create user in MySQL Database

Create user “newuser” @ “localhost” identified by “password”

Ex: create user “niloy” @ “localhost” identified by “pass@123”

Find the user:
Select user from mysql.user

create table employee(

id int auto_increment,

firstname varchar(20) not null,

middlename varchar(20),

lastname varchar(20) not null ,

age int not null ,

salary int not null,

location varchar(20) default 'dhaka',

primary key(id)

);

insert into employee(firstname,lastname,age,salary)
values("Mehedi","hasan",25,10000,"sherpur"), ("Taukir", "Ahmed",29,28000, "savar");

 # Order of SQL Execution

From>where>group by>having>select>order by>limit

Basic operation

select * from employee where binary firstname='mohaiminul';(case sensitive)

select firstname as name ,lastname as surname from employee;

update employee set location="sherpur" where firstname="mehedi" and lastname='hasan';

update employee set salary=salary+5000;

update employee set location="Comilla" where lastname='niloy';

delete from employee where id=1;

alter table employee add column jobtitle varchar(255);(add new column)

alter table employee modify column firstname varchar(35);(modify column)

alter table employee modify column course_duration_months decimal(3,1) not null;(decimal value like 12.5)

alter table courses add column changed_at TIMESTAMP DEFAULT NOW(); (show me the real time and I got to know when data is inserted)

alter table courses modify column changed_at timestamp default current_timestamp() on update current_timestamp(); (give me the exact time table of timestamp when we update any data)

// current_timestamp and now() both are same, we can use any of them.

create table students(

student_id int auto_increment,

student_fname varchar (255) not null,

student_mname varchar (255) ,

selected_courses int DEFAULT 1,

student_lname varchar (255) not null,

student_email varchar (255) not null,

student_phone varchar (255) not null,

student_alternate_phone varchar (255) ,

enrollment_date TIMESTAMP not null,

years_of_exp int not null,

student_company varchar (30),

batch_date varchar (30) not null,

source_of_joining varchar (30) not null,

location varchar (50) not null,

primary key(student_id),

unique key (student_email),

FOREIGN KEY(selected_courses) REFERENCES courses(course_id)

);

parent table-courses

child-students(students table depended on courses)

insert into students(student_fname, student_lname,student_email,
student_phone,years_of_exp ,student_company, batch_date , source_of_joining, location)
values
('tamim','ibal','tamim@gmail.com','01717294603',10,'facebook','5-02-2021','linkedin','cittagong');

create table courses(

course_id int not null ,

course_name varchar (55) not null,

course_duration_months int not null ,

course_fee int not null,

changed_at timestamp default now(), // give me the time when data is inserted

changed_at timestamp default now() on update now();// when data is update give us the that time.

primary key(course_id)

);

Aggregate Function:

Count:

select count(*) from students;(show me the total rows)

select count(Distinct location) from students; (don’t show me the duplicate values)

select count(*) from students where batch_date like "%-02-%";

select count(*) from students where batch_date like "19-%";

Group By:

select source_of_joining , count(*) from students group by source_of_joining; (use comma after field name and before count when we use group by)

+-------------------+----------+

| source_of_joining | count(*) |

+-------------------+----------+

| bdjobs | 1 |

| linkedin | 7 |

When we use group by, in what field we want to group by we have to select that field.

select source_of_joining , count(*) from students group by location; (this will not work)

select location,source_of_joining from students group by location,source_of_joining;

+-----------+-------------------+

| location | source_of_joining |

+-----------+-------------------+

| magura | bdjobs |

| cittagong | linkedin |

| savar | linkedin |

| dhaka | linkedin |

| tangail | linkedin |

| bi-baria | linkedin |

select location,source_of_joining, count(*) from students group by

location,source_of_joining;

Min & Max:

select min(years_of_exp) from students;

select source_of_joining,min(years_of_exp) from students group by source_of_joining;

select source_of_joining,max(years_of_exp) from students group by source_of_joining;

+-------------------+-------------------+

| source_of_joining | max(years_of_exp) |

+-------------------+-------------------+

| bdjobs | 10 |

| linkedin | 10 |

Sum & Avg:

select source_of_joining,sum(years_of_exp) from students group by source_of_joining;

+-------------------+-------------------+

| source_of_joining | sum(years_of_exp) |

+-------------------+-------------------+

| bdjobs | 10 |

| linkedin | 70 |

select source_of_joining,avg(years_of_exp) from students group by source_of_joining;

+-------------------+-------------------+

| source_of_joining | avg(years_of_exp) |

+-------------------+-------------------+

| bdjobs | 10.0000 |

| linkedin | 10.0000 |

Logical Operators:

Equal(=):select * from students where location="savar";

Not Equal: select * from students where location !="savar"; // to get students who are not from savar

Like("%"): select * from courses where course_name like "%data%";

SELECT DISTINCT(CITY) FROM STATION WHERE CITY LIKE 'A%' OR CITY LIKE 'E%' OR CITY LIKE 'I%' OR CITY LIKE 'O%' OR CITY LIKE 'U%' ORDER BY CITY ASC; // contain the first value of a word

select distinct(city) from station where city like "%a" or city like "%e" or city like "%i" or city like "%o" or city like "%u";// contain the last value of a word

Not Like: select * from courses where course_name not like "%data%"; (give me the value who are not belong "data)

And: select * from students where years_of_exp < 15 and source_of_joining="linkedin" and location="dhaka"; (all condition should be true)

Or: select * from students where years_of_exp < 15 or years_of_exp > 8;(if one condition is true then it will execute)

Between: select * from students where years_of_exp between 8 and 12;(we will get the value who are in 8 to 12)

Not Between : select * from students where years_of_exp not between 8 and 12;(we will get the value who are not in 8 to 12)

In: select * from students where location in("magura","dhaka");

Not in: select * from students where location not in("magura","dhaka");

Exists: is a keyword in MySQL used to check whether a subquery returns any rows or not. It can be used in combination with a SELECT, INSERT, UPDATE or DELETE statement to determine if a particular condition is true or false.exists only use in a subquery.

```
SELECT *

FROM orders

WHERE EXISTS (

  SELECT *

  FROM customers

  WHERE orders.customer_id = customers.customer_id

);
```

String Function:

MySQL provides a variety of string functions that can be used to manipulate and process strings. Some of the commonly used string functions in MySQL include

CONCAT() - This function is used to concatenate two or more strings together.

SUBSTRING() - This function is used to extract a substring from a given string.

LENGTH() - This function is used to get the length of a string.

UPPER() - This function is used to convert a string to uppercase.

LOWER() - This function is used to convert a string to lowercase.

TRIM() - This function is used to remove leading and trailing spaces from a string.

REPLACE() - This function is used to replace a specific substring in a string with a new substring.

REVERSE() - This function is used to reverse a string.

LEFT() - This function is used to get a specified number of characters from the left side of a string.

RIGHT() - This function is used to get a specified number of characters from the right side of a string.

Case General Statement : **

In SQL, a `CASE` statement is used to evaluate conditions and return a result based on those conditions.(like if-else condition). If in “when” condition is not true then it will return “else” condition. And if “else” Condition is not true it will return null.

```
Select column_name

CASE

    WHEN condition1 THEN result1

    WHEN condition2 THEN result2

    ...

    ELSE default_result

END

From Table_name.
```

example Query:

```
SELECT

    customer_name,

    CASE

    WHEN purchase_amount >= 1000 THEN 'Gold'

    WHEN purchase_amount >= 500 THEN 'Silver'

    ELSE 'Bronze'

    END AS membership_level

FROM
```

Customers

Case Expression: This is sightly different from case general statement, we dont use any conditional statement like (>,=,<,>= etc). Here we have to give our expression after case.

Example:

SELECT

customer_name,

CASE purchase_amount // purchase amount is expression

WHEN 1000 THEN 'Gold'

WHEN 500 THEN 'Silver'

ELSE 'Bronze'

END AS membership_level

FROM

Customers

1. If a course is more than 4 month we want to catagorize it as a masters else diploma. For that we want to build a new column where value shows and use case method:

select course_id, course_name,course_fee,

case

when course_duration_months >4 then "masters"

else "diploma"

end as course_type

from courses;

2.

select student_id,student_fname,student_lname,location,

case

when location in("magura","savar") then "rural"

else "city"

end as loc_type

from students;

Nested Query:

Sub-Query is a inner query or nested query.

Suppose we have a payment table and it has customer_id, amount field. Now we want to find out the customer_id details whose amount is greater than the average amount.

1. Query solve: select customer_id,amount from payment where amount >

    (select avg(amount) from payment); --->inner query

2. select course_name from courses where course_id=(select selected_courses from students where student_fname="mehedi");

+-------------+

| course_name |

+-------------+

| big data |

* Nth highest salary :

SELECT salary

FROM employee

ORDER BY salary DESC

LIMIT n-1, 1;

Replace "employee" with the name of your table, and replace "n" with the desired value to find the nth highest salary. For example, if you want to find the 3rd highest salary, replace "n" with 3.

- The offset specifies the number of rows to skip from the beginning of the result set. In this case, `(n-1)` is the offset, which means the query will skip the first two rows.
- The count specifies the maximum number of rows to be returned. In this case, `1` is the count, so the query will return only one row.

Copy of a table:

create table students_latest as select * from students; (it will give the exact same schema as students table but foreign key and others will be not there)

Joins:

*/in students table selected_course(foreign key)

in courses table course_id (primary key)

so join will be happening based on these columns.

Inner Join: only matching records are considered , not matching records are discarded.

select student_fname,student_lname,course_name from students join courses on/*

students.selected_courses=courses.course_id;

```
+----------------+----------------+--------------+
| student_fname | student_lname | course_name |
```

+---------------+---------------+-------------+

| shakib | hasan | big data |

| tamim | ibal | big data |

| niloy | islam | devops |

| niloy | islam | blockchain |

| nayan | ahmed | big data |

| mehedi | hasan | big data |

| piyas | shah | big data |

| rabiul | islam | big data |

Multiple join Query:

select s.Amount,s.SaleDate,p.Salesperson,p.SPID,s.SPID,pr.Product,pr.PID from sales s

join people p on s.SPID=p.SPID //muliple join in a query

join products pr on s.PID=pr.PID

where s.Amount<200;

Left Outer Join:

all the matching records from left and right table are considered + all the non matching records in the left table which does not have the match in the right padded with null.

select student_fname, student_lname,course_name from students_latest left join courses_latest on students_latest.selected_courses=courses_latest.course_id;

+---------------+---------------+-------------+

| student_fname | student_lname | course_name |

+---------------+---------------+-------------+

| shakib | hasan | big data |

| tamim | ibal | big data |

| niloy | islam | NULL |

| niloy | islam | blockchain |

| nayan | ahmed | big data |

| mehedi | hasan | big data |

| piyas | shah | big data |

| rabiul | islam | big data |

+---------------+---------------+-------------+

Right Outer Join:

all the matching records from left and right table are considered + all the non matching records in the right table which does not have the match in the left padded with null.

+---------------+---------------+-----------------+

| student_fname | student_lname | course_name |

+---------------+---------------+-----------------+

| rabiul | islam | big data |

| piyas | shah | big data |

| mehedi | hasan | big data |

| nayan | ahmed | big data |

| tamim | ibal | big data |

| shakib | hasan | big data |

| NULL | NULL | Data Science |

| niloy | islam | blockchain |

| NULL | NULL | Web DEVELOPMENT |

+---------------+---------------+-----------------+

Full Outer Join: Combination of left and right outer join

all the matching records + non matching records from left + non matching records from right

/in mysql we cant use full outer join, but we can use it through left,right join and combine them with union.

/here is direct keyword for this so here we have to union left and right join combinedly for query

select student_fname, student_lname,course_name from students_latest left join courses_latest on students_latest.selected_courses=courses_latest.course_id

UNION

select student_fname, student_lname,course_name from students_latest right join courses_latest on students_latest.selected_courses=courses_latest.course_id;

+---------------+---------------+-----------------+

| student_fname | student_lname | course_name |

+---------------+---------------+-----------------+

| shakib | hasan | big data |

| tamim | ibal | big data |

| niloy | islam | NULL |

| niloy | islam | blockchain |

| nayan | ahmed | big data |

| mehedi | hasan | big data |

| piyas | shah | big data |

| rabiul | islam | big data |

| NULL | NULL | Data Science |

| NULL | NULL | Web DEVELOPMENT |

Union All:

We use UNION ALL to concatenate (append) two result sets into a single result set when they have the same columns (number and type).

```
(
  SELECT CONCAT(NAME, '(', SUBSTRING(OCCUPATION, 1, 1), ')') as THETEXT
          FROM OCCUPATIONS
)
UNION ALL
(
  SELECT CONCAT('There are a total of ', COUNT(*),' ', LOWER(OCCUPATION) ,(IF
(COUNT(*) > 1, 's','')), '.') as THETEXT
   FROM OCCUPATIONS  group by  OCCUPATION
)
ORDER by THETEXT ASC;
```

Cross Join:

if one table has 8 records and another has 5 records then it will combine both each records with multiply.. (8*5) so total 40 records. Thats why it is costly.

Select * from students,courses; (this will show total 40 records)

select * from students join courses; (same as its immediate query) (show 40 records)

where & Having:

1. select source_of_joining, count(*) from students group by source_of_joining;

2. select source_of_joining, count(*) as total from students group by source_of_joining where total > 1;(this query will not work)

// where clause is used to filter the individual records before aggregation. after “group by” we can’t use “where”. Instead of where we can use “having” here.

3. select source_of_joining, count(*) as total from students where source_of_joining=”linkedin” group by source_of_joining ; (this will work cuase we use where before group by)

+-------------------+-------+

| source_of_joining | total |

+-------------------+-------+

| linkedin | 2 |

+-------------------+-------+

Having:

select source_of_joining, count(*) as total from students group by source_of_joining having source_of_joining='linkedin';

+-------------------+-------+

| source_of_joining | total |

+-------------------+-------+

| linkedin | 2 |

select source_of_joining, count(*) as total from students group by source_of_joining having total >1;

+-------------------+-------+

| source_of_joining | total |

+-------------------+-------+

| linkedin | 2 |

| pathao | 2 |

Use where and having in same query:

select location, count(*) as total from students where years_of_exp >10 group by location having total >1;

+-----------+-------+

| location | total |

+-----------+-------+

| cittagong | 3 |

* where is more fast than having

Window Function:

A window function in SQL is a type of analytic function that performs a calculation across a set of rows that are related to the current row. It allows you to perform calculations on a set of rows, without grouping or aggregating them.it applies aggregate,ranking and analytic function over a particular window.

An over clause is used with window functions to define that window.

Some common examples of window functions in SQL include:

Aggregate:SUM(), AVG(), MAX(), MIN(): performs calculations on a set of rows within a window.

Ranking:

- ROW_NUMBER(): assigns a unique sequential number to each row within a window.
- RANK(): assigns a rank to each row within a window based on the order of the values.
- DENSE_RANK(): assigns a rank to each row within a window, but unlike RANK(), it does not leave gaps in the ranking.
- PARCENT RANK:

Analytic: Lead,Lag,First_value,Last_value

Select column_name,func() OVER (

[PARTITION BY <column1>, <column2>, ...]

[ORDER BY <column3> [ASC|DESC], <column4> [ASC|DESC], ...]

[ROWS <frame specification>])

From table_name

* func()-1. aggregate function   * Over-1.partition by, 2.order by, 3.rows

2. Ranking Function

3. Analytic function

Over (Partition by clause):

create table employee (

firstname varchar(20),

lastname varchar(20),

age int,

salary int,

location varchar(20)

);

insert into employee values("sachin","tendulkar",28,10000,"bangalore"),

("shane","warne",30,20000,"bangalore"),

("rohit","sharma",32,30000,"hyderbad"),

("shikhar","dhawan",32,25000,"hyderbad"),

("rahul","dravid",31,20000,"bangalore"),

("saurav","ganguly",32,15000,"kolkata"),

("virat","kohli",19,10000,"delhi");

select location,count(location), avg(salary) from employee group by location;

+-----------+-----------------+-------------+

| location | count(location) | avg(salary) |

+-----------+-----------------+-------------+

| bangalore | 3 | 16666.6667 |

| hyderbad | 2 | 27500.0000 |

| kolkata | 1 | 15000.0000 |

| delhi | 1 | 10000.0000 |

//We want to show firstname, lastname ,location ,avg salary but based on “location”. We can do this using group by . In this case we use over clause.

select firstname,lastname,location, count(location) over (partition by location) as total ,avg(salary) over(partition by location) as average from employee;

+-----------+-----------+-----------+-------+------------+

| firstname | lastname | location | total | average |

+-----------+-----------+-----------+-------+------------+

| sachin | tendulkar | bangalore | 3 | 16666.6667 |

| shane | warne | bangalore | 3 | 16666.6667 |

| rahul | dravid | bangalore | 3 | 16666.6667 |

| virat | kohli | delhi | 1 | 10000.0000 |

| rohit | sharma | hyderbad | 2 | 27500.0000 |

| shikhar | dhawan | hyderbad | 2 | 27500.0000 |

| saurav | ganguly | kolkata | 1 | 15000.0000 |

Row Number:

order it first then it assign the row number starting from one.

select firstname,lastname,salary, row_number() over(order by salary desc)as rownumber from employee;

+-----------+-----------+--------+-----------+

| firstname | lastname | salary | rownumber |

+-----------+-----------+--------+-----------+

| rohit | sharma | 30000 | 1 |

| shikhar | dhawan | 25000 | 2 |

| shane | warne | 20000 | 3 |

| rahul | dravid | 20000 | 4 |

| saurav | ganguly | 15000 | 5 |

| sachin | tendulkar | 10000 | 6 |

| virat | kohli | 10000 | 7 |

* want to find 5th highest salary

select * from (select firstname,lastname,salary, row_number() over(order

by salary desc)as rownumber from employee) temptable where rownumber=5;

+-----------+----------+--------+-----------+

| firstname | lastname | salary | rownumber |

+-----------+----------+--------+-----------+

| saurav | ganguly | 15000 | 5 |

// we use temptable to store records in a temptable

//use partition here

select firstname,lastname,salary,location, row_number() over(partition by location order by salary desc)as rownumber from employee;

+-----------+-----------+--------+-----------+-----------+

| firstname | lastname | salary | location | rownumber |

+-----------+-----------+--------+-----------+-----------+

| shane | warne | 20000 | bangalore | 1 |

| rahul | dravid | 20000 | bangalore | 2 |

| sachin | tendulkar | 10000 | bangalore | 3 |

| virat | kohli | 10000 | delhi | 1 |

| rohit | sharma | 30000 | hyderbad | 1 |

| shikhar | dhawan | 25000 | hyderbad | 2 |

| saurav | ganguly | 15000 | kolkata | 1 |

select * from (select firstname,lastname,salary,location, row_number() over(partition by location order by salary desc)as rownum from employee) temptable

where rownum=1;

+-----------+----------+--------+-----------+--------+

| firstname | lastname | salary | location | rownum |

+-----------+----------+--------+-----------+--------+

| shane | warne | 20000 | bangalore | 1 |

| virat | kohli | 10000 | delhi | 1 |

| rohit | sharma | 30000 | hyderbad | 1 |

| saurav | ganguly | 15000 | kolkata | 1 |

// when we use row_number, we should be using the order by clause. We can also use partition by but its optional.

Rank and Dense Rank:

Row number can't handle duplicate and Rank solve this issue.

Row_number ():

select firstname,lastname,salary, row_number() over(order by salary desc)

from employee;

+-----------+-----------+--------+--+

| firstname | lastname | salary | row_number() over(order by salary desc) |

+-----------+-----------+--------+--+

| rohit | sharma | 30000 | 1 |

| shikhar | dhawan | 25000 | 2 |

| shane | warne | 20000 | 3 |

| rahul | dravid | 20000 | 4 |

| saurav | ganguly | 15000 | 5 |

| sachin | tendulkar | 10000 | 6 |

| virat | kohli | 10000 | 7 |

Rank():

select firstname,lastname,salary, rank() over(order by salary desc) from

employee;

+-----------+-----------+--------+----------------------------------+

| firstname | lastname | salary | rank() over(order by salary desc) |

+-----------+-----------+--------+----------------------------------+

| rohit | sharma | 30000 | 1 |

| shikhar | dhawan | 25000 | 2 |

| shane | warne | 20000 | 3 |

| rahul | dravid | 20000 | 3 |

| saurav | ganguly | 15000 | 5 |

| sachin | tendulkar | 10000 | 6 |

| virat | kohli | 10000 | 6 |

//in rank it will resolve the duplicacy problem and for matching value it keep its raw_number() same.if duplicacy occur then it skip the rank.

Dense Rank():

select firstname,lastname,salary,dense_rank() over(order by salary desc)

from employee;

+-----------+-----------+--------+--+

| firstname | lastname | salary | dense_rank() over(order by salary desc) |

+-----------+-----------+--------+--+

| rohit | sharma | 30000 | 1 |

| shikhar | dhawan | 25000 | 2 |

| shane | warne | 20000 | 3 |

| rahul | dravid | 20000 | 3 |

| saurav | ganguly | 15000 | 4 |

| sachin | tendulkar | 10000 | 5 |

| virat | kohli | 10000 | 5 |

// in dense rank it doesn’t skip any ranks in between if there has any duplicate value.

*If there are no duplicates the row number,rank and dense rank lead to similar results.

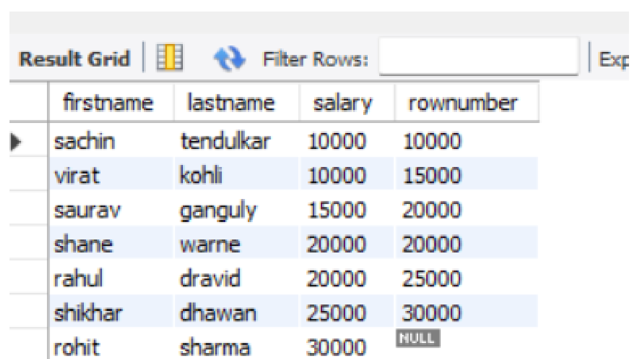
Only difference comes when there are duplicates

Analytic Function:

1.select firstname,lastname,salary, first_value(salary) over(order by salary) as rownumber from employee;

2.select firstname,lastname,salary, last_value(salary) over(order by salary rows between unbounded preceding and unbounded following) as rownumber from employee;-it shows the last value(3000)

3.select firstname,lastname,salary, lead(salary) over(order by salary) as rownumber from employee;// it eliminate the first value(10000) and count from 2nd value of a column and it gives the last value null



Result Grid | Filter Rows: | Exp

firstname	lastname	salary	rownumber
sachin	tendulkar	10000	10000
virat	kohli	10000	15000
saurav	ganguly	15000	20000
shane	warne	20000	20000
rahul	dravid	20000	25000
shikhar	dhawan	25000	30000
rohit	sharma	30000	NULL

4.select firstname,lastname,salary, lag(salary) over(order by salary) as rownumber from employee; // it put the first column as null and count from the first value and eliminate the last value(3000).

Result Grid | Filter Rows:

firstname	lastname	salary	rownumber
sachin	tendulkar	10000	NULL
virat	kohli	10000	10000
saurav	ganguly	15000	10000
shane	warne	20000	15000
rahul	dravid	20000	20000
shikhar	dhawan	25000	20000
rohit	sharma	30000	25000

* **Extract Function**: we use Extract to find out year, month day from a date.

select extract(day from payment_date), payment_date from payment;

select extract(year from payment_date), payment_date from payment;

Common Table Expression(CTE):

A Common Table Expression (CTE) is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement in SQL

We can define CTEs by adding "WITH" clause before select,insert etc statement.

We can write more than one CTE query using "with" clause.

```
WITH cte_name  AS (

        SELECT column1, column2, … //CTE Query

        FROM table_name

        WHERE condition

)

SELECT column1, column2, … // Main Query

FROM cte_name;
```

Example:

1.

```
with niloy as (

select customer_id,payment.mode,amount

from payment where amount >50

)

select customer_id,mode,amount from niloy
```

2.

```
with my_cte as (

select *,avg(p.amount) over(order by p.customer_id) as total, count(c.address_id) over(order by c.customer_id) as total_count

from niloy.`payment-cte` p inner join niloy.`customer-cte` c on c.customer_id=p.customer_id

)

select first_name,last_name from my_cte
```

Date Time Function :

The datetime functions in SQL are a set of functions that are used to manipulate and perform operations on date and time values. These functions are built into SQL and are used to retrieve or manipulate date and time information stored in a database. Here are some common datetime functions in SQL:

1. GETDATE(): Returns the current date and time.

   ```
   SELECT GETDATE();
   ```

2. DATEADD(): Adds a specified number of intervals (such as days, months, or years) to a date.

   ```
   SELECT DATEADD(day, 7, '2022-04-01');
   ```

3. DATEDIFF(): Calculates the difference between two dates.

```
SELECT DATEDIFF(day, '2022-04-01', '2022-04-08');
```

4. DATEPART(): Returns a specific part of a date (such as year, month, or day).

```
SELECT DATEPART(year, '2022-04-01');
```

5. DATENAME(): Returns the name of a specific part of a date (such as month or weekday).

```
SELECT DATENAME(month, '2022-04-01');
```

6. CONVERT(): Converts a date value from one data type to another.

```
SELECT CONVERT(varchar(10), '2022-04-01', 101);
```

These functions are used in SQL queries to perform various operations on date and time values. They can be used to calculate date intervals, format dates, and perform complex calculations involving date and time values. The datetime functions are a powerful tool for working with date and time data in SQL.

Query Optimization:

Query optimization in MySQL is the process of improving the performance of SQL queries by minimizing the time taken to execute the queries and reducing resource consumption. Query optimization involves identifying and improving the queries that take longer to execute, consume more resources, or return a large number of records. Here are some tips for optimizing queries in MySQL:

1. Use indexes: Indexes help MySQL find records faster by creating a lookup table that maps the values in a column to their corresponding records. Use indexes on columns that are frequently searched or sorted.
2. Use EXPLAIN to analyze queries: EXPLAIN is a MySQL command that shows how MySQL executes a query. It can be used to analyze and optimize queries by identifying the slow parts of the query and suggesting ways to improve the performance.

3. Use LIMIT to limit the number of records returned: LIMIT is a MySQL keyword that can be used to limit the number of records returned by a query. This can improve the performance of queries that return a large number of records.
4. Avoid subqueries and correlated subqueries: Subqueries and correlated subqueries are often slow because they require MySQL to execute multiple queries. Instead of subqueries, use joins or temporary tables to retrieve the required data.
5. Use caching: MySQL has a built-in query cache that can be used to cache the results of queries. This can improve the performance of queries that are frequently executed.
6. Optimize the database structure: The structure of the database can also affect query performance. Normalize the database structure and use appropriate data types to improve the performance of queries.
7. Use stored procedures: Stored procedures can improve query performance by reducing the number of queries executed by the database