Experiment – 9

Student Name: Garv Khurana	UID: 21BCS6615
Branch: BE CSE AIML	Section/Group: 21AML-9A
Semester: 03	Date of Performance: 1st Nov 2022
Subject Name: DBMS	Subject Code: 21CSH-243

AIM-To implement the concept of indexes, cursors, triggers and views

SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries. Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against. CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed: CREATE INDEX index_name

ON table_name (column1, column2, ...);

CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index_name

ON table_name (column1, column2, ...);

Note: The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

CREATE INDEX Example

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas: CREATE INDEX idx_pname

ON Persons (LastName, FirstName); DROP

INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

MS Access:

```
DROP INDEX index_name ON table_name;
```

SQL Server:

```
DROP INDEX table_name.index_name;
```

DB2/Oracle:

```
DROP INDEX index_name; MySQL:
```

```
ALTER TABLE table_name

DROP INDEX index_name;
```

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. 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 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 attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes −

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.

Any SQL cursor attribute will be accessed as sql%attribute_name as shown below in the example. Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal |
8500.00 | | 6 | Komal | 22 |
MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

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;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
6 customers selected
```

If you check the records in customers table, you will find that the rows have been updated –

```
Select * from customers;
```

```
+ ---- + ----------+  ----- + -----------+  ----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+ ---- + ----------+  ----- + -----------+  ----------+
| 1  | Ramesh | 32   | Ahmedabad | 2500.00   |
| 2  | Khilan | 25   | Delhi | 2000.00      |
| 3  | kaushik | 23   | Kota | 2500.00     |
| 4  | Chaitali | 25   | Mumbai | 7000.00        |
| 5  | Hardik | 27   | Bhopal | 9000.00          |
| 6  | Komal | 22   | MP | 5000.00        |
+ ---- + ----------+  -----+  -----------+  ----------+
```

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

Declaring the Cursor

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

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

Example

above-opened cursor as follows − CLOSE c_customers;

Following is a complete example to illustrate the concepts of explicit cursors &minua;

When the above code is executed at the SQL prompt, it produces the following result −

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
```

PL/SQL procedure successfully completed.

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.

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

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. Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters

–

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

The following program 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 –

When the above code is executed at the SQL prompt, it produces the following result −

```
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 recordlevel 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

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement,

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

which will create a new record in the table −

When a record is created in the CUSTOMERS table, the above create trigger,

```
Old salary:

New salary: 7500
Salary difference:
```

display_salary_changes will be fired and it will display the following result −

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

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform 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 −

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW

statement. CREATE VIEW Syntax CREATE

VIEW view_name AS

SELECT column1, column2, ...

FROM table_name

WHERE condition;

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.
SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil: Example

CREATE VIEW [Brazil Customers] AS

```
SELECT CustomerName, ContactName
FROM Customers

WHERE Country = 'Brazil';
```

We can query the view above as follows: Example

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price: Example

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

We can query the view above as follows:

Example

```
SELECT * FROM [Products Above Average Price];
```

Evaluation Grid (To be created as per the SOP and Assessment guidelines by the faculty):

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.			
2.			
3.			