Cursors

Cursors

- To process an SQL statement, ORACLE needs to create an area of memory known as the *context area*; this will have the information needed to process the statement.

- This information includes the number of rows processed by the statement, a pointer to the parsed representation of the statement.

- In a query, the active set refers to the rows that will be returned.

- A cursor is a handle, or pointer, to the context area.
- Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed.
- Two important features about the cursor are
  1. Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
  2. A cursor is named so that it can be referenced.

- The set of rows the cursor holds is referred to as the **active set**.
- You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time.

- There are two types of cursors:

✔ Implicit cursors

✔ Explicit cursors

Types of Cursors

- **There are two types of cursors:**

  1. An ***IMPLICIT*** cursor is automatically declared by Oracle every time an SQL statement is executed. The user will not be aware of this happening and will not be able to control or process the information in an implicit cursor.

  2. An ***EXPLICIT*** cursor is defined by the program for any query that returns more than one row of data. That means the programmer has declared the cursor within the PL/SQL code block.

Implicit Cursors

- Any given PL/SQL block issues an implicit cursor whenever an SQL statement is executed, as long as an explicit cursor does not exist for that SQL statement.

- A cursor is automatically associated with every DML (Data Manipulation) statement (UPDATE, DELETE, INSERT).

- All UPDATE and DELETE statements have cursors that identify the set of rows that will be affected by the operation.

- An INSERT statement needs a place to receive the data that is to be inserted in the database; the implicit cursor fulfills this need.

- The most recently opened cursor is called the “SQL%” Cursor.

Implicit Cursors

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.

- Programmers cannot control the implicit cursors and the information in it.

- Whenever a DML statement *INSERT*, *UPDATE and DELETE* is issued, an implicit cursor is associated with this statement.

- For INSERT operations, the cursor holds the data that needs to be inserted.

- For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.

- The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement.

- Any SQL cursor attribute will be accessed as **sql%attribute_name.**

The Processing of An Implicit Cursor

- The implicit cursor is used to process INSERT, UPDATE, DELETE, and SELECT INTO statements.
- During the processing of an implicit cursor, Oracle automatically performs the OPEN, FETCH, and CLOSE operations.
- An implicit cursor cannot tell you how many rows were affected by an update. SQL%ROWCOUNT returns numbers of rows updated. It can be used as follows:

```
  BEGIN
     UPDATE student
     SET first_name = 'B‘
     WHERE first_name LIKE 'B%';
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
     END;
```

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Syntax for Cursors

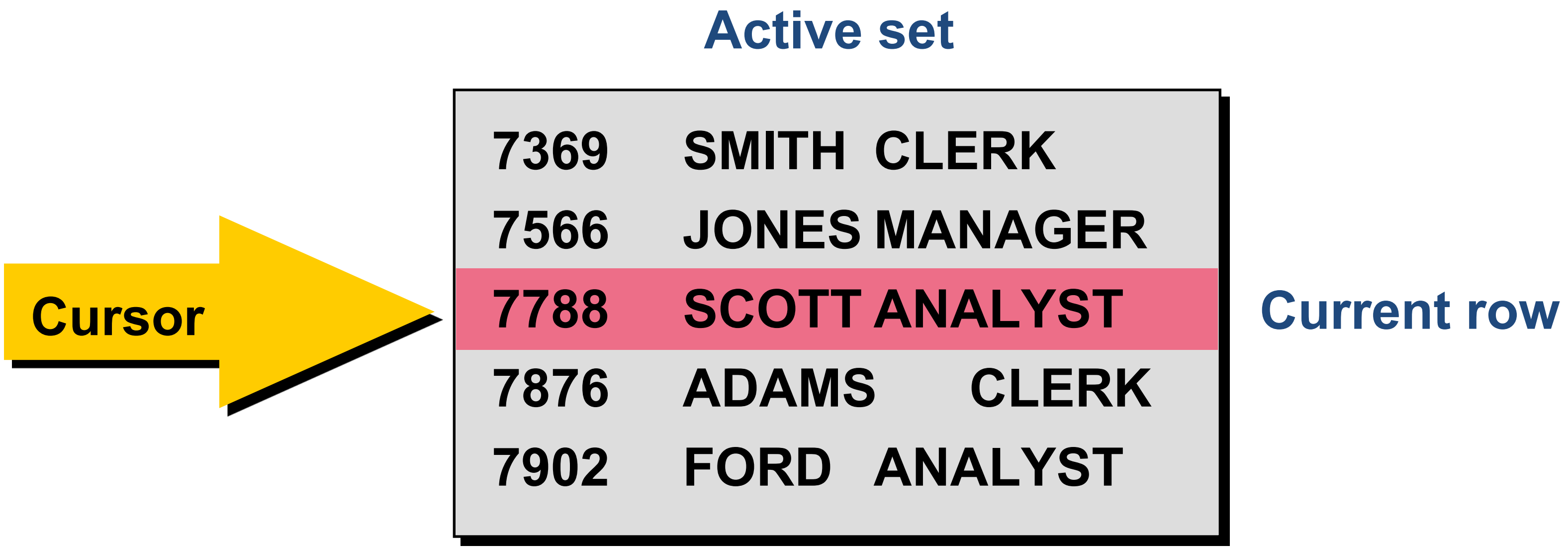
- Declared as a variable in the same way as standard variables
- Identified as cursor type
- SQL included
- E.g.

```
Cursor cur_emp is
     Select emp_id, surname name, grade, salary
      From employee
         Where grade is true;
```

Cursors

- A cursor is a temp store of data.
- The data is populated when the cursor is opened.
- Once opened the data must be moved from the temp area to a local variable to be used by the program.
- These variables must be populated in the same order that the data is held in the cursor.
- The data is looped round till an exit clause is reached.

Cursor Functions



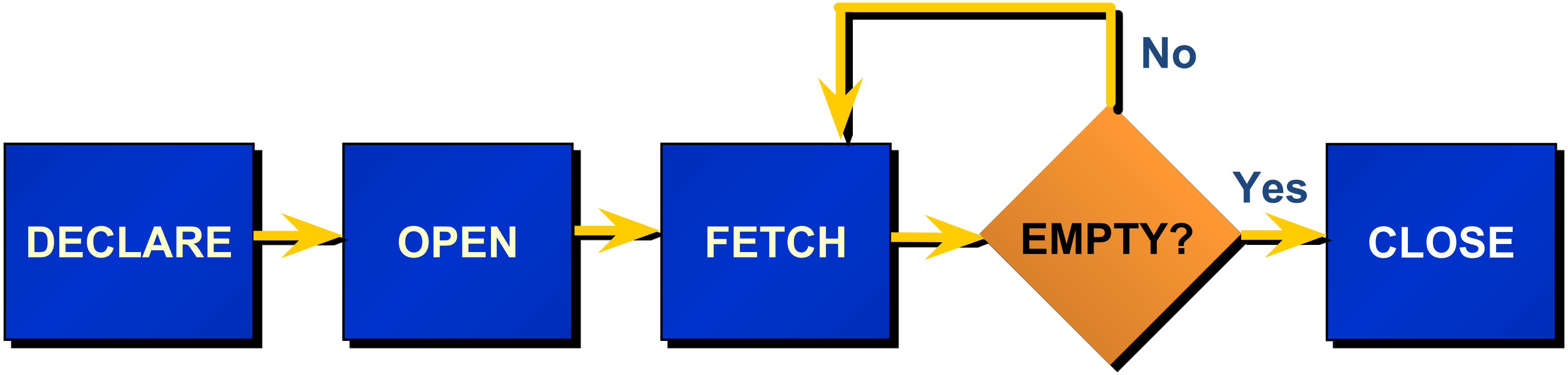
Active set

7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

Cursor

Current row

Controlling Cursor



DECLARE

OPEN

FETCH

EMPTY?

No

Yes

CLOSE

- Create a named SQL area

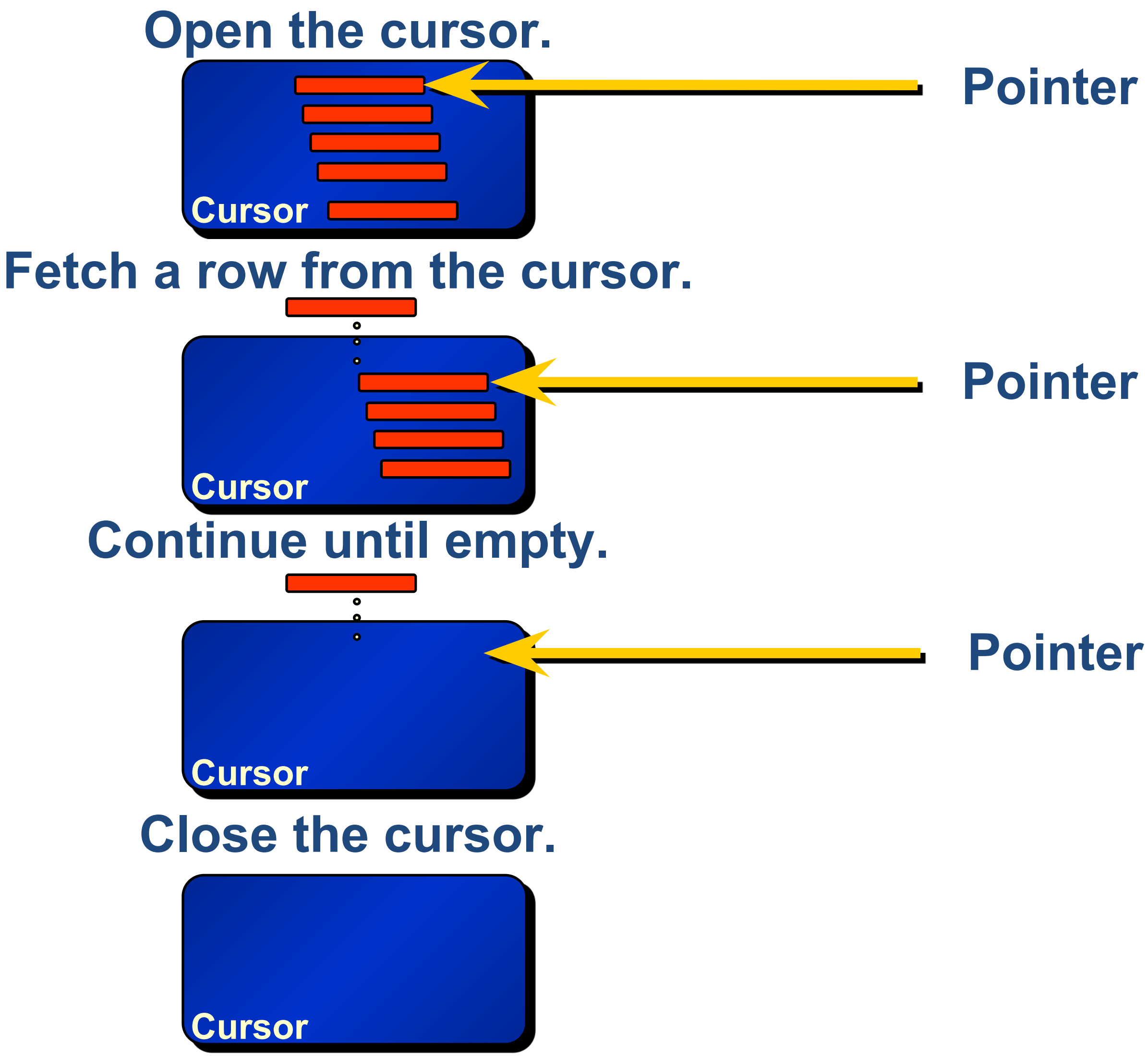
- Identify the active set

- Load the current row into variables

- Test for existing rows
- Return to FETCH if rows found

- Release the active set

Controlling Cursor…



Open the cursor.

Cursor

Pointer

Fetch a row from the cursor.

Cursor

Pointer

Continue until empty.

Cursor

Pointer

Close the cursor.

Cursor

```
Create or replace procedure proc_test as

v_empid   number;

Cursor cur_sample is
        Select empid from employee
          where grade > 4;

Begin
        open cur_sample;
        loop
        fetch cur_sample into v_empid;
        exit when cur_sample%notfound;
            update employee
            set salary = salary + 500
            where empid = v_empid;
        end loop;
End;
```

Declare Cursor

Data returned by cursor

25463
12245
55983
12524
98543

Open cursor for use.

Loops round each value returned by the cursor

Place the value from the cursor into the variable v_empid

Stop when not more records are found

The %ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.
- Example

```
IF NOT cur_sample%ISOPEN THEN
    OPEN cur_sample;
END IF;
LOOP
  FETCH cur_sample...
```

Cursors and Records

- Process the rows of the active set conveniently by fetching values into a PL/SQL RECORD.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT   empno, ename
    FROM     emp;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

Cursor FOR Loops

- Syntax

```
FOR record_name IN cursor_name LOOP
   statement1;
   statement2;
   . . .
END LOOP;
```

- The cursor FOR loop is a shortcut to process cursors.
- Implicitly opens, fetches, and closes cursor.
- The record is implicitly declared.

Cursor FOR Loops: An Example

- Retrieve employees one by one until no more are left.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Example

- The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
        dbms_output.put_line( total_rows || ' customers selected ');
    END IF;
END;
/
```

- **output**
- 6 customers selected
- PL/SQL procedure successfully completed.

❖ Explicit Cursors

- Explicit cursors are programmer-defined cursors for gaining more control over the **context area**.
- An explicit cursor should be defined in the declaration section of the PL/SQL Block.
- It is created on a SELECT Statement which returns more than one row.
- **The syntax for creating an explicit cursor is –**

  CURSOR cursor_name IS select_statement;

- Working with an explicit cursor includes the following steps –

✔ Declaring the cursor for initializing the memory

✔ Opening the cursor for allocating the memory

✔ Fetching the cursor for retrieving the data

✔ Closing the cursor to release the allocated memory

EXPLICIT CURSOR

- The only means of generating an explicit cursor is for the cursor to be named in the DECLARE section of the PL/SQL Block.

- The advantages of declaring an explicit cursor over the indirect implicit cursor are that the explicit cursor gives more programmatic control to the programmer.

- Implicit cursors are less efficient than explicit cursors and thus it is harder to trap data errors.

- The process of working with an explicit cursor consists of the following steps:

- **DECLARING** the cursor. This initializes the cursor into memory.

- **OPENING** the cursor. The previously declared cursor can now be opened; memory is allotted.

- **FETCHING** the cursor. The previously declared and opened cursor can now retrieve data; this is the process of fetching the cursor.

- **CLOSING** the cursor. The previously declared, opened, and fetched cursor must now be closed to release memory allocation.

❑ Declaring the Cursor

- Declaring the cursor defines the cursor with a name and the associated SELECT statement.
- **For example –**

```
        CURSOR c_customers IS SELECT id, name, address
FROM customers;
```

❑ Opening the Cursor

- Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.

- **For example,** we will open the above defined cursor as follows –

```
        OPEN c_customers;
```

Fetching the Cursor

- Fetching the cursor involves accessing one row at a time.
- For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

- Closing the cursor means releasing the allocated memory.
- For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

- **Example:**
- Following is a complete example to illustrate the concepts of **explicit cursors:**

```
DECLARE
    c_id customers.id%type;
    c_nam e customers.name%type;
    c_addr customers.address%type;
    CURSOR c_customers is
        SELECT id, name, address FROM customers;
BEGIN
    OPEN c_customers;
    LOOP
    FETCH c_customers into c_id, c_name, c_addr;
        EXIT WHEN c_customers%notfound;
        dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
    END LOOP;
    CLOSE c_customers;
END;
/
```

Example-Implicit Cursors

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

```
SQL>set serveroutput on
SQL>edit implicit_cursor
 BEGIN
      UPDATE emp_information  SET emp_dept='Web Developer‘
      WHERE emp_name='Saulin';
      IF SQL%FOUND THEN
             dbms_output.put_line('Updated - If Found');
      END IF;
      IF SQL%NOTFOUND THEN
            dbms_output.put_line('NOT Updated - If NOT Found');
      END IF;
      IF SQL%ROWCOUNT>0 THEN
            dbms_output.put_line(SQL%ROWCOUNT||' Rows Updated');

      ELSE dbms_output.put_line('NO Rows Updated Found');
END;
 /
```

- **SQL>@implicit_cursor**

  ```
  Updated - If Found
  1 Rows Updated

  PL/SQL procedure successfully created.
  ```

Example-Explicit Cursor

```
SQL>set serveroutput on
SQL>edit cursor_for_loop
 DECLARE
       cursor c is
       select * from emp_information where emp_no <=2;
       tmp emp_information%rowtype;
BEGIN
       OPEN c;
       FOR tmp IN c LOOP FETCH c into tmp;
       dbms_output.put_line('EMP_No: '||tmp.emp_no);
dbms_output.put_line('EMP_Name: '||tmp.emp_name);
dbms_output.put_line('EMP_Dept: '||tmp.emp_dept);
dbms_output.put_line('EMP_Salary:'||tmp.emp_salary); END Loop;
CLOSE c;
END; /
```

- **SQL>@cursor_for_loop**

  ```
  EMP_No:    1
  EMP_Name:  Forbs ross
  EMP_Dept:  Web Developer
  EMP_Salary:45k

  EMP_No:    2
  EMP_Name:  marks jems
  EMP_Dept:  Program Developer
  EMP_Salary:38k

  PL/SQL procedure successfully completed.
  ```

Triggers

- In programs sometimes it is required to execute certain code followed by certain events and this requirement can be achieved in PL/SQL through triggers.

- Triggers are stored programs that are fired automatically when some event occurs. The code to be fired can be defined as per the requirement.

- Oracle has also provided the facility to mention the event upon which the trigger needs to be fire and the timing of the execution.

- Triggers are stored programs, which are automatically executed or fired when some events occur.

- Triggers are, in fact, written to be executed in response to any of the following events −
- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

- Triggers can be defined on the table, view, schema, or database with which the event is associated.

Trigger Classification

- Triggers can be classified based on the following parameters.
- Classification based on the timing
  - **BEFORE Trigger**: It fires before the specified event has occurred.
  - **AFTER Trigger**: It fires after the specified event has occurred.
  - **INSTEAD OF Trigger:** A special type.

- Classification based on the level
  - **STATEMENT level Trigger**: It fires one time for the specified event statement.
  - **ROW level Trigger:** It fires for each record that got affected in the specified event. (only for DML)

- Classification based on the Event
  - **DML Trigger:** It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  - **DDL Trigger:** It fires when the DDL event is specified (CREATE/ALTER)
  - **DATABASE Trigger:** It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)
- So each trigger is the combination of above parameters.

Creating Triggers

- **The syntax for creating a trigger is −**

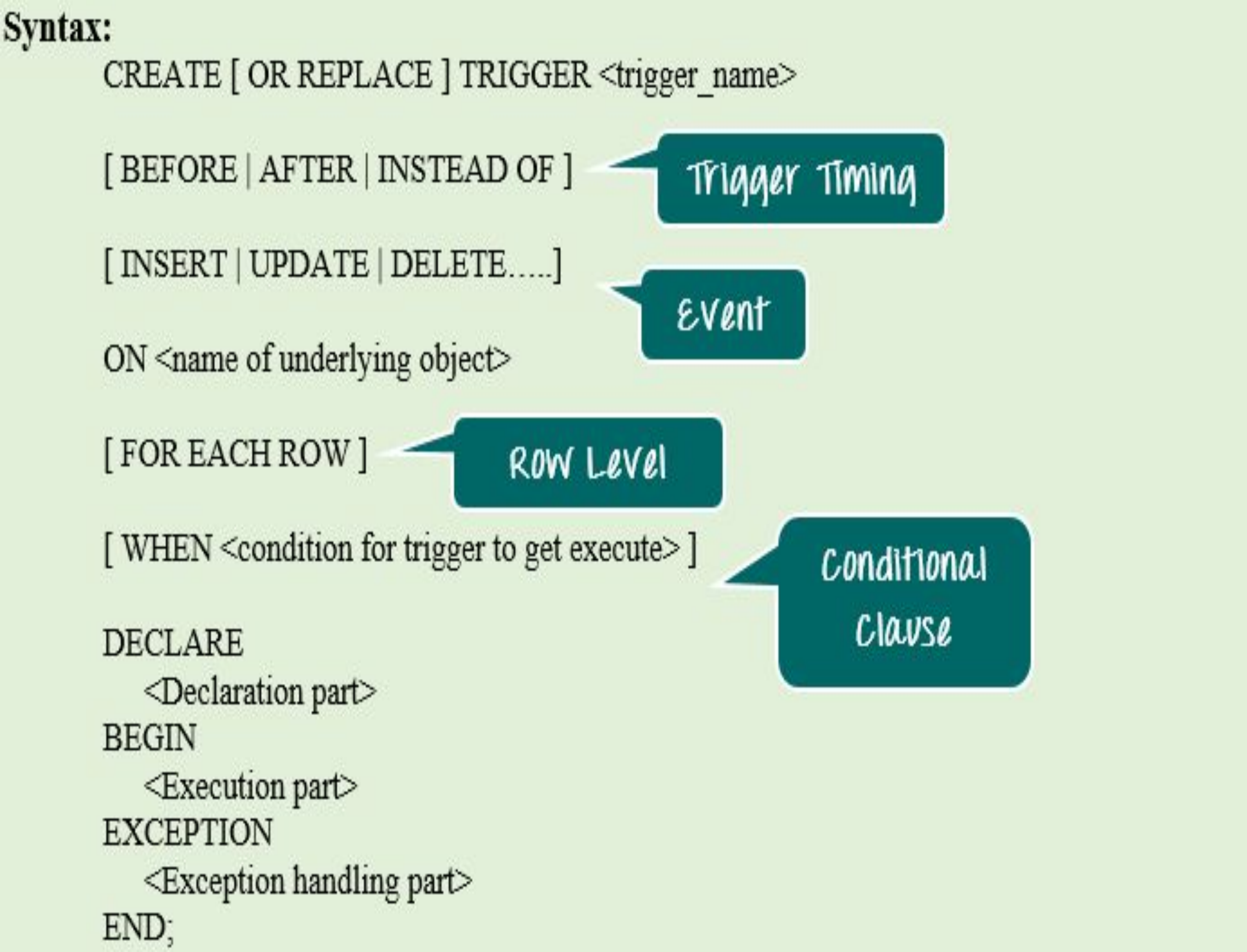
```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
 {INSERT [OR] | UPDATE [OR] | DELETE}
 [OF col_name]
 ON table_name
 [REFERENCING OLD
AS o NEW AS n]
 [FOR EACH ROW]
 WHEN (condition)
DECLARE
     Declaration-statements
BEGIN
     Executable-statements
 EXCEPTION
     Exception-handling-statements
END;
```

- Where,
- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.

- [REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

- WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.



Syntax:

```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>

[ BEFORE | AFTER | INSTEAD OF ]

[ INSERT | UPDATE | DELETE.....]

ON <name of underlying object>

[ FOR EACH ROW ]

[ WHEN <condition for trigger to get execute> ]

DECLARE
   <Declaration part>
BEGIN
   <Execution part>
EXCEPTION
   <Exception handling part>
END;
```

Trigger Timing
Event
Row Level
Conditional Clause

- **Syntax Explanation:**
- **BEFORE/ AFTER** will specify the event timings.

- **INSERT/UPDATE/LOGON/CREATE/etc.** will specify the event for which the trigger needs to be fired.

- **ON clause** will specify on which object the above mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.

- Command **"FOR EACH ROW"** will specify the ROW level trigger.
- **WHEN clause** will specify the additional condition in which the trigger needs to fire.

- The **declaration part, execution part, exception handling part** is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

Example

- Creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

- This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW WHEN (NEW.ID > 0)
DECLARE
    sal_diff number;
BEGIN
     sal_diff := :NEW.salary - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Output:-

```
Trigger created.
```

- The following points need to be considered here −
- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

- Consider DML operations on the CUSTOMERS table.

- **INSERT INTO CUSTOMERS** (ID,NAME,AGE,ADDRESS,SALARY) **VALUES** (7, 'Kriti', 22, 'HP', 7500.00);

- When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
 Salary difference:
```

- Because this is a new record, old salary is not available and the above result comes as null.
- Consider one more DML operation on the CUSTOMERS table.
- The UPDATE statement will update an existing record in the table –
- **UPDATE** customers **SET** salary = salary + 500 WHERE id = 2;

- When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
 Salary difference: 500
```

Benefits of Triggers

- **Triggers can be written for the following purposes** –
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

References

- https://www.tutorialspoint.com/plsql/

END