**Trigger**

1. It monitors the database from the application
2. It uses to enforce integrity constraints on the other hand, those constraint is not enforced by the constraint system. And one reason is that some constraint system are limited.

|  |
| --- |
| **Create** **Trigger** name  **Before** | **After** | Instead of events  [referenceing-variables]  [For each row]  **WHEN** (condition)  Action |

|  |
| --- |
| **insert** on Table(T),  **delete** on Table(T),  **update** on Table(T) |

**Instead of event/ specific event:** It can be specified are:

|  |
| --- |
| **insert** on Table T,  or **delete** on a Table T,  or **update** two particular columns on a Table T  (actually column themselves are optional) |

**For each row:** is optional clause that states that’s the trigger should be activated once for each modified tuple.

* For example: when we run say **delete** command on the database; that **delete** command might delete, say 10 tuples.

If we specify **for each row** in our trigger, the n we will run the trigger 10 times, once for each delete tuple.

* On the other hand, if for each row is not present, then we will execute the trigger once.

|  |
| --- |
| **old row,**  **new row,**  **old table,**  **new table** |

**Referencing variables:** They give us a way to reference the data that was modified that caused the trigger to be activated. So we have in the **referencing variables**, these are **key** words,

It’s possible to have up to all four of these on a single tirgger, **but there are certain restriction**.

* First of all we have a trigger that is based on **insertions**, then we can only refer to **new** data. That would be the new insert data.
* If we have a trigger activated by **deletions**, then we can only refer to the **old** data.
* If we have a trigger that is based on **update,** then we can referto both **old** and **new**. And we will get the previous version of the update value and new version of those values.

So we can only have both **old** and **new** in the case when our trigger is activated by on **update**.

If we have **row level tirgger** that will be triggered once for each modified tuple , but after the entire statement has run.

Example of **row level(delete):** In the case of delete we can only have **old,** but we could have for a row level trigger, both **old row** and **old table**.

* The **old row** would refer to the specific tuple(row) that the trigger is activated for. Again, if we have 10 rows, and it will be activated 10 times. Here, old table is that which is refering specifically to **set of tuples**(rows) that were in this case deleted.
* If our tuple(row), if our trigger is **not for each row**, if it’s a **statement level trigger**, then we can not refer to row level variables, but we have only this table level variables.

But we have **statement level insert**, then we can only have **new table**.

* If we have **row level delete**, then we can have both **old row** and **old table**.

But if it’s a **statement level delete**, then we can only have **old table**.

* **Update:** If we have **row level** update, then we can have four of these variables (**old row, new row, old table, new table).**

But if we have **statement level** update, then we can only have the **old table** and the **new table** variables.

**Note:**

1. Row level means for each row is present.
2. Statement level means for each row is not present.

**Action:** It is the last part in trigger statements. It is expressed like a sql statement.

|  |
| --- |
| **BEGIN**  Sql statement to execute after triggered  **END** |

**Example of referential integrity:**

|  |
| --- |
| **CREATE** Trigger Cascade  **After** Delete On S  Referencing Old Row As O  **For Each Row**  [no condition]  **DELETE** **FROM** R **WHERE** A = O.B |

Here we have table **R** and its attribute is **A** and table **S** and its attribute is **B**. We want to implement cascaded delete.

**Cascaded delete:** means, if we delete from table ‘S’ then any ‘a’ values that reference the deleted B values, will themselves also be deleted.

Here row level trigger is activated once for Each deleted row. Here deleted row is to be called O.

Here there is no condition.

So whenever we have a delete from **S**, then in our action, we are going to delete from R all tuples where the **A** value equals the **B** value of the deleted tuple from **S.**

**To delete, trigger as a statement level:** Here we don’t have old row anymore. But we have **only old tables**. Here are we set up a variable called OT for old table.

Referencing old table(OT) is not the old value of table; but just the value of the tuple that has been deleted. At this time we are looking for tuple in **R** where the **A** value is among the **B** values that were deleted from **S**.

[**R.A** references **S.B** cascaded delete]

|  |
| --- |
| **CREATE** **Triggers** cascade  **After** **DELETE** **ON** S  Referencing Old Table **AS** OT  **DELETE** **FROM** R **WHERE** A **IN** (**SELECT** B **FROM** OT) |

**Not NULL:** By default, a column can hold NULL value. NOT NULL constraint enforces a column to not accept NULL value.

Example:

|  |
| --- |
| **CREATE TABLE** Persons(  ID int NOT NULL,  LastName varchar(255) **NOT NULL**,  Age int  ); |

**Key**

**Primary key:** Primary key constraint uniquely identifies each record in a database. Primary keys must certain **UNIQUE** values, and can’t contain **NULL** values. A table can have **only one primary key**, which may **consist of single or multiple columns**.

Example:

|  |
| --- |
| **CREATE** **TABLE** Persons(  ID int **NOT NULL**  LastName varchar(255) **NOT NULL**  Age int,  **CONSTRAINT** PK\_Person **Primary KEY**(ID, LastName)  ); |

**Example (Create trigger):**

|  |
| --- |
| **CREATE TRIGGER** R1  **AFTER INSERT ON** Student  **FOR EACH ROW**  **WHEN NEW.**GPA > 3.3 **AND NEW.**GPA <= 3.6  **BEGIN**  **INSERT INTO** Apply **VALUES**(**NEW.**sID, ‘Stanford’, ‘geology’, NULL);  **INSERT INTO** Apply **VALUES**(**NEW.**sID, ‘MIT’, biology’, NULL);  **END;** |

**Example 2:** (Referential Integrity Constraint)

|  |
| --- |
| **CREATE TRIGGER** R2  **AFTER DELETE ON** Student  **FOR EACH ROW**  **BEGIN**  **DELETE FROM** Apply **WHERE** sID = **Old.**sID  **END;** |

**Example 3:**

|  |  |
| --- | --- |
|  | R3 trigger will be activated for following query |
| **CREATE TRIGGER** R3  **AFTER UPDATE OF** cName **ON** College  **FOR EACH ROW**  **BEGIN**  **UPDATE** Apply  **SET** cName = **NEW.**cName  **WHERE** cName = **Old.**cName  **END;** | **UPDATE** College  **SET** cName = ‘The Farm’  **WHERE** cName = ‘stanford’;  **UPDATE** College **SET** cName = ‘Bezerkeley’  **WHERE** cName = ‘Berkeley’; |

Example of (key constraint):

|  |
| --- |
| **CREATE TRIGGER** R4  **BEFORE INSERT ON** College  **FOR EACH ROW**  **WHEN EXISTS (SELECT** \* **FROM** College **WHERE** cName = **NEW.**cName**)**  **BEGIN**  **SELECT RAISE(IGNORE);**  **END;** |
| **CREATE TRIGGER** R5  **BEFORE UPDATE OF** cName **ON** College  **FOR EACH ROW**  **WHEN EXISTS (SELECT \* FROM** College **WHERE** cName = **NEW.**cName**)**  **BEGIN**  **SELECT RAISE(IGNORE);**  **END;** |

R4 trigger will be activated when we do **INSERT** on College.

R5 trigger will be activated when we do **UPDATE** on College

**UPDATE** College **SET** cName = ‘Berkeley’ **WHERE** cName = ‘Bezerkeley’;

**Constraint**

* To create a primary key constraint on the **ID** column when the table is already created. Use following sql:

**ALTER TABLE** Persons

**ADD PRIMARY KEY(**ID**)**

* To allow naming of **PRIMARY KEY** constraint and for defining a **PRIMARY KEY**(ID,LastName);

**ALTER TABLE** Persons

**ADD CONSTRAINT** PK\_Person **PRIMARY KEY(**ID, LastName**);**

* To drop **PRIMARY KEY,** syntax:

|  |  |
| --- | --- |
| **ALTER TABLE** Persons  **DROP PRIMARY KEY;** | **ALTER TABLE** Persons  **DROP CONSTRIAINT** PK\_Person; |

**FOREIGN KEY** Constraint

It is a key used to link two table together.

Example 1:

|  |
| --- |
| Persons(PersonID **PK,** LastName, FirstName, Age); |

Here PersonID is primary key (PK) in the Persons table.

Example 2:

|  |
| --- |
| Orders(orderID, orderNumber, PersonID); |

Here **PersonID** column in **Orders** table points to the **PersonID** column in the **Persons** table. So **PersonID** column in the **Orders** Table is a **FOREIGN KEY** in the **Orders** table.

|  |
| --- |
| **CREATE TABLE** Orders(  OrderID int **NOT NULL,**  OrderNumber int **NOT NULL,**  PersonID int,  **PRIMARY KEY (**OrderID**),**  **FOREIGN KEY (**PersonID**) REFERENCES** Persons(PersonID); ); |