vrs;
```

- ❖ This drops the view vrs if it exists.

EXERCISES ON EMPLOYEES DATABASE:

- ❖ Write a SQL query for each of the following based on employee database created in the previous labs. The scheme for employee database is shown below.



1. Create a view containing all of the employees assigned to the 'Robotic Spouse' project. Include the percent time they are assigned to the project.
2. Query your view created in the previous question to find the employee first and last name with the greatest amount of time assigned to 'Robotic Spouse'.
3. Create a view of employees with their department name.
4. Query your view to find all the first and last names of employees in the Consulting department.
5. Create a view showing all of the projects assigned to Abe Advice, including his percentage time on each project.
6. Query your view to find the total amount of time Abe is assigned to projects.

7. Create an updatable view showing employees and their salaries. Give everyone a 10% raise by updating the view

TRANSACTIONS

- Databases are all about sharing data, so it is common for multiple users to be accessing and even changing the same data at the same time. The simultaneous execution of operations is called concurrency. Sometimes concurrency can get us into trouble if our changes require multiple SQL statements. In general, if two or more users access the same data and one or more of the statements changes the data, we have a conflict. This is a classic problem in database systems; it is called the isolation or serializability problem. If the users perform multiple steps, conflicts can cause incorrect results to occur. To deal with this problem, databases allow the grouping of a sequence of SQL statements into an indivisible unit of work called a transaction. A transaction ends with either a commit or a rollback:
  - **commit**—A commit permanently stores all of the changes performed by the transaction.
  - **rollback**—A rollback removes all of the updates performed by the transaction, no matter how many rows have been changed. A rollback can be executed either by the DBMS to prevent incorrect actions or explicitly by the user.

- The DBMS provides the following guarantees for a transaction, called the ACID properties: Atomicity, consistency, Isolation, and durability. These properties will be covered in the course at detail.
- SQL starts a transaction automatically when a new statement is executed if there is no currently active transaction. This means that a new transaction begins automatically with the first statement after the end of the previous transaction or the beginning of the session.
- A user may explicitly start a transaction using the START TRANSACTION statement.

```
SET TRANSACTION NAME <string>;
```

- Commit:

```
SET TRANSACTION NAME 't1';
UPDATE vrs SET inventory = inventory + 10;
SELECT * FROM vrs;
COMMIT;
SELECT * FROM vrs;
```

- Rollback:

```
SET TRANSACTION NAME 't2';
UPDATE vrs SET inventory = inventory -20;
SELECT * FROM vrs;
ROLLBACK;
SELECT * FROM vrs;
```

- Here note that without using start transaction, if you execute the update query alone, the values are automatically committed. There is no way to revert back.

SAVEPOINTS

- ❖ SQL allows you to create named placeholders, called savepoints, in the sequence of statements in a transaction. You can rollback to a savepoint instead of to the beginning of the transaction. Only the changes made after the savepoint are undone. To set a savepoint, use the SAVEPOINT command:

```
SAVEPOINT <savepoint     name>
```

- ❖ If we create a savepoint, we can rollback to that savepoint with the following:

```
ROLLBACK TO SAVEPOINT  <savepoint        name>
```

- ❖ Executing ROLLBACK without designating a savepoint or executing a COMMIT deletes all savepoints back to the start of the transaction. A rollback to a particular savepoint deletes all intervening savepoints.

```
UPDATE vrs SET inventory = inventory + 25;
SELECT * FROM vrs;
SAVEPOINT spoint1;
UPDATE vrs SET inventory = inventory - 15;
SELECT * FROM vrs;
SAVEPOINT spoint2;
UPDATE vrs SET inventory = inventory + 30;
SELECT * FROM vrs;
SAVEPOINT spoint3;
ROLLBACK TO SAVEPOINT spoint1;
SELECT * FROM vrs;
```

<u>EXERCISES ON EMPLOYEES DATABASE:</u>

- ❖ Write a queries for each of the following based on employee database created in the previous labs. The scheme for employee database is shown below.

1. Delete the Administration department and all of its subdepartments without using transactions.
2. Using a transaction, delete the Administration department and all of its subdepartments.

PL/SQL

Overview of PL/SQL

PL/SQL is a block-structured language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or sub problem to be solved. PL/SQL is not case-sensitive.

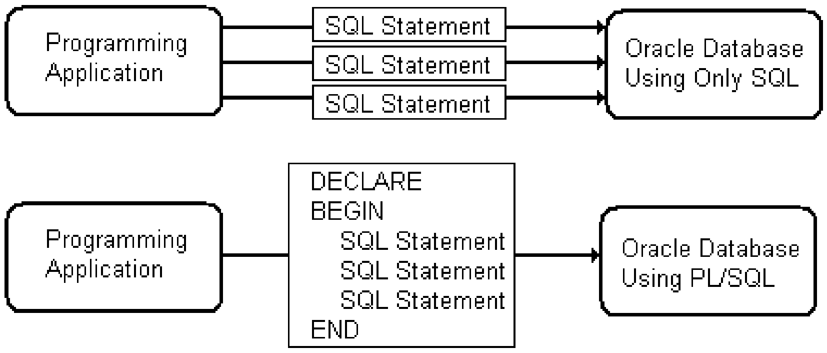
PL/SQL provides additional capabilities that SQL lacks:

- o Secure code through encryption and by storing code on a server instead of a client computer.
- o Handle exceptions that arise due to data entry errors or programming errors.

- Process record with iterative loop code that manipulates records one at a time.
- Work with variables, records, arrays, objects, and other common programming language constructs.

These are the advantages of PL/SQL.

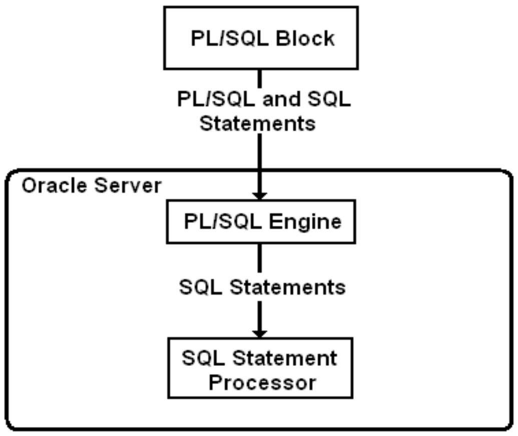
- **Better Application Performance:** PL/SQL provides the capability to define a "block" of programming statements, and can transmit this block to an Oracle database as a unit of work. When multiple SELECT statements are part of a PL/SQL block, there are still only two network trips. This improves system response time by decreasing the amount of network and performance overhead that is incurred with an approach where each SELECT statement processes individually.



- **Productivity, Portability, and Security:** PL/SQL stored procedures run on a server instead of a client computer. This means that the procedures can be secured from tampering by unscrupulous hackers by restricting access through the use of standard Oracle database security. Consider a typical business requirement to update a customer order. Instead of granting access to the customer order table, we can grant access to a procedure that has been coded to update the table.

PL/SQL BLOCK STRUCTURE

A PL/SQL blocks combine statements that represent a single logical task. When the PL/SQL engine is located on the server, the entire PL/SQL block is passed to the PL/SQL engine on the Oracle server. PL/SQL blocks can be divided into two groups: *named* and *anonymous*.



An anonymous block executes by first sending the block to the PL/SQL engine on the server where it is compiled. In contrast, a named PL/SQL block is compiled only at the time of its creation, or if it has been modified.

Structure of a PLSQL program:

```
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

/* Declare and Exception parts are optional. Every statement must be terminated by semicolon. Identifiers start with alphabets. Comments as shown. Reserved words cannot be used. Statements may span multiple lines*/

To enable output to console window in Oracle SQL Developer

From the menu go to: View > Dbms Output.
From the new window (popped-up in the log area) click on the plus sign to enable output.

anonymous block

```
DECLARE
     num_age     NUMBER(3) := 20;  -- assign value to variable
BEGIN
     num_age := 20;
     DBMS_OUTPUT.PUT_LINE('My age is: ' || TO_CHAR(num_age));
END;
```

Declarations:

Our program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

Data Type	Usage	Sample Declaration
VARCHAR2	Variable-length character strings	FirstName VARCHAR2(30);
CHAR	Fixed-length character strings	StudentGender CHAR(1);
NUMBER	Floating, fixed-point, or integer numbers	Salary NUMBER(6);
PLS_INTEGER	Integers for indexing purpose.	StudentID PLS_INTEGER;
DATE	Dates	TodaysDate DATE;
BOOLEAN	TRUE / FALSE / NULL values	OrderFlag BOOLEAN;
LOB (It has four data types: BFILE, BLOB, CLOB and NCLOB)	Large Objects	Message CLOB;
%TYPE	Assumes the data type of the database field	CustAddress customer.cadd%TYPE;
%ROWTYPE	Assumes the data type of a	CustOrderRecord

	database row	cust_order%ROWTYPE;

```
HIREDATE     DATE;
ROOTTWO        CONSTANT NUMBER: = 1.414
ACCT_ID      VARCHAR2 (5) NOT NULL: = 'PS001';
ingredients INGREDIENTS%ROWTYPE
foodgrp          ingredients.foodgroup%type
```

The first declaration names a variable of type DATE. The second declaration names a variable of CONSTANT type NUMBER and uses the assignment operator (:=) to assign an initial value of 1.414 to the variable. The third declaration names a variable of type VARCHAR2, specifies the NOT NULL constraint, and assigns an initial value of 'PS001' to the variable.

how to use data types

```
DECLARE
   wages          NUMBER;
   hours_worked   NUMBER := 40;
   hourly_salary  NUMBER := 22.50;
   bonus          NUMBER := 150;
   country        VARCHAR2(128);
   counter        NUMBER := 0;
   done           BOOLEAN;
   valid_id       BOOLEAN;
  TYPE myarr IS VARRAY(10) of NUMBER; --array
 TYPE commissions IS TABLE OF NUMBER INDEX BY VARCHAR2(10); --
associative array
  t_arr        myarr:= myarr();
  comm_tab       commissions;
BEGIN
  t_arr.extend; --append element
  t_arr(1):=1;
  t_arr.extend(2);
  t_arr(2):=1;
  t_arr(3):=1.9;
 wages := (hours_worked * hourly_salary) + bonus;
   country := 'France';
   done := (counter > 100);
   valid_id := TRUE;
   comm_tab('France') := 20000 * 0.15;
        DBMS_OUTPUT.PUT_LINE( to_char(wages) || ' ' || country || ' '
|| to_char(comm_tab('France')) );
END;
```

Error Handling: This section contains statements that are executed whenever a runtime error occurs within the block. When a runtime error occurs, program control is passed to the exception-handling section of the block. The runtime error is then evaluated, and a specific exception is raised or executed..

Exceptions :

Exceptions are of two types : user defined and predefined.

```
declare
over_withdrawal exception;  --user defined exception
begin
if  creditlimit < withdrawal_amount then
      raise over_withdrawal;
            end  if;
exception
when  over_withdrawal then
insert into log values('balance insufficient');
end;
```

Important predefined exceptions are NO_DATA_FOUND (when the query did not return any row) and TOO_MANY_ROWS (when the query returned more than one row). we don't have to declare these exceptions . If we want to catch other exceptions (other predefined exceptions we don't know but we want a default action for) then we can use the default exception handler OTHERS.

```
EXCEPTION
WHEN OTHERS THEN
ROLLBACK;
END;
```

Control Constructs

```
IF <condition> THEN <actions> [ ELSEIF <condition> THEN <actions>][ELSE
<actions>] ENDIF;
```

```
LOOP <sequence of statements>
IF <condition> THEN EXIT;
ENDIF;
END LOOP;
```

Important: there is no default exiting of loop so we **must** specify an explicit exit condition.

```
WHILE <condition is true> LOOP
               < sequence of statement >
END LOOP;

 FOR <counter> IN <lower bound> …<higher bound> LOOP
               <sequence of statement> END LOOP
```

example of if

```
DECLARE
    v_PurchaseAmount NUMBER(9,2) := 1001;
    v_DiscountAmount NUMBER(9,2);
BEGIN
    IF NOT (v_PurchaseAmount <= 1000) THEN
        v_DiscountAmount := v_PurchaseAmount * 0.05;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Discount: ' || TO_CHAR(v_DiscountAmount));
END;
```

example of if else

```
DECLARE
    v_CustomerStatus CHAR(3) := '&CustomerStatus';
    v_PurchaseAmount NUMBER(9,2) := '&PurchaseAmount';
    v_DiscountAmount NUMBER(9,2);
BEGIN
    IF v_CustomerStatus = 'AAA' THEN
        IF v_PurchaseAmount > 1000 then
            v_DiscountAmount := v_PurchaseAmount * 0.05;
        ELSE
            v_DiscountAmount := v_PurchaseAmount * 0.02;
        END IF;
    ELSE
        v_DiscountAmount := 0;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Discount: ' || TO_CHAR(v_DiscountAmount));
END;
```

```
/
```

Example of LOOP:

```
DECLARE
     v_Balance  NUMBER(9,2) := 100;
BEGIN
    LOOP
        v_Balance := v_Balance - 15;
        IF v_Balance <= 0 THEN
            EXIT;
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('You may have paid too much.');
    DBMS_OUTPUT.PUT_LINE('Ending balance: ' || TO_CHAR(v_Balance));
END;
```

Example of WHILE LOOP:

```
DECLARE
            v_Counter NUMBER := 1;
BEGIN
     WHILE v_Counter < 5 LOOP
                DBMS_OUTPUT.PUT_LINE('Count = ' || TO_CHAR(v_Counter));
                 v_Counter := v_Counter + 1;
     END LOOP;
END;
```

Example of FOR LOOP:

```
DECLARE
     v_Rate NUMBER(5,4) := 0.06/12;
     v_Periods NUMBER := 12;
     v_Balance NUMBER(9,2) := 0;
BEGIN
    FOR i IN 1..v_Periods LOOP   -- loop number of periods
         v_Balance := v_Balance + 50;
         v_Balance := v_Balance + (v_Balance * v_Rate);
          DBMS_OUTPUT.PUT_LINE('Balance for Period ' || TO_CHAR(i) || '
' || TO_CHAR(v_Balance));
     END LOOP;
END;
```

<u>CURSORS</u>

When Oracle processes an SQL statement as part of a PL/SQL program, it creates an area of memory known as the context area. A cursor is a handle, or a pointer, to a context area. Through use of a cursor, a PL/SQL program lets us control the context area, access the information, and process the data rows individually. There are two types of cursors.

- An *implicit cursor* is automatically declared by Oracle every time an SQL statement is executed.
- An *explicit cursor* is defined by a program for any query that returns more than one row of data.

Implicit Cursors

If a PL/SQL block executes a SELECT statement that returns a single row, an implicit cursor is created. Likewise, PL/SQL creates implicit cursors for all SQL data manipulation statements (INSERT, UPDATE, and DELETE). No need to do anything to handle these cursors, and one can use the attributes of implicit cursors to access information about a cursor. Implicit cursor attributes store information about DML and DDL statements. The cursor attributes of a SQL statement are given in Table B.

Table B

%ROWCOUNT	When its cursor or cursor variable is opened, %ROWCOUNT has a value zero. Before the fetch of the first row, %ROWCOUNT is equal to zero; thereafter, it equates to the number of rows fetched thus far by Oracle processing. The numeric value of %ROWCOUNT is incremented if the most recent fetch returns a row.
%FOUND	The attribute is TRUE when a cursor has remaining rows to fetch, and FALSE when a cursor has no rows left to fetch. The attribute is NULL until a DML statement is executed. The attribute is TRUE if an INSERT, UPDATE, DELETE, or SELECT INTO statement affected or returned one or more rows.
%NOTFOUND	The attribute is TRUE if a cursor has no rows to fetch, and FALSE when a cursor has some remaining rows to fetch.
%ISOPEN	The attribute is TRUE if a cursor is open, or FALSE if cursor has not been opened or has been closed. This attribute is only used with explicit cursors.

The program makes use of the SQL%FOUND and SQL%ROWCOUNT attributes. This example updates unitprice for a given ingredient id.

```
DECLARE
    unit_price ingredients_new.unitprice%type;
    old_price ingredients_new.unitprice%type;
    ingredient ingredients_new.ingredientid%type :='&ingredient';
BEGIN
    select unitprice into unit_price from ingredients_new where
ingredientid=ingredient;
    IF SQL%NOTFOUND THEN --implicit cursor
      DBMS_OUTPUT.PUT_LINE('Ingredient ' || ingredient || ' not
found');
    else
      old_price:=unit_price;
      unit_price:=unit_price+ unit_price*0.1;
```

```
        UPDATE ingredients_new set unitprice=unit_price where
ingredientid=ingredient;

    IF SQL%FOUND THEN  -- ingredient updated

      DBMS_OUTPUT.PUT_LINE('Updated unitprice for ' || ingredient ||'
from '|| TO_CHAR(old_price) || ' to ' || TO_CHAR(unit_price));

    END IF;

END IF;

EXCEPTION

WHEN NO_DATA_FOUND THEN

      DBMS_OUTPUT.PUT_LINE('no ingredient found');

WHEN TOO_MANY_ROWS THEN

      DBMS_OUTPUT.PUT_LINE('Too many rows selectd');

END;
```

This example updates deletes all ingredients except WATER. The success of this is reported using attributes. The program makes use of the SQL%FOUND and SQL%ROWCOUNT attributes.

```
BEGIN

    DELETE FROM ingredients_new WHERE ingredientid <> 'WATER';

    IF SQL%FOUND THEN  -- ingredients were deleted

        DBMS_OUTPUT.PUT_LINE('Number deleted: ' ||
TO_CHAR(SQL%ROWCOUNT));

    END IF;

END;
```

<u>Explicit Cursors:</u>

The set of rows returned by a query often includes many rows. When a query returns more than one row, we need to declare an explicit cursor in order to process the rows. Declare cursors in the DECLARE section of a PL/SQL block program. Explicit cursors are processed by following a **four-step** model.

- First, declare a cursor. The general format of this declaration is:

  ```
  CURSOR <cursorname> IS <SELECT statement>;
  ```

- The second step in using a cursor is to open it. When we open a cursor, the SQL statement associated with it is parsed (checked for syntax errors).

  ```
  OPEN <cursorname>;
  ```

- Now we are ready to execute a FETCH statement. A FETCH statement actually retrieves the result set produced by a cursor's associated SELECT query. This loads the row addressed by the cursor pointer into variables and moves the cursor pointer to the next row so that it is ready for the next fetch.

  ```
  FETCH <cursorname> INTO <record variable(s)>;
  ```

- Finally, a cursor is disabled with the CLOSE statement. This also causes the result set to become undefined. A closed cursor cannot be reopened. The general format of the CLOSE statement is:

```
CLOSE <cursorname>;
```

Explicit Cursor Attributes:

cursorname%NOTFOUND : evaluates to true, if the last fetch has failed because no more rows were available.
cursorname %FOUND : evaluates to true, if the last fetch succeed because a row was available.
Cursorname%ISOPEN : evaluates to true, if explicit cursor is open.
Cursorname%ROWCOUNT: returns number of rows fetched until now. For eg. a piece of code could look like EXIT when empl%ROWCOUNT=10;

Suppose we wanted to print the name and price of all menu items in descending order by price.

```
DECLARE
c_name   menuitems.name%TYPE;
c_price    menuitems.price%TYPE;
CURSOR menuitemcur  IS SELECT name, price FROM menuitems ORDER BY price
DESC;
BEGIN
OPEN menuitemcur ;
      LOOP
      FETCH  menuitemcur  INTO c_name, c_price;
      DBMS_OUTPUT.PUT_LINE(to_char(c_name) ||'  and  '  ||  c_price);
      EXIT WHEN  menuitemcur%NOTFOUND ;
END LOOP;
CLOSE menuitemcur;
end;
```

Procedure

A procedure is a subprogram that performs a specific action. We specify the name of the procedure, its parameters, its local variables, and the `BEGIN-END` block that contains its code and handles any exceptions.
Create Procedure Syntax

The general syntax to create (or replace) a procedure is shown here.

```
CREATE [OR REPLACE] PROCEDURE <procedure_name> (<parameter1_name>
<mode> <data type>, <parameter2_name> <mode> <data type>, ...) {AS|IS}
    <Variable declarations>
BEGIN
    Executable statements
[EXCEPTION
    Exception handlers]
END <optional procedure name>;
```

Declarative Part

- It contains local declarations
- Keyword DECLARE is NOT used to declare variables and exceptions.
- All declarations are placed in between the IS and BEGIN keywords.

Executable Part

- It contains statements placed between the keywords BEGIN and EXCEPTION (or END).
- At least one statement must appear in executable part of a procedure.

Exception Handling Part

- Contains exception handlers placed between keywords EXCEPTION and END.

Note: At least one statement must appear in the executable part of a procedure (may be NULL). A procedure is called as a PL/SQL statement.

- E.g., proc_raise_sal (empno, amt);

Invoking a procedure

A procedure can be called from any PL/SQL program by specifying their names followed by the parameters.
Syntax:

```
<procedure_name>[(parameter1, paremeter2,…)];
```

Procedures can also be invoked from the SQL prompt, using the EXECUTE command.
Syntax:

```
SQL>EXECUTE <procedure_name>[(parameter1, paremeter2,…)];
```

Parameters

Both procedures and functions can take parameters. Values passed as parameters to a procedure as arguments in a calling statement are termed *actual parameters*. The parameters in a procedure declaration are called *formal parameters*. The difference between these two classifications is critical to understanding the use of the parameter modes. The values stored in actual parameters are values passed to the formal parameters – the formal parameters are like placeholders to store the incoming values. When a procedure completes, the actual parameters are assigned the values of the formal parameters. A formal parameter can have one of three possible modes: (1) IN, (2), OUT, or (3) IN OUT. These modes are outlined in Table

Mode	**Description**
IN	This type of parameter is passed to a procedure as a read-only value tha cannot be changed within the procedure.
OUT	This type of parameter is write-only, and can only appear on the left side of ai assignment statement in the procedure.
IN OUT	This type of parameter combines both IN and OUT; a parameter of this mod is passed to a procedure, and its value can be changed within the procedure.

Creates a stored procedure named AllergyMenu that takes a single parameter, allergen, and finds the items that do not contain the specified allergen

```
create or replace
PROCEDURE AllergyMenu (allergen VARCHAR  ) is
namee varchar2(50);
```

```
pricee number(20);
CURSOR allergycur  IS SELECT name, price
FROM items IT
WHERE NOT EXISTS
(SELECT *
FROM madewith m JOIN ingredients ig ON (m.ingredientid =
ig.ingredientid)
WHERE it.itemid = m.itemid AND ig.name = allergen);
BEGIN

OPEN allergycur ;
      LOOP
      FETCH  allergycur  INTO namee, pricee;
      DBMS_OUTPUT.PUT_LINE(to_char(namee) ||'  and  '  ||
to_char(pricee));
      EXIT WHEN  allergycur%NOTFOUND ;
END LOOP;
CLOSE allergycur;
end AllergyMenu ;
```

Dropping a Procedure:

The SQL statement to drop a procedure is the straight-forward DROP PROCEDURE <procedureName> command. Keep in mind that this is a data definition language (DDL) command, and so an implicit commit executes prior to and immediately after the command.

```
DROP PROCEDURE < PROCEDURE _NAME>;
```

Triggers

Triggers provide a procedural technique to specify and maintain integrity constraints. Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (event) occurs on that table. Modifications on a table may include insert, update, and delete operations.

Basic Trigger Syntax

```
CREATE TRIGGER <trigger name>
{AFTER | BEFORE}
{DELETE | INSERT | UPDATE [OF <column list>]}
ON <table name>
[REFERENCING <reference list>]
```

```
[FOR EACH {ROW | STATEMENT}]
<triggered SQL statement>
```

Some important points to note:

- We can create only BEFORE and AFTER triggers for tables. (INSTEAD OF triggers are only available for views; typically they are used to implement view updates.)
- We may specify up to three triggering events using the keyword OR. Furthermore, UPDATE can be optionally followed by the keyword OF and a list of attribute(s) in <table_name>. If present, the OF clause defines the event to be only an update of the attribute(s) listed after OF. Here are some examples:

```
... INSERT ON R ...
... INSERT OR DELETE OR UPDATE ON R ...
... UPDATE OF A, B OR INSERT ON R ...
```

If FOR EACH ROW option is specified, the trigger is row-level; otherwise, the trigger is statement-level.

Only for row-level triggers:

- The special variables NEW and OLD are available to refer to new and old tuples respectively. **Note:** In the trigger body, NEW and OLD must be preceded by a colon (":"), but in the WHEN clause, they do not have a preceding colon! See example below.
- The REFERENCING clause can be used to assign aliases to the variables NEW and OLD.
- A trigger restriction can be specified in the WHEN clause, enclosed by parentheses. The trigger restriction is a SQL condition that must be satisfied in order for Oracle to fire the trigger. This condition cannot contain subqueries. Without the WHEN clause, the trigger is fired for each row.

<trigger_body> is a PL/SQL block, rather than sequence of SQL statements. Oracle has placed certain restrictions on what we can do in <trigger_body>, in order to avoid situations where one trigger performs an action that triggers a second trigger, which then triggers a third, and so on, which could potentially create an infinite loop. The restrictions on <trigger_body> include:

- We cannot modify the same relation whose modification is the event triggering the trigger.
- We cannot modify a relation connected to the triggering relation by another constraint such as a foreign-key constraint.

We illustrate creating a trigger through an example based on the following two tables:

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);
```

We create a trigger that may insert a tuple into T5 when a tuple is inserted into T4. Specifically, the trigger checks whether the new tuple has a first component 10 or less, and if so inserts the reverse tuple into T5:

```
CREATE TRIGGER oninsert
AFTER INSERT ON T4
REFERENCING NEW AS newRow
```

```
FOR EACH ROW
WHEN (newRow.a <= 10)
BEGIN
INSERT INTO T5 VALUES(:newRow.b, :newRow.a);
END oninsert;
```

Running the CREATE TRIGGER statement only creates the trigger; it does not execute the trigger. Only a triggering event, such as an insertion into T4 in this example, causes the trigger to execute.

```
insert into T4 values (3,'mytigger');
```

Then see whether trigger executed or not by using following command.

```
select * from T5;
```

Write a trigger to prevent any update on b in T4.

```
CREATE OR REPLACE
TRIGGER T4_b
BEFORE UPDATE OF b ON T4
FOR EACH ROW
BEGIN
RAISE_APPLICATION_ERROR(-20000,'CANNOT CHANGE b in T4');
END;
```

Now execute

```
update T4 set b=7;
```

Viewing Defined Procedures, Triggers

```
select * from user_procedures;
```

Dropping Triggers

The DROP TRIGGER statement drops a trigger from the database. Also, if we drop a table, all associated table triggers are also dropped. The syntax is:

```
drop trigger <trigger_name>;
drop trigger oninsert ;
```

Enabling and Disabling Triggers

It is useful to be able to enable and disable triggers. For example, if we need to run a script that does a bulk load of the *equipment* table, we may not want to generate audit trail information regarding the bulk load. Having a table's triggers fire can seriously degrade the performance of a bulk load operation.

An enabled trigger executes the trigger body if the triggering statement is issued. By default, triggers are enabled. A disabled trigger does not execute the trigger body even if the triggering statement is issued. The syntax for enabling and disabling triggers is:

```
-- Disable an individual trigger by name.
ALTER TRIGGER trigger_name DISABLE;
```

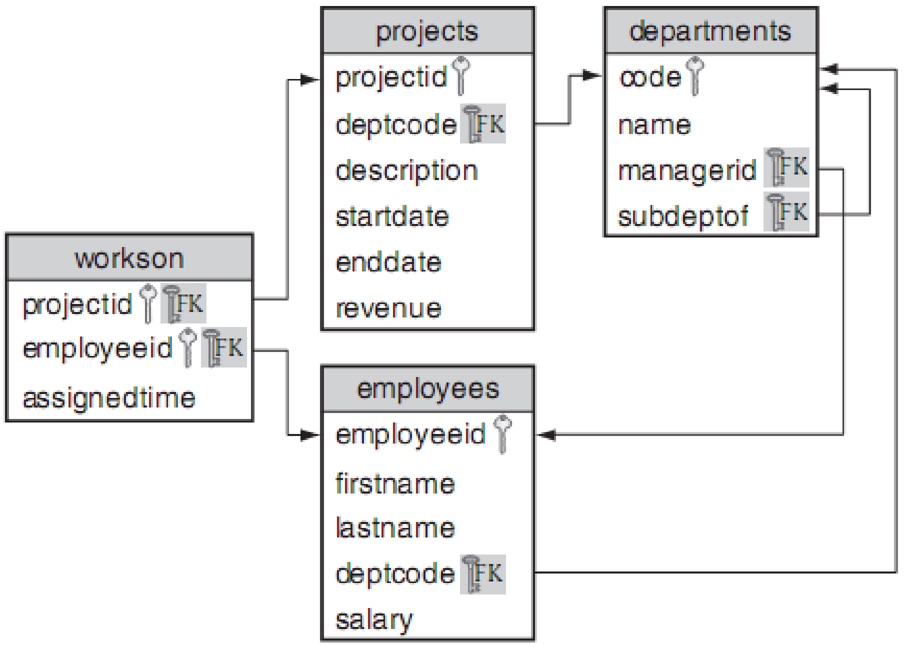
```
-- Disable all triggers associated with a table.
ALTER TABLE table_name DISABLE ALL TRIGGERS;

-- Enable a trigger that was disabled.
ALTER TRIGGER trigger_name ENABLE;

-- Enable all triggers associated with a table.
ALTER TABLE table_name ENABLE ALL TRIGGERS;
```

EXERCISES:

Write a single PL/SQL procedures for each of the following based on employee database created in previous labs. The scheme for employee database is shown below.



1. Print employee name and average salary and standard deviation on salary.
2. Find the first and last name of the highest paid employee(s) in each department.
3. Add a field 'all_assigned' to department table. Write a trigger that sets the flag true after update to 'works-on' table if all the employees in department are assigned projects.

---------&----------