**SQL**

* MYSQL workbench
  + Test\_db
    - Tables
      * Employees
        + Columns
        + Indexes
        + Foreign Keys
        + Triggers
    - Views
    - Stored Procedures
    - Functions

**SQL Datatypes**

|  |  |  |
| --- | --- | --- |
| Binary Datatypes | Binary | The maximum length of 8000 bytes(Fixed-Length binary data) |
| varbinary | The maximum length of 8000 bytes(Variable Length binary data) |
| varbinary(max) | The maximum length of 231 byte |
| image | Maximum Length of 2,147,483,647 bytes(Variable Length binary data) |
| Exact Numeric Datatype | BigInt |  |
| Int | -2e31 (-2,147,483,648)  to  2e31 (2,147,483,647) |
| smallint |  |
| tinyint |  |
| bit |  |
| decimal |  |
| numerical |  |
| money |  |
| small money |  |
| Approximate Numeric Datatype | float | -1.79E+308  to  1.79E+308 |
| real | -3.40E+38  to  3.40E+38 |
| Character String Datatype | char | The maximum length of 8000 characters.(Fixed-Length non-Unicode Characters) |
| varchar | The maximum length of 8000 characters.(Variable-Length non-Unicode Characters) |
| text | The maximum length of 2,127,483,647 characters(Variable Length non-Unicode data) |
| date time | date |  |
| time |  |
| datetime |  |
| XML | XML | A Datatype used to store data in the format of XML datatype |
| spatial | Geometry | A datatype is used for storing planar spatial data, such as points, lines, and polygons, in a database table. |
| array | ARRAY |  |
| JSON |  |

SQL Comments

-- this is a comment

/\*this is a comment\*/

SQL Alias

Temporary name, only exists for the duration of the query.

Select c.customer\_id as id, c.customer\_name AS name, o.order\_region

From customers AS c, orders o

**Databases**

A database is used to store the collection of records in an organized form. It allows us to hold the data into tables, rows, columns, and indexes to find the relevant information frequently.

|  |
| --- |
| **CREATE DATABASE** employees\_db;  CREATE DATABASE IF NOT EXISTS employees\_db;  SHOW CREATE DATABASE employee\_db;  **SHOW** DATABASES;  **SHOW DATABASES LIKE "%schema";**  SELECT schema\_name  FROM **information\_schema**.schemata;  **USE** emplyees\_db;  **DROP DATABASE** employees\_db;  DROP DATABASE IF EXISTS employees\_db;  # schema is the synonym for database  DROP SCHEMA employees\_db; |

MySQL copy or clone database is a feature that allows us to create a duplicate copy of an existing database, including the table structure, indexes, constraints, default values, etc. Making a duplicate copy of an original database into a new database is very useful when accidentally our database is lost or failure. The most common use of making a duplicate copy of the database is for data backups. It is also useful when planning the major changes to the structure of the original database. In MySQL, making the clone of an original database is a three-step process:

1. The original database records are dumped (copied) to a temporary file that holds the SQL commands for reinserting the data into the new database.
2. It is required to create a new database.
3. The SQL file is processed, and the data will be copied into the new database.

|  |
| --- |
| **BACKUP DATABASE testDB**  TO DISK = 'D:\backups\testDB.bak'  WITH DIFFERENTIAL;  # only backs up the parts of the database that have changed since the last full database backup. |

**Tables**

Creating table

Needed: - Name of table, name of fields, field constraints

|  |
| --- |
| **CREATE TABLE employee\_table(**  **emp\_id int NOT NULL AUTO\_INCREMENT,**  name varchar(45) NOT NULL,  occupation varchar(35) NOT NULL,  age int NOT NULL,  city varchar(30) DEFAULT ‘Germany’  PRIMARY KEY (emp\_id)  ); |
| SHOW TABLES;  SHOW TABLES FROM mystudentdb;  **SHOW TABLES FROM mystudentdb LIKE "stud%";**  # tables could be either base/view type  SHOW TABLES FROM sakila WHERE table\_type= "VIEW";  DESCRIBE employee\_table; |

Alter table

|  |
| --- |
| ### Add column  **ALTER TABLE cus\_tbl**  **ADD cus\_address varchar(100) NOT NULL**  ADD cus\_salary int(100) NOT NULL  AFTER cus\_name; --Optional  ### To modify existing column’s datatype; in mysql use MODIFY COLUMN  ALTER TABLE Persons  **ALTER COLUMN DateOfBirth year NULL;**  ### to drop column  ALTER TABLE cus\_tbl  **DROP COLUMN cus\_address;**  ### to rename  ALTER TABLE cus\_tbl  **RENAME TO cus\_table;**  ALTER TABLE employees  **RENAME COLUMN old\_name to new\_name;** |

Deleting tables: - Drop/Truncate

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself. Whereas drop table drops the entire table from database.

|  |
| --- |
| **DROP TABLE Shippers;** |
| **TRUNCATE TABLE Shippers;** |

**Records in table: - Insert/Update/Delete/Select**

Insert Into

|  |
| --- |
| **INSERT INTO People (id, name, occupation, age)**  **VALUES (101, 'Peter', 'Engineer', 32);** |
| INSERT INTO People  VALUES  (102, 'Joseph', 'Developer', 30),  (103, 'Mike', 'Leader', 28),  (104, 'Stephen', 'Scientist', 45); |
| INSERT INTO table\_name (column\_name, column\_date)  VALUES ('DATE: Manual Date', '2008-7-04'); |

Select Into / Insert Into Select

Copying data from one table to another.

INSERT INTO SELECT inserts into an existing table. Requires that datatype in source and target tables matches.

SELECT INTO creates a new table and puts the data in it. Uses the schema of old table to create a new table.

|  |
| --- |
| **SELECT \***  **INTO CustomerGermany**  FROM Customers  WHERE Country=’Germany’; |
| **INSERT INTO Customers (CustomerName, City, Country)**  **SELECT** SupplierName, City, Country  FROM Suppliers  WHERE Country=’Germany’; |

Update

The UPDATE statement is used with the SET and WHERE clauses.

Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

|  |
| --- |
| **UPDATE Customers**  **SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'**  **WHERE CustomerID = 1;** |
| UPDATE Trainer\_table  SET email = REPLACE(  email,'@javatpoint.com','@tutorialandexample.com'  )  WHERE course\_name = 'Testing'; |

Delete

This query in MySQL deletes a full row from the table and produces the count of deleted rows. Once we delete the records using this query, we cannot recover it. If we omit the WHERE clause into the statement, this query will remove whole records from the database table.

If we want to delete all records from a table without knowing the count of deleted rows, we must use the TRUNCATE TABLE statement that gives better performance.

|  |
| --- |
| **DELETE FROM Employees**  **WHERE emp\_id=107;**  # to delete only top n records  DELETE FROM Employees  ORDER BY name  LIMIT 3; |
| # to delete records from multiple tables using a single query  DELETE Employees, Payment  FROM Employees  INNER JOIN Payment  ON Employees.emp\_id = Payment.emp\_id  WHERE Employees.emp\_id = 102; |

|  |
| --- |
| Drop – drops entire table  Truncate – deletes all records from table  Delete – deletes mentioned records from table |

**SQL Constraints**

|  |  |
| --- | --- |
| **Not Null** | column cannot have a NULL value |
| **Unique** | all values in a column are different |
| **Primary Key** | (NotNull + Unique) Uniquely identifies each row in a table.  Only one per table.  Can be made up of multiple columns, called **(Composite Key)** |
| **Foreign Key** | Prevents actions that would destroy links between tables |
| **Check** | Ensures that the values in a column satisfies a specific condition.  To limit the value range that can be placed. |
| **Default** | Sets a default value for a column if no value is specified |
| **Create Index** | Used to create and retrieve data from the database very quickly. |

|  |
| --- |
| Create Table employees (  ID int NOT NULL **AUTO\_INCREMENT=1,**  Age int **NOT NULL**,  Year int,  City CHAR(20) **DEFAULT (‘Bombay’)**,  **UNIQUE(Age),**  **CHECK(Age>=18),**  Age int DEFAULT (18),  **PRIMARY KEY (Age, Year),**  **FOREIGN KEY (ID) REFERENCES persons(id),**  **CREATE INDEX**  );  ALTER TABLE employees  **DROP PRIMARY KEY;**  ALTER TABLE employyes  **ADD CHECK(Year>=2011)**  ALTER TABLE employees  **ALTER Year SET DEFAULT 2015;**  **-- ALTER City DROP DEFAULT;** |

**SQL INDEXES**

* Used by the server **to speed up the retrieval of rows by using a pointer (based on index column), rather than having to scan the entire table.**
* Helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements.
* Can be created or dropped with **no effect on the data**.
* Users **can NOT see** the indexes.
* Each table can have **more than one index.**
* We can create an index by using **one or more columns** of the table for efficient access to the records.
* When a table is created with a primary key or unique key, it automatically creates a special index named PRIMARY. We called this index as a **clustered index**. All indexes other than PRIMARY indexes are known as a non-clustered index or secondary index.

When should indexes be created:

* A column contains a **wide range of values**.
* A column does not contain a large number of null values.
* One or more columns are **frequently used columns** together in a where clause or a join condition.

When should indexes be avoided:

* The table is **small table**
* The columns are **NOT often used col** as a condition in the query
* The **column is updated frequently**
* Do not make an index for each query as creating indexes also take storage
* periodically review the indexes in your database and remove any that are no longer needed.
* Updating a table with indexes takes more time than updating a table without.

|  |
| --- |
| -- create table  CREATE TABLE Colleges (  college\_id INT PRIMARY KEY,  college\_code VARCHAR(20) NOT NULL,  college\_name VARCHAR(50)  );  -- create index  **CREATE INDEX college\_index**  **ON Colleges(college\_code);** |
| -- create unique index  CREATE UNIQUE INDEX college\_index  ON Colleges(college\_code); |
| ALTER TABLE Colleges  DROP INDEX college\_index; |

**SQL Clauses**

Where

|  |
| --- |
| SELECT \*  FROM officers  WHERE (address = 'Mau' AND officer\_name = 'Ajeet')  OR (officer\_id < 5); |
| AND  OR  LIKE  RLIKE  (SELECT…) |

Distinct

|  |
| --- |
| SELECT **DISTINCT officer\_name**, address  FROM officers; |

From

|  |
| --- |
| SELECT \*  FROM officers |

Order by

|  |
| --- |
| SELECT officer\_name, address  FROM officers  WHERE officer\_id < 5  **ORDER BY officer\_name DESC, address ASC;** |

Group by

If no aggregated function is used or that aggregate function can be used on a column that is used in select clause, then the top record for that group by object is returned.

|  |
| --- |
| SELECT emp\_name, **SUM(working\_hours) AS "Total working hours"**  FROM employees  **GROUP BY emp\_name;** |
| Aggregate functions available:-   * Count * Sum * Min * Max * Avg |

Having

HAVING Clause is used with GROUP BY clause. It always returns the rows where condition is TRUE. It was added to the SQL **because where condition can not be used with aggregated functions.**

|  |
| --- |
| SELECT emp\_name,  SUM(working\_hours) AS "Total\_working\_hours"  FROM employees  **GROUP BY emp\_name**  **HAVING SUM(working\_hours) > 5;** |

window

* A window function **performs a calculation across a set of table rows that are somehow related to the current row.** This is comparable to the type of calculation that can be done with an aggregate function.
* But unlike regular aggregate functions, use of a window function **does not cause rows to become grouped into a single output row — the rows retain their separate identities.**
* It means window functions perform operations on a set of rows and **produces an aggregated value for each row.**
* Helps constructs **comparison with data from current row WITHOUT LOSING THE GRANULARITY.**

Types of window functions:

|  |
| --- |
| 1. Aggregating Functions   **AVG, SUM, MIN, MAX, COUNT,**  **STRING\_AGG, ARRAY\_AGG**   1. Ranking functions.   **RANK, DENSE\_RANK, PERCENT\_RANK,**  **ROW\_NUMBER**   1. Analytical functions   **LAG, LEAD** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RANK and DENSE\_RANK are used to order values and assign them numbers depending on where they fall in relation to one another. Unlike DENSE\_RANK , RANK skips positions after equal rankings. The number of positions skipped depends on how many rows had an identical ranking. For example, Mary and Lisa sold the same number of products and are both ranked as #2. With RANK , the next position is #4; with DENSE\_RANK , the next position is #3.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | RANK (SKIPS POSITIONS)  25.86 1  23.86 2  **21.86 3**  **21.86 3**  **18.86 5**  18.86 5  17.91 7 | DENSE RANK  25.86 1  23.86 2  **21.86 3**  **21.86 3**  **18.86 4**  18.86 4  17.91 5 | ROW NUMBER  25.86 1  23.86 2  **21.86 3**  **21.86 4**  **18.86 5**  18.86 6  17.91 7 | LEAD (of 2)  1.98 5.94  3.96 0.99  5.94 1.98  0.99 13.86  1.98 **8.91**  13.86 **NULL**  **8.91** **NULL** | LAG (of 1)  **1.98** **NULL**  3.96 **1.98**  5.94 3.96  0.99 5.94  1.98 0.99  13.86 1.98  8.91 13.86 | |

Window functions are executed after all other filter, aggregations, etc. Hence windw functions are NOT allowed in where, having, or group by clause. To filter results based on window function outputs, we can use sub queries/CTEs.

# Clauses:

|  |  |
| --- | --- |
| OVER | Must have clause  Decides how postgres divides and sorts data for window function calculations  With no additional arguments, it gets applied to entire table rows. |
| PARTITION BY | Determines how data is grouped.  If no partitionby clause, then group is the entire table.  If all columns are included then each group is one row. |
| ORDER BY | Determines how data is sorted for windows calculations. |
| WINDOW | In order to reuse the same winodw for multiple calculations.  Enables us to define the reusable window after the from clause. |

|  |  |
| --- | --- |
| SELECT Year, Product,  SUM(Sale) AS Total\_Sales  FROM Sales  GROUP BY Year  ORDER BY Product; | SELECT Year, Product, Sale,  **SUM(Sale) OVER(PARTITION BY Year) AS Total\_Sales**  FROM Sales; |
| MySQL Window Functions | MySQL Window Functions |

|  |
| --- |
| select  surgery\_id,  (discharge\_date - admission\_date) as length\_of\_stay,  **avg(discharge\_date - admission\_date)**  **over() as average\_length\_of\_stay**  from surgical\_encounters; |
| SELECT  account\_id,  acct\_icd,  total\_acct\_balance,  **rank()**  **over(PARTITION BY acct\_icd**  **ORDER BY total\_acct\_balance desc)**  as rank\_by\_accnt\_icd  from accounts; |
| SELECT  s.surgery\_id,  p.full\_name,  s.total\_profit,  **avg(total\_profit)**  **over w as avg\_total\_profit,**  s.total\_cost,  sum(total\_cost) over w as total\_surgeon\_cost  from surgical\_encounters s  left outer join pysicians p  on p.id = s.surgeon\_id  window w as (partition by s.surgery\_id); |
| SELECT  patient\_encounter\_id,  master\_patient\_id,  admission\_date,  discharge\_date,  **lag(discharge\_date) over w as previous\_discharge\_date,**  **lead(admission\_date) over w as next\_admission\_date**  from encounters  window w as (partition by master\_patient\_id  order by admission\_date asc)  order by master\_patient\_id, admission\_date; |
| SELECT Year, Product, Sale,  **LEAD(Sale, 1) OVER(ORDER BY Year) AS Total\_Sales**  FROM Sales;  MySQL Window Functions |
| **select \*,**  **lead(Total, 2) over W as total\_lead**  **from Invoice**  **window W as (partition by BillingCountry**  **order by InvoiceDate ASC);**  **select \*,**  **lead(Total, 2) over (partition by BillingCountry order by total ASC) as total\_lead**  **from Invoice;**  **# lead(Total, 2) over () as total\_lead**  **# lead(Total, 2) over (partition by BillingCountry) as total\_lead** |
| **select InvoiceID, InvoiceDate, BillingCountry, Total,**  **MAX(Total) over W as country\_max,**  **MIN(Total) over W as country\_min,**  **AVG(Total) over W as country\_avg,**  **SUM(Total) over W as country\_total,**  **lag(Total, 1) over W as total\_lag\_2,**  **lead(Total, 2) over W as total\_lead\_2,**  **rank() over w as total\_rank,**  **dense\_rank() over w as total\_dense\_rank,**  **row\_number() over w as total\_row\_num**  **from Invoice**  **window W as (partition by BillingCountry**  **order by InvoiceDate ASC);** |
| **# finding second highest salary**  **select \***  **from (**  **select \*,**  **ROW\_NUMBER() over(order by total desc) AS rnk**  **from Invoice i**  **) a**  **where rnk=2;** |

**Control flow functions**

IF()

The IF function returns a value YES when the given condition evaluates to true and returns a NO value when the condition evaluates to false.

The IF function takes three expressions, where the first expression will be evaluated. If the first expression evaluates to true, not null, and not zero, it returns the second expression. If the result is false, it returns the third expression.

If() functions is different from If statement.

|  |
| --- |
| # Syntax  IF ( expression 1, expression 2, expression 3)  SELECT IF(251 = 251,' Correct','Wrong');  SELECT IF(STRCMP('Rinky Ponting','Yuvraj Singh')=0, 'Correct', 'Wrong'); |
| **SELECT lastname,**  **IF(age>20, "Mature", "Immature") As Result**  **FROM student**  MySQL IF |

IFNULL()

You should avoid the use of the IFNULL() function in the WHERE clause because this function reduces the performance of the query.

|  |
| --- |
| **SELECT IFNULL(NULL,5);**  >>> 5  **SELECT IFNULL("Hello", "javaTpoint");**  >>> Hello |
| # return the **home phone** number if the cell phone is NULL.  MySQL IFNULL  **SELECT**  **contactname,**  **IFNULL(cellphone, homephone) AS phone**  FROM  student\_contact;  MySQL IFNULL |

NULLIF()

If both expressions are the same, it returns NULL. Otherwise, it will return the first expression.

|  |
| --- |
| **SELECT NULLIF("javaTpoint", "javaTpoint");**  >>> NULL  SELECT NULLIF("Hello", "404");  >>> Hello  # helps prevents division by zero  SELECT 1/NULLIF(0,0);  >>> NULL |
| SELECT cust\_name,  occupation,  qualification,  **NULLIF (qualification,"Btech") AS result**  FROM myproductdb.customer;  MySQL NULLIF |

CASE

if-else or if-then-else logic

alongwith SELECT, WHERE, ORDER BY clause, etc.

Once the condition is met, it stops traversing and gives the output. If it will not find any condition true, it executes the else block.

**When the else block is not found, it returns a NULL value.**

|  |
| --- |
| SELECT CASE 1  WHEN 1 THEN 'one'  WHEN 2 THEN 'two'  ELSE 'more'  END; |
| SELECT studentid,  firstname,  CASE  WHEN 'CS' THEN 'Computer Science'  WHEN 'EC' THEN 'Electronics and Communication'  ELSE 'Electrical Engineering'  END AS department  from students;  MySQL CASE Expression |
| SELECT orderid,  quantity,  **CASE**  **WHEN quantity>30 THEN ‘greater than 30’**  **WHEN quantity<30 THEN ‘lesser than 30’**  **ELSE ‘equal to 30'**  **END AS quantity\_text**  from order\_details; |
| select customer\_name, city, country  from customers  order by  (CASE  WHEN city IS NULL THEN country  ELSE city  END); |

If Statement

The IF statement is used in stored procedure programs that implement the basic conditional construct in MySQL. Based on a certain condition, it allows us to execute a set of SQL statements. It returns one of the three values True, False, or NULL.

|  |
| --- |
|  |
|  |
|  |
|  |

**SQL OPERATORS and Conditions**

|  |
| --- |
| **SQL Operators**  **...** |

|  |
| --- |
| AND & OR  WHERE cus\_firstname = 'Ajeet' AND cus\_id > 3;  WHERE cus\_firstname = 'Ajeet' OR cus\_id > 100;  **WHERE (course\_name = 'Java' AND student\_name = 'Aryan')**  **OR (student\_id < 2);** |
| Boolean  ... |
| LIKE & NOT LIKE & RLIKE  Used to find the pattern matching, used in SELECT, INSERT, UPDATE and DELETE statement with the combination of WHERE clause.  Two wildcards (wildcard nothing but representing character):   1. **(%) – represents zero, one or multiple characters** 2. **(\_) – represents single character**   SELECT officer\_name  FROM officers  WHERE address LIKE 'Luck%';   |  |  | | --- | --- | | WHERE country LIKE **‘a%’** | find any values that start with a | | WHERE country LIKE **‘%a’** | find any values that ends with a | | WHERE country LIKE **‘%st%’** | find any values that have ‘st’ in any position | | WHERE country LIKE **‘\_r%’** | find any values that have ‘r’ in second position | | WHERE country LIKE **‘a\_%’** | find any values that start with ‘a’ and are at least 2 chars in length |   WHERE address **NOT** **LIKE** 'Luck%'; |
| IN & NOT IN  SELECT \*  FROM officers  **WHERE officer\_name NOT IN ('Ajeet', 'Vimal', 'Deepika');**  WHERE country NOT IN (‘’, ‘’, ‘’);  SELECT \* FROM customers  **WHERE country IN (SELECT country FROM suppliers);** |
| ANY & ALL  ANY : -   * Returns the Boolean value TRUE if the comparison is TRUE for ANY of the subquery condition * The keyword must follow the comparison operator * works like **comparing the value of a table to each value in the result set** provided by the subquery condition * to make coparison between a single column value and a range of other values   ALL : -   * Returns true when all the subquery values are satisfied by the condition.   # lists the Product Names if it finds ANY records in the Order Details table has Quantity equal to 10  SELECT prod\_name  FROM products  **WHERE prod\_id = ANY (SELECT prod\_id FROM orders WHERE qty=10);** |
| BETWEEN  The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.  The BETWEEN operator is inclusive: **begin and end values are included.**  SELECT \* FROM Products  **WHERE Price BETWEEN 10 AND 20;**  SELECT \* FROM Products  **WHERE ProductName NOT BETWEEN 'Carnarvon’ AND 'Mozzarella’**  ORDER BY ProductName;  SELECT \* FROM Orders  **WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';** |
| EXISTS  The EXISTS operator is used to test for the existence of any record in a subquery.  The EXISTS operator **returns TRUE if the subquery returns one or more records.**  SELECT SupplierName  FROM Suppliers  WHERE EXISTS (  SELECT ProductName  FROM Products  WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20  );  # to find the customers who has placed at least 1 order  SELECT name, occupation FROM customer  WHERE EXISTS (SELECT \*  FROM Orders  WHERE customer.cust\_id = Orders.cust\_id  );  DELETE FROM Orders WHERE EXISTS (  SELECT \* FROM customer  WHERE order\_id=3);  # to check whether a row exists in a table or not  SELECT EXISTS(SELECT \* from customer WHERE cust\_id=104)  AS Result;   |  |  | | --- | --- | | IN | EXISTS | | It is used to minimize the multiple OR conditions in MySQL. | It is used to check the existence of data in a subquery. | | SELECT col\_names FROM tab\_name WHERE col\_name IN (subquery); | SELECT col\_names  FROM tab\_name  WHERE [NOT] EXISTS (subquery); | | It compares all values inside the IN operator. | It stops for further execution as soon as it finds the first true occurrence. | | It can use for comparing NULL values. | It cannot use for comparing NULL values. | | It executes faster when the subquery result is less. | It executes faster when the subquery result is large. | | It performs a comparison between parent query and child query or subquery. | It does not perform a comparison between parent query and child query or subquery. |   . |
| IS NULL & IS NOT NULL  SELECT \*  FROM officers  **WHERE officer\_name IS NULL;**  WHERE officer\_name IS NOT NULL; |
| NOT & NOT EQUAL  SELECT \* FROM customers  **WHERE NOT (country=’india’ or region=’asia’);**  SELECT \* FROM students WHERE city <> "England";  **SELECT \* FROM students WHERE city != "England";** |

**SQL Regex**

Regular expressions allow us to search data matching even more complex ways compared to wildcards.

The advantage of using regular expression is that we are not limited to search for a string based on a fixed pattern with the percent (%) sign and underscore (\_) in the LIKE operator. The regular expression has more meta-characters that allow more flexibility and control while performing pattern matching.

We can use either **RLIKE or REGEXP**, both are same.

For finding the negation of a specific string pattern we can use NOT\_RLIKE also.

|  |  |  |
| --- | --- | --- |
| SELECT column\_lists  FROM table\_name  WHERE field\_name RLIKE 'pattern'; | | |
| ^ | the beginning | # student names starting with a or b  WHERE stud\_name RLIKE '^[ab]' |
| $ | the end | WHERE stud\_name RLIKE 'k$' |
| . | any single character except for a new line. | # name contains exactly six characters  WHERE stud\_name RLIKE '^.{6}$' |
| [abc] | to match any characters enclosed in the square brackets |  |
| [^abc] | any characters not specified in the square brackets. |  |
| \* | matches zero (0) or more instances of the preceding strings. |  |
| + | matches one or more instances of preceding strings. |  |
| {n} | match n instances of the preceding element. |  |
| {m, n} | match m to n instance of the preceding element |  |
| p1|p2 | It is used to isolate alternatives that match any of the patterns p1 or p2. |  |
| ? | The question mark (?) character matches zero (0) or one instance of preceding strings. |  |
| [A-Z] | match any upper case character |  |
| [a-z] | any lower case character |  |
| [0-9] | It is used to match numeric digits from 0 to 9 |  |

Regular expression functions and operators: -

|  |  |
| --- | --- |
| [NOT\_REGEXP](https://www.javatpoint.com/mysql-not-regexp-operator) | It is the negation of a REGEXP operator. |
| [REGEXP](https://www.javatpoint.com/mysql-regexp-operator) | This operator represents whether the string matches regular expression or not. |
| [RLIKE](https://www.javatpoint.com/mysql-rlike-operator) | This operator represents whether the string matches regular expression or not. |
| [REGEXP\_INSTR()](https://www.javatpoint.com/mysql-regexp-instr-function) | It is a function that gives a result when the starting index of substring matches a regular expression. |
| [REGEXP\_LIKE()](https://www.javatpoint.com/mysql-regexp-like-function) | This function represents whether the string matches regular expression or not. |
| [**REGEXP\_REPLACE()**](https://www.javatpoint.com/mysql-regexp-replace-function) | This function replaces the specified string character by matching characters and then returns the resulting string. If any expression, pattern, or replaceable string is not found, it will return NULL.  # Examines the string, looking for two or more dashes. Each occurrence of two or more dashes will be replaced by a single astrik sign.  SELECT  REGEXP\_REPLACE(  'the---web-----development---tutorial--w3',  '( ){2,}',  ' ') AS "trimmed\_name"  FROM DUAL;  >>> the\*web\*development\*tutorial\*w3  # Examines country\_name and puts a space after each non-null character in the string.  SELECT REGEXP\_REPLACE(  country\_name,  '(.)',  '\1 ') AS country\_name\_explanded  FROM countries;  >>> A u s t r a l i a |
| [REGEXP\_SUBSTRING()](https://www.javatpoint.com/mysql-regexp-substr-function) | This function return substring that matches a regular expression. |

**SQL Joins**

Inner

The MySQL Inner Join is used to returns only those results from the tables that match the specified condition and hides other rows and columns. MySQL assumes it as a default Join, so it is optional to use the Inner Join keyword with the query.

Sometimes, the name of the columns is the same in both the tables. In that case, we can use a USING keyword to access the records.

**The resulting table can be grouped by or ordered by.**

|  |
| --- |
| SELECT s.stud\_fname, s.stud\_lname, s.city, t.technology  FROM students s  **INNER JOIN technologies t**  **ON s.student\_id = t.tech\_id**  **GROUP BY inst\_name;** |
| # if name of common column is exactly same  SELECT student\_id, inst\_name, city, technology  FROM students  **INNER JOIN technologies**  **USING (student\_id);** |
| # joining multiple tables  SELECT student\_id, inst\_name, city, technology, cellphone  FROM students  **INNER JOIN technologies USING (student\_id)**  **INNER JOIN contact ORDER BY student\_id;** |

Left

It will return all the records from the first (left-side) table, even no matching records found from the second (right side) table. If it will not find any matches record from the right side table, then returns null.

|  |
| --- |
| SELECT customers.customer\_id, cust\_name, price, date  FROM customers  **LEFT JOIN orders**  **ON customers.customer\_id = orders.customer\_id**  **GROUP BY price**  **HAVING price>200;** |
| SELECT customers.customer\_id, cust\_name, order\_id, price, cellphone  FROM customers  LEFT JOIN contacts  ON customer\_id = contact\_id  LEFT JOIN orders  ON customers.customer\_id = orders.customer\_id ORDER BY income; |

The LEFT JOIN clause is also useful in such a case when we want to get records in the table that does not contain any matching rows of data from another table.

|  |
| --- |
| SELECT customer\_id, cust\_name, cellphone, homephone  FROM customers  LEFT JOIN contacts ON customer\_id = contact\_id  WHERE cellphone IS NULL; |

|  |  |  |
| --- | --- | --- |
| Que. To find non-matching records between two tables   |  |  | | --- | --- | | table\_a  id account  1 Acc1  2 Acc2  3 Acc3 | table\_b  opp\_num account\_id  Opp1 1  Opp2 2  Opp3 4 |   **select account\_id**  **from tableB**  **where account\_id NOT IN table\_a.id;**  SELECT B.Accountid  FROM TableB AS B  LEFT JOIN TableA AS A  ON A.ID = B.Accountid  WHERE A.ID IS NULL; |

Right

Same as left join except...right table is the base table.

Full Outer / Outer

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

|  |
| --- |
| SELECT c.CustomerName, o.OrderID  **FROM Customers c**  **FULL OUTER JOIN Orders o**  **ON c.CustomerID=o.CustomerID**  **ORDER BY c.CustomerName;** |

Cross

CROSS JOIN is used **to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables**. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table.

|  |
| --- |
| 🡪  SELECT \*  FROM customers  CROSS JOIN contacts; |
| SELECT customers.customer\_id, customers.cust\_name, customers.income, orders.order\_id, orders.price  FROM customers c  **CROSS JOIN orders o**  **ON c.customer\_id = o. customer\_id**  **WHERE price>1500 AND price<5000;**  SELECT \* FROM customer  LEFT JOIN(orders CROSS JOIN contacts)  ON customer.customer\_id=contact\_id  ORDER BY income; |

Self

...

Union

### Union Vs Join

They produce a result by combining data from two or more tables. But, the way of combining the data from two or more relations differ in both clauses.

MySQL Union clause allows us to combine two or more relations using multiple SELECT queries into a single result set. By default, it has a feature to remove the duplicate rows from the result set.

* **The order and number of the columns must be the same in all tables.**
* **The data type must be compatible with the corresponding positions of each select query.**
* **The column name in the SELECT queries should be in the same order.**

Join in MySQL is used with SELECT statement to retrieve data from multiple tables. It is performed whenever we need to fetch records from more than one tables. It returns only those records from the tables that match the specified conditions.

|  |  |
| --- | --- |
| UNION | JOIN |
| It is used to combine the result from multiple tables using SQL queries. | It is used to fetch the record from more than one table using SQL queries. |
| **It combines the records into new rows.**  **VERTICALLY**  **[]**  **+**  **[]** | **It combines the records into new columns.**  **HORIZONTALLY**  **[] + []** |
| It allows us to connect the tables vertically. | It will enable us to join the tables vertically. |
| It works as the conjunction of the more than one tables that sum ups all records. | It produces results in the intersection of the tables. |
| In this, the **order and number of the columns must be the same in all tables.** | In this, the order and number of the columns do not need to be the same in all tables. |
| It has a default feature to remove the duplicate rows from the result set. | It does not eliminate the duplicate rows from the result set. |
| In this, **the data type must be the same in all SELECT statements.** | In this, there is no need to be the same data type. It can be different. |
| The Union clause is applicable only when the number of columns and corresponding attributes has the same domain. | The Join clause is applicable only when the two tables that are going to be used have at least one column. |
| The Union clause can have mainly two types that are given below:   * **Union (Removes duplicates)** * **Union All (Doesn’t remove duplicates)** | The Join clause can have different types that are given below:   * Inner Join (Sometimes Join) * Left Join (Left Outer Join) * Right Join (Right Outer Join) * Full Join (Outer Join) |
| A union always combines the result set vertically while the join appends the output horizontally.  MySQL Union | |

### Union Vs Union All

This operator returns all rows by combining two or more results from multiple SELECT queries into a single result set. It does not remove the duplicate rows from the result set.

|  |
| --- |
| # syntax  SELECT column\_name(s) FROM table\_1  UNION  SELECT column\_name(s) FROM table\_2; |
| SELECT stud\_name AS student\_name, subject AS course FROM student1  UNION  SELECT stud\_name, subject FROM student2; |
| **(SELECT stud\_name, subject, marks FROM students)**  **UNION**  **(SELECT stud\_name, subject, marks FROM student2)**  **ORDER BY marks;** |
| SELECT column\_list FROM table1  UNION ALL  SELECT column\_list FROM table2; |
| (SELECT stud\_name, subject, marks FROM students)  UNION ALL  (SELECT stud\_name, subject, marks FROM student2)  ORDER BY marks; |

**SQL Functions**

SQL Coalesce() function

The COALESCE(v1, v2, v3, ...) function in MySQL is used **to return the first non-null value in a specified series of expressions.** If this function evaluates all values of the list are null, or **if it does not find any non-null value, then it returns NULL.**

IFNULL(v1, v2) function takes only two expressions. After doing an evaluation, it returns the first expression if this expression is not NULL; otherwise, it returns the second expression.

|  |
| --- |
| **SELECT COALESCE(NULL, 'Mango', 'Apple', 'Orange', NULL);**  **>>> Mango**  SELECT COALESCE(NULL, 'A', 'B', NULL);  >>> A  # to returns only the mobile number if the employee has all contacts number, else Office no., else Phone number.  SELECT Emp\_id,  Name,  **COALESCE(Mobile, Office, Phone)**  FROM employee; |

Aggregate functions

To perform calculations on multiple values and return the result in a single value.

We mostly use the aggregate functions with SELECT statements.

|  |  |  |
| --- | --- | --- |
| [count()](https://www.javatpoint.com/mysql-count) | It returns the number of rows, including rows with NULL values in a group. | SELECT **COUNT(name)** FROM employee;  SELECT **COUNT(\*)** FROM employee;  SELECT **COUNT(DISTINCT(name))** FROM employee;  # returns the count of the element in each group  SELECT emp\_name, city, COUNT(\*)  FROM employees GROUP BY city;  **SELECT emp\_name, emp\_age, COUNT(\*) FROM employees**  **GROUP BY emp\_age**  **HAVING COUNT(\*)>=2**  **ORDER BY COUNT(\*);** |
| sum() | It returns the total summed values (Non-NULL) in a set. | SELECT SUM(working\_hours) AS "Total working hours" FROM employee;  # to return the total summed value for each group.  SELECT emp\_id, emp\_name, occupation, SUM(working\_hours) AS "Total working hours"  FROM employees  GROUP BY occupation  HAVING SUM(working\_hours)>24; |
| avg() | It returns the average value of an expression. | SELECT AVG(working\_hours) AS "Average working hours" FROM employee;  SELECT emp\_name, occupation,  AVG(DISTINCT working\_hours)  FROM employees  GROUP BY occupation; |
| min() |  | SELECT MIN(working\_hours) AS Minimum\_working\_hours FROM employee; |
| max() |  | SELECT MAX(working\_hours) AS Maximum\_working\_hours FROM employee;  SELECT \* FROM employees  WHERE emp\_age = (  SELECT MAX(emp\_age) FROM employees  ); |
| first() | It returns the first value of an expression. Supported only in MS Access. | SELECT working\_date  FROM employee  **LIMIT 1;** |
| last() | It returns the last value of an expression. Supported only in MS Access. | SELECT working\_hours  FROM employee  **ORDER BY name DESC**  **LIMIT 1;** |
| [group\_concat()](https://www.javatpoint.com/mysql-group_concat-function) | It returns a concatenated string. It returns the concatenated string from multiple rows into a single string. | MySQL Aggregate Functions  SELECT emp\_id,  emp\_fname,  emp\_lname,  dept\_id,  **GROUP\_CONCAT(designation)**  FROM employee  GROUP BY emp\_id;  MySQL Aggregate Functions  # can mention your own separator  GROUP\_CONCAT(DISTINCT designation SEPARATOR '; ') |

**SQL Subquery**

Query within another query.

Subquery is a query that is embedded in WHERE clause of another SQL query.

You can place the Subquery in a number of SQL clauses: WHERE clause, HAVING clause, FROM clause, JOIN clause.

Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.

ORDER BY command cannot be used in a Subquery. GROUPBY command can be used to perform same function as ORDER BY command.

Using from clause

|  |
| --- |
| select \*  **from (**  **select \***  **from patients**  **where dob >= "2000-01-01"**  **order by master\_ids**  **) p**  where p.name ilike '%m'; |

Using where clause

|  |
| --- |
| Just like arithmatic operators subqueries can be used with IN and NOT IN statements. With operators with compare with a single value but with IN statement we compare with a column.  Select NAME, LOCATION, PHONE\_NUMBER  from DATABASE  **WHERE ROLL\_NO IN**  **(**  **SELECT ROLL\_NO**  **from STUDENT**  **where SECTION=’A’**  **);** |
| DELETE FROM Student2  WHERE ROLL\_NO IN ( SELECT ROLL\_NO  FROM Student1  WHERE LOCATION = ’chennai’); |
| **UPDATE Student2**  **SET NAME=’geeks’**  **WHERE LOCATION NOT IN ( SELECT LOCATION**  **FROM Student1**  **WHERE NAME IN (‘Raju’,’Ravi’)**  **);** |
| select \*  from surgical\_encounters  where total\_profit > ALL (  select avg(total\_cost)  from surgical\_encounters  group by diagnosis\_description  ); |
| To see if the subquery returns any result  When subquery returns NULL, result of EXISTS evaluates to True.  select \*  from encounters e  **where EXISTS(**  **select 1**  **from orders o**  **where e.id = o.id**  **);** |

|  |
| --- |
| **select se.\***  **from (**  **select \***  **from surgical\_encounters**  **where surgery\_adimssion\_date**  **between '2016-11-01' and '2016-11-30'**  **) se**  **inner join (**  **select \***  **from patients**  **where dob >= '2000-12-31'**  **) p**  **on p.master\_ids = se.master\_ids;**  ###### Above query can be simplified for understanding as below ####  select se.\*  from sales\_entry se  inner join partners p  on p.ids = se.ids |

Using CTEs – Common Table Expressions

* Used to break down complex queries into simpler one's.
* **CTEs create table that exists only for that query and can be reused multiple times within that single query.**
* As compared to FROM clause options, CTEs can be kept at start of query can be resued later multiple times, making it more readable.
* **WITH table\_name AS (select ...)**
* can be used with create, update, delete, select (CRUD) operations.

|  |
| --- |
| with young\_patients as (  select \*  from patients  where dob >= '2000-01-01'  )  select \*  from young\_patients  where name ilike 'm%'; |
| **with top\_counties as (**  select county, count(\*) as num\_patients  from patients  group by county  having count(\*) > 1500  **),**  **county\_patients as (**  select p.master\_ids, p.county  from patients as p  inner join top\_counties as t  on p.county = t.county  )  select cp.county, count(se.surgery\_id) as num\_surgeries  from surgical\_encounters as se  inner join county\_patients cp  on cp.master\_id = se.master\_id  group by cp.county; |

**Views**

* Its is basically a **stored SELECT query**.
* Has no data of its own
* It **does NOT store the underlying data**.
* The View and table have one main difference that the views are **definitions built on top of other tables** (or views).
* If any changes occur in the underlying table, the same changes reflected in the View also.
* Whereas CTEs create table for that particular query’s runtime, Views can be stored in database separately.

Why create views?

* **Simplify/replace complex queries.**
* **Avoid breakage with schema changes.**
* **Restrict data access.**

|  |
| --- |
| **CREATE VIEW v\_monthly\_surgery\_stats\_by\_deparment AS (**  select  unit\_name,  count(surgery\_id) as num\_surgeries,  sum(total\_cost) as total\_cost,  sum(total\_profit) as total\_profit  from surgical\_encounters  group by departments  order by units  );  select \*  from v\_monthly\_surgery\_stats\_by\_deparment  limit 5; |

Modifying and Deleting views

|  |
| --- |
| SELECT \*  FROM information\_schema.views;  **CREATE OR REPLACE VIEW my\_view AS (**  Select ...  );  **ALTER VIEW if exists** v\_monthly\_surgey\_stats  rename to view\_monthly\_stats;  DROP VIEW v\_monthly\_surgey\_stats;  **DROP VIEW IF EXISTS** v\_monthly\_surgey\_stats; |

By default, view in postgress are updatable. INSERT, UPDATE, DELETE will work when certain conditions are met. Useful for certain users to update certain columns.

View can be **updatable if** it has:

* No window function
* No aggregate function
* No limit
* No groupby/having clause
* No set operations
* Is built on single table.

with check option

Issue is that, users can alter/insert into the underlying table data through the view. So to avoid this we can use with check option where **user can alter/insrto into underlying table only if they have access** to that particular id in the given view.

|  |
| --- |
| create or replace view v\_monthly\_surgeries\_22100005 as  select  deparmnt\_id,  encounter\_id,  patinet\_flag  from encounters  where department\_id = 22100005  **with check option;**  insert into v\_monthly\_surgeries values\_22100005 values  (12346, 5611, 22100006, 'Yes');  -- this query throws error because, id 22100006 is not visible in view v\_monthly\_surgeries\_22100005, so it can not be inserted also into the underlying encounters table |

Materialized view

We know, views don’t store underlying data inside the view. With materialized views, we can store the underlying data in a view. This helps give speed of access along with conveneince of view.

**WITH DATA:** the view is populated with the underlying data

**WITH NO DATA:** the view is populated without the underlying data.

**REFRESH MATERIALIZED VIEW:** to refresh the view from the updated underlying data

|  |
| --- |
| CREATE MATERIALIZED VIEW  v\_high\_frequency\_calls AS (  select prod\_vod\_c  from call2\_vod\_c  group by product\_vod\_c  )  WITH DATA;  # to refresh the view from the updated underlying data  REFRESH MATERIALIZED VIEW  v\_monthly\_surgeries;  DROP materialized view  If exists v\_monthly\_surgeries;  ALTER materialized view v\_monthly\_surgeries  rename to view\_monthly\_stats; |

Recursive Views

...

**Stored Procedures**

A procedure (often called a stored procedure) is **a collection of pre-compiled SQL statements stored inside the database.** It is a subroutine or a subprogram in the regular computing language. A procedure always contains a name, parameter lists, and SQL statements. **We can invoke the procedures by using triggers, other procedures and applications** such as Java, Python, PHP, etc.

First select a database that will store the newly created procedure.

* Stored Procedure **increases the** **performance** of the applications. Once stored procedures are created, they are **compiled and stored in the database.**
* Stored procedure **reduces the traffic between application and database** server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
* Stored procedures are reusable and transparent to any applications.
* A procedure is always secure. **The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.**
* if we overuse many logical applications inside stored procedures, the CPU usage will increase. It is because the database server is not well designed for logical operations.
* Stored procedure's constructs are not designed to develop complex and flexible business logic.
* It is **difficult to debug** stored procedures. Only a few database management systems allow us to debug stored procedures.
* Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess.

|  |
| --- |
| ### MySQL  USE employee\_db;  **CREATE PROCEDURE us\_customers ()**  **BEGIN**  **SELECT customer\_id, first\_name**  **FROM Customers**  **WHERE Country = 'USA';**  **END;**  **CALL us\_customers();**  DROP PROCEDURE us\_customers;  ### SQL Server  CREATE PROCEDURE us\_customers  AS  SELECT customer\_id, first\_name  FROM Customers  WHERE Country = 'USA';  EXEC us\_customers; |
| CREATE PROCEDURE get\_merit\_student ()  BEGIN  SELECT \* FROM student\_info WHERE marks > 70;  SELECT COUNT(stud\_code) AS Total\_Student FROM student\_info;  END; |
| **### Used the IN parameter as 'var1' of integer type to accept a number from users.**  **CREATE PROCEDURE get\_student (IN var1 INT)**  BEGIN  SELECT \* FROM student\_info LIMIT var1;  SELECT COUNT(stud\_code) AS Total\_Student FROM student\_info;  END;  mysql>>> CALL get\_student(4); |
| ###  CREATE PROCEDURE display\_max\_mark (OUT highestmark INT)  BEGIN  SELECT MAX(marks) INTO highestmark FROM student\_info;  END;  mysql>>> CALL display\_max\_mark(@M);  mysql>>> SELECT @M; |
| SHOW PROCEDURE STATUS WHERE db = 'mystudentdb';  DROP PROCEDURE display\_marks; |

**Triggers (a specific class of stored procudeure)**

**A trigger is a stored procedure in a database that AUTOMATICALLY invokes whenever a special event in the database occurs.**

For example, a trigger **can be invoked when a row is inserted** into a specified table or when specific table columns are updated in simple words a trigger is a collection of SQL statements with particular names that are stored in system memory.

It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. **Every trigger has a table attached to it.**

Because a trigger **cannot be called directly, unlike a stored procedure**, it is referred to as a special procedure. A trigger is automatically called whenever a data modification event against a table takes place, which is the main distinction between a trigger and a procedure. On the other hand, a stored procedure must be called directly.

The following are the key differences between triggers and stored procedures:

* **Not Manual** - Triggers cannot be manually invoked or executed.
* **No Parameter**- There is no chance that triggers will receive parameters.
* **No rollback** - A transaction cannot be committed or rolled back inside a trigger.

Ex. we could create a trigger to insert the required data into a different table if the primary table underwent any changes.

Disadvantage of Triggers

* Automatic triggers are used, and the **user is unaware of when they are being executed**. Consequently, it is **difficult to troubleshoot** issues that arise in the database layer.
* The **database server’s overhead** may increase as a result of triggers.
* In a single CREATE TRIGGER statement, we can specify the same trigger action for multiple user actions, such as INSERT and UPDATE.
* Only the current database is available for creating triggers, but they can still make references to objects outside the database.

Advantage of Triggers

* Database **object rules are established** by triggers, which cause changes to be undone if they are not met.
* We can **enforce data integrity** thanks to triggers.
* Data is validated using triggers before being inserted or updated.
* Triggers assist us in maintaining a **records log**.
* Due to the fact that they do not need to be compiled each time they are run, triggers improve the performance of SQL queries.
* The client-side **code is reduced** by triggers, saving time and labor.
* Trigger maintenance is simple.

Types of triggers:

* **DDL trigger**
* **DML trigger**
* **LOGON trigger** - When a user session is created with a SQL Server instance after the authentication process of logging is finished. Used to track login activity or set a limit on the number of sessions that a given login can have in order to audit.

Before Triggers are used to update or validate record values before they’re saved to the database.

After Triggers are used to access field values that are set by the system and to effect changes in other records. The records that activate the after trigger are read-only. We cannot use After trigger if we want to update a record because it will lead to read-only error. Row-level trigger gets executed before or after any column value of a row changes. Column Level Trigger gets executed before or after the specified column changes.

The trigger body can access the column's values, which are affected by the DML statement. The NEW and OLD modifiers are used to distinguish the column values BEFORE and AFTER the execution of the DML statement. We can use the column name with NEW and OLD modifiers as OLD.col\_name and NEW.col\_name. The OLD.column\_name indicates the column of an existing row before the updation or deletion occurs. NEW.col\_name indicates the column of a new row that will be inserted or an existing row after it is updated.

|  |  |  |
| --- | --- | --- |
| Trigger Event | OLD | NEW |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| DELETE | Yes | No |

|  |
| --- |
| # SYNTAX  **CREATE TRIGGER** Trigger\_Name  (Before | After) [ Insert | Update | Delete]  **ON** [Table\_Name]  [ for each row | for each column ]  **BEGIN**  [ trigger\_body ]  **END;** |
| DROP TRIGGER trigger name;  DROP TRIGGER IF EXISTS employeedb.before\_update\_salaries; |
| SHOW TRIGGERS;  SHOW TRIGGERS  IN database\_name;  SHOW TRIGGERS  FROM mysqltestdb  WHERE table = 'employee'; |
| # calculate percentage of the student as soon as his details are updated to the database.  CREATE TRIGGER sample\_trigger  before INSERT  ON student  FOR EACH ROW  SET new.total = new.marks/6; |
| # Automatically add 100 to marks column.  CREATE TRIGGER calculate\_total  before INSERT  ON student  FOR EACH ROW  SET new.marks = new.marks+100; |
| CREATE TRIGGER total\_mark  after insert  ON student  FOR EACH ROW  insert into Final\_mark values(new.marks); |
| # automatically insert the **working\_hrs = 0** if someone tries to insert **working\_hrs < 0**.  Create Trigger before\_insert\_empworkinghours  BEFORE INSERT  ON employee  FOR EACH ROW  BEGIN  IF NEW.working\_hours < 0 THEN SET NEW.working\_hours = 0;  END IF  END; |
| # automatically insert new record in another table also  Create Trigger after\_insert\_details  AFTER INSERT  ON student\_info  FOR EACH ROW  BEGIN  INSERT INTO student\_detail VALUES (new.stud\_id, new.stud\_code,  new.stud\_name, new.subject, new.marks, new.phone, CURTIME());  END; |
| # create the backup of records before it is deleted  CREATE TRIGGER before\_delete\_salaries  BEFORE DELETE  ON salaries  FOR EACH ROW  BEGIN  INSERT INTO salary\_archives (emp\_num, valid\_from, amount)  VALUES(OLD. emp\_num, OLD.valid\_from, OLD.amount);  END; |
| # updates the total salary into the total\_salary\_budget table after a row is deleted from the salaries table.  CREATE TRIGGER after\_delete\_salaries  AFTER DELETE  ON salaries FOR EACH ROW  BEGIN  UPDATE total\_salary\_budget SET total\_budget = total\_budget - old.amount;  END; |

**SQL Injection**

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the **placement of malicious code in SQL statements, via web page input.**

**Normalization in SQL**

Normalization is **a database design technique that reduces data redundancy/repetition** and **eliminates undesirable characteristics** like Insertion, Update and Deletion Anomalies.

Normalization rules **divides larger tables into smaller tables and links them using relationships, ensures data is stored logically.**

Data Integrity: The purpose of Normalisation in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

Data consistent by storing the data in one table and referencing it everywhere else.

Storage optimization although that is not an issue these days because Database storage is cheap.

It is a multi-step process that puts data into tabular form, removes duplicate data, and set up the relationship between tables.

Insertion, Updation, and Deletion Anomalies are very frequent if the database is not normalized.

|  |
| --- |
|  |
| Insertion Anomaly :  **If you have to repeat the same data in every row of data, it's better to keep the data separately and reference that data in each row.** We can keep the branch information separately, and just use the branch\_id in the student table |
| Updation Anomaly :  What if Mr. X leaves the college? or Mr. X is no longer the HOD of the computer science department? In that case, all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is an **Updation anomaly because you need to update all the records in your table just because one piece of information got changed.** |
| Deletion Anomaly :  If only a single student is enrolled in a branch, and that student leaves the college, or **for some reason, the entry for the student is deleted, we will lose the branch information too.** So never in DBMS, we should keep two different entities together, which in the above example is Student information and Branch information. |

The inventor of the relational model Edgar Codd proposed the theory of normalization of data with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

* 1NF (First Normal Form)
* 2NF (Second Normal Form)
* 3NF (Third Normal Form)
* BCNF (Boyce-Codd Normal Form)
* 4NF (Fourth Normal Form)
* 5NF (Fifth Normal Form)
* 6NF (Sixth Normal Form)

However, in most practical applications, normalization achieves its best in 3rd Normal Form.

1NF (First Normal Form) Rules

* **Each table cell should contain a single value, and not concatenated strings/values.**
* **Each record needs to be unique.**

|  |  |
| --- | --- |
| Database Normalization With Example | Example of 1NF in DBMS |

Solution: -

1. Remove the movies\_rented column from the table and keep it in some other table. OR
2. Add multiple rows for the names and each row is linked with one rented movie.

2NF

A primary key is a single column value used to identify a database record uniquely. A composite key is a primary key composed of multiple columns used to identify a record uniquely.

* Rule 1- Be in 1NF
* Rule 2- **It should not have partial dependency. Single Column Primary Key that does not functionally dependant on any subset of candidate key relation.**

|  |
| --- |
| 2NF Example  2NF Example in DBMS |

You will only be able to insert values into your foreign key that exist in the unique key in the parent table. This helps in referential integrity.

|  |
| --- |
| Now in the above table, the primary key is student\_id + subject\_id, because both these information are required to select any row of data.  But in the Score table, we have a column teacher\_name, which depends on the subject information or just the subject\_id, so we should not keep that information in the Score table.  The column teacher\_name should be in the Subjects table. And then the entire system will be Normalized as per the Second Normal Form. |

3NF

**A transitive functional dependency is when changing a non-key column, might cause any of the other non-key columns to change.** Ex. A value from full name (Robert) if changed to (Lia) might cause value in column salutation to change from (Mr.) to (Mrs.).

* Rule 1- Be in 2NF
* Rule 2- **Has no transitive functional dependencies**

|  |
| --- |
|  |

BCNF

Boyce and Codd Normal Form is a higher version of the Third Normal Form.

Transaction

...

Grouping Sets

...

Table Inheritence

...