SQL Bootcamp:-

Some Basics:

In a database -> In a table each row is called as record and each column is known as field.

Entity -> The smallest unit that can contain a meaningful set of data. Therefore the rows represent the horizontal entity and columns represent vertical entity. A single row can also be known as entity instance.

Relational algebra allows us to use mathematical logic and create a relational connection between a few tables in a way that allows us to retrieve data efficiently.

SQL is a Declarative programming language (Non Procedural) not like other languages like C, Java etc which follows some procedural orders while coding.

The SQL Optimiser will separate your task into smaller steps and do the magic to give you the desired output.

SQL is not case sensitive.

DDL(Data Definition Language) :

CREATE TABLE object_name (column_name data_type);

The table name can coincide with the database name.

The ALTER statement is used to alter the existing objects. -> ADD, REMOVE, RENAME

ALTER TABLE object_name

ADD COLUMN column_name data_type;

The DROP statement is used to delete the entire table:

DROP TABLE object_name;

RENAME is used to rename the table name:

RENAME TABLE object_name TO new_object_name;

Instead of deleting (or) dropping the entire table, we can just delete its entire data and continue to have the table as an object in the database and this can be achieved by using TRUNCATE statement:

TRUNCATE TABLE object_name;

Once if you TRUNCATE a table, if it contains Auto-increment column then those values will be reset if you insert values in that table in future.

NOTE: Keywords in SQL cannot be variable names.

DML(Data Manipulation Language):

SELECT Statement :

Used to retrieve data from database objects like from tables.

SELECT * FROM object_name;

SELECT column1, column2….. FROM table_name;

SELECT column1, column2…. FROM table_name WHERE condition;

INSERT is used to insert more records into the table.

INSERT INTO object_name (column_name1, column_name2,…..) VALUES ( );

Note: If you are inserting a record with all the column values then no need of using (column_name1, column_name2,……) explicitly just we can use:

INSERT INTO object_name VALUES ( );

[Please remember to type integers as plain numbers while inserting, without using quotes to fasten the execution time but if we mention the integers inside the quotes also it works and we must put the values in the exact order we have listed the column names].

Inserting data into a new Table:-

Ex: Consider the departments table

Aim: To create a new duplicate table for the departments table and copy all the records from the departments table to the duplicate departments table.

Sol: create table departments_dup(

dept_no CHAR(4) NOT NULL,

dept_name VARCHAR(40) NOT NULL);

Now let us insert the values:

insert into departments_dup(dept_no, dept_name)

select * from departments;

UPDATE [Row value can be updated]:

UPDATE object_name

SET column_name=value

WHERE conditions;

DELETE Statement, it is similar to TRUNCATE but here we can specify which data has to be deleted i.e the DELETE statement removes records from a database.

DELETE from table_name

WHERE conditions;

For Example:

DELETE FROM object_name

WHERE purchase_id=1;

Auto increment values are not reset with DELETE statement.

DCL(Data Control Language):

The GRANT and REVOKE Statements:

Allow us to manage the right users have in a database.

GRANT Statement:

Gives or grants certain permission to users.

REVOKE Statement:

The REVOKE clause is used to revoke permissions and privileges of database users.

People who have complete rights to a database like database administrators, they can GRANT and REVOKE access to the users.

GRANT type_of_permission ON database_name.table_name TO “username” @“localhost”;

REVOKE type_of_permission ON database_name.table_name FROM “username”@“localhost”;

Example::

GRANT SELECT ON sales.customers TO “frank”@“localhost”;

REVOKE SELECT ON sales.customers FROM “frank”@“localhost”;

Creating a USER :

CREATE USER “frank”@“localhost” IDENTIFIED BY “pass” [Here the username is frank, server name is localhost[since he is using local machine] and password is pass]

GRANT ALL ON sales.* TO “frank”@“localhost”;

[This allows frank to access all the tables under sales database and has access to do all operations]

TCL(Transaction Control Language):

Not every change you make to the database is saved automatically.

The Commit Statement - Related to INSERT, DELETE and UPDATE.

Suppose you are an administrator and if you made any changes in the database then you have to use COMMIT statement at the end, then only the updated statements can be seen by the other users also.

If you use the statement ROLLBACK then all the updates made will be removed and it will be rolled back.

The COMMIT Statement : saves the transaction in the database. Changes cannot be undone.

The ROLLBACK clause : allows you to take a step back, the last change made will not count, reverts to the last non-committed state[it will refer to the state corresponding to the last time you executed COMMIT] and you cannot restore data to a state corresponding to an earlier commit.

[In MySQL Workbench go to preferences, click on sql editor and uncheck the safe updates to execute COMMIT and ROLLBACK statements].

The current state of the database can be saved using the COMMIT statement.

Ex:Use employees;

```
COMMIT;
Select * from employees where emp_no=10001;
Delete from employees
Where emp_no=10001;
Select * from employees where emp_no=10001;
ROLLBACK; # Now undoing the change that we have made
Select * from employees where emp_no=10001;
```

[Now it has rolled back to the last committed state].

Note: Once if you DROP a table from the database you can't ROLLBACK.

Primary Key :

A column(or a set of columns) whose value exists and is unique for every record in a table is called a Primary Key.

* Each table can have a maximum of only 1 Primary Key.
* In one table you can have 3 or 4 Primary Keys.
* Primary Key may be composed of set of columns.
* Primary keys are the unique identifiers of a table.
* Cannot contain Null values.
* Not all tables we work with should have a Primary Key.

Foreign Key :

Foreign Key identifies the relationships between tables, not the tables themselves.

Unique Key :

Used whenever you would like to specify that you don't want to see duplicate data in a given field.

The main difference between primary key and unique key is : primary key cannot contain NULL values whereas unique key can contain NULL values. There can more than 1 unique key in a table.

Relational Schema :

The term "**schema**" refers to the organisation of data as a blueprint of how the database is constructed (divided into database tables in the case of **relational** databases).

One to Many Relationship:
Ex: one value from the customer_id column the "Customers" table can be found many times in the customer_id column in the "Sales" table.
Similarly One to One and Many to Many Relationships also exists.

CREATE DATABASE [IF NOT EXISTS] database_name;
(It is the best practise to use IF NOT EXISTS)

(OR)

We can also use as : CREATE SCHEMA [IF NOT EXISTS] database_name;
[ ] -> Indicates optional.

Execute the Command : Use database_name;
[This helps us to execute SQL commands in that database]
(OR)
Database_name.table_name can also be used.

DATA TYPES :

String Data Types:-
Data types in SQL: VARCHAR( ) is not a fixed storage type, more responsive and occupies less memory but still CHAR( ) [fixed storage data type] also exists because CHAR( ) is 50% faster than VARCHAR( ) datatype while processing the data.

The maximum size of CHAR( ) is 255 bytes whereas VARCHAR( ) has a maximum size of 65,535 bytes.[Allowed by the SQL]

ENUM(enumerate) : This data type is assigned to columns where we have to provide only certain values for ex : for gender column we can provide data type as ENUM(“M”, “F”), thereby only these two values can be assigned to the values in that specific column.

Signed / unsigned integers :
If the encompassed range includes both positive and negative values then it is known as signed integers.
If the integers are allowed to be only positive then it is known as unsigned integers.

Numeric Data Types[Integer, fixed and floating point data types]: TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT.

Integer data types are signed by default.

Fixed and floating point data types :-
Precision refers to the number of digits in a number for ex: 10.123 has a precision of 5.
Scale refers to the number of digits to the right of the decimal point in a number.
Ex: DECIMAL(5,3), here 5 represents precision and 3 represents scale.

Fixed point data represents exact values ex: DECIMAL(5,3) -> this contains exactly 5 integers.

When only one digit is specified within the parentheses, it will be treated as the precision of the data type.
DECIMAL(7) = DECIMAL(7,0)

Also, DECIMAL = NUMERIC therefore NUMERIC(7,2) = DECIMAL(7,2).
NUMERIC AND DECIMAL are two data types that are available in fixed point data type.

Floating point Data Type:-
Used for approximate values only, for ex: if we mention data type as FLOAT(5,3) and if we assign a value 10.5236789 then it will consider it as 10.524 and does not give any warning symbol instead if we use DECIMAL(5,3) then it will round off and gives a warning symbol.

FLOAT(occupies 4 bytes)[Maximum number of digits = 23] and DOUBLE(8 bytes)[Maximum number of digits = 53] are two different data types that are present in the floating point data type.

The main difference b/w the fixed and floating point type is the way the value is represented in the memory of the computer.

DATE Data Type:
Format is YYYY-MM-DD

DATETIME Data Type:
Format is YYYY-MM-DD HH:MM:SS [even micro seconds can be specified]

TIMESTAMP Data Type:
Used for a well defined, exact point in time.
The **TIMESTAMP** data type is used for values that contain both date and time parts. **TIMESTAMP** has a much lower range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.

It records the moment in time as the number of seconds passed after the 1st January 1970 00:00:00 UTC.

Representing a moment in time as a number allows you to easily obtain difference b/w two TIMESTAMP values.

The difference b/w two TIMESTAMP values is obtained in seconds.

TIMESTAMP is appropriate if you need to handle Time Zones.

BLOB Data Type:
Binary Large Object.
Refers to a file of binary data - data with 1s and 0s.

Consider if you have a column where you have to store photo (.jpg format) then the Data Type of that column can be mentioned as BLOB.
For ex: File formats like .jpg, .doc, .xlsx, .xml, .wav etc

CREATING A TABLE :

CREATE TABLE table_name(column_1 datatype constraint_1, …….);

[Name of the column, datatype of the column and constraints]

We can assign AUTO_INCREMENT constraint that frees you from having to insert all values manually through the insert command at a latter stage.

* It assigns 1 to the first record of the table and automatically increments by 1 for every subsequent row.

CONSTRAINTS :

Constraints are specific rules, or limits, that we define in our tables.

While assigning PRIMARY KEY constraint to a particular column, there are two possible ways in doing this:

Ex: purchase_number INT AUTO_INCREMENT PRIMARY KEY (OR)

purchase_number INT AUTO_INCREMENT

PRIMARY KEY(purchase_number) [This code can be added anywhere inside the ( )]

AUTO_INCREMENT can be applied to only primary key or unique key or an index.

Foreign key points to a column of another table and thus links the two tables.

Ex : FOREIGN KEY(customer_id) REFERENCES customer(customer_id)

Ex : FOREIGN KEY(customer_id) REFERENCES customer(customer_id) ON DELETE

CASCADE.

[Here the customer table is the parent table].

ON DELETE CASCADE:-

If a specific value from the parent's table primary key has been deleted, then all the records from the child table referring to that value will be removed as well.

Ex : Adding a foreign key to already existing table :

ALTER TABLE sales

ADD FOREIGN KEY (customer_id) REFERENCES customer(customer_id) ON DELETE CASCADE;

Ex: Dropping a foreign key column from a table :

ALTER TABLE sales

DROP FOREIGN KEY sales_ibfk_1; [See in the schema section under DDL for the name of the constraint]

[Here it is the name of the constraint]

UNIQUE KEY Constraint:-

Used whenever you would like to specify that you don't want to see duplicate data in a given data field.

Ex: UNIQUE KEY(email_address)

INDEX :-

Unique keys in mysql have the same role as indexes

Index of a table is an organisational unit that helps retrieve data easily.

If we want to remove any UNIQUE KEY from our table then we have to use the following syntax:-

ALTER TABLE table_name

DROP INDEX unique_key_field; [We should not mention DROP UNIQUE for dropping unique key]

Ex: DROP INDEX email_address;

DEFAULT CONSTRAINT :-

Helps us to assign a particular default value to every row of a column.

Ex : In create statement at the end we can mention like this :

number_of_complaints INT DEFAULT 0; (or) "0" (or) '0'

[Ex while any customer registers for the 1st time then this column value will be set to

0 by default]

Ex: ALTER TABLE customers

CHANGE COLUMN number_of_complaints number_of_complaints INT DEFAULT 0;

Here we can change the name of the column also if we want too else repeat the same old name again]

To drop DEFAULT :-

Ex: ALTER TABLE customers

ALTER COLUMN number_of_complaints DROP DEFAULT ;

NOT NULL CONSTRAINT :-

Ex: company_name VARCHAR(255) NOT NULL

To modify the column:

ALTER TABLE companies

MODIFY company_name VARCHAR(255) NULL;

If we want to add a NOT NULL constraint to already exist column in a table:

ALTER TABLE companies

CHANGE COLUMN company_name company_name VARCHAR(255) NOT NULL;

COMMENTS:

/* …… */ (OR) # for a single line comment.

OPERATORS:

AND, OR, =(equal), IN - NOT IN, LIKE - NOT LIKE, BETWEEN, EXISTS - NOT EXISTS, IS NULL - IS NOT NULL, comparison operators etc.

Operator Precedence:

AND > OR -> So, use brackets ( ) if you would like to execute a particular condition first because whichever condition is present inside the ( ) -> brackets that will be executed first.

Ex: Select * from employees where gender="F" and (first_name="Kellie" or first_name="Aruna");

IN Operator:

Ex: Select * from employees where first_name IN (“Cathie”, “Mark”, “Nathan”);

(OR)

Select * from employees where first_name = “Cathie” OR first_name = “Mark” OR first_name = “Nathan”;

But computationally IN Operator is cheaper than OR Operator.

NOT IN Operator is just opposite to IN Operator.

LIKE Operator -> pattern matching operator:

Ex: select * from employees where first_name LIKE (“Mar%”);

Here “%” sign is a substitute for a sequence of characters.

Also “_” sign helps you match a single character.

NOT LIKE is just opposite to LIKE Operator.

BETWEEN Operator:

Helps us to designate the interval to which a given value belongs.

Ex: select * from employees where hire_date between “1990-01-01” and “2000”-“01”-“01”;

NOT BETWEEN Operator is just opposite of BETWEEN Operator.

IS NOT NULL Operator:

Used to extract values that are not null.

Ex: Select column1, column2,….. from table_name where column_name is not null;

IS NULL Operator is just opposite of IS NOT NULL Operator.

Other useful Operators[>=, <=, != etc]:

Ex: select * from employees where first_name <> “Mark”;

(This will select all the columns whose first_name does not contain “Mark”, even != can be used instead of <> operator).

WildCard Characters:

“%”, “_” and “ * “ are known as wildcard characters.

You would need a wildcard character whenever you wished to put “anything” on its place.

SELECT DISTINCT:

Selects all distinct, different data values.

SELECT DISTINCT column1, column2,…. from table_ name;

Ex: Select distinct gender from employees;

Aggregate Functions:

They are applied on multiple rows of a single column of a table and return an output of a single value.

COUNT( ), SUM( ), MIN( ), MAX( ), AVG( ).

Ex: Select COUNT(column_name) as count_column_name from table_name;

[Note: The parentheses after the COUNT( ) must start right after the keyword, not after a whitespace].

Ex: Select count(DISTINCT column_name) from table_name;

Normally the aggregate functions won't consider NULL values but when we mention COUNT(*) here all the values including the NULL values are counted.

ROUND(#, decimal_places) (or) ROUND(#), where # is the numerical value:

Ex: Select ROUND(AVG(salary), 2) from salaries;

IF NULL( ) and COALESCE( ) are among the advanced SQL functions.

They are used when null values are dispersed in your data table and you would like to substitute the null values with another value.

IF NULL(expression_1, expression_2)

Returns the first of the two indicated values if the data value found in the table is not null, and returns the second value if there is a null value.

- prints the returned output in the column of the output.

Ex: select dept_no, IFNULL(dept_name, "Department name not provided")

From departments; # This will print columns dept_no and dept_name.

IFNULL( ) cannot contain more than two arguments in it.

COALESCE(expression_1, expression_2, ……); # In coalesce( ) more than one argument can be sent.

COALESCE( ) will always return a *single* value of the ones we have within parentheses, and this value will be *the first non-null value* of this list, reading the values from left to right.

if COALESCE( ) has two arguments, it will work precisely like IFNULL( ).

IFNULL( ) and COALESCE( ) do not make any changes to the data set. They merely create an output where certain data values appear in place of NULL values.

Ex: Select dept_no, dept_name, coalesce(dept_manager, dept_name, "N/A") as dept_manager from departments_dup;

Here in the 3rd column[Output] if there are no null values in the dept_manager then those values are printed else if there are null values in 3rd column then the values in dept_name column is printed, if the dept_name has null values then N/A is printed.

COALESCE(expression_1): COALESCE( ) can also have only one argument.

Ex: Select dept_no, dept_name, COALESCE(“department manager name”)

As fake_col from departments_dup;

This will print a 3rd column with “department manager name” as values in

All the rows.

Some more Examples:-

Que: Select coalesce(null, null, “Third”) as coalesce_test;

O/p: Third

Que: Select coalesce(“First”, null, “Third”) as coalesce_test;

O/p: First

Que: Select coalesce(“First”, “Second”, “Third”) as coalesce_test;

O/p: First

Que: Select coalesce(null, “Second”, null) as coalesce_test;

O/p: Second

GROUP BY:

When working in SQL, results can be grouped according to a specific field or fields.

GROUP BY must be placed immediately after the WHERE condition, if any, and just before the ORDER BY clause.

In most cases, when you need an aggregate function, you must add a GROUP BY clause in your query too.

Thumb Rule for Professionals:

Note: Always include the field you have grouped your results by the SELECT statement for better visualisation as shown in Ex 2.

Ex 1: Select column_name(s) from table_name where conditions GROUP BY column_name(s) ORDER BY column_name(s);

Using Aliases(As):-

Ex 2: Select salary, count(emp_no) as emps_with_same_salary from salaries where salary>80000 group by salary order by salary;

[Note: Here if we don't use GROUP BY clause then the application throws an error so we have to use GROUP BY clause in the above query].

ORDER BY:

ORDER BY in descending order:

Ex: Select * from employees order by hire_date DESC;

HAVING:

Frequently implemented with GROUP BY.

HAVING is like WHERE but applied to the GROUP BY block.
HAVING is written in between the clauses GROUP BY and ORDER BY.

Ex: Select column_name(s)
From table_name
Where conditions
Group by column_name(s)
Having conditions
Order by column_name(s);

Where (vs) Having:
After HAVING, you can have a condition with an [aggregate function], while WHERE cannot use aggregate functions within its conditions.

Ex: Select first_name, count(first_name) as names_count from employees
Group by first_name having count(first_name)>250 order by first_name;

Example:-

Que: Extract a list of all names that are encountered less than 200 times. Let the data refer to people hired after the "1st of January 1999".

Sol: Select first_name, count(first_name) as names_count from employees
Where hire_date>"1999-01-01"

Group by first_name

Having count(first_name)<200

Order by first_name desc;

Note: Don't mix multiple conditions in the Having clause because it won't work if you mention multiple conditions.

LIMIT:

This is used before the semicolon to show case how many records (or) rows we need. Normally in the MySQL Workbench if we execute any query at-most only 1000 rows are displayed so to increase the limit we can change this in the preferences tab.

Ex: Select * from salaries order by salary desc LIMIT 10;

[Thereby only 10 rows are displayed]

JOINS:

INNER JOIN:

Select t1.column_name(s), t2.column_name(s) from table_1 t1 JOIN table_2 t2 on t1.column_name=t2.column_name;

[Here t1 and t2 are Aliases for the tables table_1 and table_2]

Inner joins will extract only records in which the values in the related columns match. Null values, or values appearing in just one of the two tables and not appearing in the other, are not displayed. Thereby INNER JOIN does not displays NULL values.

Duplicate Records:

Consider if we are having duplicate records that is duplicate rows in our table then to avoid displaying duplicate records GROUP BY clause is used. GROUP BY the field that differs most among records.

Ex: Select m.dept_no, m.emp_no, d.dept_name from dept_manager_dup m JOIN departments_dup d ON m.dept_no=d.dept_no

GROUP BY m.emp_no;

LEFT JOIN (or) LEFT OUTER JOIN:

Retrieves all matching values of the two tables + all values from the left table that match no values from the right table. Thereby, the order in which you join table matters.

RIGHT JOIN (or) RIGHT OUTER JOIN:

Retrieves all matching values of the two tables + all values from the right table that match no values from the left table. Thereby, the order in which you join table matters.

New Syntax using WHERE clause:

Select t1.column_name, t1.column_name,… t2.column_name, t2.column_name, ….. From table_1 t1, table_2 t2 WHERE t1.column_name=t2.column_name;

[Thereby, instead of INNER JOIN WHERE can be used but there are few disadvantages]:-

Using WHERE is more time-consuming.

The WHERE syntax is perceived as morally old and is rarely employed by the professionals.

Whereas the join syntax allows you to modify the connection between tables easily.

JOIN and WHERE clause:

JOIN is used for connecting two tables whereas WHERE clause is used to define a condition while retrieving data from tables.

Thereby JOIN and WHERE clause can be used at a time in a query to join the tables and to mention conditions.

CROSS JOIN:

A CROSS JOIN will take the values from a certain table and connect them with all the values from the tables we want to JOIN it with. CROSS JOIN is just opposite of INNER JOIN because the INNER JOIN connects only the matching values.

CROSS JOIN is the Cartesian product of the values of two or more sets.

Particularly useful when the tables in a Database are not well connected.

While writing a query if we mention only JOIN or INNER JOIN instead of CROSS JOIN, and forget to mention the ON condition then we will obtain a CROSS JOIN result on the data.

JOIN can be applied to more than two tables also.

Some Tips and Tricks in JOIN’s:

While joining multiple tables to obtain the desired output, one should look for key columns which are common between the tables involved in the analysis and are necessary to solve the task at hand. These columns do not need to be foreign or private keys.

UNION (vs) UNION ALL:

UNION ALL:

Used to combine a few Select statements in a single output.

Syntax:

Select N columns FROM table_1 UNION ALL Select N columns FROM table_2;

[Note: We have to select the same number of columns from each table.

These columns should have the same order, and should contain related data types].

UNION:

Select N columns FROM table_1 UNION Select N columns FROM table_2;

Note:

When uniting two identically organised tables:-

UNION displays only distinct values in the output.

UNION is computationally expensive and requires more storage space while executing.

UNION ALL retrieves the duplicates as well.

Consider the following example for better understanding the difference between UNION and UNION ALL:

```
drop table if exists employees_dup;

create table employees_dup(

emp_no int(11),

birth_date date,

first_name varchar(14),

last_name varchar(16),

gender enum("M", "F"),
```

```
hire_date date
);

insert into employees_dup
select * from employees limit 20;
# inserting 20 rows from employees table

select * from employees_dup;
insert into employees_dup values("10001", "1953-09-02", "Georgi",
"Facello", "M", "1986-06-26");
# inserting a duplicate row

UNION ALL:
select e.emp_no, e.first_name, e.last_name, null as dept_no, null as
from_date from
employees_dup e where e.emp_no=10001 UNION ALL
select null as emp_no, null as first_name, null as last_name, m.dept_no,
m.from_date from
dept_manager m;
UNION:
select e.emp_no, e.first_name, e.last_name, null as dept_no, null as
from_date from
employees_dup e where e.emp_no=10001 UNION
select null as emp_no, null as first_name, null as last_name, m.dept_no,
m.from_date from
dept_manager m;

One more interesting Example on UNION:
SELECT * FROM (SELECT e.emp_no, e.first_name,  e.last_name, NULL AS
dept_no, NULL AS from_date FROM
employees e WHERE last_name = 'Denis'
UNION
```

SELECT NULL AS emp_no, NULL AS first_name, NULL AS last_name, dm.dept_no, dm.from_date FROM

dept_manager dm) as a ORDER BY -a.emp_no DESC;

[Here it is -a not just a so, the output is displayed in ascending order].

Subqueries (or) Inner query (or) Nested query:

Queries embedded in a query.

A Subquery should always be placed within parenthesis.

You can have a lot more than one subquery in your outer query.

Process:
1)The SQL engine starts by running the *inner query.*
2) Then it uses its returned output, which is intermediate, to execute the outer query.

IN:
Ex: select e.first_name, e.last_name from employees e where e.emp_no IN (Select dm.emp_no from dept_manager dm);
Selecting employees[Managers] from employees table who are present in the dept_manager table

EXISTS:
Checks whether certain row values are found within a subquery.
-> This check is conducted row by row.
-> It returns a boolean value.

If a row value of a subquery exists -> True -> The corresponding record of the outer query is extracted.

If a row value of a subquery doesn't exists -> False -> No row value from the outer query is extracted.

Ex: select e.first_name, e.last_name from employees e WHERE EXISTS(Select * from dept_manager dm where dm.emp_no=e.emp_no) order by e.emp_no;

EXISTS (vs) IN:

EXISTS tests row values for existence whereas IN searches among values.
EXISTS is quicker in retrieving large amounts of data than using IN operator because IN is faster with smaller datasets.
Another Example:

Question: Select the entire information for all employees whose job title is “Assistant Engineer”.

Solution: Select e.emp_no, e.first_name, e.last_name from employees e JOIN titles t on e.emp_no=t.emp_no where t.title="Assistant Engineer";

(OR)

Select e.emp_no, e.first_name, e.last_name from employees e WHERE EXISTS(Select * from titles t where t.emp_no=e.emp_no and t.title="Assistant Engineer”);

(OR)

Select e.emp_no, e.first_name, e.last_name from employees e WHERE e.emp_no in (Select t.emp_no from titles t where t.title="Assistant Engineer");

(Computationally the 2nd query is cheaper than the 1st and 3rd query)

NOTE: SQL subqueries can also be mentioned inside FROM clause not only after WHERE clause and also see -> Subqueries.sql file for better understanding.

SELF JOIN:

It is applied when a table must join itself.

If you would like to combine rows of a table with other rows of the same table, you need a SELF JOIN.

VIEWS:

A virtual table whose contents are obtained from an existing table or tables, called as base tables.

The VIEW itself does not contain any real data; the data is physically stored in the base table. The VIEW simply shows the data contained in the base table.

Syntax:

CREATE OR REPLACE VIEW view_name AS

Select column_1, column_2,….. FROM table_name;

[REPLACE is not mandatory but if we include that keyword also consider the VIEW with the same name already exists then it is replaced by the new one].

Executing a VIEW:

Select * FROM database_name.view_name;

SQL VIEW act as a dynamic table because it instantly reflects data and structural changes in the base table. These are like temporary virtual data tables retrieving information from base tables. So, INSERT, UPDATE etc operations are not possible on VIEWS.

STORED ROUTINES:

A Stored Routine is an SQL statement, or a set of SQL statements, that can be stored on the database server.

Whenever a user needs to run the query in question, they can call, reference, or invoke the routine.

This routine can bring the desired result multiple times.

There are two types of Stored Routines:

1) Stored Procedures
2) Functions

STORED PROCEDURES:

Syntax:

```
DELIMITER $$
CREATE PROCEDURE procedure_name(param_1, param_2…)
BEGIN
  # Body(Query)
END$$
```

[$$ is a temporary Delimiter]

[Parameters represent certain values that the procedure will use to complete the calculation it is supposed to execute]

A PROCEDURE can be created without parameters too but the parenthesis must always be attached to its name.

```
DELIMITER $$
CREATE PROCEDURE procedure_name(IN param_1 datatype, ….)
BEGIN
```

END$$

DELIMITER ;

[We have to mention whether the parameter is an IN (or) OUT Parameter and also its Data Type]

Dropping a PROCEDURE:

DROP PROCEDURE IF EXISTS procedure_name;

(OR)

DROP PROCEDURE procedure_name;

[Note: When dropping a non parameterised PROCEDURE, we should not write the parenthesis at the end]

Ex:

DELIMITER $$

Create PROCEDURE select_employees( )

BEGIN

Select * from employees limit 1000;

END$$

DELIMITER ;

Invoking a PROCEDURE:

CALL database_name.procedure_name( );

CALL select_employees( );

(OR)

CALL select_employees; # But mentioning parenthesis also is best practise

Stored Procedures with an OUTPUT parameter:

The OUTPUT parameter will represent the variable containing the output value of the operation executed by the query of the Stored Procedure.

Ex:

```
DELIMITER $$
USE employees$$
CREATE PROCEDURE emp_avg_salary_out(IN p_emp_no INTEGER, OUT
p_avg_salary DECIMAL(10,2))
BEGIN
SELECT AVG(s.salary) INTO p_avg_salary
FROM employees e JOIN salaries s
ON e.emp_no=s.emp_no
WHERE e.emp_no=p_emp_no;
END$$
DELIMITER ;
```

Invoking above PROCEDURE:

```
CALL emp_avg_salary_out(11300, @p_avg_salary);
SELECT @p_avg_salary AS AVG_Salary;
```

(OR)

```
SET @avg_salary=0;
CALL emp_avg_salary_out(11300, @avg_salary);
SELECT @avg_salary;
```

[Note: SET is used to assign values to a variable]

USER DEFINED FUNCTIONS:

Syntax:

```
DELIMITER $$
CREATE FUNCTION function_name(parameter_1 datatype,….) RETURNS
datatype
# Here we should not indicate object name only datatype is enough
BEGIN
DECLARE variable_name datatype
……..
RETURN variable_name
```

```
END$$
DELIMITER ;
```

Ex:

```
DELIMITER $$
USE employees$$
CREATE FUNCTION f_emp_avg_salary(p_emp_no INTEGER) RETURNS
DECIMAL(10,2)
DETERMINISTIC # This is added to omit Error Code 1418
BEGIN
DECLARE v_avg_salary DECIMAL(10,2);
SELECT AVG(s.salary) INTO v_avg_salary
FROM employees e JOIN salaries s
ON e.emp_no=s.emp_no
WHERE e.emp_no=p_emp_no;

RETURN v_avg_salary;
END$$
DELIMITER ;
```

Executing a Function:

```
SELECT f_emp_avg_salary(11300);
```

The main differences between User Defined Functions (vs) Stored Procedures are as follows:

- In Functions there are no OUT parameters, but there is a RETURN value.
- We cannot CALL Functions we can just SELECT it[CALL procedure_name, SELECT function_name].
- If you need to obtain more than one value as a result of a calculation, you are better off using a Procedure. If you need to just one value to be returned, then you can use a function.

- INSERT, UPDATE, DELETE operations can be done only using Stored Procedures without any OUT Parameters.

ADVANCED SQL TOPICS:

MySQL Variables:

There are three types of Variables:-

Local, Session and Global variables.

Local variable: a variable that is visible only in the BEGIN and END block in which it was created.

Declare is a keyword that can be used when creating local variables only.

Session: A session is a series of information exchange interactions, or a dialogue, between a computer and a user. A session can contain only one connection.

A session begins at a certain point of time and terminates at another, later point.

There are certain SQL objects that are valid for a specific session only.

Consider if there are 100 people accessing the database at a time so, there would be 100 different sessions and if you have created a session variable, then it is visible only to the created person and once if the connection or session is ended then the variable is lost.

Syntax for creating a session variable:

SET @s_var1=3;

Now if we try to open a new tab and type the following command:

SELECT @s_var1;

[It will display 3 because the root connection is same but if we try to establish a new connection and execute the above command then it prints NULL as value]

Global variable:

Global variables apply to all connections related to a specific server.

You cannot set just any variable as GLOBAL.

Only System variables can be set as GLOBAL variables.

For Ex:

.max_connection( ) - indicated the maximum number of connections to a server that can be established at a certain point in time.

Syntax:

SET GLOBAL var_name=value;

(OR)

SET @@global.var_name=value;

User - defined (vs) System variables:

User - defined variables are created by user manually whereas, variables that are pre-defined on our System- the My SQL server are known as System variables.

[User - defined and System variables can be set as Session variables but with few Limitations like not every GLOBAL variable can be used as a Session variable]

Triggers:

A Trigger **in MySQL** is a set of SQL statements that reside in a system catalog. It is a special type of stored procedure that is invoked automatically in response to an event. Each **trigger** is associated with a table, which is activated on any DML statement such as INSERT, UPDATE, or DELETE.

A Trigger is a MySQL object that can “trigger” a specific action or calculation “before” or “after” an INSERT, UPDATE, or DELETE statement has been executed.

For Syntax and everything else on TRIGGERS please REFER to sql document [Triggers.sql]

INDEXES:

The INDEX of a table functions like the index of a book.

Syntax:

CREATE INDEX index_name

ON table_name(column_1, column_2,…..);

Ex on employees table:

```
USE employees;
SELECT * FROM employees WHERE hire_date>"2000-01-01";

CREATE INDEX i_hire_date ON employees(hire_date);
select * from employees where hire_date>"2000-01-01";
# The output was displayed very quickly due to INDEX
```

Dropping an INDEX on a table:

```
ALTER TABLE employees
DROP INDEX i_hire_date;
```

Composite Indexes:
These are applied on multiple columns, not just a single column.
[Carefully pick the columns that would optimise your search]

Primary and Unique keys in SQL are also known as Indexes.

To display the indexes that are present on a table can be achieved as follows:

```
SHOW INDEX FROM table_name FROM database_name;
```

Disadvantages of using INDEX on a column in a table:

It requires more space and could be redundant.

Thereby for small datasets the costs of having an index might be higher that the benefits. Whereas for large datasets a well optimised index can make a positive impact on the search process.

The CASE Statement:

Ex:

```
SELECT emp_no, first_name, last_name,
CASE
WHEN gender="M" THEN "MALE"
```

ELSE "FEMALE"

END AS Gender

FROM employees;

(OR)

Another Example using IF( ):

SELECT emp_no, first_name, last_name,
IF(gender="M", "MALE", "FEMALE") AS Gender

FROM employees;

Here if the 1st condition is TRUE then "MALE" is displayed else "FEMALE" is displayed.

IF (vs) CASE:
IF can have just one conditional expression whereas, CASE can have multiple conditional expressions.

Some more Useful Examples:

Que 1:

Similar to the exercises done in the lecture, obtain a result set containing the employee number, first name, and last name of all employees with a number higher than 109990. Create a fourth column in the query, indicating whether this employee is also a manager, according to the data provided in the *dept_manager* table, or a regular employee.

Sol 1:

```
SELECT e.emp_no, e.first_name, e.last_name,
CASE
WHEN dm.emp_no IS NOT NULL THEN "Manager"
else "Employee"
END AS is_Manager
FROM employees e LEFT JOIN dept_manager dm
ON e.emp_no=dm.emp_no
```

```
WHERE e.emp_no > 109990;
```

Que 2:

Extract a dataset containing the following information about the managers: employee number, first name, and last name. Add two columns at the end – one showing the difference between the maximum and minimum salary of that employee, and another one saying whether this salary raise was higher than $30,000 or NOT.

Sol 2:

```
SELECT e.emp_no, e.first_name, e.last_name,
MAX(s.salary) - MIN(s.salary) AS Salary_Difference,
CASE
WHEN MAX(s.salary) - MIN(s.salary) > 30000 THEN "Greater than
$30000"
ELSE "Lesser than $30000"
END AS Salary_Raise
FROM employees e JOIN dept_manager dm ON
e.emp_no=dm.emp_no
JOIN salaries s ON dm.emp_no=s.emp_no GROUP BY s.emp_no;
```

Que 3:

Extract the employee number, first name, and last name of the first 100 employees, and add a fourth column, called “current_employee” saying “Is still employed” if the employee is still working in the company, or “Not an employee anymore” if they aren’t.

Hint: You’ll need to use data from both the ‘employees’ and the ‘dept_emp’ table to solve this exercise.

Sol 3:

```
SELECT e.emp_no, e.first_name, e.last_name,

CASE
```

```
WHEN MAX(de.to_date) > date_format(sysdate(), '%Y-%m-%d')
THEN

"Is still employed" ELSE

"Not an employee anymore"

END AS current_employee

FROM employees e JOIN dept_emp de on e.emp_no=de.emp_no
GROUP BY de.emp_no LIMIT 100;



################## THE END ###################################
```