Experiment No: 1

Title: Study & Implementation of SQL Triggers

Objective:

Outcomes:

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Experiment No: 1

Title: Implementation of DDL commands of SQL with suitable examples
- Create table
- Alter table
- Drop Table

Objective: To study and execute the DDL commands in RDBMS

DDL aims to establish and modify the structure of objects stored in a database.

These definitions and modifications control descriptions of the database schema.

Outcomes:

SQL DDL commands are used for creating new database objects (CREATE command), modifying existing database objects (ALTER command), and deleting or removing database objects (DROP and TRUNCATE commands).

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Query:

Create table student(roll number(3) primary key, name varchar(20) not null, dateofbirth date, fees number(7,2));

- Alter table

Alter table student add (address varchar2(50));

- Drop Table

drop table student;

Experiment No: 2

Title: Implementation of DML commands of SQL with suitable examples

- Insert
- Update
- Delete

Objective:

purpose of DML commands is to select, insert, delete, update, and merge data records in RDBMS.

Data manipulation language is used to update existing data within a database table with the update command.

Outcomes:

Query:

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

INSERT

Insert into Employee(Emp_id, Emp_name) values (001, " bhanu");

Insert into Employee(Emp_id, Emp_name) values (002, " hari");

Insert into Employee(Emp_id, Emp_name) values (003, " bob");

Update

UPDATE Employee SET Emp_name= Ram WHERE Emp_id= 001;

Delete

DELETE from Employee WHERE Emp_id=002;

Experiment No: 3

Title: Implementation of different types of function with suitable examples

- Number function
- Aggregate Function
- Character Function
- Conversion Function
- Date Function

Objective: The Database functions perform basic operations, such as Sum, Average, Count, etc., and additionally use criteria arguments, that allow you to perform the calculation only for a specified subset of the records in your Database.

Outcomes:

- Manages a designed database.
- Arranges database using Relational algebra.
- Organizes database using SQL

Query:

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Number function

ABS(number)

Returns the absolute positive value of an expression.
Syntax:

```
ABS(expression)
```

Example:

```
SELECT ABS(-1.0), ABS(0.0), ABS(1.0)
```

Output:

```
1.0   .0   1.0
```

FLOOR(number)

Returns the largest integer less than, or equal to, the specified numeric expression.
Syntax:

```
FLOOR(expression)
```

Example:

```
SELECT FLOOR($223.45), CEILING($-223.45), CEILING($0.0)
```

Output:

```
223.00   -224.00   0.00
```

MOD(number, divisor)

Returns the remainder of the division from 2 integer values.
Syntax:

```
MOD(dividend, divisor)
```

Example:

```
SELECT MOD(20,3)
```

Output:

```
2
```

POWER(number, power)

Returns the exponential value for the numeric expression.
Syntax:

```
POWER(number, power)
```

Example:

```
SELECT POWER(2.0, 3.0)
```

Output:

```
8.0
```

Aggregate Function

An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Various types of SQL aggregate functions are:

- Count()
- Sum()
- Avg()
- Min()
- Max()

COUNT() Function
The COUNT() function returns the number of rows in a database table.
Syntax:
COUNT(*)
or
COUNT([ALL|DISTINCT] expression)
The SUM() function returns the total sum of a numeric column.
Syntax:
SUM()
or
SUM([ALL|DISTINCT] expression)

AVG() Function
The AVG() function calculates the average of a set of values.
Syntax:
AVG()
or
AVG([ALL|DISTINCT] expression)

MIN() Function
The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.
Syntax:
MIN()
or
MIN([ALL|DISTINCT] expression)

The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
Syntax:
AVG()
or
AVG([ALL|DISTINCT] expression)

Character Function
LOWER(SQL course)
Input1: SELECT LOWER('GEEKSFORGEEKS') FROM DUAL;
Output1: geeksforgeeks

UPPER(SQL course)
Input1: SELECT UPPER('geeksforgeeks') FROM DUAL;
Output1: GEEKSFORGEEKS
INITCAP(SQL course)
Input1: SELECT INITCAP('geeksforgeeks is a computer science portal for geeks') FROM DUAL;

CONCAT('String1', 'String2')
Input1: SELECT CONCAT('computer' ,'science') FROM DUAL;
Output1: computerscience

LENGTH(Column|Expression)
Input1: SELECT LENGTH('Learning Is Fun') FROM DUAL;
Output1: 15

Syntax: INSTR(Column|Expression, 'String', [,m], [n])
Input: SELECT INSTR('Google apps are great applications','app',1,2) FROM DUAL;
Output: 23

 Conversion Function

SELECT TO_CHAR(salary, ’$99,999.00’) SALARY

FROM employees

WHERE last_name = ’Ernst’;

Date Function

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.

We use the following SELECT statement:

SELECT * FROM Orders WHERE OrderDate='2008-11-11'

The result-set will look like this:

Experiment No: 4

Title: Implementation of different types of operators in SQL

- Arithmetic Operators
- Logical Operators
- Comparison Operator
- Special Operator
- Set Operation

Objective:

+	It adds the value of both operands.	a+b will give 30
-	It is used to subtract the right-hand operand from the left-hand operand.	a-b will give 10
*	It is used to multiply the value of both operands.	a*b will give 200
/	It is used to divide the left-hand operand by the right-hand operand.	a/b will give 2
%	It is used to divide the left-hand operand by the right-hand operand and returns reminder.	a%b will give 0

Outcomes:

An operator is a reserved word or a character that is used to query our database in a SQL expression. To query a database using operators, we use a WHERE clause. Operators are necessary to define a condition in SQL, as they act as a connector between two or more conditions.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Processor: Four or more 3.3 GHz Intel Xeon class or higher processor cores.

Memory: 8 GB RAM or more

Addition Operator

```
-- returns new column named total_amount which is
-- 100 added to the amount field
SELECT item, amount, amount+100 AS total_amount
FROM Orders;
```

Run Code

Subtraction Operator

```
-- returns new column named offer_price which is
-- 20 subtracted to the amount field
SELECT item, amount, amount-20 AS offer_price
FROM Orders;
```

Run Code

Multiplication Operator

```
-- returns new column named total_amount  which is
-- 4 multiplied to the amount field
SELECT item, amount, amount*4 AS total_amount
FROM Orders;
```

Run Code

Division Operator

```
-- returns new column named half_amount which is
-- divided by 2 to the amount field
SELECT item, amount, amount/2 AS half_amount
FROM Orders;
```

Run Code

- **Logical Operators**

The Logical Operators in SQL are as follows:

1. SQL AND OPERATOR

```
mysql> SELECT * FROM employees WHERE City = "Mumbai" AND Designation = "Project Manager";
```

2. SQL OR OPERATOR

```
mysql> SELECT * FROM employees WHERE Designation = "System Engineer" OR City = "Mumbai";
```

3. SQL NOT OPERATOR

```
mysql> SELECT * FROM employees WHERE NOT Designation = "Project Manager";
```

4. SQL BETWEEN OPERATOR

```
mysql> SELECT * FROM employees WHERE Salary BETWEEN 50000 AND 90000;
```

5. SQL IN OPERATOR

```
mysql> SELECT * FROM employees WHERE City IN ("Mumbai", "Bangalore", "Pune");
```

• Comparison Operator

Equal to operator(=)

```
SELECT *
FROM suppliers
WHERE supplier_name = 'Microsoft';
```

```
SELECT *
FROM suppliers
WHERE supplier_name <> 'Microsoft';
```

```
SELECT *
FROM customers
WHERE customer_id > 6000;
```

• Special Operator

ALL operator

The ALL operator compares a value with all the values returned by the subquery and is true only if the given condition is satisfied for all the values. For example −

```
Select * from Employee
Where Emp_Salary > ALL (select Emp_Salary from Employee
where Emp_DeptID=30);
```

• Set Operation

1. mysql> SELECT *FROM t_employees UNION SELECT *FROM t2_employees; mysql> SELECT *FROM t_employees UNION ALL SELECT *FROM t2_employees;
2. mysql> SELECT *FROM t_students UNION ALL SELECT *FROM t2_students;

Experiment No: 5

Title: Implementation of different types of Joins
- Inner Join
- Outer Join
- Natural Join etc.

Objective: (INNER) JOIN :To Returns records that have matching values in both tables.

- LEFT (OUTER) To JOIN : Returns all records from the left table, and the matched records from the right table.
- RIGHT (OUTER) JOIN : To Returns all records from the right table, and the matched records from the left table.

Outcomes: A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Example Queries(INNER JOIN)

This query will show the names and age of students enrolled in different courses.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Natural Join in SQL refers to joining two or more tables based on common columns, which have the same name and data type. We do not need to specify the column used for joining two tables in natural join. Natural join is used to retrieve data from more than one table in a single place.

Create Employee Table

```
CREATE TABLE employee (
   EmployeeID varchar(10),
   FirstName varchar(50),
   LastName varchar(50),
   DeptID varchar(10)
);
```

Insert Records into Employee Table

```
INSERT INTO employee
VALUES("E62549", "John", "Doe", "D1001"),
    ("E82743", "Priya", "Sharma", "D3002"),
    ("E58461", "Raj", "Kumar", "D1002"),
    ("E95462", "Ravi", "", "D1001"),
    ("E25947", "Shreya", "P", "D3000"),
    ("E42650", "Jane", "Scott", "D3001")
```

We can view the employee table using SELECT query.

```
SELECT * FROM employee
```

Output

```
| EmployeeID | FirstName | LastName | DeptID |
|:----------:|:---------:|:--------:|:------:|
| E62549     | John      | Doe      | D1001  |
| E82743     | Priya     | Sharma   | D3002  |
| E58461     | Raj       | Kumar    | D1002  |
| E95462     | Ravi      |          | D1001  |
| E25947     | Shreya    | P        | D3000  |
| E42650     | Jane      | Scott    | D3001  |
```

Create department table

```
CREATE TABLE department (
   DeptID varchar(10),
   DeptName varchar(40),
   Location varchar(40)
);
```

Insert records into department table

```
INSERT INTO department
VALUES("D1001", "Technology", "Bangalore"),
```

```
    ("D1002", "Technology", "Hyderabad"),
    ("D3001", "Sales", "Gurugram"),
    ("D3002", "Operations", "Hyderabad"),
    ("D4002", "Finance", "Mumbai")
```

We can view the department table using SELECT query.

```
SELECT * FROM department
```

Output

```
| DeptID | DeptName   | Location  |
|:------:|:----------:|:---------:|
| D1001  | Technology | Bangalore |
| D1002  | Technology | Hyderabad |
| D3001  | Sales      | Gurugram  |
| D3002  | Operations | Hyderabad |
| D4002  | Finance    | Mumbai    |
```

Natural Join on Employee and Department Tables

Code

```
SELECT * FROM employee NATURAL JOIN department
```

Output

```
| EmployeeID | FirstName | LastName | DeptID | DeptName   | Location  |
|:----------:|:---------:|:--------:|:------:|:----------:|:---------:|
| E62549     | John      | Doe      | D1001  | Technology | Bangalore |
| E82743     | Priya     | Sharma   | D3002  | Operations | Hyderabad |
| E58461     | Raj       | Kumar    | D1002  | Technology | Hyderabad |
| E95462     | Ravi      |          | D1001  | Technology | Bangalore |
| E42650     | Jane      | Scott    | D3001  | Sales      | Gurugram  |
```

Experiment No: 6

Title: Study and Implementation of
- Group By & having clause
- Order by clause
- Indexing

Objective: The GROUP BY Clause is used together with the SQL SELECT statement. The SELECT statement used in the GROUP BY clause can only be used contain column names, aggregate functions, constants and expressions. SQL Having Clause is used to restrict the results returned by the GROUP BY clause.

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Outcomes:

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Group By & having clause

```
SELECT `gender` FROM `members` ;
SELECT `gender` FROM `members` GROUP BY `gender`;
```

```
SELECT `category_id`,`year_released` FROM `movies` ;
SELECT `category_id`,`year_released` FROM `movies` GROUP BY `category_id`,`year_released`;
```

```
SELECT * FROM `movies` GROUP BY `category_id`,`year_released` HAVING `category_id` = 8;
```

Order by clause

```
SELECT * FROM Customers
ORDER BY Country DESC;
```

Indexing

```
CREATE INDEX index_name
ON table_name (column_name);
```

CREATE INDEX index_state **ON** Employee (Emp_State);

Experiment No: 7

Title: Study & Implementation of

- Sub queries
- Views

Objective: To know how to use sub query for solving nested queries.

To know how to display data using view.

Outcomes: It is used to return data from a table, and this data will be used in the main query as a condition to further restrict the data to be retrieved.

Views display only selected data. We can also use Sql Join s in the Select statement in deriving the data for the view.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
  (SELECT column_name [, column_name ]
  FROM table1 [, table2 ]
  [WHERE])
```

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
```

```
| 3 | kaushik  |  23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 6 | Komal    |  22 | MP        |  4500.00 |
| 7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT *
   FROM CUSTOMERS
   WHERE ID IN (SELECT ID
      FROM CUSTOMERS
      WHERE SALARY > 4500) ;
```

This would produce the following result.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
| 4 | Chaitali |  25 | Mumbai  |  6500.00 |
| 5 | Hardik   |  27 | Bhopal  |  8500.00 |
| 7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
  SELECT [ *|column1 [, column2 ]
  FROM table1 [, table2 ]
  [ WHERE VALUE OPERATOR ]
```

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP
  SELECT * FROM CUSTOMERS
  WHERE ID IN (SELECT ID
  FROM CUSTOMERS) ;
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> DELETE FROM CUSTOMERS

   WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP

      WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

A view is created with the `CREATE VIEW` statement.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Creating a View in SQL

We can create views in SQL by using the `CREATE VIEW` command. For example,

```
CREATE VIEW us_customers AS
SELECT customer_id, first_name
FROM Customers
WHERE Country = 'USA';
```

Updating a View

It's possible to change or update an existing **view** using the `CREATE OR REPLACE VIEW` command. For example,

```
CREATE OR REPLACE VIEW us_customers AS
SELECT *
FROM Customers
WHERE Country = 'USA';
```

Here, the `us_customers` view is updated to show all the fields.

Deleting a View

We can delete views using the `DROP VIEW` command. For example,

```
DROP VIEW us_customers;
```

Experiment No: 8

Title: Study & Implementation of PL/SQL

Objective: To understand PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).
PL/SQL is a combination of SQL along with the procedural features of programming languages.

Outcomes: learn programming, management, and security issues of working with PL/SQL program units

Programming topics will include the built-in packages that come with Oracle, the creation of triggers, and stored procedure features.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

3! = 3*2*1 = 6

```
Declare
num   number:= #
fact number:= 1;
temp number;
begin
  temp := num;
   while (num > 0)
loop
    fact := fact * num;
    num   := num - 1;
end loop;
  Dbms_Output.Put_line('factorial of ' || temp ||  '  is ' || fact);
end;
/
```

Experiment No: 9

Title: Study & Implementation of SQL Cursors

Objective: To understand how to use SQL cursor for solving SQL queries.

Outcomes: The cursor's purpose is to update the data row by row, change it, or perform calculations that are not possible when we retrieve all records at once.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Syntax: To declare a cursor.

CURSOR cursor_name

IS

SELECT columns

FROM table_name

WHERE conditions;

Syntax: To open a cursor.

OPEN cursor_name;

Syntax: To fetch rows from a cursor.

FETCH cursor_name INTO variables;

Syntax: To close cursor.

CLOSE cursor_name;

Example 1: Using cursor in a function.

```
CREATE OR REPLACE Function Search_Students
( name IN varchar2 )
RETURN number
IS
num number;
CURSOR cur
IS
SELECT student_name
FROM students
WHERE student_name = name;
BEGIN
OPEN cur;
FETCH cur INTO num;
 if cur % notfound then
num := 9999;
```

```
end if;
CLOSE cur;
RETURN num;
END;
```

Output:

Function created.

0.1 seconds

Experiment No: 10

Title: Study & Implementation of SQL function and procedure

Objective: Both function as well as stored procedure have a **unique named block** of code which is compiled and stored in the database.

Outcomes: Any stored procedure or function created, remains useless unless it is called.

now we will come to know how to define and use a stored procedure in PL/SQL and how to define a function and use the function in PL/SQL.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Demo Database

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

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country

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

---

Stored Procedure Example

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

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;

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

In the code below we have a program to demonstrate the use of function for adding two numbers.

```
set serveroutput on;

CREATE OR REPLACE FUNCTION Sum(a IN number, b IN
number) RETURN Number IS

c number;

BEGIN

     c := a+b;

     RETURN c;

END;
```

```
output

Function Created
```

For calling the function Sum following code will be executed:

```
set serveroutput on;

DECLARE

     no1 number;

     no2 number;
```

```
      result number;
BEGIN
      no1 := &no1;
      no2 := &no2;
      result := Sum(no1,no2);
dbms_output.put_line('Sum of two nos='||result);
END;
```

Output

```
Enter value for no1:5
Enter value for no2:5
Sum of two nos=10
PL/SQL procedure successfully created.
```

Experiment No: 11

Title: Study & Implementation of SQL Triggers

Objective: It is a specialized category of stored procedure that is called automatically when a database server event occurs. Each trigger is always associated with a table.

- **Outcomes:** We cannot manually execute/invoked triggers.
- Triggers have no chance of receiving parameters.
- A transaction cannot be committed or rolled back inside a trigger.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Syntax of Trigger

We can create a trigger in SQL Server by using the **CREATE TRIGGER** statement as follows:

```
1. CREATE TRIGGER schema.trigger_name
2. ON table_name
3. AFTER {INSERT, UPDATE, DELETE}
4. [NOT FOR REPLICATION]
5. AS
6. {SQL_Statements}
```

The parameter descriptions of this syntax illustrate below:

schema: It is an optional parameter that defines which schema the new trigger belongs to.

trigger_name: It is a required parameter that defines the name for the new trigger.

table_name: It is a required parameter that defines the table name to which the trigger applies. Next to the table name, we need to write the AFTER clause where any events like INSERT, UPDATE, or DELETE could be listed.

NOT FOR REPLICATION: This option tells that SQL Server does not execute the trigger when data is modified as part of a replication process.

SQL_Statements: It contains one or more SQL statements that are used to perform actions in response to an event that occurs.

Example of Trigger in SQL Server

Let us understand how we can work with triggers in the SQL Server. We can do this by first creating a table named '**Employee**' using the below statements:

```
1. CREATE TABLE Employee
2. (
3.   Id INT PRIMARY KEY,
```

```
4.   Name VARCHAR(45),
5.   Salary INT,
6.   Gender VARCHAR(12),
7.   DepartmentId INT
8.  )
```

Next, we will insert some record into this table as follows:

```
1.  INSERT INTO Employee VALUES (1,'Steffan', 82000, 'Male', 3),
2.  (2,'Amelie', 52000, 'Female', 2),
3.  (3,'Antonio', 25000, 'male', 1),
4.  (4,'Marco', 47000, 'Male', 2),
5.  (5,'Eliana', 46000, 'Female', 3)
```

We can verify the insert operation by using the SELECT statement. We will get the below output:

```
1.  SELECT * FROM Employee;
```

We will also create another table named '**Employee_Audit_Test'** to automatically store transaction records of each operation, such as INSERT, UPDATE, or DELETE on the Employee table:

```
1.  CREATE TABLE Employee_Audit_Test
2.  (
3.  Id int IDENTITY,
4.  Audit_Action text
5.  )
```

Now, we will **create a trigger that stores transaction records of each insert operation** on the Employee table into the Employee_Audit_Test table. Here we are going to create the insert trigger using the below statement:

```
1.  CREATE TRIGGER trInsertEmployee
2.  ON Employee
3.  FOR INSERT
4.  AS
```

```
5. BEGIN
6.   Declare @Id int
7.   SELECT @Id = Id from inserted
8.   INSERT INTO Employee_Audit_Test
9.   VALUES ('New employee with Id = ' + CAST(@Id AS VARCHAR(10)) + ' is added at ' + CAST(G
     etdate() AS VARCHAR(22)))
10. END
```

After creating a trigger, we will try to add the following record into the table:

```
1. INSERT INTO Employee VALUES (6,'Peter', 62000, 'Male', 3)
```

If no error is found, execute the SELECT statement to check the audit records. We will get the output as follows:

We are going to **create another trigger to store transaction records of each delete operation** on the Employee table into the Employee_Audit_Test table. We can create the delete trigger using the below statement:

```
1. CREATE TRIGGER trDeleteEmployee
2. ON Employee
3. FOR DELETE
4. AS
5. BEGIN
6.   Declare @Id int
7.   SELECT @Id = Id from deleted
8.   INSERT INTO Employee_Audit_Test
9.   VALUES ('An existing employee with Id = ' + CAST(@Id AS VARCHAR(10)) + ' is deleted at ' +
     CAST(Getdate() AS VARCHAR(22)))
10. END
```

After creating a trigger, we will delete a record from the Employee table:

```
1. DELETE FROM Employee WHERE Id = 2;
```

If no error is found, it gives the message as below:

Finally, execute the SELECT statement to check the audit records:

In both the triggers code, you will notice these lines:

Group B Large Scale Databases

Experiment No: 1

Title: Study of Open Source NOSQL Database: MongoDB (Installation, Basic CRUD operations, Execution)

Objective: To examine how to execute MongoDB CRUD operations using the MongoDB Query Language (MQL).

Outcomes: we came to know A MongoDB is a schema less or NoSQL database. By means of 'schema less' is that we can store different documents having different schema inside a same collection.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

2) Setup and Installation

First, let's see how to configure MongoDB locally to your machine.

Here are the Steps to install MongoDB:

Firstly, Open this URL from your browser: https://www.mongodb.com/

How to Perform CRUD Operations

, reading, updating, and deleting documents, looking at each operation in turn.

Create Operations

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true},
      {name: "Kitana",
      age: "4 years",
      species: "Cat",
      ownerAddress: "521 E. Cortland",
      chipped: true}])
```

```
db.RecordsDB.insertMany([{ name: "Marsh", age: "6 years", species: "Dog",
ownerAddress: "380 W. Fir Ave", chipped: true}, {name: "Kitana", age: "4 years",
```

```
species: "Cat", ownerAddress: "521 E. Cortland", chipped: true}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

replaceOne()

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
> db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"specie
```

Experiment No: 2

Title: Study of Open Source NOSQL Database: MongoDB (Installation, Basic CRUD operations, Execution)

Objective: To examine how to execute MongoDB CRUD operations using the MongoDB Query Language (MQL).

Outcomes: we came to know A MongoDB is a schema less or NoSQL database. By means of 'schema less' is that we can store different documents having different schema inside a same collection.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

MongoDB CRUD Operations

Share Feedback

On this page

CRUD operations *create*, *read*, *update*, and *delete* documents.

Create Operations

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

- `db.collection.insertOne()` *New in version 3.2*

- `db.collection.insertMany()` *New in version 3.2*

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

For examples, see Insert Documents.

Read Operations

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

- `db.collection.find()`

You can specify query filters or criteria that identify the documents to return.

click to enlarge

For examples, see:

- Query Documents
- Query on Embedded/Nested Documents
- Query an Array
- Query an Array of Embedded Documents

Update Operations

Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

- `db.collection.updateOne()` *New in version 3.2*
- `db.collection.updateMany()` *New in version 3.2*
- `db.collection.replaceOne()` *New in version 3.2*

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

For examples, see Update Documents.

Delete Operations

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

- `db.collection.deleteOne()` *New in version 3.2*
- `db.collection.deleteMany()` *New in version 3.2*

In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

Experiment No: 3

Title Design and Implement any 5 query using MongoDB

Objective: To examine how to execute MongoDB CRUD operations using the MongoDB Query Language (MQL).

Outcomes: we came to know A MongoDB is a schema less or NoSQL database. By means of 'schema less' is that we can store different documents having different schema inside a same collection.

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows −

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as −

```
> use sampleDB
switched to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below −

```
> db.mycol.insert([
        {
                title: "MongoDB Overview",
```

```
                description: "MongoDB is no SQL database",
                by: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
                title: "NoSQL Database",
                description: "NoSQL database doesn't have tables",
                by: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 20,
                comments: [
                        {
                                user:"user1",
                                message: "My first comment",
                                dateCreated: new Date(2013,11,10,2,35),
                                like: 0
                        }
                ]
        }
])
```

Following method retrieves all the documents in the collection −

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" :
"MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com",
"tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" :
"NoSQL database doesn't have tables", "by" : "tutorials point", "url" :
"http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20,
"comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-
12-09T21:05:00Z"), "like" : 0 } ] }
>
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

>db.COLLECTION_NAME.find().pretty()

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

```
> db.mycol.find().pretty()
{
```

```
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
        "title" : "NoSQL Database",
        "description" : "NoSQL database doesn't have tables",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 20,
        "comments" : [
                {
                        "user" : "user1",
                        "message" : "My first comment",
                        "dateCreated" : ISODate("2013-12-09T21:05:00Z"),
                        "like" : 0
                }
        ]
}
```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

Example

Following example retrieves the document with title MongoDB Overview.

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
```

```
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

Values in an array	{<key>:{$in:[<value1>, <value2>,……<valueN>] }}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
```

```
}
>
```

For the above given example, equivalent where clause will be **' where by = 'tutorials point' AND title = 'MongoDB Overview' '**. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** −

```
>db.mycol.find(
   {
      $or: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
```

```
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $not: [
                        {key1: value1}, {key2:value2}
                ]
        }
)
```

Example

Assume we have inserted 3 documents in the collection **empDetails** as shown below −

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Age: "27",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
```

```
        ]
)
```

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

```
> db.empDetails.find(
        {
                $nor:[
                        40
                        {"First_Name": "Radhika"},
                        {"Last_Name": "Christopher"}
                ]
        }
).pretty()
{
        "_id" : ObjectId("5dd631f270fb13eec3963bef"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $NOT: [
                        {key1: value1}, {key2:value2}
                ]
        }
).pretty()
```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
        "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
```

```
}
```