SQL – Structured Query Language

SQL was developed by IBM back in 1970s. Originally, it was called as **SEQUEL – Structured English Query Language** but they changed acronym as **SQL – Structured Query Language** because **SEQUEL** was a trademark of an Airplane company. So, both **SQL** and **SEQUEL** are right way to pronouncing it. Mostly English speaking people pronounce it as **SEQUEL** because it is shorter and in non-English speaking country it is called as **SQL**. SQL is a relational database management system, the data are stored in a tabular format. Columns of the table are called Variable and Rows of the table are called records.

1. Selecting from Single Table

Select Everything from table

SELECT *

FROM table_name;

Select some columns from table

SELECT column_1, column_2

FROM table_name;

Info: we can rename the column name using **AS** keyword.

SELECT amount * 10 AS result

FROM table_name;

Note: *SQL is not case sensitive*, you can write `select * from table_name` as well but usually all the SQL commands are written in UPPERCASE and all the variables & values are written in lowercase.

Note: You don't need to terminate the syntax with semicolon ';' but when you're writing multiple syntax, it will through an error, so make an habit of terminating the syntax with semicolon.

```
# Select the unique/distinct values, remove duplicates
SELECT DISTINCT column_name
FROM table_name;
```

Example: Return all the products name, unit price and new price (unit price * 1.1)

```
SELECT name, unit_price, unit_price * 1.1 AS new_price
FROM products;
```

1.1 WHERE clause (Conditional Selection)

WHERE clause is used to filter data

```
SELECT *
FROM table
WHERE age > 30;
```

AND, OR, NOT operator

```
SELECT *
FROM table
WHERE age > 30 AND weight < 50;
```

Info: We can add multiple AND OR selector but we should add parenthesis so that the order of the operation will not change.

```
SELECT *
FROM table
WHERE (age > 30 AND weight < 50) OR height > 5;

SELECT *
FROM table
WHERE NOT (age > 30 AND weight < 50);
```

IN Operator

```
SELECT *
FROM table
WHERE age = 5 OR age = 10 OR age = 15;
```

We can simplify above syntax using IN operator.

```
SELECT *
FROM table
WHERE age IN (5, 10, 15);
```

To negate the above expression, we can use NOT operator

```
SELECT *
FROM table
WHERE age NOT IN (5, 10, 15);
```

BETWEEN Operator

```
SELECT *
```

FROM table

WHERE age >= 20 AND age <= 30;

Alternatively,

SELECT *

FROM table

WHERE age BETWEEN 20 AND 30;

Note: The range values in BETWEEN operators are inclusive.

LIKE Operator

Select rows matching specific string patterns

SELECT *

FROM table

WHERE name LIKE 'D_ira_';

Info: % denotes any number of characters, _ denotes exactly one characters. To add more characters using _ we add underscore eg. _ _.

'D%' -- String starts with D

'%L' -- String ends with L

'%h%' -- String contains h

'D_ _ _ _ _' -- String starts with D and have exactly 6 characters long.

REGEXP Operator – Alternative of old LIKE operator and powerful as well

Select the rows containing 'ra' in their name

SELECT *

FROM table

WHERE name REGEXP ‘ra’;

Info: ^ carat sign is used for beginning of string eg. ‘^D’ – Name starts with ‘D’

$ Dollar sign is used for ending of string eg ‘j$’ – Name ends with ‘J’,

To use OR operator, we can use pipe ‘|’ operator eg. ‘^dh | aj$ | a’ name starts with ‘dh’ or ends with ‘aj’ or contain ‘a’

‘i[ts]’ – string contains either ‘it’ or ‘is’

‘[dhi | bi]raj’ – String contains either ‘dhiraj’ or ‘biraj’

‘[a-z]’ – range of character from a to z

IS NULL Operator

To select the null or missing values.

SELECT *

FROM table

WHERE age IS NULL;

To exclude the missing values,

SELECT *

FROM table

WHERE age IS NOT NULL;

Order By – Order the results in Ascending or Descending

Primarily the table is sorted by ID or Primary Key by default, however SQL provides a method to sort the table using specific columns.

Sort the table by column_1 in ascending order

```
SELECT *
FROM table
ORDER BY column_1;
 # Sort the table by column_1 in descending order,
SELECT *
FROM table
ORDER BY column_1 DESC;
```

Info:- We can sort the table by multiple columns as well as alias…

```
SELECT *
FROM table
ORDER BY age, height;

SELECT  *, unit_price * quantity as total
FROM products
ORDER BY total

LIMIT Clause to limit the results
SELECT *
FROM table
LIMIT 5

# Using Offset -- LIMIT offset, total_records
# Below code will generate the records starting from 7 having 4 records (7, 8, 9, 10)
SELECT                                                                                *
FROM table
```

LIMIT 6, 4

Note: Order of the SQL clauses

1 – SELECT

2 – FROM

3 – JOIN (optional)

4 – WHERE (optional)

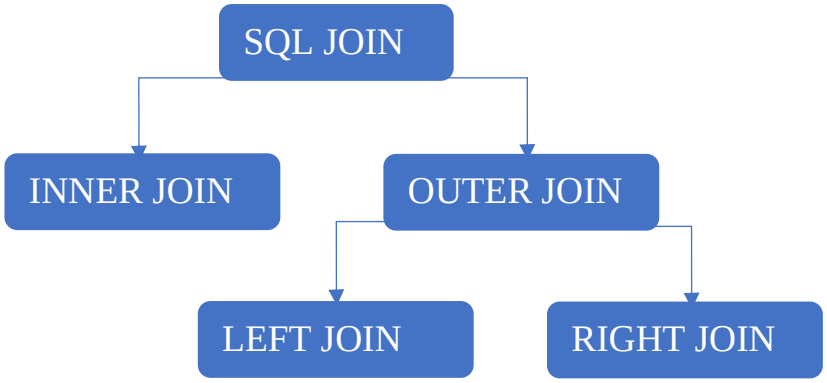
5 – GROUP BY (optional)

6 – ORDER BY (optional)

7 – HAVING (optional)

8 – LIMIT (optional)

SQL JOIN



SQL JOIN			
INNER JOIN		OUTER JOIN	
		LEFT JOIN	RIGHT JOIN

1. INNER JOIN

SQL JOIN are useful when we have to select records from multiple tables, following expression shows how to access data from two tables

```
SELECT *
FROM table_1
JOIN table_2                    # We can write INNER JOIN instead of JOIN as well,
ON table_1.id = table_2.id;     # but it's not necessary


# Selecting from multiple tables, - more than two tables
SELECT *
FROM table_1
JOIN table_2
```

```
ON table_1.id = table_2.id
JOIN table_3
ON table_1.id = table_3.id;
```

Info: While selecting the specific columns in SQL JOIN, we have to prefix the columns with table name otherwise it will through an error. We can exclude the prefix if the column name is existed in only one table and if both tables have the same column name prefix is required.

```
# Select the records from customers and orders table having column name customer_id, customer_name in both tables.
SELECT c.customer_id, o.customer_name
FROM customers c
JOIN orders o
    ON c.customer_id = o.customer_id;
```

Joining Tables across multiple databases.

```
# Let's join the table customers from user database and orders from product database
# Let's use the user database as our current database
USE user;
SELECT *
FROM customers c
JOIN product.orders po
    ON c.customer_id = po.customer_id;
```

SELF JOIN

Joining the table itself is called Self Join. Self-Join is used to select the relationship.

```
# Find out the manager of each employee from employees table
```

```
SELECT e.employee_name AS employee, m.employee_name AS manager
FROM employees e
JOIN employees m
    ON m.employee_id = e.report_to;
```

Compound JOIN – Joining using multiple id/columns

Use to Join the table having composite primary key (i.e. primary key made up with two columns, for example product_id and order_id)

```
SELECT *
FROM orders_items oi
JOIN order_items_notes oin
    ON oi.order_id = oin.order_id
    AND oi.product_id = oin.product_id;
```

Implicit JOIN – Joining Table Without Explicitly Defining the Join Condition

```
# Let's change the following explicit syntax to implicit
SELECT *
FROM table_1 t1
JOIN table_2 t2
ON t1.id = t2.id;
# Let's write the expression for the implicit join
SELECT *
FROM table_1, table_2
WHERE t1.id = t2.id;
```

Warning: Even the MySQL supports implicit join, don't use it because if you accidently forget to write the Where clause, you'll get **CROSS JOIN**.

OUTER JOIN

Outer Join (Left and Right Join) used to select the records even if the condition doesn't match. For example, if you want to select all the records from one table even if the second table doesn't have value for it i.e. NULL value, we'll use Outer Join.

Let's write an Inner Join and Convert it to Outer

Select all the customer from customers table even if they don't have placed any order.

```
SELECT *
FROM customers c
JOIN orders o
    ON c.customer_id = o.customer_id;
```

Outer Join of Above syntax is

```
SELECT *
FROM customers c
LEFT JOIN orders o
ON c.customer_id = o.customer_id;
```

Info: You can write **LEFT OUTER JOIN** instead of **LEFT JOIN** but not necessary.

OUTER JOIN BETWEEN MULTIPLE TABLES

Join multiple tables using Outer join, we can use inner join as well or mix of Left and Right Join.

```
SELECT c.customer_id, c.first_name, o.order_id
FROM customers c
LEFT JOIN orders o
```

```
ON c.customer_id = o.customer_id
LEFT JOIN shippers sh
ON o.shipper_id = sh.shipper_id
ORDER BY c.customer_id;
```

SELF OUTER JOIN

```
# Write a SQL Query to find out the managers from employee table
SELECT e.employee_id, e.first_name, m.reports_to AS manager
FROM employee e
LEFT JOIN employee m
ON e.reports_to = m.employee_id;
```

USING CLAUSE

Using clause is used to select the records when both tables have same column name.

```
SELECT *
FROM customer c
JOIN orders o
ON c.customer_id = o.customer_id;

# Let's simplify the above expression using USING clause.
SELECT *
FROM customer c
JOIN orders o
USING (customer_id);
```

NATURAL JOIN

In natural join we don’t specify the column which is used for join, instead the DBMS will automatically pick the column to join on.

```
SELECT *
FROM orders o
NATURAL JOIN customers c;
```

CROSS JOIN

Used to join every record of first table with every record of second table, so we don’t specify the condition here.

```
# Bellow is the explicit syntax of cross join
SELECT *
FROM customers c
CROSS JOIN products p

# For implicit syntax, use follows
SELECT c.name, p.name
FROM customers c, products p
```

UNION and UNION ALL

Use to join the Row/records instead of column, so it must have the equal number of column selection.

```
SELECT *
FROM orders
WHERE order_date >= ‘2022-03-05’
UNION
```

```
SELECT *
FROM orders
WHERE order_date < ‘2022-03-05’;
```

Info:- The column names of the resulted table is similar with first table column name.

Let’s use UNION ALL to select all duplicates data from the tables.

INSERT, UPDATE, DELETE

Char Vs VarChar

If char(50) & varchar(50) but your string is only 5 character long, CHAR will add remaining 45 spaces to fill the column which is waste of computer space/memory, but VARCHAR will take only 5 characters.

INSERTING ROWS

While inserting the records/row in database table, we use INSERT INTO keyword, and one important thing is we have to insert/write the values serially as table columns.

INSERT INTO table_name VALUES (list of values);

If we want to insert the values randomly, we can specify the column name and insert the values according to column name.

INSERT INTO table_name (name of the columns) VALUES (list of values);

Inserting Multiple Rows,

INSERT INTO table_name (name of the columns) VALUES (list of values), (another list of values), (another list of values);

INSERTING INTO MULTIPLE TABLES

INSERT INTO table_1 (column name) VALUES (list of values)

When two tables are linked and the primary key of the second table is in table 1, we can get the updated/new id as

INSERT INTO table_2 (column name) VALUES (LAST_INSERT_ID(), other values)

COPYING DATA FROM ONE TABLE TO ANOTHER

CREATE TABLE new_table AS SELECT * FROM old_table;

Use SELECT statement as a Sub-query of INSERT statement.

INSERT INTO table_1

SELECT *

FROM orders

WHERE column < “something”,

Here we don’t have to specify the column name and Values because the sub query already have values for every column.

Updating Single Row

Update the single record of the table using Update and Set clause.

UPDATE invoices

SET payment_total = 80, payment_date = “2019-03-01”

Where invoive_id = 1;

Updating Multiple Records

By Default, MySQL runs on safe mode which will not let you update multiple records at the same time so we have to change the workbench setting before updating.

Update the multiple records associated with client_id of invoices table.

UPDATE invoices

SET payment_total = invoice_total * 0.5,

    Payment_date = due_date

```
WHERE client_id = 3;


## Also, we can select multiple conditions as well,
UPDATE invoices
SET payment_total = invoice_total * 0.5,
    Payment_date = due_date
WHERE client_id IN (3,5);
```

Update Using Sub-Queries

We use this syntax to update the records where we have to

```
UPDATE invoices
SET   payment_total = invoice_total * 1.1
      Payment_date = due_date
WHERE client_id = (
            SELECT client_id
            FROM clients
            WHERE name = “Dhiraj”);
```

Info: If the sub-query returns the multiple client_id, we have to modify the expression as follows.

```
UPDATE invoices
SET   payment_total = invoice_total * 1.1
      Payment_date = due_date
WHERE client_id IN
            (SELECT client_id
```

```
    FROM clients
    WHERE name IN (“Dhiraj”, “Dyroz”);
```

Deleting Records

To delete the record, we use DELETE keyword in our SQL expression.

```
DELETE FROM invoices
WHERE customer_id = 1;
```

Warning: If we forget to write the WHERE clause or condition, all of the records from our table will be deleted. So, while deleting the records from table, be cautious and never execute the following.

```
DELETE FROM invoices;
```

```
# To delete the multiple records, you can use the Sub-Query as above
DELETE FROM invoices
WHERE customer_id IN (
    SELECT customer_id
    FROM customers
    WHERE name IN (‘dhiraj’, ‘dyroz’);
```

Summarizing The Data

SQL provides the built-in functions to summarize the data of the table, for example aggregate function, group by function etc.

Aggregate Functions

Some of the aggregate functions are

MAX() – To Find the maximum from the respective columns/records of table
MIN()
SUM()
AVG()

COUNT()

```
SELECT MAX(amount_total) AS highest,

MIN(amount_total) AS lowest,

AVG(amount_total) AS average,

SUM(amount_total) AS total,

COUNT(amount_total) AS number_of_product

FROM products;
```

GROUP BY

Groups the records which has similar conditional parameters, for example group by gender, nationality etc

```
SELECT *

FROM customers

WHERE age BETWEEN 19 AND 70
```

```
GROUP BY gender
ORDER BY age;
```

HAVING Clause

Having keyword works as WHERE clause but after aggregating the function we can use HAVING.

Info: We cannot select the column if that column is not in the select clause in Having expression, but in Where clause we don't need to explicitly give the column name.

```
SELECT SUM(unit_price) AS total
FROM products
WHERE product_price > 10
GROUP BY customer_id
HAVING total >1000;

# Get the customers, located in Virginia, who have spent more than $100
# USE the database sql_store, where you find all the tables
USE sql_store;

SELECT c.customer_id,
    c.first_name,
    c.last_name,
    SUM(oi.quantity * oi.unit_price) AS total_sales
FROM customers c
JOIN orders o USING (customer_id)
JOIN order_items oi USING (order_id)
```

```
WHERE state = 'VA'
GROUP BY c.customer_id,
    c.first_name,
    c.last_name
HAVING total_sales > 100;
```

ROLLUP Clause (Only Available in MySQL)

RollUp clause generate the extra row summarizing the data.

```
SELECT client_id,
    SUM(invoice_total) AS total_sales
FROM invoices
GROUP BY client_id WITH ROLLUP
```

Above expression will generate an extra row/record at the end of the records summing up all the total_sales and leaving the client_id null.

ROLLUP only assigns value in Aggregate columns.

Warning: When you're using ROLLUP method, you cannot use alias, you have to use actual column name.

Writing Complex Query

Complex SQL queries are generated adding multiple sub-queries.

Example: In sql_hr database, find employees who earn more than average.

```
USE sql_hr;


SELECT *
FROM employees
WHERE salary > (
	SELECT AVG(salary)
	FROM employees
);
```

Write Sub-Queries using IN operator

```
# Find the products that have never been ordered.
USE sql_store;
SELECT *
FROM products
WHERE product_id NOT IN (
	SELECT DISTINCT product_id
	FROM order_items
);


# Find clients without invoices
SELECT *
FROM clients
WHERE client_id NOT IN (
```

```
    SELECT DISTINCT client_id
    FROM invoices
);
```

Subqueries Vs Joins

We can write above expression using Join

```
SELECT *
FROM clients
LEFT JOIN invoices USING (client_id)
WHERE invoice_id IS NULL;
```

Info: We can use both Sub-Queries and Join to solve the problems. To select the best query is to see the readability and performance time.

Example: Find customers who have ordered lettuce (id = 3), Select customer_id, first_name, last_name

```
SELECT DISTINCT customer_id, first_name, last_name
FROM customers
WHERE customer_id IN (
    SELECT o.customer_id
    FROM order_items oi
    JOIN orders o USING (order_id)
    WHERE product_id = 3
)
```

ALL KeyWord

When we get multiple values while applying sub-queries, we use ALL keyword to select and apply it to the query parameters.

```
# Select invoices larger than all invoices of client 3
SELECT *
FROM invoices
WHERE invoice_total > ALL (
	SELECT invoice_total
	FROM invoices
	WHERE client_id = 3
);


# Alternately we can re-write the above expression using MAX functions
SELECT *
FROM invoices
WHERE invoice_total > (
	SELECT MAX(invoice_total)
	FROM invoices
	WHERE client_id = 3
);
```

ANY Keyword/ SOME Keyword

We use ANY/SOME keyword to select the value at least one.

```
# Select clients with at least two invoices
SELECT *
FROM clients
WHERE client_id = ANY (
```

```
SELECT client_id
FROM invoices
GROUP BY client_id
HAVING COUNT(*) >= 2
);
```

Info: In above expression we can use IN operator as well in the place of ANY, both are equivalent so you can use whichever you prefer.

Correlated Sub-Queries

The subquery is correlated with outer query so we call it correlated query, It has to do lots of calculation so, correlated query is a bit slow than other query, but it is very powerful and has real life implementations.

```
# Select employees whose salary is above the average in their office
USE sql_hr;

SELECT *
FROM employees e
WHERE salaray > (
        SELECT AVG(salary)
        FROM employees
        WHERE office_id = e.office_id
);

# Get invoices that are larger than the client's average invoice amount
USE sql_invoicing;
```

```
SELECT *
FROM invoices i
WHERE invoice_total > (
SELECT AVG(invoice_total)
FROM invoices
WHERE client_id = i.client_id
);
```

EXISTS Operator – We use IN clause

In fact, we can use **IN** operator in place of **EXISTS** operator, but when we have the list of thousands of values, the execution becomes slower because it returns the results of list having thousands or millions of values. In this case we can simplify and speedup our execution using EXISTS operator, which doesn't return values, only checks if it is exists or not.

```
# Select clients that have an invoice
SELECT *
FROM clients
WHERE client_id IN (
    SELECT DISTINCT client_id
    FROM invoices
);

# Same example using INNER JOIN
SELECT *
FROM clients
JOIN invoices USING (client_id)
```

```
# Alternatively, we can use EXISTS operator using Correlated subqueries.
SELECT *
FROM clients c
WHERE EXISTS (
	SELECT client_id
	FROM invoices
	WHERE client_id = c.client_id
);
```

Example: Find the products that have never been ordered

```
USE sql_store;

SELECT *
FROM products
WHERE product_id NOT IN (
	SELECT product_id
	FROM order_items
);

# Now using ESIXTS operator
SELECT *
FROM products p
WHERE NOT EXISTS (
	SELECT product_id
```

```
    FROM order_items
    WHERE product_id = p.product_id
);
```

Sub-Queries in SELECT clause

We can use sub-queries in select clause as well.

```
SELECT
    Invoice_id,
    Invoice_total,
    (SELECT AVG(invoice_total)
        FROM invoices) AS invoice_average,
    Invoices_total – (SELECT invoice_average) AS difference
FROM invoices;
```

Info: In above expressions, we cannot use the invoice_average directly so we use "SELECT invoice_average".

```
# Sub-Queries using Correlated queries
SELECT client_id,
    name,
    (SELECT SUM(invoice_total)
        FROM invoices
        WHERE client_id = c.client_id) AS total_sales,
    (SELECT AVG(invoice_total)
        FROM invoices) AS average
```

```
    (SELECT total_sales – average) AS difference
FROM clients c;
```

Sub-Queries in FROM clause

Whenever we use Sub-Queries in FROM clause, we must give an alias.

```
SELECT *
FROM (SELECT client_id,
            name,
            (SELECT SUM(invoice_total)
                FROM invoices
                WHERE client_id = c.client_id) AS total_sales,
            (SELECT AVG(invoice_total)
                FROM invoices) AS average
            (SELECT total_sales – average) AS difference
        FROM clients c
) AS sales_summary
WHERE total_sales IS NOT NULL;
```

Info: We can write subqueries in FROM clause but it looks a bit messy, so it's better to write the VIEWS and select the data from there.

Essential MySQL Functions

There is various SQL built-in function, some of them are explained as follows.

Numeric Functions

1. ROUND(number, num_of_precisions) – the second parameter/argument is optional, which is used to roundup to nearest high/low number.
   Example: SELECT ROUND(52.7472, 2) will output 52.75
2. TRUNCATE(52.7472, 2) will truncate the digit after decimal, result will be 52.74
3. CEIL(52.7472) OR CEILING(52.7472) output the nearest high value, this will result 53
4. FLOOR(52.7472) outputs nearest low value, results 52 -5
5. ABS(-5.3) OR ABS(5.3) convert the number to positive number, both results 5.3
6. RAND() generates the random floating point number between 0 & 1

Note: There are more numeric functions, however these are the functions we use mostly, to know all the list search MySQL Numeric Functions.

String Functions

1. LENGTH('dhiraj') outputs the total characters in the string
2. LTRIM(' dhiraj') removes the leading white spaces
3. RTRIM('dhiraj ') removes the trailing white spaces
4. TRIM(' Dhiraj ') removes both leading and trailing white spaces
5. UPPER('dhiraj') Capitalized the strings, uppercase string
6. LOWER('DHIraj') returns lowercase string
7. LEFT('dhiraj', 2) returns 2 characters from left
8. RIGHT('dhiraj', 2) returns 2 characters from right
9. SUBSTRING/SUBSTR('dhiraj bhattarai', starting_index/position, length_of_string) example: SUBSTRING('dhirajbhattarai', 3, 5) returns substring starting at position 3 and having length 5 &&& the third parameter is optional, if we remove the 3rd parameter it will return the string all the characters starting from 3
10. LOCATE('search_string', 'string') example LOCATE('d', 'dhiraj') returns the starting position of search char/strings i.e 1 here &&& and we search for character which doesn't exists in the string will return 0

11. REPLACE(‘String’, ‘what_you_want_to_replace’, ‘replace_with’) example REPLACE(‘Dhiraj’, ‘hiraj’, ‘yroz’) returns Dyroz
12. CONCAT(‘String1’, ‘String2’) concatenate the two strings example CONCAT(‘Dhiraj’, ‘ ‘, ‘Bhattarai’) returns ‘Dhiraj Bhattarai’

Info: For more, search MySQL string functions

Date Time Functions

Functions to work with date and time

1. NOW() returns the current date and time
2. CURDATE() returns the current date without time component
3. CURTIME() returns the current time without date
4. YEAR(NOW()) this nested functions returns the current year, we can pass CURDATE() instead of NOW() inside YEAR() method
5. DAYOFYEAR(NOW()) returns day of the years, i.e 201th day of this year
6. DAYOFMONTH(NOW()) returns which day of current month, i.e 20th day
7. DAYOFWEEK(NOW()) returns which day of week, i.e 4th day
8. DAYNAME(NOW()) returns day name i.e Monday
9. MONTHNAME(NOW()) returns month name i.e Jun
10. EXTRACT() method is useful while working with other database as well, for example EXTRACT(DAY FROM NOW()) returns today’s day

Formatting Dates and Times

In MySQL dates are represented as String, 4 digits for Year followed by hyphen(-) 2 digits for month followed by hyphen(-) 2 digits for days

1. DATE_FORMAT(‘date_value’, ‘format_string) example, DATE_FORMAT(NOW(), ‘%d %M %Y’) returns 20 July 2022. We can format the date as our requirement.
2. TIME_FORMAT(NOW(), ‘%h:%i %p’) returns the Hour:Minute AM/PM

Info: For more information about formatting date, search ‘MySQL date format string’

Calculating Dates and Times

Built-in functions to add or subtract data/time.

1. SELECT DATE_ADD(NOW(), INTERVAL 1 DAY) returns the next/tomorrows date with same time. Also we can add 1 year by changing the interval DATE_ADD(NOW(), INTERVAL 1 YEAR)

Info: To select the previous date either we can pass negative date in DATE_ADD interval parameter or we can use DATE_SUB().

2. SELECT DATEDIFF('date_1', 'date_2') returns the difference in those dates in days. It calculates the total days.

IFNULL and COALESCE Functions

This function helps to assign the value for null characters

Example: Select all the shipper_id and replace null with 'Not assigned'

```
SELECT
    order_id,
    IFNULL(shipper_id, 'Not Assigned') AS shipper
FROM orders
```

Same expression can be written using COALESCE functions, COALESCE(shipper_id, 'Not assigned') AS shipper

Select the replace the records in shipper_id column if shipper_id and comments both are null, otherwise replace null with comments

```
SELECT
    Order_id,
    COALESCE(shipper_id, comments, 'Not assigned') AS shipper
FROM orders
```

IF Functions

If function will execute a logical function to choose either or expression, If the method/expression is true the first value will be returned otherwise second value will be returned.

IF(expression, first, second)

```
SELECT order_id,
    order_date,
    IF(YEAR(order_date) = YEAR(NOW()), 'Active', 'Archieved') AS category
FROM orders
```

CASE Operator

In order to apply multiple conditions (Which cannot be done using IF functions), we use CASE operator/functions

```
SELECT
    Order_id,
    CASE
        WHEN YEAR(order_date) = YEAR(NOW()) THEN 'Active'
        WHEN YEAR(order_date) = YEAR(NOW()) -1 THEN 'Last Year'
        WHEN YEAR(order_date) < YEAR(NOW()) – 1 THEN 'Archieved'
        ELSE 'Future' – Optional
    END AS category
FROM orders
```

VIEWS

Whenever our Query becomes complex and messy, views come in rescue by storing the query temporary in view to access easily.

It’s like a data warehouse, we curate the data and save it as a view which behaves as an actual table, but in fact it’s just an stored query which makes our query easy.

```
CREATE VIEW sales_by_client AS
SELECT
    c.client_id,
    c.name,
    SUM(invoice_total) AS total_sales
FROM clients c
JOIN invoices i USING (client_id)
GROUP BY client_id, name
```

Info: After creating VIEW we can use it as a regular table and perform all the SQL queries we do in other tables.

Altering and Dropping Views

Once you created your views and you realized you want to change it, we can apply these functions.

```
# Let’s delete the views table
DROP VIEW sales_by_client

# The better way to do it is by using CREATE OR REPLACE clause
CREATE OR REPLACE VIEW sales_by_client AS
SELECT
```

```
    c.client_id,
    c.name,
    SUM(invoice_total) AS total_sales
FROM clients c
JOIN invoices i USING (client_id)
GROUP BY client_id, name
```

Updatable Views

To update the table using view, we shouldn't use the aggregate functions in views. Otherwise, we cannot apply it for change/update the table.

The following VIEW can be used for updating table, because we are not using any aggregate function.

```
CREATE OR REPLACE VIEW invoices_with_balance AS
SELECT
    invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_total – payment_total AS balance,
    invoice_date,
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total – payment_total) > 0
```

```
# Let's use the VIEW to delete records
DELETE FROM invoices_with_balance
WHERE invoice_id = 1;

# Let's update the records using VIEW
UPDATE invoices_with_balance
SET due_date = DATE_ADD(due_date, INTERVAL 2 DAY)
WHERE invoice_id = 2
```

The WITH CHECK OPTION Clause

When we update or delete the records using VIEW we may encounter with problems, we can exclude such scenarios using WITH CHECK OPTION clause.

```
CREATE OR REPLACE VIEW invoices_with_balance AS
SELECT
    invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_total – payment_total AS balance,
    invoice_date,
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total – payment_total) > 0
WITH CHECK OPTION
```

Benefits of Views

1. Simplify queries
2. Reduce the impact of changes
3. Restrict access to the data
4. Data Securities

Stored Procedures

Stored procedure is a database object that contains a block of SQL code, In our applications code we simply call this procedures to get or save the data.

Benefits:

a. Store and organize SQL
b. Faster execution
c. Data security

Creating a Stored Procedure

Stored procedure is similar to GitLab jobs/stages, which will be executed in order after creating it.

```
CREATE PROCEDURE get_clients()

BEGIN

    SELECT * FROM clients;

END
```

Info: Store procedure starts with CREATE PROCEDURE followed by get_clients() method, we use lowercase words separated by underscore to create variable/method. And the expression between the BEGIN and END clause is called Body, which can have multiple queries so we have to terminate the queries using semicolon (;).

To execute the stored procedure as a single expression we have to assign the delimiter (i.e by convention $$) and change it back to default after end of the expression so that SQL engine execute it in a one goes.

```
DELIMITER $$

CREATE PROCEDURE get_clients()

BEGIN

    SELECT * FROM clients;

END$$
```

```
DELIMITER ;
```

Note: We can call the stored procedure using CALL get_clients() or by clicking the button on workbench/tools.

Create a stored procedure called, get_invoices_with_balance to return all the invoices with a balance > 0

```
CREATE PROCEDURE get_invoices_with_balance()
BEGIN
    SELECT *
    FROM invoices
    WHERE invoice_total – payment_total > 0;
END
```

OR by using VIEWS

```
DELIMITER $$
CREATE PROCEDURE get_invoices_with_balance()
BEGIN
    SELECT *
    FROM invoices_with_balance
    WHERE balance > 0;
END$$
DELIMITER ;
```

Drop Stored Procedures

```
DROP PROCEDURE get_clients
```

Info: If we execute the above expression multiple time it will throw an error to prevent this we use IF EXISTS keywords.

```
DROP PROCEDURE IF EXISTS get_clients
```

Note: Let's create the basic template for the stored procedure and store it in a separate file to save for later use in version control system like git/github.

```
DROP PROCEDURE IF EXISTS get_clients;

DELIMITER $$
CREATE PROCEDURE get_clients()
BEGIN
    SELECT * FROM clients;
END$$
DELIMITER ;
```

Add Parameters in Stored Procedure

We use parameters to execute and get some specific value, stored procedure will return the results.

```
DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITER $$
CREATE PROCEDURE get_clients_by_state
```

```
(
    state CHAR(2)
)
BEGIN
    SELECT * FROM clients c
    WHERE c.state = state;
END$$
DELIMITER ;
```

Info: We can call the above procedure using CALL get_clients_by_state('VA');

Parameter With Default Value

```
DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITER $$
CREATE PROCEDURE get_clients_by_state
(
    state CHAR(2)
)
BEGIN
    IF state IS NULL THEN
        SET state = 'CA';
    END IF;
    SELECT * FROM clients c
    WHERE c.state = state;
END$$
```

```
DELIMITER ;
```

Info: We can select All value if the state parameters is NULL as well,

```
DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITER $$
CREATE PROCEDURE get_clients_by_state
(
	state CHAR(2)
)
BEGIN
	IF state IS NULL THEN
		SELECT * FROM clients;
	ELSE
		SELECT * FROM clients c
		WHERE c.state = state;
	END IF;
END$$
DELIMITER ;
```

Note: Above expression looks a bit messy so we can make it more concise.

```
DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITER $$
```

```
CREATE PROCEDURE get_clients_by_state
(
    state CHAR(2)
)
BEGIN
    SELECT * FROM clients c
    WHERE c.state = IFNULL(state, c.state);
END$$
DELIMITER ;
```

Parameter Validation

We can update using stored procedure

```
CREATE PROCEDURE make_payment (
    invoice_id INT,
    payment_amount DECIMAL(9, 2),
    payment_date DATE
)
BEGIN
    UPDATE invoices i
    SET
        i.payment_total = payment_amount,
        i.payment_date = payment_date
    WHERE i.invoice_id = invoice_id
END
```

Warning: Above expression will lead to problem, for example we can store negative payment value as well, so before storing/updating date we need to validate it. We can generate SQLSTATE ERROR while validating or raising error.

Let's write an expression before update command

```
IF payment_amount <= 0 THEN
        SIGNAL SQLSTATE '22003'
                SET MESSAGE_TEXT  = "Invalid payment amount';
```

Add above statement before your store procedure statement.

Output Parameter

```
CREATE PROCEDURE get_unpaid_invoices_for_client
(
        Client_id INT,
        OUT invoices_count INT,
        OUT invoices_total DECIMAL(9, 2)
)
BEGIN
        SELECT COUNT(*), SUM(invoice_total)
        INTO invoices_count, invoices_total
        FROM invoices i
        WHERE i.client_id = client_id
                AND payment_total = 0;
END
```

Variables

```
SET @invoices_count = 0
```

```
# Let’s declare variables
CREATE PROCEDURE get_risk_factor ()
BEGIN
    DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
    DECLARE invoices_total DECIMAL(9, 2);
    DECLARE invoices_count INT;

    SELECT COUNT(*), SUM(invoice_total)
    INTO invoices_count, invoices_total
    FROM invoices;

    SET risk_factor = invoices_total / invoices_count * 5;

    SELECT risk_factor;

END
```

Functions

Returns a single value

```
# Let’s create a function to calculate the risk factor
CREATE FUNCTION get_risk_factor_for_client
(
    Client_id INT
)
RETURNS INTEGER
READS SQL DATA
```

```
BEGIN
    DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
    DECLARE invoices_total DECIMAL(9, 2);
    DECLARE invoices_count INT;

    SELECT COUNT(*), SUM(invoice_total)
    INTO invoices_count, invoices_total
    FROM invoices i
    WHERE i.client_id = client_id;

    SET risk_factor = invoices_total / invoices_count * 5;
    RETURN IFNULL(risk_factor, 0);
END
```

Triggers and Events

Trigger is a block of SQL code that automatically gets executed before or after an insert, update or delete statement. Quite often we use trigger for data consistency.

```
DELIMITER $$

CREATE TRIGGER payments_after_insert
    AFTER INSERT ON payments        # we can use BEFORE INSERT /UPDATE /DELETE
    FOR EACH ROW                    # Execute for each row affected
BEGIN
    UPDATE invoices
    SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;
END $$

DELIMITER ;
```

Warning: We can modify any table except the table specified in CREATE TRIGGER clause (i.e. payments in above expression), otherwise it will run infinite times because trigger execute automatically.

Viewing Created Triggers

To view the triggers created, we can use the expression `SHOW TRIGGERS` which will show all the triggers created. To select the specific triggers, we have to explicitly specify the name of the triggers.

Example: SHOW TRIGGERS LIKE ‘payments%’

The above expression shows all the triggers starting with `payments`

---- table_after_insert /table_after_delete /table_before_insert

These are the table names we created earlier, usually we follow this conventions.

Dropping Triggers

We can use the dropping **store procedures** like expressions to drop/delete triggers, and optionally we can use IF EXISTS keyword and as a best practice, we can use the drop and create triggers expression in a same file.

```
DROP TRIGGER IF EXISTS payment_after_insert;


# Let's add the drop trigger expression in create trigger query.
DELIMITER $$


DROP TRIGGER IF EXISTS payment_after_insert;


CREATE TRIGGER payments_after_insert
    AFTER INSERT ON payments        # we can use BEFORE INSERT /UPDATE /DELETE
    FOR EACH ROW                    # Execute for each row affected
BEGIN
    UPDATE invoices
    SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;
END $$


DELIMITER ;
```

Using Triggers for Auditing

Using audit table query we can log our changes in our auditing table.

```
USE sql_invoicing;


CREATE TABLE payments_audit
(
    client_id      INT
    date           DATE
    amount         DECIMAL(9, 2)
    action_type    VARCHAR(50)
    action_date    DATETIME
)
```

Now we can use the above expression to add the log in our audit table, let's add the following expression below the previous trigger query inside BEGIN/End clause.

```
INSERT INTO payments_audit
VALUES (NEW.client_id, NEW.date, NEW.amount, 'Insert', NOW() )
```

EVENTS

A task (or block of SQL code) that gets executed according to a schedule, for example (once a month, once a week, twice a year etc, for example the Viber ot Whatsapp ask you to backup your messages every day/week or month, that's also a event method)

```
# To show the systems variables type, SHOW VARIABLES
SHOW VARIABLES LIKE 'event%';
```

If the above even variable is OFF turn it ON, if you don't require it turn it OFF because it is regularly checking for event to execute, it uses background memory/processes… to save system resources.

SET GLOBAL event_scheduler = ON

Info: Use the keyword like once, yearly, monthly, weekly to easily search or select the EVENT which are created or running.

DELIMITER $$

CREATE EVENT yearly_delete_stale_audit_rows

ON SCHEDULE

    # Here we have to specify the date or run the even once or in regular interval

    AT '2022-12-30'

    # To execute on regular basis, give start date and optional end date

    - - EVERY 1 DAY STARTS '2022-12-30' ENDS '2025-12-30'

DO BEGIN

    DELETE FROM payments_audit

    WHERE action_date < NOW() – INTERVAL 1 YEAR;

END $$

DELIMITER ;

Viewing, Dropping and Altering Events

View the Events, optionally specify the required keyword using LIKE

SHOW EVENTS LIKE 'yearly%';

```
# Drop the EVENTS
DROP EVENTS IF EXISTS yearly_delete_stale_audit_rows;

# Alter EVENTS, we can also enable or disable the Event using alter keyword.
ALTER EVENT yearly_delete_stale_audit_rows DISABLE;
ALTER EVENT yearly_delete_stale_audit_rows ENABLE;
```

Transactions and Concurrency

Transaction is multiple queries grouped as a single unit of expression, if one of the queries fail to execute, the whole expression must be aborted and rollback.

A group of SQL statements that represent a single unit of work.

ACID Properties:

a. Atomicity
b. Consistency
c. Isolation
d. Durability

Let’s start creating a transaction

START TRANSACTION;

INSERT INTO orders (customer_id, order_date, status)

VALUES (1, ‘2022-07-20’, 1);

INSERT INTO order_items

VALUES (LAST_INSERT_ID(), 1, 1, 1);

COMMIT;

Sometimes we want to deal with error checking, we use ROLLBACK instead of COMMIT.

Concurrency and Locking

By default, MySQL lock the system while one query is running, other have to wait until the first query execute successfully or rollback.

To change the Transaction_isolation use following query

SHOW VARIABLES LIKE ‘transaction_isolation’;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

READ UNCOMMITTED Isolation Level (It generates concurrency problem)

USE database_name;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT points

FROM customers

WHERE customer_id = 1;

Let’s change the above syntax line to

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

This will let you read only the committed data.

Data Types

1. Strings
    a. CHAR(x) - Fixed-length
    b. VARCHAR(x) - Variable-length
    c. TEXT
    d. TINTTEXT
    e. MEDIUMTEXT
    Etc….
2. Integers
    a. TINYINT 1byte [-128, 127]
    b. UNSIGNED TINYINT [0, 255] Useful for storing positive integer
    c. SMALLINT 2byte [-32K, 32K]
    d. MEDIUMINT 3byte [-8M, 8M]
    e. INT 4byte [-2B, 2B]
    f. BIGINT 8byte [-9Z, 9Z]

    Try to use small byte size data types, this will help to execute your query fast, for example if you want to store age use UNSIGNED TINYINT, because nobody going to live more than 255 years.
3. Rationals
    a. DECIMAL(p, s) Precision and Scale must be supplied
4. Booleans
    a. BOOL / BOOLEAN
5. Enums and Set Types (Don’t use it)
    a. Used to store the values from the choices
6. Date/Time
    a. DATE / TIME / DATETIME / TIMESTAMP / YEAR

Designing Database

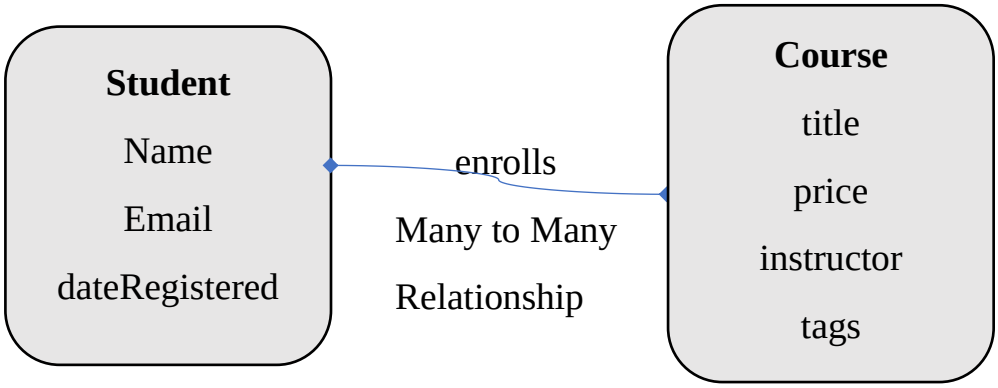
Database design affects your querying time and speed of executions. So when we design our database, we have to be conscious. Designing a good database take time but, poorly designed database takes more time while querying and maintaining.

Some fundamental steps we need to consider while designing database…

1. Data Modelling
    a. Understand the requirements
    b. Build a Conceptual Model
    c. Build a Logical Model
    d. Build a Physical Model

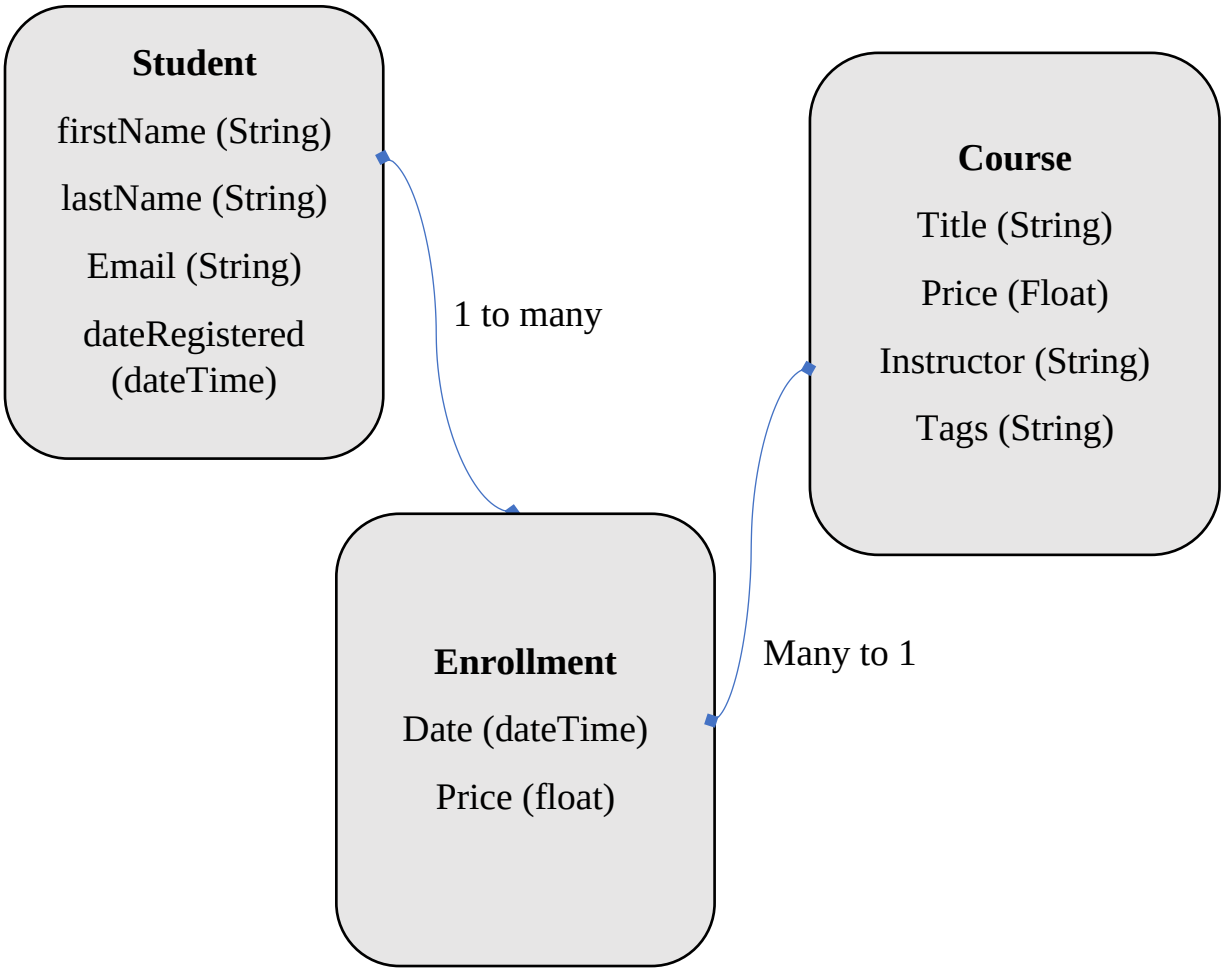
Conceptual Models

Before we design our logical or physical model of database, we need to design/build the conceptual models which represents the entities and their relationships. Use Entity Relationship (ER) or Unified Modeling Language (UML) diagrams.



Logical Model

Now we have to specify the data types of all the entity/attributes, logical model is independent of database. Now we have to breakdown the name into first and last name, which will be easy while searching or sorting the data. If we have a single attribute containing first and last name, it will be hard/time taking to query and extract the first and last name from that data.



Physical Model

Physical model is the database/physical implementation of logical model. Whatever we designed logically will be applied in the respective database design tool (i.e. MySQL, PostgreSQL etc)