Birla Institute of Technology and Science, Pilani
Database Systems
Lab No #8

Today's Topics

- ❖ Triggers
- ❖ Indexing

Triggers

TSQL block or TSQL procedure associated with table, view, schema, or the database
It implicitly gets executed whenever a particular event takes place and the event might be in the form of:

- System event
- DML statement being issued against the table
- SQL Server also allows creating triggers for views

Features of Triggers

- Trigger includes execution of SQL and TSQL statements as a unit.
- Triggers are stored separately from their associated tables in Db.
- Triggers are similar to subprograms in the following ways:
  - Implementing TSQL blocks with declarative, executable, and exception-handling sections
  - Can be stored in Database and can invoke other stored procedures

Benefits of Triggers

- Audit data modifications
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicated tables
- Enable building complex updatable views
- Useful in tracking system events

<u>Types of Triggers</u>

- Database trigger
  - Executes whenever a data event or system event occurs on a schema or database
    - ✓ Data Event implies DML or DDL.
    - ✓ System Event implies SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.

<u>Difference between Triggers and Subprograms</u>

Triggers	Subprograms
Defined with CREATE TRIGGER	Defined with CREATE PROCEDURE / CREATE FUNCTION
Are executed implicitly whenever the triggering event occurs	Are executed explicitly by a user, application, or a trigger
Do not accept arguments	Can accept arguments
Data dictionary contains source code in USER_TRIGGERS	Data dictionary contains source code in USER_SOURCE
Implicitly invoked	Explicitly invoked
COMMIT, SAVEPOINT, and ROLLBACK are not allowed	COMMIT, SAVEPOINT, and ROLLBACK are allowed

<u>Application Triggers</u>

- Are executed implicitly when a DML event occurs with respect to an application (usually a frontend application)
- Operates in the Client layer in client/server architecture

<u>Database Triggers</u>

- Executes whenever a data event or system event occurs on a schema or database
  - Data Event implies DML or DDL.
  - System Event implies SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.
- Are TSQL program units that are associated with a specific table or view
- Operate in Client/Server architecture in the server layer
- Are compiled and stored permanently in the database
  - Hence the name “database” triggers!
- Are integrated with middle tier applications

Parts of Database Triggers

- Whenever the trigger event occurs the database trigger is implicitly fired and the TSQL block performs the action.
- The four parts of database trigger are:
  - Trigger Timing
    - ✓ Determines when the trigger executes in relation to the triggering event
    - ✓ Can be FOR, AFTER, or INSTEAD OF
  - Triggering Event
    - ✓ (or statement) is the SQL statement that causes a trigger to be executed
    - ✓ Can be an INSERT, UPDATE, or DELETE statemznt for a specific table or view
  - Trigger Action
    - ✓ Is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed
    - ✓ Is executed when a triggering statement is issued and the trigger restriction (if present) evaluates to TRUE

Defining Database Triggers

- Database triggers can be defined on:
  - DML statements
    - ✓ INSERT, UPDATE, and DELETE
  - DDL events
    - ✓ CREATE, DROP, and ALTER
  - Database operations
    - ✓ SERVERERROR, LOGON, LOGOFF, STARTUP, and SHUTDOWN

Creating Database Trigger

- CREATE TRIGGER is used to create a new trigger.
- ALTER TRIGGER is used to replace an existing trigger.
- To create a trigger the user must have:

```
CREATE [ OR ALTER ] TRIGGER trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]  ]
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
GO
```

A trigger that fires before an operation will make sure the operation can execute correctly. A trigger that fires after the table modification usually will cause another action to take place.

Triggers are added to the database using the CREATE TRIGGER command.

FOR | AFTER :
AFTER specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires.
AFTER is the default when FOR is the only keyword specified. AFTER triggers cannot be defined on views.

INSTEAD OF :
Specifies that the DML trigger is executed *instead of* the triggering SQL statement, therefore, overriding the actions of the triggering statements. At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. However, you can define views on views where each view has its own INSTEAD OF trigger. INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. The user must remove that option by using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] } :
Specifies the data modification statements that activate the DML trigger when it is tried against this table or view. At least one option must be specified. Any combination of these options in any order is allowed in the trigger definition.
For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON UPDATE.

WITH APPEND
Specifies that an additional trigger of an existing type should be added. WITH APPEND cannot be used with INSTEAD OF triggers or if AFTER trigger is explicitly stated. WITH APPEND can be used only when FOR is specified, without INSTEAD OF or AFTER, for backward compatibility reasons.

Execution of Trigger

- It is executed implicitly when DML, DDL, or system event occurs.
- The act of executing a trigger is also known as **firing the trigger**.
- When multiple triggers are to be executed, they have to be executed in a sequence.
- Database triggers execute with the privileges of the owner, not the current user.
- The following list shows the sample execution sequence:

Example:

Assume that we always have a 100% markup on the price of items. In other words, the price of an item is always twice the cost of the sum of the ingredients. We can create a trigger to update the price of a meal when the ingredient costs are modified.

```
CREATE TRIGGER markup ON ingredients
AFTER UPDATE as
UPDATE items SET price =
(SELECT 2 * SUM(quantity * unitprice)
FROM madewith m , ingredients i
WHERE m.ingredientid = i.ingredientid AND items.itemid = m.itemid)
```

Two special tables inserted and deleted can be used in a trigger.

Inserted: Table containing the new values of row / rows being updated / inserted.
Deleted: Table containing the previous values of the row / rows being updated / deleted.

Example:

/* create a view new_emp_dept */

```
Create view new_emp_dept as
Select e.employeeid, firstname, e.deptcode, e.salary, d.name
From employees e, departments d
Where e.deptcode = d.code;
```

/* Now suppose when a delete statement is executed for the view specifying a department number, following actions are to be performed simultaneously:

1. Delete all rows from the EMP table for a particular department number
2. Delete the corresponding department information from the DEPT table

```
CREATE TRIGGER new_emp_dept_remove
ON new_emp_dept
INSTEAD OF DELETE
AS
DELETE FROM employees where employeeid in
(select employeeid from deleted);
DELETE FROM departments where code in (select code from deleted)
GO
```

Managing Triggers

- Triggers can be enabled, disabled, and recompiled.
  - To disable or enable one database trigger:

```
DISABLE TRIGGER { [ schema_name . ] trigger_name [ ,...n ] |
ALL }
```

```
ON { object_name | DATABASE | ALL SERVER } [ ; ]
```

- To disable or enable or alter all triggers for a table:

  ```
  DISABLE Trigger ALL ON ALL SERVER;
  ```

- To recompile a trigger for a table:

  ```
  sp_recompile [ @objname = ] 'object'
  ```

Modifying Triggers

- It is possible to straight away modify the existing triggers using the ALTER triggers command and the same syntax as in CREATE.
- ALTER TRIGGER supports manually updatable views through INSTEAD OF triggers on tables and views. SQL Server applies ALTER TRIGGER the same way for all kinds of triggers (AFTER, INSTEAD-OF).

Removing Triggers

- Triggers can be deleted from the system memory.
- They are removed using the DROP TRIGGER command.
- To drop a trigger one must either :
  - Own the trigger or
  - Have the DROP ANY TRIGGER system privilege

Syntax:

```
DROP TRIGGER [ IF EXISTS ] [schema_name.]trigger_name [ ,...n ] [ ; ]
```

Example:

```
DROP TRIGGER new_emp_dept_remove;
```

Exercise:

1. Create a trigger that restores the values before an update operation on the employees table if the salary exceeds 100000. (Hint: Use the inserted and deleted tables to look at the old and new values in the table respectively.)
2. Create one single trigger which raises different errors depending on whether the operation is INSERT, DELETE and UPDATE. (Hint: Check the existence of rows in the inserted, deleted tables to identify each operation. Also, use RAISERROR to raise an error)
3. Create a trigger on the table employees, which after an update or insert, converts all the values of first and last names to upper case. (Hint: Use cursors to retrieve the values of each row and modify them.)

Indexes

An index for a database table is similar in concept to a book index.

- The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.
- Generally, you should create an index on a column when you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is
  Create an index when a query retrieves <= 10 percent of the total rows in a table.
- This means the column for the index should contain a wide range of values. These types of indexes are called "B-tree" indexes

Creating a B-Tree Index

Syntax:

```
CREATE [UNIQUE] INDEX index_name ON table_name(column_name[,
column_name ...]);
```

where

- UNIQUE means that the values in the indexed columns must be unique.
- index_name is the name of the index.
- table_name is a database table.
- column_name is the indexed column. You can create an index on multiple columns (such an index is known as a composite index).

Query to be executed

```
SELECT customer_id, first_name, last_name FROM customers
WHERE last_name = 'Brown';
```

The following CREATE INDEX statement creates an index named i_customers_last_name on the last_name column of the customers table (I always put i_ at the start of index names):

```
CREATE INDEX i_customers_last_name
ON customers(last_name);
```

Once the index has been created, the previous query will take less time to complete. You can enforce uniqueness of column values using a unique index. For example, the following statement creates a unique index named i_customers_phone on the customers.phone column:

```
CREATE UNIQUE INDEX i_customers_phone ON
customers(phone);
```

You can also create a composite index on multiple columns. For example, the following statement creates a composite index named i_employees_first_last_name on the first_name and last_name columns of the employees table:

```
CREATE INDEX i_employees_first_last_name  ON
employees(first_name, last_name);
```

Creating a Function-Based Index

```
SELECT first_name, last_name FROM customers
WHERE last_name = UPPER('BROWN');
```

Because this query uses a function—UPPER(), in this case—the i_customers_last_name index isn't used. If you want an index to be based on the results of a function, you must create a function-based index, such as:

```
CREATE INDEX i_func_customers_last_name ON
customers(UPPER(last_name));
```

CONNECT system/manager
ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;

Retrieving Information on Indexes

Customers and employees tables:

```
COLUMN table_name FORMAT a15 COLUMN column_name FORMAT a15
SELECT index_name, table_name, column_name FROM user_ind_columns
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

```
INDEX_NAME TABLE_NAME COLUMN_NAME
------------------------------ --------------- ------------
CUSTOMERS_PK CUSTOMERS CUSTOMER_ID
```

EMPLOYEES_PK EMPLOYEES EMPLOYEE_ID
I_CUSTOMERS_LAST_NAME CUSTOMERS
LAST_NAME

I_CUSTOMERS_PHONE CUSTOMERS PHONE
I_EMPLOYEES_FIRST_LAST_NAME EMPLOYEES
LAST_NAME I_EMPLOYEES_FIRST_LAST_NAME
EMPLOYEES FIRST_NAME
I_FUNC_CUSTOMERS_LAST_NAME CUSTOMERS
SYS_NC00006$

Modifying an Index

You modify an index using ALTER INDEX. The following example renames the i_customers_phone index to i_customers_phone_number:

```
ALTER INDEX i_customers_phone
RENAME TO i_customers_phone_number;
```

Dropping an Index

You drop an index using the DROP INDEX statement. The following example drops the I_customers_phone_number index:

```
DROP INDEX i_customers_phone_number;
```

****