**Introduction to SQL**

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

SQL is used to perform all types of *data operations in RDBMS*.

**SQL Command -** SQL defines following ways to manipulate data stored in an RDBMS.

**1) DDL: Data Definition Language**

This includes changes to the structure of the table like *creation of table, altering table, deleting a table* etc.

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 but not structure of table |
| drop | to drop a table |
| rename | to rename a table |

**2) DML: Data Manipulation Language**

DML commands are used for manipulating the data stored in the table and not the table itself.

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 |

**3) 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 the data back to its original state. It can also make any temporary change permanent.

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

**4) DCL: Data Control Language**

Data control language are the commands to grant and take back authority from any database user.

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

**5) DQL: Data Query Language**

Data query language is used to fetch data from tables based on conditions that we can easily apply.

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

**SQL Commands**

**SQL: create command**

create is a DDL SQL command used to create a table or a database in relational database management system.

1. **Creating a Database**

To create a database in RDBMS, create command is used.

Syntax:

CREATE DATABASE <DB\_NAME>;

Example:

CREATE DATABASE Test;

1. **Creating a Table**

Syntax:

CREATE TABLE <TABLE\_NAME>

(

column\_name1 datatype1,

column\_name2 datatype2,

column\_name3 datatype3,

column\_name4 datatype4

);

Example:

CREATE TABLE Student(

student\_id INT,

name VARCHAR(100),

age INT

);

Note : if we have a database with name Test and we want to create a table Student in it, then we can do so using the following query:

Query:- Create a table (Test) in the database (student)

Solution:

CREATE TABLE Test.Student(

Student\_id INT,

name VARCHAR(100),

age INT

);

Most commonly used datatypes for Table columns

|  |  |
| --- | --- |
| Datatype | Use |
| INT | used for columns which will store integer values. |
| FLOAT | used for columns which will store float values. |
| DOUBLE | used for columns which will store float values. |
| VARCHAR | used for columns which will be used to store characters and integers, basically a string. |
| CHAR | used for columns which will store char values(single character). |
| DATE | used for columns which will store date values. |
| TEXT | used for columns which will store text which is generally long in length. For example, if you create a table for storing profile information of a social networking website, then for about me section you can have a column of type TEXT. |

**SQL: ALTER command**

alter command is used for altering the table structure, such as,

1. to add a column to existing table
2. to rename any existing column
3. to change datatype of any column or to modify its size.
4. to drop a column from the table.
5. **ALTER Command: Add a new Column**

Syntax:

ALTER TABLE table\_name ADD COLUMN ( column\_name datatype);

Example:

ALTER TABLE student ADD COLUMN ( address VARCHAR(200) );

1. **ALTER Command: Add multiple new Columns**

Syntax:

ALTER TABLE table\_name ADD COLUMN (

column\_name1 datatype1,

column-name2 datatype2,

column-name3 datatype3

);

Example:

ALTER TABLE student ADD COLUMN (

father\_name VARCHAR(60),

mother\_name VARCHAR(60),

dob DATE

);

1. **ALTER Command: Add Column with default value**

Syntax,

ALTER TABLE table\_name ADD COLUMN (

column-name1 datatype1 DEFAULT some\_value

);

Example:

ALTER TABLE student ADD COLUMN (

dob DATE DEFAULT '01-Jan-99'

);

1. **ALTER Command: Modify an existing Column**

Syntax,

ALTER TABLE table\_name modify COLUMN (

column\_name datatype

);

Example:

ALTER TABLE student MODIFY COLUMN (

address varchar(300)

);

1. **ALTER Command: Rename a Column**

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

Syntax:

ALTER TABLE table\_name RENAME COLUMN

old\_column\_name TO new\_column\_name;

Example:

ALTER TABLE student RENAME COLUMN

address TO location;

The above command will rename address column to location.

1. **ALTER Command: Drop a Column**

*ALTER command can also be used to drop or remove columns.* Following is the Syntax,

Syntax:

ALTER TABLE table\_name DROP COLUMN column\_name;

Here is an example for this,

ALTER TABLE student DROP COLUMN address;

The above command will drop the address column from the table **student**.

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

In this tutorial we will learn *about the various DDL commands* which are used to re-define the tables.

1. **TRUNCATE command**

*TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure.* When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized.

Syntax:

TRUNCATE TABLE table\_name

Example:

TRUNCATE TABLE student;

The above query will delete all the records from the table **student**.

*In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command. We will also learn about the difference between the two in that tutorial.*

1. **DROP command**

*DROP command completely removes a table from the database.*

This command will also **destroy the table structure and the data stored in it.**

Syntax:

DROP TABLE table\_name

Example:

DROP TABLE student;

The above query will delete the **Student** table completely.

It can also be used on Databases, to delete the complete database.

For example, to drop a database,

DROP DATABASE Test;

The above query will drop the database with name **Test** from the system.

1. **RENAME query**

*RENAME command is used to set a new name for any* ***existing table.***

Syntax,

RENAME TABLE old\_table\_name to new\_table\_name

Example:

RENAME TABLE student to students\_info;

The above query will rename the table **student** to **students\_info**.

**Using INSERT SQL 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.

Talking about the Insert command, whenever we post a Tweet on Twitter, the text is stored in some table, and as we post a new tweet, a new record gets inserted in that table.

**INSERT command**

Insert command is used to insert data into a table.

Syntax,

INSERT INTO table\_name VALUES(data1, data2, ...)

Lets see an example,

Consider a table **student** with the following fields.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |

INSERT INTO student VALUES(101, 'Adam', 15);

The above command will insert a new record into **student** table.

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

***Insert value into only specific columns***

We can use the INSERT command to insert values for only some specific columns of a row. We can specify the column names along with the values to be inserted like this,

INSERT INTO student(id, name) values(102, 'Alex');

The above SQL query will only insert id and name values in the newly inserted record.

**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 values and the other column is set to null.

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

**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 column age in our tabel has a 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')

**Using UPDATE SQL command**

Let's take an example of a real-world problem. These days, Facebook provides an option for **Editing** your status update, how do you think it works? Yes, using the **Update** SQL command.

Let's learn about the Syntax and usage of the UPDATE command.

**UPDATE command**

UPDATE command is used to update any record of data in a table. Following is its general Syntax,

UPDATE table\_name SET column\_name = new\_value WHERE some\_condition;

WHERE is used to add a condition to any SQL query, we will soon study about it in detail.

Lets take a sample table **student**,

|  |  |  |
| --- | --- | --- |
| **student\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

UPDATE student SET age=18 WHERE student\_id=102;

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

In the above statement, if we do not use the WHERE clause, then our update query will update age for all the columns of the table to **18**.

**Updating Multiple Columns**

We can also update values of multiple columns using a single UPDATE statement.

UPDATE student SET name='Abhi', age=17 where s\_id=103;

The above command will update two columns of the record which has s\_id 103.

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

**UPDATE Command: Incrementing Integer Value**

When we have to update any integer value in a table, then we can fetch and update the value in the table in a single statement.

For example, if we have to update the age column of **student** table every year for every student, then we can simply run the following UPDATE statement to perform the following operation:

UPDATE student SET age = age+1;

As you can see, we have used age = age + 1 to increment the value of age by 1.

**NOTE:** This style only works for integer values.

**Using DELETE SQL command**

When you ask any question in [Studytonight's Forum](https://www.studytonight.com/studyroom/) it gets saved into a table. And using the **Delete** option, you can even delete a question asked by you. How do you think that works? Yes, using the Delete DML command.

Let's study about the Syntax and the usage of the Delete command.

**DELETE command**

DELETE command is used to delete data from a table.

Following is its general Syntax,

DELETE FROM table\_name;

Let's take a sample table **student**:

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

**Delete all Records from a Table**

DELETE FROM student;

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

**Delete a particular Record from a Table**

In our **student** table if we want to delete a single record, we can use the WHERE clause to provide a condition in our DELETE statement.

DELETE FROM student WHERE s\_id=103;

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

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

**Isn't DELETE same as TRUNCATE**

TRUNCATE command is different from DELETE command. The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).

**For eg:** If you have a table with 10 rows and an **auto\_increment** primary key, and if you use DELETE command to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the DELETE command, the auto\_increment primary key will start from 11. But in case of TRUNCATE command, primary key is re-initialized, and it will again start from 1.

**Commit, Rollback and Savepoint SQL commands**

Transaction Control Language(TCL) commands are used to manage transactions in the [database](https://www.studytonight.com/dbms/overview-of-dbms.php).

Before moving forward with TCL commands, check these topics out first:

[DML commands](https://www.studytonight.com/dbms/dml-command.php)

[DDL COMMAND](https://www.studytonight.com/dbms/create-query.php)

These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

**COMMIT command**

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's Syntax,

COMMIT;

**ROLLBACK command**

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

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 required.

Following is savepoint command's Syntax,

SAVEPOINT savepoint\_name;

In short, using this command we can **name** the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

**Using Savepoint and Rollback**

Following is the table **class**,

|  |  |
| --- | --- |
| **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;

**NOTE:** SELECT statement is used to show the data stored in the table.

The resultant table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |
| 7 | Bravo |

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

ROLLBACK TO B;

SELECT \* FROM class;

Now our **class** table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

ROLLBACK TO A;

SELECT \* FROM class;

Now the table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |

So now you know how the commands COMMIT, ROLLBACK and SAVEPOINT works.

**Using GRANT and REVOKE**

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

**System:** This includes permissions for creating session, table, etc and all types of other system privileges.

**Object:** This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

GRANT: Used to provide any user access privileges or other priviliges for the database.

REVOKE: Used to take back permissions from any user.

**Allow a User to create session**

When we create a user in SQL, it is not even allowed to login and create a session until and unless proper permissions/priviliges are granted to the user.

Following command can be used to grant the session creating priviliges.

GRANT CREATE SESSION TO username;

**Allow a User to create table**

To allow a user to create tables in the database, we can use the below command,

GRANT CREATE TABLE TO username;

**Provide user with space on tablespace to store table**

Allowing a user to create table is not enough to start storing data in that table. We also must provide the user with priviliges to use the available tablespace for their table and data.

ALTER USER username QUOTA UNLIMITED ON SYSTEM;

The above command will alter the user details and will provide it access to unlimited tablespace on system.

**NOTE:** Generally unlimited quota is provided to Admin users.

**Grant all privilege to a User**

sysdba is a set of priviliges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the sysdba permission.

GRANT sysdba TO username

**Grant permission to create any table**

Sometimes user is restricted from creating come tables with names which are reserved for system tables. But we can grant privileges to a user to create any table using the below command,

GRANT CREATE ANY TABLE TO username

**Grant permission to drop any table**

As the title suggests, if you want to allow user to drop any table from the database, then grant this privilege to the user,

GRANT DROP ANY TABLE TO username

**To take back Permissions**

And, if you want to take back the privileges from any user, use the REVOKE command.

REVOKE CREATE TABLE FROM username

**Using the WHERE SQL clause**

WHERE clause is used to specify/apply any condition while retrieving, updating or deleting data from a table. This clause is used mostly with SELECT, UPDATE and DELETEquery.

When we specify a condition using the WHERE clause then the query executes only for those records for which the condition specified by the WHERE clause is true.

**Syntax for WHERE clause**

Here is how you can use the WHERE clause with a DELETE statement, or any other statement,

DELETE FROM table\_name WHERE [condition];

The WHERE clause is used at the end of any SQL query, to specify a condition for execution.

**Time for an Example**

Consider a table **student**,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

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

Let's write a simple SQL query to display the record for student with s\_id as 101.

SELECT s\_id,

name,

age,

address

FROM student WHERE s\_id = 101;

Following will be the result of the above query.

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

**Applying condition on Text Fields**

In the above example we have applied a condition to an integer value field, but what if we want to apply the condition on name field. In that case we must enclose the value in single quote ' '. Some databases even accept double quotes, but single quotes is accepted by all.

SELECT s\_id,

name,

age,

address

FROM student WHERE name = 'Adam';

Following will be the result of the above query.

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

**Operators for WHERE clause condition**

Following is a list of operators that can be used while specifying the WHERE clause condition.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal to |
| != | Not Equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or Equal to |
| >= | Greate than or Equal to |
| BETWEEN | Between a specified range of values |
| LIKE | This is used to search for a pattern in value. |
| IN | In a given set of values |

**SQL LIKE clause**

LIKE clause is used in the condition in SQL query with the WHERE clause. LIKE clause compares data with an expression using wildcard operators to match pattern given in the condition.

**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 a single 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 |

**Using \_ and %**

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 |

**Using % only**

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 |

**SQL ORDER BY Clause**

Order by clause is used with SELECT statement for arranging retrieved data in sorted order. The **Order by** clause by default sorts the retrieved data in ascending order. To sort the 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;

**Using default 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 the resultant data 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 |

**Using Order by DESC**

Consider the **Emp** table described above,

SELECT \* FROM Emp ORDER BY salary DESC;

The above query will return the resultant data 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 |

Check out other DCL commands and their usage:

[SELECT query](https://www.studytonight.com/dbms/select-query.php)

[WHERE clause](https://www.studytonight.com/dbms/where-clause.php)

[LIKE clause](https://www.studytonight.com/dbms/like-clause.php)

[Group BY clause](https://www.studytonight.com/dbms/groupby-clause.php)

[Having clause](https://www.studytonight.com/dbms/having-clause.php)

**SQL 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** or in other words, we will be grouping employees based on their salaries, hence, as a result, we will get a data set, with unique salaries listed, along side the first employee's name and age to have that salary. Hope you are getting the point here!

group by is used to group different row of data together based on any one column.

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 of the SQL query, just like the Order by clause.

**SQL 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 by based SQL queries, just like WHERE clause is used with SELECT query.

**Syntax** for HAVING clause is,

SELECT column\_name, function(column\_name)

FROM table\_name

WHERE column\_name condition

GROUP BY column\_name

HAVING function(column\_name) condition

**Example of SQL Statement using HAVING**

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 BY customer

HAVING sum(previous\_balance) > 3000

Result will be,

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

The main objective of the above SQL query was to find out the name of the customer who has had a **previous\_balance** more than **3000**, based on all the previous sales made to the customer, hence we get the first row in the table for customer 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 records from any table in the database.

**Syntax for DISTINCT Keyword**

SELECT DISTINCT column-name FROM table-name;

**Example using DISTINCT Keyword**

Consider the following **Emp** table. As you can see in the table below, there is employee **name**, along with employee **salary** and **age**.

In the table below, multiple employees have the same salary, so we will be using DISTINCT keyword to list down distinct salary amount, that is currently being paid to the employees.

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

**SQL AND & OR operator**

The AND and OR operators are used with the 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 the WHERE clause, alongside, SELECT, UPDATE or DELETE SQL queries.

**Example of AND operator**

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**. Hope you get the concept here. We have used the AND operator to specify two conditions with WHERE clause.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 specified conditions will be there in the result.

But in case of OR operator, atleast one condition from the conditions specified must be satisfied by any record to be in the resultset.

**Example of OR operator**

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 is greater than 25.

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

**Division Operator in SQL**

The division operator is used when we have to evaluate queries which contain the keyword ALL.

Some instances where division operator is used are:

Which person has account in all the banks of a particular city?

Which students have taken all the courses required to graduate?

In above specified problem statements, the description after the keyword 'all' defines a set which contains some elements and the final result contains those units which satisfy these requirements.

Another way how you can identify the usage of division operator is by using the logical implication of if...then. In context of the above two examples, we can see that the queries mean that,

If there is a bank in that particular city, that person must have an account in that bank.

If there is a course in the list of courses required to be graduated, that person must have taken that course.

Do not worry if you are not clear with all this new things right away, we will try to expain as we move on with this tutorial.

We shall see the second example, mentioned above, in detail.

**Table 1: Course\_Taken** → It consists of the names of Students against the courses that they have taken.

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| Robert | Databases |
| Robert | Programming Languages |
| David | Databases |
| David | Operating Systems |
| Hannah | Programming Languages |
| Hannah | Machine Learning |
| Tom | Operating Systems |

**Table 2: Course\_Required** → It consists of the courses that one is required to take in order to graduate.

|  |
| --- |
| **Course** |
| Databases |
| Programming Languages |

**Using Division Operator**

So now, let's try to find out the correct SQL query for getting results for the first requirement, which is:

**Query:** Find all the students who can graduate. (i.e. who have taken all the subjects required for one to graduate.)

Unfortunately, there is no direct way by which we can express the division operator. Let's walk through the steps, to write the query for the division operator.

**1. Find all the students**

Create a set of all students that have taken courses. This can be done easily using the following command.

CREATE TABLE AllStudents AS SELECT DISTINCT Student\_Name FROM Course\_Taken

This command will return the table **AllStudents**, as the resultset:

|  |
| --- |
| **Student\_name** |
| Robert |
| David |
| Hannah |
| Tom |

**2. Find all the students and the courses required to graduate**

Next, we will create a set of students and the courses they need to graduate. We can express this in the form of Cartesian Product of **AllStudents** and **Course\_Required** using the following command.

CREATE table StudentsAndRequired AS

SELECT AllStudents.Student\_Name, Course\_Required.Course

FROM AllStudents, Course\_Required

Now the new resultset - table **StudentsAndRequired** will be:

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| Robert | Databases |
| Robert | Programming Languages |
| David | Databases |
| David | Programming Languages |
| Hannah | Databases |
| Hannah | Programming Languages |
| Tom | Databases |
| Tom | Programming Languages |

**3. Find all the students and the required courses they have not taken**

Here, we are taking our first step for finding the students who cannot graduate. The idea is to simply find the students who have not taken certain courses that are required for graduation and hence they wont be able to graduate. This is simply all those tuples/rows which are present in **StudentsAndRequired** and not present in **Course\_Taken**.

CREATE table StudentsAndNotTaken AS

SELECT \* FROM StudentsAndRequired WHERE NOT EXISTS

(Select \* FROM Course\_Taken WHERE StudentsAndRequired.Student\_Name = Course\_Taken.Student\_Name

AND StudentsAndRequired.Course = Course\_Taken.Course)

The table **StudentsAndNotTaken** comes out to be:

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| David | Programming Languages |
| Hannah | Databases |
| Tom | Databases |
| Tom | Programming Languages |

**4. Find all students who cannot graduate**

All the students who are present in the table **StudentsAndNotTaken** are the ones who cannot graduate. Therefore, we can find the students who cannot graduate as,

CREATE table CannotGraduate AS SELECT DISTINCT Student\_Name FROM StudentsAndNotTaken

|  |
| --- |
| **Student\_name** |
| David |
| Hannah |
| Tom |

**5. Find all students who can graduate**

The students who can graduate are simply those who are present in **AllStudents** but not in **CannotGraduate**. This can be done by the following query:

CREATE Table CanGraduate AS SELECT \* FROM AllStudents

WHERE NOT EXISTS

(SELECT \* FROM CannotGraduate WHERE

CannotGraduate.Student\_name = AllStudents.Student\_name)

The results will be as follows:

|  |
| --- |
| **Student\_name** |
| Robert |

Hence we just learned, how different steps can lead us to the final answer. Now let us see how to write all these 5 steps in one single query so that we do not have to create so many tables.

SELECT DISTINCT x.Student\_Name FROM Course\_Taken AS x WHERE NOT

EXISTS(SELECT \* FROM Course\_Required AS y WHERE NOT

EXISTS(SELECT \* FROM Course\_Taken AS z

WHERE z.Student\_name = x.Student\_name

AND z.Course = y.Course ))

|  |
| --- |
| **Student\_name** |
| Robert |

This gives us the same result just like the 5 steps above.

**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 the 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**

By default, a [column](https://www.studytonight.com/dbms/rdbms-concept.php) can hold NULL values. If you do not want a column to have a NULL value, use the NOT NULL constraint.

It restricts a column from having a NULL value.

We use [ALTER](https://www.studytonight.com/dbms/alter-query.php) statement and [MODIFY](https://www.studytonight.com/dbms/alter-query.php) statement to specify this constraint.

One important point to note about this 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.

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

ALTER TABLE Student

MODIFY s\_id int NOT NULL;

**UNIQUE Constraint**

It ensures that a column will only have unique values. A UNIQUE constraint field cannot have any duplicate data.

It prevents two records from having identical values in a column

We use [ALTER](https://www.studytonight.com/dbms/alter-query.php) statement and [MODIFY](https://www.studytonight.com/dbms/alter-query.php) statement to specify this constraint.

**Example of UNIQUE Constraint:**

Here we have a simple CREATE query to create a table, which will have a column **s\_id** with unique values.

CREATE TABLE Student

( s\_id int NOT NULL,

name varchar(60),

age int NOT NULL UNIQUE

);

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

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

ALTER TABLE Student

MODIFY age INT NOT NULL UNIQUE;

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.

**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.

**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](https://www.studytonight.com/dbms/database-key.php) is used to relate two tables. The relationship between the two tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

This is also called a referencing key.

We use [ALTER](https://www.studytonight.com/dbms/alter-query.php) statement and [ADD](https://www.studytonight.com/dbms/alter-query.php) statement to specify this constraint.

To understand FOREIGN KEY, let's see its use, with help of the below tables:

**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.

If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

**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 in Customer\_Detail table.

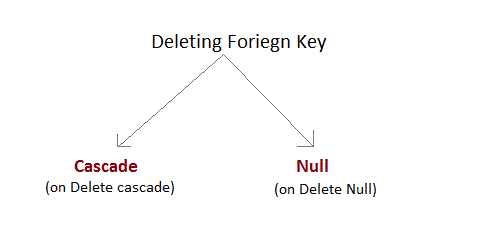
**FOREIGN KEY constraint at Column Level**

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

**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.

**Using CHECK constraint at Column Level**

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

**Related Tutorials**:

[SQL function](https://www.studytonight.com/dbms/sql-function.php)

[SQL Join](https://www.studytonight.com/dbms/joining-in-sql.php)

[SQL Alias](https://www.studytonight.com/dbms/sql-alias.php)

[SQL SET operation](https://www.studytonight.com/dbms/set-operation-in-sql.php)

[SQL Sequences](https://www.studytonight.com/dbms/sql-sequences.php)

[SQL Views](https://www.studytonight.com/dbms/sql-views.php)

**What are 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 categories,

Aggregate Functions

Scalar Functions

**Aggregate Functions**

These functions **return a single value** after performing calculations on a group of values. Following are some of the frequently used Aggregrate functions.

**AVG() Function**

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

Its general **Syntax** is,

SELECT AVG(column\_name) FROM table\_name

**Using AVG() function**

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 |

SQL query to find average salary will be,

SELECT avg(salary) from Emp;

Result of the above query will be,

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

**COUNT() Function**

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

**Using COUNT() function**

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 |

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 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 |

SQL query is,

SELECT COUNT(DISTINCT salary) FROM emp;

Result of the above query will be,

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

**FIRST() Function**

First function returns first value of a selected column

**Syntax** for FIRST function is,

SELECT FIRST(column\_name) FROM table-name;

**Using FIRST() function**

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 |

SQL query will be,

SELECT FIRST(salary) FROM Emp;

and the result will be,

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

**LAST() Function**

LAST function returns the return last value of the selected column.

**Syntax** of LAST function is,

SELECT LAST(column\_name) FROM table-name;

**Using LAST() function**

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 |

SQL query will be,

SELECT LAST(salary) FROM emp;

Result of the above query will be,

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

**MAX() Function**

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

**Syntax** of MAX function is,

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

**Using MAX() function**

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 |

SQL query to find the Maximum salary will be,

SELECT MAX(salary) FROM emp;

Result of the above query will be,

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

**MIN() Function**

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

**Syntax** for MIN function is,

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

**Using MIN() function**

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 |

SQL query to find minimum salary is,

SELECT MIN(salary) FROM emp;

Result will be,

|  |
| --- |
| **MIN(salary)** |
| 6000 |

**SUM() Function**

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

**Syntax** for SUM is,

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

**Using SUM() function**

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 |

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 some frequently used Scalar Functions in SQL.

**UCASE() Function**

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

**Syntax** of UCASE,

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

**Using UCASE() function**

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 |

SQL query for using UCASE is,

SELECT UCASE(name) FROM emp;

Result is,

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

**LCASE() Function**

LCASE function is used to convert value of string columns to Lowecase characters.

**Syntax** for LCASE is,

SELECT LCASE(column\_name) FROM table-name;

**Using LCASE() function**

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 |

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 |

**MID() Function**

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;

**Using MID() function**

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 |

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 |

**ROUND() Function**

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;

**Using ROUND() function**

Consider the 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 |

**SQL JOIN**

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It 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, which is known as, **Self Join**.

**Types of JOIN**

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 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**

Following is the **class** table,

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

and 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 resultset 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 |

As you can see, this join returns the cross product of all the records present in both the tables.

**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 SQL 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**

Consider a **class** table,

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

and the **class\_info** table,

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

**Inner** JOIN query will be,

SELECT \* from class INNER JOIN class\_info where class.id = class\_info.id;

The resultset 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.

The Syntax for Natural Join is,

SELECT \* FROM

table-name1 NATURAL JOIN table-name2;

**Example of Natural JOIN**

Here is the **class** table,

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

and 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 resultset 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 resultset table with the **matched data** from the two tables and then the remaining rows of the **left** table and null from the **right** table's columns.

Syntax for Left Outer Join is,

SELECT column-name-list FROM

table-name1 LEFT OUTER JOIN table-name2

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

To specify a condition, we use the ON keyword with Outer Join.

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**

Here is the **class** table,

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

and 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 resultset 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 |

**RIGHT Outer Join**

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

Syntax for Right Outer Join 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**

Once again the **class** table,

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

and 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 resultant 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 resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Syntax of Full Outer Join 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 |

and 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 resultset 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 |

Now that we have learned SQL JOIN, you can check these SQL topics as well and their usage:

[SQL function](https://www.studytonight.com/dbms/sql-function.php)

[SQL Alias](https://www.studytonight.com/dbms/sql-alias.php)

[SQL SET operation](https://www.studytonight.com/dbms/set-operation-in-sql.php)

[SQL Views](https://www.studytonight.com/dbms/sql-views.php)

**SQL Alias - AS Keyword**

**Alias** is used to give an alias name to a table or a column, which can be a resultset table too. 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 SQL query using **alias**,

SELECT \* FROM Employee\_detail AS ed;

**Syntax for defining alias 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 |

and 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;

and the resultset table will look like,

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

SQL Alias seems to be quite a simple feature of SQL, but it is highly useful when you are working with more than 3 tables and have to use JOIN on them.

**SET Operations in SQL**

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

In this tutorial, we will cover 4 different types of SET operations, along with example:

UNION

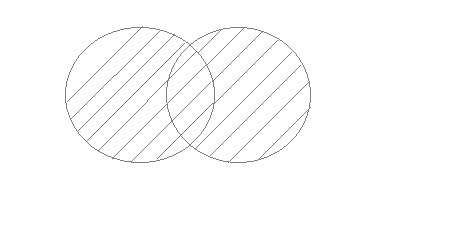
UNION ALL

INTERSECT

MINUS

**UNION Operation**

**UNION** is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.



**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

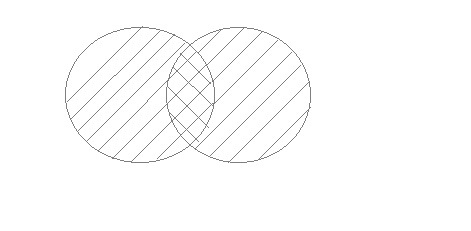
SELECT \* FROM Second;

The resultset 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;

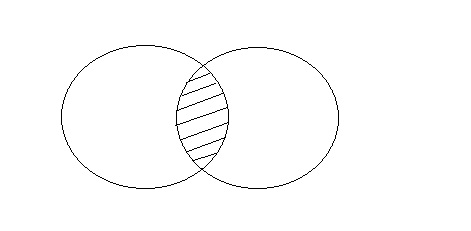
The resultset table will look like,

|  |  |
| --- | --- |
| **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.

**NOTE:** 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

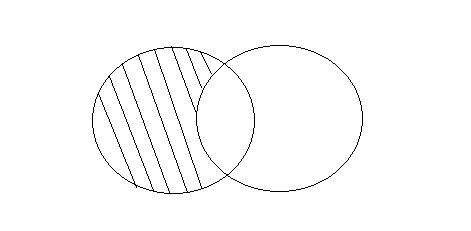
SELECT \* FROM Second;

The resultset table will look like

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

**MINUS**

The Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.



**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 resultset table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |

**Query Optimization**

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. A query optimizer decides the best methods for implementing each query.

The query optimizer selects, for instance, whether or not to use indexes for a given query, and which join methods to use when joining multiple tables. These decisions have a tremendous effect on SQL performance, and query optimization is a key technology for every application, from operational Systems to data warehouse and analytical systems to content management systems.

* **Query:** A query is a request for information from a database.
* **Query Plans:**A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.
* **Query Optimization:**A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

There are broadly two ways a query can be optimized:

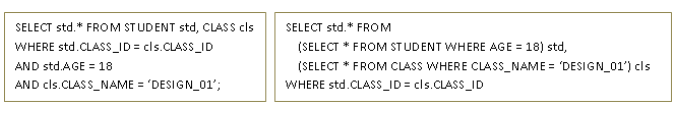
* **Analyze and transform equivalent relational expressions:** Try to minimize the tuple and column counts of the intermediate and final query processes (discussed here).
* **Using different algorithms for each operation:** These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block accesses (discussed in query processing).

There are two methods of query optimization.

**1. Cost based Optimization (Physical):** This is based on the cost of the query. The query can use different paths based on indexes, constraints, sorting methods etc. This method mainly uses the statistics like record size, number of records, number of records per block, number of blocks, table size, whether whole table fits in a block, organization of [tables](https://www.tutorialcup.com/dbms/tables.htm), uniqueness of column values, size of columns etc.

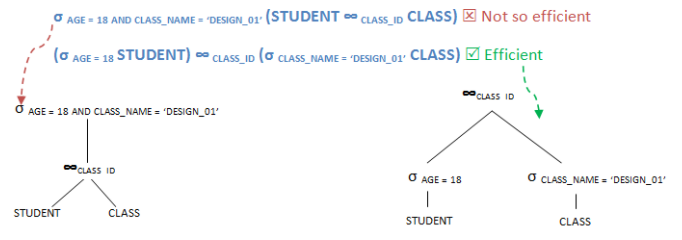
**2. Heuristic Optimization (Logical):** This method is also known as rule based optimization. This is based on the equivalence rule on relational expressions; hence the number of combination of queries get reduces here. Hence the cost of the query too reduces. This method creates relational tree for the given query based on the equivalence rules. These equivalence rules by providing an alternative way of writing and evaluating the query, gives the better path to evaluate the query. This rule need not be true in all cases. It needs to be examined after applying those rules.

Suppose we have a query to retrieve the students with age 18 and studying in class DESIGN\_01. We can get all the student details from  STUDENT table, and class details from CLASS table. We can write this query in two different ways.

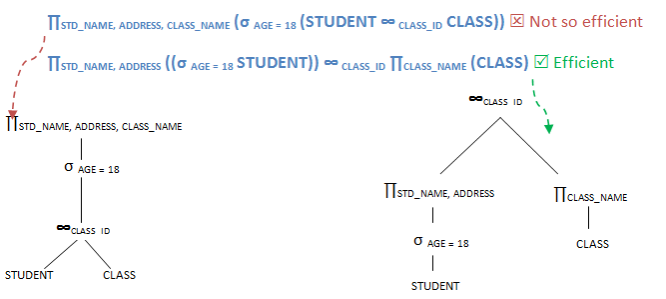


Here both the queries will return same result. But when we observe them closely we can see that first query will join the two tables first and then applies the filters. That means, it traverses whole table to join, hence the number of records involved is more. But he second query, applies the filters on each table first. This reduces the number of records on each table (in class table, the number of record reduces to one in this case!). Then it joins these intermediary tables. Hence the cost in this case is comparatively less.

Instead of writing query the optimizer creates relational algebra and tree for above case.



Suppose for example, we have to select only student name, address and class name of students with age 18 from STUDENT and CLASS tables.



Here again, both the queries look alike, results alike. But when we compare the number of records and attributes involved at each stage, second query uses less records and hence more efficient.