**Normalization of Database**

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anamolies.(non cosisitance) It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Normalization is used for mainly two purpose,

* Eliminating redundant (useless) data.
* Ensuring data dependencies make sense i.e data is logically stored.

**Problem Without Normalization**

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **S\_Address** | **Subject\_opted** |
| 401 | Adam | Noida | Bio |
| 402 | Alex | Panipat | Maths |
| 403 | Stuart | Jammu | Maths |
| 404 | Adam | Noida | Physics |

* **Updation Anomoly :** To update address of a student who occurs twice or more than twice in a table, we will have to update **S\_Address** column in all the rows, else data will become inconsistent.
* **Insertion Anomaly :** Suppose for a new admission, we have a Student id(S\_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anomaly.
* **Deletion Anomaly :** If (S\_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

**Normalization Rule**

Normalization rule are divided into following normal form.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF

**First Normal Form (1NF)**

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The **Primary key** is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

**Student Table :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology, Maths |
| Alex | 14 | Maths |
| Stuartg | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

**Student Table following 1NF will be :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology |
| Adam | 15 | Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

**Second Normal Form (2NF)**

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {**Student**, **Subject**}, **Age** of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

**New Student Table following 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Age** |
| Adam | 15 |
| Alex | 14 |
| Stuart | 17 |

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

**New Subject Table introduced for 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Subject** |
| Adam | Biology |
| Adam | Maths |
| Alex | Maths |
| Stuart | Maths |

In Subject Table the candidate key will be {**Student**, **Subject**} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

**Third Normal Form (3NF)**

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key. The *transitive functional dependency* should be removed from the table. The table must be in **Second Normal form**. For example, consider a table with following fields.

**Student\_Detail Table :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Street** | **city** | **State** | **Zip** |

In this table Student\_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called **transitive dependency**. Hence to apply **3NF**, we need to move the street, city and state to new table, with **Zip** as primary key.

**New Student\_Detail Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Zip** |

**Address Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Zip** | **Street** | **city** | **state** |

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

**Boyce and Codd Normal Form (BCNF)**

**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF.

**Introduction to SQL**

Structure Query Language(SQL) is a programming language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) uses **SQL** as the standard database language. SQL is used to perform all type of data operations in RDBMS.

**SQL Command**

SQL defines following data languages to manipulate data of RDBMS.

**DDL : Data Definition Language**

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

|  |  |
| --- | --- |
| **Command** | **Description** |
| create | to create new table or database |
| alter | for alteration |
| truncate | delete data from table |
| drop | to drop a table |
| rename | to rename a table |

**DML : Data Manipulation Language**

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

|  |  |
| --- | --- |
| **Command** | **Description** |
| insert | to insert a new row |
| update | to update existing row |
| delete | to delete a row |
| merge | merging two rows or two tables |

**TCL : Transaction Control Language**

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling back to original state. It can also make changes permanent.

|  |  |
| --- | --- |
| **Command** | **Description** |
| commit | to permanently save |
| rollback | to undo change |
| savepoint | to save temporarily |

**DCL : Data Control Language**

Data control language provides command to grant and take back authority.

|  |  |
| --- | --- |
| **Command** | **Description** |
| grant | grant permission of right |
| revoke | take back permission. |

**DQL : Data Query Language**

|  |  |
| --- | --- |
| **Command** | **Description** |
| select | retrieve records from one or more table |

**create command**

**create** is a DDL command used to create a table or a database.

**Creating a Database**

To create a database in RDBMS, *create* command is uses. Following is the Syntax,

**create** database *database-name*;

**Example for Creating Database**

create database Test;

The above command will create a database named **Test**.

**Creating a Table**

*create* command is also used to create a table. We can specify names and datatypes of various columns along.Following is the Syntax,

**create** table *table-name*

{

*column-name1* datatype1,

*column-name2* datatype2,

*column-name3* datatype3,

*column-name4* datatype4

};

create table command will tell the database system to create a new table with given table name and column information.

**Example for creating Table**

create table Student(id int, name varchar, age int);

The above command will create a new table **Student** in database system with 3 columns, namely id, name and age.

**alter command**

*alter* command is used for alteration of table structures. There are various uses of *alter* command, such as,

* to add a column to existing table
* to rename any existing column
* to change datatype of any column or to modify its size.
* *alter* is also used to drop a column.

**To Add Column to existing Table**

Using alter command we can add a column to an existing table. Following is the Syntax,

**alter** table *table-name* add(**column-name** *datatype*);

Here is an Example for this,

alter table Student add(address char);

The above command will add a new column *address* to the **Student** table

**To Add Multiple Column to existing Table**

Using alter command we can even add multiple columns to an existing table. Following is the Syntax,

**alter** table *table-name* add(**column-name1** *datatype1*, **column-name2** *datatype2*, **column-name3** *datatype3*);

Here is an Example for this,

alter table Student add(father-name varchar(60), mother-name varchar(60), dob date);

The above command will add three new columns to the **Student** table

**To Add column with Default Value**

alter command can add a new column to an existing table with default values. Following is the Syntax,

**alter** table *table-name* add(**column-name1** *datatype1* **default** *data*);

Here is an Example for this,

alter table Student add(dob date default '1-Jan-99');

The above command will add a new column with default value to the **Student** table

**To Modify an existing Column**

alter command is used to modify data type of an existing column . Following is the Syntax,

**alter** table *table-name* modify(**column-name** *datatype*);

Here is an Example for this,

alter table Student modify(address varchar(30));

The above command will modify *address* column of the **Student table**

**To Rename a column**

Using alter command you can rename an existing column. Following is the Syntax,

**alter** table *table-name* **rename** old-column-name to column-name;

Here is an Example for this,

alter table Student rename address to Location;

The above command will rename *address* column to *Location*.

**To Drop a Column**

alter command is also used to drop columns also. Following is the Syntax,

**alter** table *table-name* drop(column-name);

Here is an Example for this,

alter table Student drop(address);

The above command will drop *address* column from the **Student table**

**SQL queries to Truncate, Drop or Rename a Table**

**truncate command**

*truncate* command removes all records from a table. But this command will not destroy the table's structure. When we apply truncate command on a table its Primary key is initialized. Following is its Syntax,

**truncate** table *table-name*

Here is an Example explaining it.

truncate table Student;

The above query will delete all the records of **Student** table.

**truncate** command is different from **delete** command. delete command will delete all the rows from a table whereas truncate command re-initializes a table(like a newly created table).

**For eg.** If you have a table with 10 rows and an auto\_increment primary key, if you use *delete* command to delete all the rows, it will delete all the rows, but will not initialize the primary key, hence if you will insert any row after using delete command, the auto\_increment primary key will start from 11. But in case of *truncate* command, primary key is re-initialized.

**drop command**

*drop* query completely removes a table from database. This command will also destroy the table structure. Following is its Syntax,

**drop** table *table-name*

Here is an Example explaining it.

drop table Student;

The above query will delete the **Student** table completely. It can also be used on Databases. For Example, to drop a database,

drop database Test;

The above query will drop a database named **Test** from the system.

**rename query**

*rename* command is used to rename a table. Following is its Syntax,

**rename** table *old-table-name* to *new-table-name*

Here is an Example explaining it.

rename table Student to Student-record;

The above query will rename **Student** table to **Student-record**.

**DML command**

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

**1) INSERT command**

Insert command is used to insert data into a table. Following is its general syntax,

**INSERT** into *table-name* values(data1,data2,..)

Lets see an example,

Consider a table **Student** with following fields.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |

INSERT into Student values(101,'Adam',15);

The above command will insert a record into **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |

**Example to Insert NULL value to a column**

Both the statements below will insert NULL value into **age** column of the Student table.

INSERT into Student(id,name) values(102,'Alex');

Or,

INSERT into Student values(102,'Alex',null);

The above command will insert only two column value other column is set to null.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex |  |

**Example to Insert Default value to a column**

INSERT into Student values(103,'Chris',default)

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

Suppose the **age** column of student table has default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT into Student values(103,'Chris')

**2) UPDATE command**

Update command is used to update a raw of a table. Following is its general syntax,

**UPDATE** *table-name* set column-name = value *where* **condition**;

Lets see an example,

update Student set age=18 where s\_id=102;

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | chris | 14 |

**Example to Update multiple columns**

UPDATE Student set s\_name='Abhi',age=17 where s\_id=103;

The above command will update two columns of a record.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

**3) Delete command**

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row. Following is its general syntax,

**DELETE** from *table-name*;

**Example to Delete all Records from a Table**

DELETE from Student;

The above command will delete all the records from **Student** table.

**Example to Delete a particular Record from a Table**

Consider the following **Student** table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

DELETE from Student where s\_id=103;

The above command will delete the record where s\_id is 103 from **Student** table.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **Age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |

**TCL command**

Transaction Control Language(TCL) commands are used to manage transactions in database. These are used to manage the changes made by DML statements. It also allows statements to be grouped together into logical transactions.

**Commit command**

Commit command is used to permanently save any transaaction into database.

Following is Commit command's syntax,

***commit***;

**Rollback command**

This command restores the database to last commited state. It is also use with savepoint command to jump to a savepoint in a transaction.

Following is Rollback command's syntax,

**rollback** to *savepoint-name*;

**Savepoint command**

**savepoint** command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Following is savepoint command's syntax,

**savepoint** *savepoint-name*;

**Example of Savepoint and Rollback**

Following is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |

Lets use some SQL queries on the above table and see the results.

INSERT into class values(5,'Rahul');

commit;

UPDATE class set name='abhijit' where id='5';

savepoint **A**;

INSERT into class values(6,'Chris');

savepoint **B**;

INSERT into class values(7,'Bravo');

savepoint **C**;

SELECT \* from class;

The resultant table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |
| 7 | Bravo |

Now **rollback** to **savepoint B**

rollback to B;

SELECT \* from class;

The resultant table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |

Now **rollback** to **savepoint A**

rollback to A;

SELECT \* from class;

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |

**DCL command**

Data Control Language(DCL) is used to control privilege in Database. To perform any operation in the database, such as for creating tables, sequences or views we need privileges. Privileges are of two types,

* **System :** creating session, table etc are all types of system privilege.
* **Object :** any command or query to work on tables comes under object privilege.

DCL defines two commands,

* **Grant :** Gives user access privileges to database.
* **Revoke :** Take back permissions from user.

**To Allow a User to create Session**

**grant** create session to *username*;

**To Allow a User to create Table**

**grant** create table to *username*;

**To provide User with some Space on Tablespace to store Table**

**alter** user *username* quota unlimited on system;

**To Grant all privilege to a User**

**grant** sysdba to *username*

**To Grant permission to Create any Table**

**grant** *create* any table to *username*

**To Grant permission to Drop any Table**

**grant** *drop* any table to *username*

**To take back Permissions**

**revoke** create table from *username*

**WHERE clause**

Where clause is used to specify condition while retriving data from table. *Where* clause is used mostly with*Select*, *Update* and *Delete* query. If condititon specified by *where* clause is true then only the result from table is returned.

**Syntax for WHERE clause**

*SELECT* column-name1,

column-name2,

column-name3,

column-nameN

from table-name **WHERE [condition]**;

**Example using WHERE clause**

Consider a **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

Now we will use a SELECT statement to display data of the table, based on a condition, which we will add to the SELECT query using WHERE clause.

SELECT s\_id,

s\_name,

age,

address

from Student **WHERE** s\_id=101;

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

**SELECT Query**

Select query is used to retrieve data from a tables. It is the most used SQL query. We can retrieve complete tables, or partial by mentioning conditions using WHERE clause.

**Syntax of SELECT Query**

**SELECT** column-name1, column-name2, column-name3, column-nameN from *table-name*;

**Example for SELECT Query**

Conside the following **Student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **age** | **address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

SELECT s\_id, s\_name, age from Student.

The above query will fetch information of s\_id, s\_name and age column from Student table

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |
| 104 | Ankit | 22 |

**Example to Select all Records from Table**

A special character **asterisk** \* is used to address **all the data(belonging to all columns)** in a query.*SELECT* statement uses \* character to retrieve all records from a table.

SELECT \* from student;

The above query will show all the records of Student table, that means it will show complete Student table as result.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **age** | **Address** |
| 101 | Adam | 15 | Noida |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Rohtak |
| 104 | Ankit | 22 | Panipat |

**Example to Select particular Record based on Condition**

SELECT \* from Student **WHERE** s\_name = 'Abhi';

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Abhi | 17 | Rohtak |

**Example to Perform Simple Calculations using Select Query**

Conside the following **Employee** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **Name** | **age** | **Salary** |
| 101 | Adam | 26 | 5000 |
| 102 | Ricky | 42 | 8000 |
| 103 | Abhi | 22 | 10000 |
| 104 | Rohan | 35 | 5000 |

SELECT eid, name, salary+3000 from Employee;

The above command will display a new column in the result, showing 3000 added into existing salaries of the employees.

|  |  |  |
| --- | --- | --- |
| **eid** | **Name** | **salary+3000** |
| 101 | Adam | 8000 |
| 102 | Ricky | 11000 |
| 103 | Abhi | 13000 |
| 104 | Rohan | 8000 |

**Like clause**

**Like** clause is used as condition in SQL query. **Like** clause compares data with an expression using wildcard operators. It is used to find similar data from the table.

**Wildcard operators**

There are two wildcard operators that are used in like clause.

* **Percent sign %** : represents zero, one or more than one character.
* **Underscore sign \_** : represents only one character.

**Example of LIKE clause**

Consider the following **Student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

SELECT \* from Student where s\_name like = 'A%';

The above query will return all records where **s\_name** starts with character 'A'.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

**Example**

SELECT \* from Student where s\_name like = '\_d%';

The above query will return all records from **Student** table where **s\_name** contain 'd' as second character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |

**Example**

SELECT \* from Student where s\_name like = '%x';

The above query will return all records from **Student** table where **s\_name** contain 'x' as last character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 102 | Alex | 18 |

**Order By Clause**

Order by clause is used with **Select** statement for arranging retrieved data in sorted order. The **Order by**clause by default sort data in ascending order. To sort data in descending order **DESC** keyword is used with**Order by** clause.

**Syntax of Order By**

*SELECT* column-list|\* from table-name **order by** *asc*|*desc*;

**Example using Order by**

Consider the following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SELECT \* from Emp **order by** salary;

The above query will return result in ascending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 403 | Rohan | 34 | 6000 |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 8000 |
| 401 | Anu | 22 | 9000 |
| 404 | Scott | 44 | 10000 |

**Example of Order by DESC**

Consider the **Emp** table described above,

SELECT \* from Emp order by salary DESC;

The above query will return result in descending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 404 | Scott | 44 | 10000 |
| 401 | Anu | 22 | 9000 |
| 405 | Tiger | 35 | 8000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |

**Group By Clause**

Group by clause is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Syntax for using Group by in a statement.

SELECT column\_name, function(column\_name)

FROM table\_name

WHERE condition

GROUP BY column\_name

**Example of Group by in a Statement**

Consider the following **Emp** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 9000 |
| 405 | Tiger | 35 | 8000 |

Here we want to find name and age of employees grouped by their salaries

SQL query for the above requirement will be,

SELECT name, age

from Emp **group by** salary

Result will be,

|  |  |
| --- | --- |
| **name** | **age** |
| Rohan | 34 |
| shane | 29 |
| anu | 22 |

**Example of Group by in a Statement with WHERE clause**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 9000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

select name, salary

from Emp

where age > 25

**group by** salary

Result will be.

|  |  |
| --- | --- |
| **name** | **salary** |
| Rohan | 6000 |
| Shane | 8000 |
| Scott | 9000 |

You must remember that Group By clause will always come at the end, just like the Order by clause.

**HAVING Clause**

having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in Group based SQL functions, just like WHERE clause.

Syntax for having will be,

select column\_name, function(column\_name)

FROM table\_name

WHERE column\_name condition

GROUP BY column\_name

**HAVING** function(column\_name) condition

**Example of HAVING Statement**

Consider the following **Sale** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

Suppose we want to find the customer whose previous\_balance sum is more than 3000.

We will use the below SQL query,

SELECT \*

from sale group customer

having sum(previous\_balance) > 3000

Result will be,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |

**Distinct keyword**

The **distinct** keyword is used with **Select** statement to retrieve unique values from the table. **Distinct**removes all the duplicate records while retrieving from database.

**Syntax for DISTINCT Keyword**

**SELECT** *distinct* column-name from *table-name*;

**Example**

Consider the following **Emp** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 10000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

select distinct salary from Emp;

The above query will return only the unique salary from **Emp** table

|  |
| --- |
| **salary** |
| 5000 |
| 8000 |
| 10000 |

**AND & OR operator**

**AND** and **OR** operators are used with **Where** clause to make more precise conditions for fetching data from database by combining more than one condition together.

**AND operator**

AND operator is used to set multiple conditions with *Where* clause.

**Example of AND**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary < 10000 **AND** age > 25

The above query will return records where salary is less than 10000 and age greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **Salary** |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 9000 |

**OR operator**

OR operator is also used to combine multiple conditions with *Where* clause. The only difference between AND and OR is their behaviour. When we use AND to combine two or more than two conditions, records satisfying all the condition will be in the result. But in case of OR, atleast one condition from the conditions specified must be satisfied by any record to be in the result.

**Example of OR**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **Salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* from Emp WHERE salary > 10000 **OR** age > 25

The above query will return records where either salary is greater than 10000 or age greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

**SQL Constraints**

SQl Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into following two types,

* **Column level constraints :** limits only column data
* **Table level constraints :** limits whole table data

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

**NOT NULL Constraint**

NOT NULL constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value. One important point to note about NOT NULL constraint is that it cannot be defined at table level.

**Example using NOT NULL constraint**

CREATE table Student(s\_id int NOT NULL, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will not take NULL value.

**UNIQUE Constraint**

UNIQUE constraint ensures that a field or column will only have unique values. A UNIQUE constraint field will not have duplicate data. UNIQUE constraint can be applied at column level or table level.

**Example using UNIQUE constraint when creating a Table (Table Level)**

CREATE table Student(s\_id int NOT NULL UNIQUE, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will only have unique values and wont take NULL value.

**Example using UNIQUE constraint after Table is created (Column Level)**

ALTER table Student add UNIQUE(s\_id);

The above query specifies that **s\_id** field of **Student** table will only have unique value.

**Primary Key Constraint**

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

**Example using PRIMARY KEY constraint at Table Level**

CREATE table Student (s\_id int **PRIMARY KEY**, Name varchar(60) NOT NULL, Age int);

The above command will creates a PRIMARY KEY on the s\_id.

**Example using PRIMARY KEY constraint at Column Level**

ALTER table Student add PRIMARY KEY (s\_id);

The above command will creates a PRIMARY KEY on the s\_id.

**Foreign Key Constraint**

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also **used to restrict actions that would destroy links between tables**. To understand FOREIGN KEY, let's see it using two table.

**Customer\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **c\_id** | **Customer\_Name** | **address** |
| 101 | Adam | Noida |
| 102 | Alex | Delhi |
| 103 | Stuart | Rohtak |

**Order\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **Order\_id** | **Order\_Name** | **c\_id** |
| 10 | Order1 | 101 |
| 11 | Order2 | 103 |
| 12 | Order3 | 102 |

In **Customer\_Detail** table, c\_id is the primary key which is set as foreign key in **Order\_Detail** table. The value that is entered in c\_id which is set as foreign key in **Order\_Detail** table must be present in**Customer\_Detail** table where it is set as primary key. **This prevents invalid data to be inserted into c\_id column of Order\_Detail table.**

**Example using FOREIGN KEY constraint at Table Level**

CREATE table Order\_Detail(order\_id int PRIMARY KEY,

order\_name varchar(60) NOT NULL,

*c\_id int* **FOREIGN KEY** REFERENCES **Customer\_Detail**(*c\_id*));

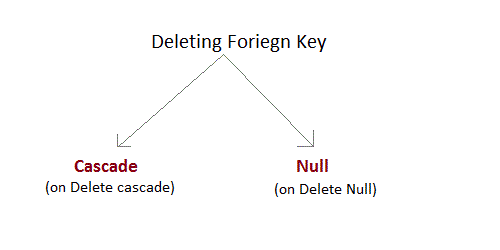
In this query, c\_id in table Order\_Detail is made as foriegn key, which is a reference of c\_id column of Customer\_Detail.

**Example using FOREIGN KEY constraint at Column Level**

ALTER table Order\_Detail add **FOREIGN KEY** (c\_id) REFERENCES Customer\_Detail(c\_id);

**Behaviour of Foriegn Key Column on Delete**

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which record exit in child table too, then we must have some mechanism to save the integrity of data in child table.



* **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
* **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is eleted from the main table.
* If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

ERROR : Record in child table exist

**CHECK Constraint**

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

**Example using CHECK constraint at Table Level**

create table Student(s\_id int NOT NULL **CHECK(s\_id > 0)**,

Name varchar(60) NOT NULL,

Age int);

The above query will restrict the s\_id value to be greater than zero.

**Example using CHECK constraint at Column Level**

ALTER table Student add CHECK(s\_id > 0);

**SQL Functions**

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two catagories,

* Aggregrate Functions
* Scalar Functions

**Aggregrate Functions**

These functions return a single value after calculating from a group of values.Following are some frequently used Aggregrate functions.

**1) AVG()**

Average returns average value after calculating from values in a numeric column.

Its general Syntax is,

SELECT **AVG**(column\_name) from *table\_name*

**Example using AVG()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find average of salary will be,

SELECT **avg(salary)** from Emp;

Result of the above query will be,

|  |
| --- |
| **avg(salary)** |
| 8200 |

**2) COUNT()**

Count returns the number of rows present in the table either based on some condition or without condition.

Its general Syntax is,

SELECT **COUNT**(column\_name) from *table-name*

**Example using COUNT()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to count employees, satisfying specified condition is,

SELECT **COUNT(name)** from Emp where salary = 8000;

Result of the above query will be,

|  |
| --- |
| **count(name)** |
| 2 |

**Example of COUNT(distinct)**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query is,

SELECT COUNT(**distinct salary**) from emp;

Result of the above query will be,

|  |
| --- |
| **count(distinct salary)** |
| 4 |

**3) FIRST()**

First function returns first value of a selected column

Syntax for FIRST function is,

SELECT **FIRST**(column\_name) from *table-name*

**Example of FIRST()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query

SELECT FIRST(salary) from Emp;

Result will be,

|  |
| --- |
| **first(salary)** |
| 9000 |

**4) LAST()**

LAST return the return last value from selected column

Syntax of LAST function is,

SELECT **LAST**(column\_name) from *table-name*

**Example of LAST()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT LAST(salary) from emp;

Result of the above query will be,

|  |
| --- |
| **last(salary)** |
| 8000 |

**5) MAX()**

MAX function returns maximum value from selected column of the table.

Syntax of MAX function is,

SELECT **MAX**(column\_name) from *table-name*

**Example of MAX()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find Maximum salary is,

SELECT MAX(salary) from emp;

Result of the above query will be,

|  |
| --- |
| **MAX(salary)** |
| 10000 |

**6) MIN()**

MIN function returns minimum value from a selected column of the table.

Syntax for MIN function is,

SELECT **MIN**(column\_name) from *table-name*

**Example of MIN()**

Consider following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find minimum salary is,

SELECT MIN(salary) from emp;

Result will be,

|  |
| --- |
| **MIN(salary)** |
| 8000 |

**7) SUM()**

SUM function returns total sum of a selected columns numeric values.

Syntax for SUM is,

SELECT SUM(column\_name) from *table-name*

**Example of SUM()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find sum of salaries will be,

SELECT SUM(salary) from emp;

Result of above query is,

|  |
| --- |
| **SUM(salary)** |
| 41000 |

**Scalar Functions**

Scalar functions return a single value from an input value. Following are soe frequently used Scalar Functions.

**1) UCASE()**

UCASE function is used to convert value of string column to Uppercase character.

Syntax of UCASE,

SELECT **UCASE**(column\_name) from *table-name*

**Example of UCASE()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for using UCASE is,

SELECT UCASE(name) from emp;

Result is,

|  |
| --- |
| **UCASE(name)** |
| ANU |
| SHANE |
| ROHAN |
| SCOTT |
| TIGER |

**2) LCASE()**

LCASE function is used to convert value of string column to Lowecase character.

Syntax for LCASE is,

SELECT **LCASE**(column\_name) from *table-name*

**Example of LCASE()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for converting string value to Lower case is,

SELECT LCASE(name) from emp;

Result will be,

|  |
| --- |
| **LCASE(name)** |
| anu |
| shane |
| rohan |
| scott |
| tiger |

**3) MID()**

MID function is used to extract substrings from column values of string type in a table.

Syntax for MID function is,

SELECT **MID**(column\_name, start, length) from *table-name*

**Example of MID()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

select MID(name,2,2) from emp;

Result will come out to be,

|  |
| --- |
| **MID(name,2,2)** |
| nu |
| ha |
| oh |
| co |
| ig |

**4) ROUND()**

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values. Syntax of Round function is,

SELECT **ROUND**(column\_name, decimals) from *table-name*

**Example of ROUND()**

Consider following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **Salary** |
| 401 | anu | 22 | 9000.67 |
| 402 | shane | 29 | 8000.98 |
| 403 | rohan | 34 | 6000.45 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000.01 |

SQL query is,

SELECT ROUND(salary) from emp;

Result will be,

|  |
| --- |
| **ROUND(salary)** |
| 9001 |
| 8001 |
| 6000 |
| 10000 |
| 8000 |

**Join in SQL**

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables. **Join** Keyword is used in SQL queries for joining two or more tables. **Minimum required condition for joining table, is (n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

**Types of Join**

The following are the types of JOIN that we can use in SQL.

* Inner
* Outer
* Left
* Right

**Cross JOIN or Cartesian Product**

This type of JOIN returns the Cartesian product of rows of from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

SELECT column-name-list

from *table-name1*

**CROSS JOIN**

*table-name2*;

**Example of Cross JOIN**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Cross** JOIN query will be,

SELECT \* from class cross JOIN class\_info;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | Alex | 3 | CHENNAI |

**INNER Join or EQUI Join**

This is a simple JOIN in which the result is **based on matched data as per the equality condition** specified in the query.

Inner Join Syntax is,

SELECT column-name-list

from *table-name1*

**INNER JOIN**

*table-name2*

WHERE table-name1.column-name = table-name2.column-name;

**Example of Inner JOIN**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

SELECT \* from class, class\_info where class.id = class\_info.id;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |

**Natural JOIN**

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

Natural Join Syntax is,

SELECT \*

from *table-name1*

**NATURAL JOIN**

*table-name2*;

**Example of Natural JOIN**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Natural join query will be,**

SELECT \* from class NATURAL JOIN class\_info;

The result table will look like,

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

In the above example, both the tables being joined have ID column(same name and same datatype), hence the records for which value of ID matches in both the tables will be the result of Natural Join of these two tables.

**Outer JOIN**

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

* Left Outer Join
* Right Outer Join
* Full Outer Join

**Left Outer Join**

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left** table and null for the **right** table's column.

Left Outer Join syntax is,

SELECT column-name-list

from *table-name1*

**LEFT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

Left outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*,

*table-name2*

on table-name1.column-name = table-name2.column-name(**+**);

**Example of Left Outer Join**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id=class\_info.id);

The result table will look like,

Select \* from class left outer join class\_info ON(class.id=class\_info.id);

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

**Right Outer Join**

The right outer join returns a result table with the **matched data** of two tables then remaining rows of the **right table** and null for the **left** table's columns.

Right Outer Join Syntax is,

select column-name-list

from *table-name1*

**RIGHT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*,

*table-name2*

on table-name1.column-name(**+**) = table-name2.column-name;

**Example of Right Outer Join**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

**Full Outer Join**

The full outer join returns a result table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Full Outer Join Syntax is,

select column-name-list

from *table-name1*

**FULL OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

**Example of Full outer join is,**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 3 | Alex |
| 4 | Anu |
| 5 | Ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

**SQL Alias**

Alias is used to give an alias name to a table or a column. This is quite useful in case of large or complex queries. Alias is mainly used for giving a short alias name for a column or a table with complex names.

Syntax of Alias for table names,

**SELECT** column-name

from *table-name*

as **alias-name**

Following is an Example using Alias,

SELECT \* from Employee\_detail as **ed**;

Alias syntax for columns will be like,

**SELECT**

*column-name* as **alias-name**

from

*table-name*

Example using alias for columns,

SELECT customer\_id as **cid** from Emp;

**Example of Alias in SQL Query**

Consider the following two tables,

The **class** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

The **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

Below is the Query to fetch data from both the tables using SQL Alias,

SELECT C.id, C.Name, Ci.Address from Class as C, Class\_info as Ci where C.id=Ci.id;

Result table look like,

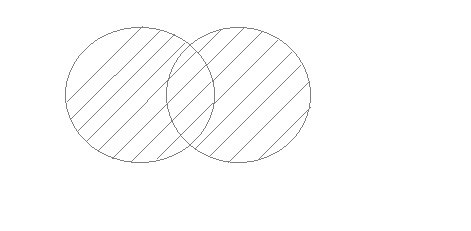
|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

**Set Operation in SQL**

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions.

**Union**

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



**Example of UNION**

The **First** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 2 | adam |
| 3 | Chester |

Union SQL query will be,

select \* from First

**UNION**

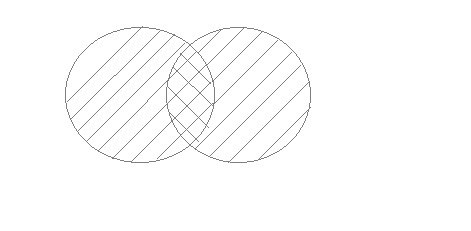
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | Chester |

**Union All**

This operation is similar to Union. But it also shows the duplicate rows.



**Example of Union All**

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |
| 3 | Chester |

Union All query will be like,

select \* from First

**UNION ALL**

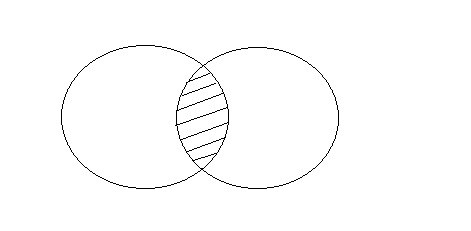
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 2 | adam |
| 3 | Chester |

**Intersect**

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.



**Example of Intersect**

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |
| 3 | Chester |

Intersect query will be,

select \* from First

**INTERSECT**

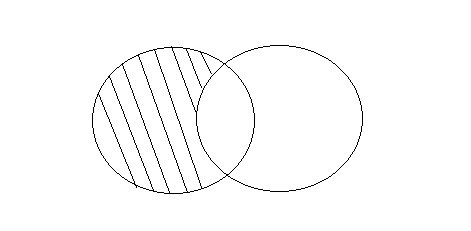
select \* from second

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | adam |

**Minus**

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



**Example of Minus**

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | Adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |
| 3 | Chester |

Minus query will be,

select \* from First

**MINUS**

select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |

## Union Query Overview

The purpose of the SQL **UNION** and **UNION** **ALL** commands are to combine the results of two or more queries into a single result set consisting of all the rows belonging to all the queries in the union. The question becomes whether or not to use the **ALL** syntax.

The main difference between **UNION ALL** and **UNION** is that, **UNION** only selects distinct values, while **UNION ALL** selects all values (including duplicates).

The syntax for **UNION {ALL}** is as follows:

[SQL Statement 1]  
**UNION {ALL}**  
[SQL Statement 2]  
[GROUP BY ...]

### Sample Data

Use Authors table in SQL Server Pubs database or just use a simple table with these values (obviously simplified to just illustrate the point):

|  |  |  |
| --- | --- | --- |
| City | State | Zip |
| Nashville | TN | 37215 |
| Lawrence | KS | 66044 |
| Corvallis | OR | 97330 |

## UNION ALL Example

This SQL statement combines two queries to retrieve records based on states. The two queries happen to both get records from Tennessee ('TN'):

SELECT City, State, Zip FROM Authors WHERE State IN ('KS', 'TN')  
**UNION ALL**  
SELECT City, State, Zip FROM Authors WHERE IN ('OR' 'TN')

### Result of UNION ALL syntax:

|  |  |  |
| --- | --- | --- |
| City | State | Zip |
| Nashville | TN | 37215 |
| Lawrence | KS | 66044 |
| Nashville | TN | 37215 |
| Corvallis | OR | 97330 |

Notice how this displays the two query results in the order they appear from the queries. The first two records come from the first SELECT statement, and the last two records from the second SELECT statement. The TN record appears twice, since both SELECT statements retrieve TN records.

## Union Query SQL Example

Using the same SQL statements and combining them with a UNION command:

SELECT City, State, Zip FROM Authors WHERE State IN ('KS', 'TN')  
**UNION**  
SELECT City, State, Zip FROM Authors WHERE IN ('OR' 'TN')

### Result of UNION Query

|  |  |  |
| --- | --- | --- |
| City | State | Zip |
| Corvallis | OR | 97330 |
| Lawrence | KS | 66044 |
| Nashville | TN | 37215 |

Notice how the TN record only appears once, even though both SELECT statements retrieve TN records. The UNION syntax automatically eliminates the duplicate records between the two SQL statements and sorts the results. In this example the Corvallis record appears first but is from the second SELECT statement.

A GROUP BY clause can be added at the end to sort the list.

## More Advanced Union Queries

The example above is very simple for illustrative purposes, and can obviously be done without using a UNION query. More common uses of UNION queries include:

### More than two SELECT statements

You can add SELECT statements with additional UNION syntax. You are not limited to just two.

### More complex SELECT statements

Each SELECT statement can include multiple tables with different types of joins and filters. Each query could be referencing completely different sets of tables. For instance, you can combine the list of zip codes with customers who purchased a particular product in the past year with zip codes that have total sales exceeding a certain amount.

What's critical is the output field names are identical for each SELECT statement in the UNION query. The fields don't have to come from the same table, or even same field names since you can use the AS syntax to give the field a different name from its source.

### Creating an Artificial Blank Row

You can create a SELECT statement that simulates a blank row, without actually having a blank record in your table. This is commonly used as the rowsource of combo boxes to give people a choice of None (""). For example:

SELECT "" as City, "" as State, "" as Zip FROM Authors  
**UNION**  
SELECT City, State, Zip FROM Authors WHERE IN ('OR' 'TN')

|  |  |  |
| --- | --- | --- |
| City | State | Zip |
|  |  |  |
| Corvallis | OR | 97330 |
| Nashville | TN | 37215 |

# SQL UNION Operator

The SQL UNION operator combines the result of two or more SELECT statements.

## The SQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

### SQL UNION Syntax

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

**Note:** The UNION operator selects only distinct values by default. To allow duplicate values, use the ALL keyword with UNION.

### SQL UNION ALL Syntax

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

**PS:** The column names in the result-set of a UNION are usually equal to the column names in the first SELECT statement in the UNION.

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 |

And a selection from the "Suppliers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SupplierID** | **SupplierName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | Londona | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA |

## SQL UNION Example

The following SQL statement selects all the **different** cities (only distinct values) from the "Customers" and the "Suppliers" tables:

## Example

SELECT City FROM Customers  
UNION  
SELECT City FROM Suppliers  
ORDER BY City;

**Note:** UNION cannot be used to list ALL cities from the two tables. If several customers and suppliers share the same city, each city will only be listed once. UNION selects only distinct values. Use UNION ALL to also select duplicate values!

## SQL UNION ALL Example

The following SQL statement uses UNION ALL to select **all** (duplicate values also) cities from the "Customers" and "Suppliers" tables:

## Example

SELECT City FROM Customers  
UNION ALL  
SELECT City FROM Suppliers  
ORDER BY City;

## SQL UNION ALL With WHERE

The following SQL statement uses UNION ALL to select **all** (duplicate values also) **German** cities from the "Customers" and "Suppliers" tables:

## Example

SELECT City, Country FROM Customers  
WHERE Country='Germany'  
UNION ALL  
SELECT City, Country FROM Suppliers  
WHERE Country='Germany'  
ORDER BY City;

Union : The union operator combines the results of two or  
more queries into a single result set. But no.of columns  
must match in both/all the queries (and also the order)  
which are used for union.  
Union -- returns with no duplicate rows  
Union all -- retruns with duplicate rows (No. of rows  
returned = No. of rows in Query1 + No. of rows in Query 2)  
Union is a set operator.  
You cannot use the union operator within a create view  
statement.  
  
You cannot use the union operator on text and image columns.  
  
You cannot use the for browse clause in statements  
involving the union operator.  
  
Joins: Joins are used to extract information from more than  
one table based on the related column/coloums (PK and RFK)  
any no. of rows can be retrived based on matching matching  
colums. Different types of joins are Inner join=equi join=  
join, outer join(Right outer join/right join, Left outer  
join/ left join), Cross join, and full outer join/outer join

# SQL joins and unions

You can use a SQL statement to create joins and unions between tables in a query in IBM® Cognos® Disclosure Management.

The following table illustrates joins and unions and their descriptions.

| Table 1. Query types for joins and unions | |
| --- | --- |
| **Query type** | **Description** |
| JOIN | A join statement allows you to combine rows from multiple tables. An inner join is the most common, and returns all rows from multiple tables where the join condition is met. An outer join is used to include rows that exist in one table (but not the other). A left outer join preserves rows from the table specified before the join statement, while right outer join preserves rows from the table specified after the join statement. For example, an inner join might look like this:  SELECT \* FROM tableA  JOIN tableB on column\_nameA = column\_nameB |
| UNION | A union statement allows you to combine the result sets of two or more SELECT queries. By default, it removes duplicate rows between the various SELECT statements. Using the ALL qualifier forces duplicate rows to persist in the query result. For example, a union of 2 tables with duplicate rows preserved might look like this:  SELECT column\_nameA  FROM tableA  UNION ALL  SELECT column\_nameB  FROM tableB |

## JOIN:

A Join is used for displaying columns with the same or different names from different tables. The output displayed will have all the columns shown individually. i.e. The columns will be aligned next to each other.

## UNION:

The UNION set operator is used for combining data from two tables which have columns with the same datatype. When a UNION is performed the data from both tables will be collected in a single column having the same datatype.

|  |  |
| --- | --- |
|  | [UNION](http://msdn.microsoft.com/en-us/library/ms180026.aspx) combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union.  By using [JOINs](http://msdn.microsoft.com/en-us/library/ms191517.aspx), you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how SQL should use data from one table to select the rows in another table.  The UNION operation is different from using JOINs that combine columns from two tables.  UNION Example:  SELECT 1 AS [Column1], 2 AS [Column2]  UNION  SELECT 3 AS [Column1], 4 AS [Column2]  Output:  Column1 Column2  -------------------  1 2  3 4  JOIN Example:  SELECT a.Column1, b.Column2 FROM TableA a INNER JOIN TableB b ON a.Id = b.AFKId  This will output all the rows from both the tables for which the condition a.Id = b.AFKId is true. |

**SQL Sequence**

Sequence is a feature supported by some database systems to produce unique values on demand. Some DBMS like **MySQL** supports AUTO\_INCREMENT in place of Sequence. AUTO\_INCREMENT is applied on columns, it automatically increments the column value by 1 each time a new record is entered into the table. Sequence is also some what similar to AUTO\_INCREMENT but its has some extra features.

**Creating Sequence**

Syntax to create sequences is,

CREATE **Sequence** *sequence-name*

**start** with *initial-value*

**increment** by *increment-value*

**maxvalue** *maximum-value*

cycle|nocycle

**initial-value** specifies the starting value of the Sequence, **increment-value** is the value by which sequence will be incremented and **maxvalue** specifies the maximum value until which sequence will increment itself. **cycle** specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the begining. **No cycle**specifies that if sequence exceeds **maxvalue** an error will be thrown.

**Example to create Sequence**

The sequence query is following

CREATE **Sequence** seq\_1

start with 1

increment by 1

maxvalue 999

cycle ;

**Example to use Sequence**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

The sql query will be,

INSERT into class value(**seq\_1.nextval**,'anu');

Result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |
| 1 | anu |

Once you use nextval the sequence will increment even if you don't Insert any record into the table.

**SELECT** max(salary) **FROM** Employee **WHERE** salary **NOT** **IN** (**SELECT** max(salary) **FROM** Employee);  
**SELECT** max(salary) **FROM** Employee **WHERE** salary < (**SELECT** max(salary) **FROM** Employee);

**SQL View**

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access.

Syntax for creating a View,

CREATE or REPLACE **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition*

**Example of Creating a View**

Consider following **Sale** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

SQL Query to Create View

CREATE or REPLACE **view** sale\_view as select \* from Sale where customer = 'Alex';

The data fetched from select statement will be stored in another object called **sale\_view**. We can use create seperately and replace too but using both together works better.

**Example of Displaying a View**

Syntax of displaying a view is similar to fetching data from table using Select statement.

SELECT \* from **sale\_view**;

**Force View Creation**

force keyword is used while creating a view. This keyword force to create View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

CREATE or REPLACE *force* **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition*

**Update a View**

Update command for view is same as for tables.

Syntax to Update a View is,

UPDATE **view-name**

set value

WHERE condition;

If we update a view it also updates base table data automatically.

**Read-Only View**

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

CREATE or REPLACE *force* **view** *view\_name* AS

SELECT *column\_name*(s)

FROM *table\_name*

WHERE *condition* with **read-only**

The above syntax will create view for read-only purpose, we cannot Update or Insert data into read-only view. It will throw an error.

**Types of View**

There are two types of view,

* Simple View
* Complex View

|  |  |
| --- | --- |
| **Simple View** | **Complex View** |
| Created from one table | Created from one or more table |
| Does not contain functions | Contain functions |
| Does not contain groups of data | Contains groups of data |

# Sql Server - How To Write a Stored Procedure in SQL Server

Stored Procedure in SQL Server can be defined as the set of logically group of SQL statement which are grouped to perform a specific task. This article will describe the Stored Procedure in SQL Server and their benefits.

**Stored Procedure**: Stored Procedure in SQL Server can be defined as the set of logical group of SQL statements which are grouped to perform a specific task. There are many benefits of using a stored procedure. The main benefit

of using a stored procedure is that it increases the performance of the database.The other benefits of using the

Stored Procedure are given below.

## Benefits of Using the Stored Procedure

1. One of the main benefits of using the Stored procedure is that it reduces the amount of information sent to the database server. It can become a more important benefit when the bandwidth of the network is less. Since if we send the SQL query (statement) which is executing in a loop to the server through network and the network gets disconnected, then the execution of the SQL statement doesn't return the expected results, if the SQL query is not used between Transaction statement and rollback statement is not used.
2. Compilation step is required only once when the stored procedure is created. Then after it does not require recompilation before executing unless it is modified and reutilizes the same execution plan whereas the SQL statements need to be compiled every time whenever it is sent for execution even if we send the same SQL statement every time.
3. It helps in re usability of the SQL code because it can be used by multiple users and by multiple clients since we need to just call the stored procedure instead of writing the same SQL statement every time. It helps in reducing the development time.
4. Stored procedure is helpful in enhancing the security since we can grant permission to the user for executing the Stored procedure instead of giving permission on the tables used in the Stored procedure.
5. Sometimes, it is useful to use the database for storing the business logic in the form of stored procedure since it makes it secure and if any change is needed in the business logic, then we may only need to make changes in the stored procedure and not in the files contained on the web server.

## How to Write a Stored Procedure in SQL Server

Suppose there is a table called tbl\_Students whose structure is given below:

CREATE TABLE tbl\_Students

(

[Studentid] [int] IDENTITY(1,1) NOT NULL,

[Firstname] [nvarchar](200) NOT NULL,

[Lastname] [nvarchar](200) NULL,

[Email] [nvarchar](100) NULL

)

Support we insert the following data into the above table:

Insert into tbl\_Students (Firstname, lastname, Email)

Values('Vivek', 'Johari', 'vivek@abc.com')

Insert into tbl\_Students (Firstname, lastname, Email)

Values('Pankaj', 'Kumar', 'pankaj@abc.com')

Insert into tbl\_Students (Firstname, lastname, Email)

Values('Amit', 'Singh', 'amit@abc.com')

Insert into tbl\_Students (Firstname, lastname, Email)

Values('Manish', 'Kumar', 'manish@abc.comm')

Insert into tbl\_Students (Firstname, lastname, Email)

Values('Abhishek', 'Singh', 'abhishek@abc.com')

Now, while writing a Stored Procedure, the first step will be to write the Create Procedure statement as the first statement:

Create Procedure Procedure-name

(

Input parameters ,

Output Parameters (If required)

)

As

Begin

Sql statement used in the stored procedure

End

Now, suppose we need to create a Stored Procedure which will return a student name whose studentid is given as the input parameter to the stored procedure. Then, the Stored Procedure will be:

*/\* Getstudentname is the name of the stored procedure\*/*

Create PROCEDURE Getstudentname(

@studentid INT *--Input parameter , Studentid of the student*

)

AS

BEGIN

SELECT Firstname+' '+Lastname FROM tbl\_Students WHERE studentid=@studentid

END

We can also collect the student name in the output parameter of the Stored Procedure. For example:

*/\**

*GetstudentnameInOutputVariable is the name of the stored procedure which*

*uses output variable @Studentname to collect the student name returns by the*

*stored procedure*

*\*/*

Create PROCEDURE GetstudentnameInOutputVariable

(

@studentid INT, *--Input parameter , Studentid of the student*

@studentname VARCHAR(200) OUT *-- Out parameter declared with the help of OUT keyword*

)

AS

BEGIN

SELECT @studentname= Firstname+' '+Lastname FROM tbl\_Students WHERE studentid=@studentid

END

**Note:-/\* \*/** is used to write comments in one or multiple lines

**--** is used to write a comment in a single line

## How to Alter a Stored Procedure in a SQL Server

In SQL Server, a stored procedure can be modified with the help of the Alter **keyword**. Now if we want to get student email address through the same procedure GetstudentnameInOutputVariable. So we

need to modify it by adding one more output parameter "@StudentEmail " which is shown below:

*/\**

*Stored Procedure GetstudentnameInOutputVariable is modified to collect the*

*email address of the student with the help of the Alert Keyword*

*\*/*

Alter PROCEDURE GetstudentnameInOutputVariable

(

@studentid INT, *--Input parameter , Studentid of the student*

@studentname VARCHAR (200) OUT, *-- Output parameter to collect the student name*

@StudentEmail VARCHAR (200)OUT *-- Output Parameter to collect the student email*

)

AS

BEGIN

SELECT @studentname= Firstname+' '+Lastname,

@StudentEmail=email FROM tbl\_Students WHERE studentid=@studentid

END

**Note**: It is not necessary that a stored procedure will have to return. There can be a case when a stored procedure doesn't returns anything. For example, a stored procedure can be used to Insert, delete or update a SQL statement. For example, the below stored procedure is used to insert value into the table tbl\_students.

*/\**

*This Stored procedure is used to Insert value into the table tbl\_students.*

*\*/*

Create Procedure InsertStudentrecord

(

@StudentFirstName Varchar(200),

@StudentLastName Varchar(200),

@StudentEmail Varchar(50)

)

As

Begin

Insert into tbl\_Students (Firstname, lastname, Email)

Values(@StudentFirstName, @StudentLastName,@StudentEmail)

End

## Execution of the Stored Procedure in SQL Server

### Execution of the Stored Procedure which doesn't have an Output Parameter

A stored procedure is used in the SQL Server with the help of the "Execute" or "Exec" Keyword. For example, if we want to execute the stored procedure "Getstudentname", then we will use the following statement.

Execute Getstudentname 1

Exec Getstudentname 1

### Execution of the Stored Procedure using the Output Parameter

If we want to execute the Stored procedure "GetstudentnameInOutputVariable" , then we first need to declare the variable to collect the output values. For example:

Declare @Studentname as nvarchar(200) *-- Declaring the variable to collect the Studentname*

Declare @Studentemail as nvarchar(50) *-- Declaring the variable to collect the Studentemail*

Execute GetstudentnameInOutputVariable 1 , @Studentname output, @Studentemail output

select @Studentname,@Studentemail *-- "Select" Statement is used to show the output from Procedure*

## Summary

In the end, we can say that a Stored procedure not only enhances the possibility of reusing the code and execution plan, but it also increases the performance of the database by reducing the traffic of the network by reducing the amount of information sent over the network.

# Triggers -- SQL Server

This article gives a brief introduction about Triggers in Sql Server 2000/2005.

## Introduction

Triggers in SQL Server

## Background

This article gives a brief introduction about Triggers in SQL Server 2000/2005.

## What is a Trigger

A trigger is a special kind of a store procedure that executes in response to certain action on the table 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 triggers. The only way to do this is by performing the required action to the table that they are assigned to.

## Types Of Triggers

There are three action query types that you use in SQL which are INSERT, UPDATE and DELETE. So, there are three types of triggers and hybrids that come from mixing and matching the events and timings that fire them. Basically, triggers are classified into two main types:

1. After Triggers (For Triggers)
2. Instead Of Triggers

## (i) After Triggers

These triggers run after an insert, update or delete on a table. They are **not supported for views.**   
AFTER TRIGGERS can be classified further into three types as:

1. AFTER INSERT Trigger.
2. AFTER UPDATE Trigger.
3. AFTER DELETE Trigger.

Let’s create After triggers. First of all, let’s create a table and insert some sample data. Then, on this table, I will be attaching several triggers.

CREATE TABLE Employee\_Test

(

Emp\_ID INT Identity,

Emp\_name Varchar(100),

Emp\_Sal Decimal (10,2)

)

INSERT INTO Employee\_Test VALUES ('Anees',1000);

INSERT INTO Employee\_Test VALUES ('Rick',1200);

INSERT INTO Employee\_Test VALUES ('John',1100);

INSERT INTO Employee\_Test VALUES ('Stephen',1300);

INSERT INTO Employee\_Test VALUES ('Maria',1400);

I will be creating an AFTER INSERT TRIGGER which will insert the rows inserted into the table into another audit table. The main purpose of this audit table is to record the changes in the main table. This can be thought of as a generic audit trigger.

Now, create the audit table as:-

CREATE TABLE Employee\_Test\_Audit

(

Emp\_ID int,

Emp\_name varchar(100),

Emp\_Sal decimal (10,2),

Audit\_Action varchar(100),

Audit\_Timestamp datetime

)

## (a) After Insert Trigger

This trigger is fired after an INSERT on the table. Let’s create the trigger as:

CREATE TRIGGER trgAfterInsert ON [dbo].[Employee\_Test]

FOR INSERT

AS

declare @empid int;

declare @empname varchar(100);

declare @empsal decimal(10,2);

declare @audit\_action varchar(100);

select @empid=i.Emp\_ID from inserted i;

select @empname=i.Emp\_Name from inserted i;

select @empsal=i.Emp\_Sal from inserted i;

set @audit\_action='Inserted Record -- After Insert Trigger.';

insert into Employee\_Test\_Audit

(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@empid,@empname,@empsal,@audit\_action,getdate());

PRINT 'AFTER INSERT trigger fired.'

GO

The CREATE TRIGGER statement is used to create the trigger. THE ON clause specifies the table name on which

the trigger is to be attached. The FOR INSERT specifies that this is an AFTER INSERT trigger. In place of FOR INSERT, AFTER INSERT can be used. Both of them mean the same.

In the trigger body, table named **inserted**has been used. This table is a logical table and contains the row that has been inserted. I have selected the fields from the logical inserted table from the row that has been inserted into different variables, and finally inserted those values into the Audit table.

To see the newly created trigger in action, lets insert a row into the main table as:

insert into Employee\_Test values('Chris',1500);

Now, a record has been inserted into the Employee\_Test table. The AFTER INSERT trigger attached to this table

has inserted the record into the Employee\_Test\_Audit as:

6 Chris 1500.00 Inserted Record -- After Insert Trigger. 2008-04-26 12:00:55.700

## (b) AFTER UPDATE Trigger

This trigger is fired after an update on the table. Let’s create the trigger as:

CREATE TRIGGER trgAfterUpdate ON [dbo].[Employee\_Test]

FOR UPDATE

AS

declare @empid int;

declare @empname varchar(100);

declare @empsal decimal(10,2);

declare @audit\_action varchar(100);

select @empid=i.Emp\_ID from inserted i;

select @empname=i.Emp\_Name from inserted i;

select @empsal=i.Emp\_Sal from inserted i;

if update(Emp\_Name)

set @audit\_action='Updated Record -- After Update Trigger.';

if update(Emp\_Sal)

set @audit\_action='Updated Record -- After Update Trigger.';

insert into Employee\_Test\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@empid,@empname,@empsal,@audit\_action,getdate());

PRINT 'AFTER UPDATE Trigger fired.'

GO

The AFTER UPDATE Trigger is created in which the updated record is inserted into the audit table. There is **no logical table updated like the logical table inserted.** We can obtain the updated value of a field from

the update(column\_name) function. In our trigger, we have used, if update(Emp\_Name) to check if the

column Emp\_Name has been updated. We have similarly checked the column Emp\_Sal for an update.

Let’s update a record column and see what happens.

update Employee\_Test set Emp\_Sal=1550 where Emp\_ID=6

This inserts the row into the audit table as:

6 Chris 1550.00 Updated Record -- After Update Trigger. 2008-04-26 12:38:11.843

## (c) AFTER DELETE Trigger

This trigger is fired after a delete on the table. Let’s create the trigger as:

CREATE TRIGGER trgAfterDelete ON [dbo].[Employee\_Test]

AFTER DELETE

AS

declare @empid int;

declare @empname varchar(100);

declare @empsal decimal(10,2);

declare @audit\_action varchar(100);

select @empid=d.Emp\_ID from deleted d;

select @empname=d.Emp\_Name from deleted d;

select @empsal=d.Emp\_Sal from deleted d;

set @audit\_action='Deleted -- After Delete Trigger.';

insert into Employee\_Test\_Audit

(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@empid,@empname,@empsal,@audit\_action,getdate());

PRINT 'AFTER DELETE TRIGGER fired.'

GO

In this trigger, the deleted record’s data is picked from the **logical deleted table** and inserted into the audit table. Let’s fire a delete on the main table. A record has been inserted into the audit table as:

6 Chris 1550.00 Deleted -- After Delete Trigger. 2008-04-26 12:52:13.867

All the triggers can be enabled/disabled on the table using the statement

ALTER TABLE Employee\_Test {ENABLE|DISBALE} TRIGGER ALL

Specific Triggers can be enabled or disabled as:

ALTER TABLE Employee\_Test DISABLE TRIGGER trgAfterDelete

This disables the After Delete Trigger named trgAfterDelete on the specified table.

## (ii) Instead Of Triggers

These can be used as an interceptor for anything that anyone tried to do on our table or view. If you define an Instead

Of trigger on a table for the Delete operation, they try to delete rows, and they will not actually get deleted (unless you issue another delete instruction from within the trigger)

INSTEAD OF TRIGGERS can be classified further into three types as:

1. INSTEAD OF INSERT Trigger.
2. INSTEAD OF UPDATE Trigger.
3. INSTEAD OF DELETE Trigger.

Let’s create an Instead Of Delete Trigger as:

CREATE TRIGGER trgInsteadOfDelete ON [dbo].[Employee\_Test]

INSTEAD OF DELETE

AS

declare @emp\_id int;

declare @emp\_name varchar(100);

declare @emp\_sal int;

select @emp\_id=d.Emp\_ID from deleted d;

select @emp\_name=d.Emp\_Name from deleted d;

select @emp\_sal=d.Emp\_Sal from deleted d;

BEGIN

if(@emp\_sal>1200)

begin

RAISERROR('Cannot delete where salary > 1200',16,1);

ROLLBACK;

end

else

begin

delete from Employee\_Test where Emp\_ID=@emp\_id;

COMMIT;

insert into Employee\_Test\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@emp\_id,@emp\_name,@emp\_sal,'Deleted -- Instead Of Delete Trigger.',getdate());

PRINT 'Record Deleted -- Instead Of Delete Trigger.'

end

END

GO

This trigger will prevent the deletion of records from the table where Emp\_Sal > 1200. If such a record is deleted, the Instead Of Trigger will rollback the transaction, otherwise the transaction will be committed. Now, let’s try to delete a record with the Emp\_Sal >1200 as:

delete from Employee\_Test where Emp\_ID=4

This will print an error message as defined in the RAISE ERROR statement as:

Server: Msg 50000, Level 16, State 1, Procedure trgInsteadOfDelete, Line 15

Cannot delete where salary > 1200

And this record will not be deleted.

In a similar way, you can code Instead of Insert and Instead Of Update triggers on your tables.

## Conclusion

In this article, I took a brief introduction of triggers, explained the various kinds of triggers – After Triggers and Instead Of Triggers along with their variants and explained how each of them works. I hope you will get a clear understanding about the Triggers in SQL Server and their usage.

**Database testing scenario:**

**Testing trigger:**

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

1. After triggers

-After Insert

-After Update

-After Delete

2. Instead of triggers

- INSTEAD OF INSERT

- INSTEAD OF UPDATE

- INSTEAD OF DELETE

1. Test size of database when migrate

1. How to connect to the database?   
2. Ability to write simple queries to retrieve data and manipulate the data using DML operations.  
3. Functional flow should be very well known!   
4. Good knowledge on table level, column level constraints, ability to understand and execute complex queries related to joins is added advantage.

Three types of database testing:

-Structural testing

-Functional testing

-Non Functional

<http://www.softwaretestinghelp.com/database-testing-practical-tips-and-insight-on-how-to-test-database/>

<http://inderpsingh.blogspot.in/2010/03/how-to-do-real-database-testing-10-tips.html>

**Questions related to data**  
1. Is the data complete?  
2. Is all data factually correct i.e. in sync with its source, for example the data entered by a user via the application UI?  
3. Is there any unnecessary data present?

1. List all database-specific requirements. You should gather the requirements from all sources, particularly technical requirements. It is quite possible that some requirements are at a high level. Break-down those requirements into the small testable requirements.  
  
2. Create test scenarios for each requirement as suggested below.  
  
3. In order to check the logical database design, ensure that each entity in the application e.g. actors, system configuration are represented in the database. An application entity may be represented in one or tables in the database. The database should contain only those tables that are required to represent the application entities and no more.  
  
4. In order to check the database performance, you may focus on its throughput and response times. For example, if the database is supposed to insert 1000 customer records per minute, you may design a query that inserts 1000 customer records and print/ store the time taken to do so. If the database is supposed to execute a stored procedure in under 5 seconds, you may design a query to execute the stored procedure with sample test data multiple times and note each time.  
  
5. If you wish to test the database objects e.g. stored procedures, you should remember that a stored procedure may be thought of as a simple program that (optionally) accepts certain input(s) and produces some output. You should design test data to exercise the stored procedure in interesting ways and predict the output of the stored procedure for every test data set.  
  
6. In order to check database constraints, you should design invalid test data sets and then try to insert/ update them in the database. An example of an invalid data set is an order for a customer that does not exist. Another example is a customer test data set with an invalid ZIP code.  
  
7. In order to check the database security, you should design tests that mimic unauthorized access. For example, log in to the database as a user with restricted access and check if you can view/ modify/ delete restricted database objects or view or view and update restricted data. It is important to backup your database before executing any database security tests. Otherwise, you may render your database unusable.  
You should also check to see that any confidential data in the database e.g. credit card numbers is either encrypted or obfuscated (masked).  
  
8. In order to test data integrity, you should design valid test data sets for each application entity. Insert/ update a valid test data set (for example, a customer) and check that the data has been stored in the correct table(s) in correct columns. Each data in the test data set should have been inserted/ updated in the database. Further, the test data set should be inserted only once and there should not be any other change in the other data.  
  
9. Since your test design would require creating SQL queries, try to keep your queries as simple as possible to prevent defects in them. It is a good idea for someone other than the author to review the queries. You should also dynamically test each query. One way to test your query is to modify it so that it just shows the resultset and does not perform the actual operation e.g. insert, delete. Another way to test your query is to run it for a couple of iteration s and verify the results.  
  
10. If you are going to have a large number of tests, you should pay special attention to organizing them. You should also consider at least partial automation of frequently run tests.  
  
Now you should know what database testing is all about, the problems that you are likely to face while doing database testing and how to design a good database test approach for the scope decided by you.

Top command:

Select top 3 \* from student;

Select \* from student limit 3;

Show second highest salary

Delete the duplicate row from table

Joins types with all query

Select distinct salary from emp order by salary DESC LIMIT 1,1;

Select distinct salary from emp order by slary desc LIMIT 1,1;

select distinct id, age from emp order by age desc limit 1,3;

1 Deepak

1 Deepak

2 Ramesh

Select \* from emp group by id, salary