# SQL Triggers

Triggers monitors a database and executes when an event occurs in the database server.

- like insertion,
- deletion or
- updation of data.

It is a database object which is bound to a table and is executed automatically.

You can’t explicitly invoke **trigger**s.

- The only way to do this is by performing the required action on the table that they are assigned to.

# SQL Triggers

Objective: to monitor a database and take action when a condition occurs

Triggers include the following:

- event (e.g., an update operation)
- condition
- action (to be taken when the condition is satisfied)

Triggers are classified into two main types:

- After Triggers (For Triggers)
- Instead Of Triggers

# SQL Triggers: An Example

**Using a trigger with a reminder message**

```
CREATE TRIGGER reminder1
ON Employee
AFTER INSERT, UPDATE
AS PRINT 'Notify employee added'
```

https://www.codeproject.com/Articles/25600/Triggers-SQL-Server

# SQL Triggers: An Example

A trigger to compare an employee's salary to his/her supervisor after insert or update operations:

```
CREATE TRIGGER Emp_Salary ON Employee
    FOR INSERT, UPDATE
    AS
    IF EXISTS (SELECT * FROM inserted as i JOIN Employee as e ON
                        i.super_SSN= e.SSN WHERE i.salary > e.salary)
    BEGIN
            PRINT 'Employee salary is greater than the Supervisor Salary'
    END
```

```
INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, Super_SSN, Salary)
VALUES ('Richard', 'Marini', '653298653','123456789',500000)
```

# SQL Triggers

- **CREATE** TRIGGER SampleTrigger **ON** Employee
- **INSTEAD OF  INSERT**
- **AS**
- SELECT * FROM Employee

- To fire the trigger we can insert a row in table and it will show list of all user instead of inserting into the table

```
INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, Super_SSN, Salary)
VALUES ('Richard', 'Marini', '653298653','123456789',500000)
```

INSTEAD OF triggers are usually used to correctly update views that are based on multiple tables.

# SQL Triggers: An Example

**Using a trigger with a reminder message**

```
CREATE TRIGGER reminder2
ON  employee
AFTER INSERT, UPDATE, DELETE
AS
        EXEC msdb.dbo.sp_send_dbmail
        @profile_name = 'The Administrator',
        @recipients = 'danw@Adventure-Works.com',
        @body = 'Don''t forget to print a report',
        @subject = 'Reminder';
```

https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-2017

# Trigger

It is required that a team do not submit more than two proposals. Write a SQL query or trigger or view to solve this issue?

- INSTEAD OF triggers are run in place of the Insert command.
  - If you run insert command in instead of trigger it will again call the trigger so on.
- You can either use After (FOR) trigger
  - check if the inserted row has violated the given condition. If yes then delete it.

**OR**

- you can handle it at frontend application using Sql query to check if the given team has already submitted two projects then donot insert.

# Why Transactions?

- *Transaction* is a process involving database queries and/or modification.
- Database systems are normally being accessed by many users or processes at the same time.
- Example- ATM
- Formed in SQL from single statements or explicit programmer control

# ACID Transactions

| | |
|---|---|
| *Atomic* | • Whole transaction or none is done. |
| *Consistent* | • Database constraints preserved. |
| *Isolated* | • It appears to the user as if only one process executes at a time. |
| *Durable* | • Effects of a process survive a crash. |

*Optional: weaker forms of transactions are often supported as well.*

# T-SQL AND Transactions

SQL has following transaction modes.

- Autocommit transactions
  - Each individual SQL statement = transaction.

- Explicit transactions

  BEGIN TRANSACTION

  [SQL statements]

  COMMIT or ROLLBACK

# Transaction Support in TSQL

- BEGIN TRAN
- UPDATE Department
- SET Mgr_ssn = 123456789
- WHERE DNumber = 1
- UPDATE Department
- SET Mgr_start_date = '1981-06-19'
- WHERE Dnumber = 1
- COMMIT TRAN

# Transaction Support in SQL

Potential problem with lower isolation levels:

- **Dirty Read**
  - Reading a value that was written by a failed transaction.
- **Nonrepeatable Read**
  - Allowing another transaction to write a new value between multiple reads of one transaction.
    - A transaction T1 reads a given value from a table.
    - If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

# Transaction Support in SQL

- Potential problem with lower isolation levels (contd.):
  - Phantoms
    - New rows being read using the same read with a condition.
      - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

# Transaction Support in TSQL

**Table 21.1** Possible Violations Based on Isolation Levels as Defined in SQL

| | Type of Violation | | |
|---|---|---|---|
| **Isolation Level** | **Dirty Read** | **Nonrepeatable Read** | **Phantom** |
| READ UNCOMMITTED | Yes | Yes | Yes |
| READ COMMITTED | No | Yes | Yes |
| REPEATABLE READ | No | No | Yes |
| SERIALIZABLE | No | No | No |

# Transaction Support in TSQL

1. "Dirty reads"

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

# ACID Transactions

| | |
|---|---|
| *Atomic* | • Whole transaction or none is done. |
| *Consistent* | • Database constraints preserved. |
| *Isolated* | • It appears to the user as if only one process executes at a time. |
| *Durable* | • Effects of a process survive a crash. |

*Optional: weaker forms of transactions are often supported as well.*

# Example of *Fund Transfer*

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Atomicity requirement** :
  - if the transaction **fails** after step 3 and before step 6,
    - the **system** should **ensure** that :
      - its **updates** are *not reflected* in the database,
      - else an *inconsistency* will result.

# Example of *Fund Transfer*

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Consistency requirement** :
  - the **sum** of **A** and **B** is:
    - **unchanged** by the execution of the transaction.

# Example of *Fund Transfer* (Cont.)

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Isolation requirement** —
  - if between steps 3 and 6, another transaction is allowed to access the partially updated database,
    - it will see an inconsistent database (the sum *A + B* will be less than it should be).
  - Isolation can be **ensured** trivially by:
    - running transactions **serially**, that is **one** after the **other**.
  - *However*, executing multiple *transactions* **concurrently** has significant **benefits**.

# Example of *Fund Transfer* (Cont.)

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Durability requirement** :
  - once the user has been notified that the transaction has **completed** :
    - (i.e., the transfer of the $50 has taken place),
    - the **updates** to the database by the transaction **must persist**
      - despite *failures*.