**.. TRIGGERS..**

What are triggers?

A trigger is an SQL code that are automatically executed in response to certain events on a particular table syntax

Or it can be defined as:

**Trigger:** A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

**Syntax:**

create trigger [trigger\_name]   
[before | after]   
{insert | update | delete}   
on [table\_name]   
[for each row]   
[trigger\_body]

**Explanation of syntax:**

1. create trigger [trigger\_name]: Creates or replaces an existing trigger with the trigger\_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table\_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger\_body]
7. : This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:  
BEFORE triggers run the trigger action before the triggering statement is run.  
AFTER triggers run the trigger action after the triggering statement is run

Example:  
Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

mysql> desc S **inserted** tudent;   
+-------+-------------+------+-----+---------+----------------+   
| Field | Type | Null | Key | Default | Extra |   
+-------+-------------+------+-----+---------+----------------+   
| tid | int(4) | NO | PRI | NULL | auto\_increment |   
| name | varchar(30) | YES | | NULL | |   
| subj1 | int(2) | YES | | NULL | |   
| subj2 | int(2) | YES | | NULL | |   
| subj3 | int(2) | YES | | NULL | |   
| total | int(3) | YES | | NULL | |   
| per | int(3) | YES | | NULL | |  
+-------+-------------+------+-----+---------+----------------+   
7 rows in set (0.00 sec)

SQL Trigger to problem statement.

create trigger stud\_marks   
before INSERT   
on   
Student   
for each row   
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total \* 60 / 100;

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);   
Query OK, 1 row affected (0.09 sec)   
  
mysql> select \* from Student;   
+-----+-------+-------+-------+-------+-------+------+   
| tid | name | subj1 | subj2 | subj3 | total | per |   
+-----+-------+-------+-------+-------+-------+------+   
| 100 | ABCDE | 20 | 20 | 20 | 60 | 36 |   
+-----+-------+-------+-------+-------+-------+------+   
1 row in set (0.00 sec)

In this way trigger can be creates and execute the d in databases.

**. ..INDEXING...**

What is indexing?

What is indexing in a database?

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

## The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

CREATE INDEX index\_name ON table\_name;

### Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

CREATE INDEX index\_name  
ON table\_name (column\_name);

### Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

CREATE UNIQUE INDEX index\_name  
on table\_name (column\_name);

### Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

CREATE INDEX index\_name  
on table\_name (column1, column2);

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

### Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

## The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows −

DROP INDEX index\_name;

### When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

* Indexes should not be used on small tables.
* Tables that have frequent, large batch updates or insert operations.
* Indexes should not be used on columns that contain a high number of NULL values.
* Columns that are frequently manipulated should not be indexed.

**...VIEW...**

**What is view?**

A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database.

Or we can say:

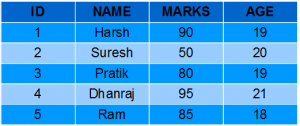
Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

In this article we will learn about creating , deleting and updating Views.  
Sample Tables:

StudentDetails



StudentMarks



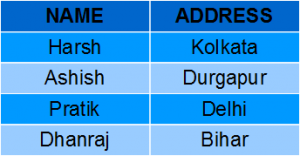
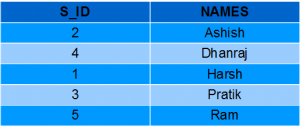
CREATING VIEWS

We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

Syntax:

CREATE VIEW view\_name AS  
SELECT column1, column2.....  
FROM table\_name  
WHERE condition;  
  
view\_name: Name for the View  
table\_name: Name of the table  
condition: Condition to select rows

Examples:

* Creating View from a single table:
  + In this example we will create a View named DetailsView from the table StudentDetails.  
    Query:CREATE VIEW DetailsView AS  
    SELECT NAME, ADDRESS  
    FROM StudentDetails  
    WHERE S\_ID < 5;
  + To see the data in the View, we can query the view in the same manner as we query a table.
  + SELECT \* FROM DetailsView;
  + Output:  
    
  + In this example, we will create a view named StudentNames from the table StudentDetails.  
    Query:
  + CREATE VIEW StudentNames AS  
    SELECT S\_ID, NAME  
    FROM StudentDetails  
    ORDER BY NAME;
  + If we now query the view as,
  + SELECT \* FROM StudentNames;
  + Output:  
    
* Creating View from multiple tables: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:CREATE VIEW MarksView AS  
  SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS  
  FROM StudentDetails, StudentMarks  
  WHERE StudentDetails.NAME = StudentMarks.NAME;
* To display data of View MarksView:
* SELECT \* FROM MarksView;
* Output:  
  

DELETING VIEWS

We have learned about creating a View, but what if a created View is not needed any more? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

DROP VIEW view\_name;  
  
view\_name: Name of the View which we want to delete.

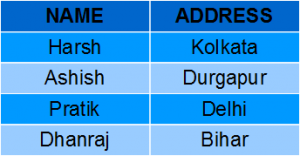
For example, if we want to delete the View MarksView, we can do this as:

DROP VIEW MarksView;

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

* We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.  
  Syntax:CREATE OR REPLACE VIEW view\_name AS  
  SELECT column1,coulmn2,..  
  FROM table\_name  
  WHERE condition;
* For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:
* CREATE OR REPLACE VIEW MarksView AS  
  SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE  
  FROM StudentDetails, StudentMarks  
  WHERE StudentDetails.NAME = StudentMarks.NAME;
* If we fetch all the data from MarksView now as:
* SELECT \* FROM MarksView;
* Output:  
  
* Inserting a row in a view:  
  We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.Syntax:INSERT INTO view\_name(column1, column2 , column3,..)   
  VALUES(value1, value2, value3..);  
    
  view\_name: Name of the View
* Example:  
  In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.
* INSERT INTO DetailsView(NAME, ADDRESS)  
  VALUES("Suresh","Gurgaon");
* If we fetch all the data from DetailsView now as,
* SELECT \* FROM DetailsView;
* Output:  
  
* Deleting a row from a View:  
  Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.Syntax:DELETE FROM view\_name  
  WHERE condition;  
    
  view\_name:Name of view from where we want to delete rows  
  condition: Condition to select rows
* Example:  
  In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.
* DELETE FROM DetailsView  
  WHERE NAME="Suresh";
* If we fetch all the data from DetailsView now as,
* SELECT \* FROM DetailsView;
* Output:  
  

WITH CHECK OPTION

The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.

* The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
* If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

Example:  
In the below example we are creating a View SampleView from StudentDetails Table with WITH CHECK OPTION clause.

CREATE VIEW SampleView AS  
SELECT S\_ID, NAME  
FROM StudentDetails  
WHERE NAME IS NOT NULL  
WITH CHECK OPTION;

In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.  
For example,though the View is updatable but then also the below query for this View is not valid:

INSERT INTO SampleView(S\_ID)  
VALUES(6);

NOTE: The default value of NAME column is *null*.

Uses of a View :  
A good database should contain views due to the given reasons:

1. Restricting data access –  
   Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
2. Hiding data complexity –  
   A view can hide the complexity that exists in a multiple table join.
3. Simplify commands for the user –  
   Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
4. Store complex queries –  
   Views can be used to store complex queries.
5. Rename Columns –  
   Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.
6. Multiple view facility –  
   Different views can be created on the same table for different users.

**...STORED PROCEDURE...**

What is stored procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

### Stored Procedure Syntax

CREATE PROCEDURE *procedure\_name*  
*AS*  
*sql\_statement*  
*GO;*

### Execute a Stored Procedure

EXEC *procedure\_name*;

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

## Stored Procedure Example

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

### Example

CREATE PROCEDURE SelectAllCustomers  
AS  
SELECT \* FROM Customers  
GO;

Execute the stored procedure above as follows:

### Example

EXEC SelectAllCustomers;

## Stored Procedure With One Parameter

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

### Example

CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)  
AS  
SELECT \* FROM Customers WHERE City = @City  
GO;

Execute the stored procedure above as follows:

### Example

EXEC SelectAllCustomers @City = 'London';

## Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

### Example

CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)  
AS  
SELECT \* FROM Customers WHERE City = @City AND PostalCode = @PostalCode  
GO;

Execute the stored procedure above as follows:

### Example

EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';