Database Systems

Lab 09

Triggers

Triggers

- A trigger is an event within the DBMS that can cause some code to execute automatically

Syntax

- CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE|AFTER|INSTEAD OF}
  {INSERT [OR] | UPDATE [OR] | DELETE}
  [OF col_name]
  ON table_name
  [FOR EACH ROW]
  WHEN (condition)
  BEGIN
  --SQL Statements
  END;

For Example:

- The price of a product changes constantly. It is important to maintain the history of the prices of
  the products

- We can create a trigger to update the 'product_price_history' table when the price of the product
  is updated in the 'product' table

1) Create the product table and product_price_history

- CREATE TABLe product_price_history
  (product_id number(5),
  product_name VARCHAR(20),
  supplier_name VARCHAR (20),
  unit_price number(7,2));

- CREATE TABLE product
  (product_id number(5),
  product_name VARCHAR(20),
  supplier_name VARCHAR (20),
  unit_price number(7,2))

2) Create the product_history_trigger and execute it

```
CREATE OR REPLACE TRIGGER PRICE_HISTORY_TRIGGER
BEFORE UPDATE OF UNIT_PRICE ON PRODUCT
FOR EACH ROW
BEGIN
INSERT INTO PRODUCT_PRICE_HISTORY
VALUES(:OLD.PRODUCT_ID,:OLD.PRODUCT_NAME,
:OLD.SUPPLIER_NAME,:OLD.UNIT_PRICE);
END;
/
```

3) Lets insert & update the price of a product.

- Insert into product values (100, ‘Laptop’, ‘Dell’, 262.22);
  Insert into product values (101, ‘Laptop’, ‘HP’, 362.22);

- UPDATE PRODUCT SET unit_price=800 WHERE product_id=100;

- Once the above query is executed, the trigger fires and updates the 'product_price_history' table.

Types of PL/SQL Triggers

- There are two types of triggers based on the which level it is triggered.

1) Row Level Trigger- An event is triggered for each row updated, inserted or deleted.

2) Statement Level Trigger- An event is triggered for each sql statement executed.

PL/SQL Triggers Execution Hierarchy

- The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level triggers fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

For Example:

- Let's create a table 'product_check' which we can use to store messages when triggers are fired.

- CREATE TABLE product_check
  (Message VARCHAR(50),
  Current_Date Date
  );

- Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level:

- This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level

- CREATE OR REPLACE TRIGGER BEFORE_UPDATE_STAT_PRODUCT
- BEFORE UPDATE ON PRODUCT
- BEGIN
- INSERT INTO PRODUCT_CHECK VALUES('BEFORE UPDATE,STATEMENT LEVEL',SYSDATE);
- END;
- /

2) BEFORE UPDATE, Row Level:

- This trigger will insert a record into the table 'product_check' before each row is update.

- CREATE OR REPLACE TRIGGER BEFORE_UPDATE_ROW_PRODUCT
- BEFORE UPDATE ON PRODUCT
- FOR EACH ROW
- BEGIN
- INSERT INTO PRODUCT_CHECK VALUES('BEFORE UPDATE ROW LEVEL',SYSDATE);
- END;
- /

3) AFTER UPDATE, Statement Level:

- This trigger will insert a record into the table 'product_check' after a sql update statement is excuted, at the statement level.

- CREATE OR REPLACE TRIGGER AFTER_UPDATE_STAT_PRODUCT
- AFTER UPDATE ON PRODUCT
- BEGIN
- INSERT INTO PRODUCT_CHECK VALUES ('AFTER UPDATE,STATEMENT LEVEL',SYSDATE);
- END;
- /

4) AFTER UPDATE, Row Level:

- This trigger will insert a record into the table 'product-check' after each row is updated.

- CREATE OR REPLACE TRIGGER AFTER_UPDATE_ROW_PRODUCT
- AFTER UPDATE ON PRODUCT
- BEGIN
- INSERT INTO PRODUCT_CHECK VALUES ('After update, row level',sysdate);
- END;
- /

Now lets execute an update statement on table product.

- UPDATE PRODUCT SET unit_price = 800
  WHERE product_id in (100,101);

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

- SELECT * FROM product_check;

Output:

- Message Current_Date

  Before update, statement level 10-Nov-2021
  Before update, row level 10-Nov-2021
  After update, Row level 10-Nov-2021
  Before update, Row level 10-Nov-2021
  After update, Row level 10-Nov-2021
  After update, statement level 10-Nov-2021

Exercise

- In this exercise, you are required to create a database for a wiki. A wiki is a website that *allows collaborative modification of its content and structure directly from the web browser* (Wikipedia)
- To make collaborative edition easier, you need to keep an history of the modifications of each page. To achieve this, you will use a simple model that simply keeps each version of each page as a separate row:
    - PageRevision(_name_, _date_, author, text)
- **Question 1:** To make it easier to access the wiki, create a TABLE Page(name, last_author, text) that shows only the latest version of each page.
- **Question 2:** Create a trigger on your newly created table so that when a user tries to update a given page, a new revision is created instead.
- Sometimes, pages on the wiki needs to be completely deleted (for instance, if a page contains sensitive information or copyrighted content). In that case, we want to remove all revisions of the page from the database but we still want to remember that the page has existed but has been deleted. To this end, we add the following table to our database:
    - DeleteLog(_pagename_, _date_)
- **Question 3:** Write a trigger on the Page table such that when a page is deleted:
    - all its revisions are removed from the database
    - the deletion is recorded in the DeleteLog.