# SQL Complete Guide

> ## BASICS:

1. Multiple Query Statements need to end with ;

2. -- represents a comment

3. SQL is not case sensitive

4. SQL query statements can be written in one single line or in multiple lines.

5. Order of clauses matter.

6. RDBMS data tables should always have a PRIMARY KEY Column.

7. Default sorting [ORDER BY] is done through PRIMARY KEY column

8. VARCHAR(50) -> Stores atmost 50 alphanumeric string values. If value is of length 5, then it takes only 5 space in memory.

9. CHAR(50) -> If value length is 5, it will fill the rest of 45 space in memory by itself.

10. Subquery is a SQL statement which resides within another SQL statement.

11. Put string data in '' or "". Mainly done in '' as per convention

### 1. Select the database

```sql

USE database\_name;

```

### 2. Retrieving Data

- FROM SINGLE TABLE:

- SELECT Statement:

```sql

SELECT [colums | \*]

FROM [table\_name] (optional)

WHERE [filter\_condition] (optional)

ORDER BY [column\_name] (optional)

-- (SELECT 1, 2 will create two columns on the go temporary)

-- We specify the columns in order to reduce the load on server, database and network.

-- Put names having spaces in ''.

SELECT

last\_name,

first\_name,

points,

points \* 10 + 100 AS discount\_factor

FROM customers

-- Order of operators is important. Works like normal maths. Parenthesis can be used to change the order of the operation

-- AS is an alias keyword to rename the column

-- DISTINCT keyword removes the duplicates.

```

- WHERE clause:

```sql

-- is used to filter the results as per the condition. The query execution engine iterates over all the records and retrieves the ones that satisfies this condition

```

- Comparison Operators for WHERE Clause:

- &gt; [greater than]

- &gt;= [greater than or equal to]

- < [less than]

- <= [less than or equal to]

- = [equality]

- != [not equal to]

- <> [not equal to]

```sql

SELECT \*

FROM customers

WHERE points > 3000

SELECT \*

FROM Customers

WHERE state = 'va'

SELECT \*

FROM cusTOMERS

WHERE birth\_DAtE > 'YYYY-MM-DD'

```

- LOGICAL Operators:

- AND {1 -> in terms of order to execute}

- OR {2 -> in terms of order to execute}

- NOT (makes every comparator operator reverse. AND <-> OR, < -> > and so on)

```sql

SELECT \*

FROM customers

WHERE points > 3000 AND birth\_date > '1990-01-01'

SELECT \*

FROM customers

WHERE points > 3000 OR birth\_date > '1990-01-01'

SELECT \*

FROM customers

WHERE points > 3000 OR birth\_date > '1990-01-01' AND state = 'VA'

```

- IN Operator:

\* In SQL we cannot combine a string with a boolean condition that produces a true or false. Thus we cannot write something like: state = "VA" OR "GA". We need: state = 'VA' OR state = 'GA'.

\* state IN ('VA', 'FL', 'GA')

\* Use this when you want to evaluate attribute to a bunch of values.

\* Wording like: stock equal to value\_1, value\_2, value\_3, etc.

- BETWEEN Operator:

- Whenever the range of value is involved, use between.

- Inclusive range values.

- points >=1000 AND points <=3000 == points BETWEEN 1000 AND 3000

```sql

SELECT

FROM customers

WHERE birth\_date BETWEEN '1990-01-01' AND '2000-01-01'

```

- LIKE Operator:

- Match pattern and filter the result as per pattern which satisfies the result.

- String pattern.

- % indicates any number of characters

- ['b%' -> starts with b or B]

- ['%b%' -> b somewhere in string]

- ['%y' -> ends with y]

- \\_ indicates single character

- [\_y -> 2 character string ending with y]

- [b\_\_y -> string having 4 characters which starts with b and ends with y]

```sql

SELECT

FROM customers

WHERE address LIKE '%Trail%' OR address LIKE '%avenue%'

```

- REGEXP Operator:

- REGular EXPression operator

- ^[carrot sign] -> ^field => must start with field

- field$ -> must end with field

- field|mac -> have field or mac [pipe]

- ^field|mac|rose -> start with field or have mac or have rose

- [gim]e -> any of characters in brackets can come before e -> ge or ie or me in reality

- e[fmq] -> ef or em or eq

- [a-z]e -> range of ae or be or ... ze.

- ^ beginning

- $ end

- | logical or

- [abcd] a or b or c or d

- [a-f] range

```sql

SELECT \*

FROM customers

WHERE address REGEXP 'Trail|Avenue';

```

- IS NULL Operator:

- Null represents absence of value

- WHERE phone IS NULL

- WHERE phone IS NOT NULL

```sql

SELECT \*

FROM orders

WHERE shipped\_date IS NULL

```

- ORDER BY Clause:

- Default done via PRIMARY KEY Column

- Used for sorting as per condition

- ORDER BY first\_name

- Default ASC

- DESC can be done

- Multiple columns can be done. first\_column sort, then each sorted result group gets sorted using second\_column

- ORDER BY state DESC, first\_name

- We can sort data in MYSQL by any column regardless they are being picked or not for display.

- Alias sorting allowed in MySQL.

- Sorting columns using their position [ORDER BY 1, 2] should be avoided.

```sql

SELECT \*

FROM order\_items

WHERE order\_id = 2

ORDER BY quantity, unit\_price DESC

```

- LIMIT Clause:

- LIMIT 6, 3

- 6 is offset here.

- 3 is the limit of records after 6 offset.

- Should always come at the end in terms of order.

```sql

SELECT \*

FROM order\_items

WHERE order\_id = 2

ORDER BY quantity, unit\_price DESC

LIMIT 6,3

```

- FROM MULTIPLE TABLE:

- INNER JOIN/JOIN Clause:

- Use to join two tables to extract data residing in the two table but are linked somehow

```sql

SELECT oi.order\_id,

p.product\_id,

p.name,

oi.quantity,

oi.unit\_price

FROM order\_items oi

INNER JOIN products p ON oi.product\_id = p.product\_id

```

- It's better practice to give aliases to the tables involved.

- It's not required to prefix alias to the unambiguous columns in SELECT clause. Example: oi.order\_id can simply be order\_id.

- INNER keyword is optional as that's the default behaviour for JOIN.

- JOINING ACROSS DATABASES:

- You simply prefix the name of the database before the table names to join tables residing in different databases.

- If the current database in use is "A" then no need to prefix table residing in "A" with "A".

```sql

SELECT oi.order\_id,

p.product\_id,

p.name,

oi.quantity,

oi.unit\_price

FROM order\_items oi

INNER JOIN sql\_inventory.products p ON oi.product\_id = p.product\_id

```

- SELF JOIN:

- Exactly the same as INNER JOIN. Just requires different aliases for the same table and prefix becomes necessary due to column ambiguity.

- JOIN MORE THAN 2 TABLES:

```sql

SELECT o.order\_id,

o.order\_date,

c.first\_name,

c.last\_name,

os.name AS status

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id

JOIN order\_statuses os ON o.status = os.order\_status\_id

```

- COMPOUND JOIN STATEMENTS:

- Sometimes, there is no one unique PRIMARY KEY in the table to help us identify unique data record from the table.

- In such situation, we need to use 2 or more column names to uniquely identify a record.

- These group of columns are all made PRIMARY KEY and are called COMPOSITE PRIMARY KEY

- JOIN table\_name ON cond\_1 AND cond\_2

- IMPLICIT JOIN Syntax:

```sql

SELECT \*

FROM orders o, customers c [till now it's CROSS JOIN]

WHERE o.customer\_id = c.customer\_id [this makes it IMPLICIT INNER JOIN]

```

- OUTER JOIN:

- Two types: LEFT & RIGHT

- OUTER keyword is optional. So, LEFT OUTER JOIN == LEFT JOIN

- Used when we want to make the ON condition true but not strictly. Meaning, when we want the all the data of either LEFT table or RIGHT table. It returns all the record whether the condition is true or not.

```sql

SELECT c.customer\_id, first\_name, order\_id

FROM customers c

LEFT JOIN orders o on c.customer\_id = o.customer\_id

ORDER BY c.customer\_id

-- (AND)

SELECT c.customer\_id, first\_name, order\_id

FROM customers c

RIGHT JOIN orders o on c.customer\_id = o.customer\_id

ORDER BY c.customer\_id

```

- OUTER JOIN on Multiple Tables:

```sql

SELECT o.order\_id,

o.order\_date,

c.first\_name,

s.name AS shipper,

os.name AS status

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id

LEFT JOIN shippers s ON o.shipper\_id = s.shipper\_id

LEFT JOIN order\_statuses os ON o.status = os.order\_status\_id

ORDER BY shipper

```

- SELF OUTER JOIN:

```sql

select e.employee\_id,

e.first\_name,

m.first\_name

from employees e

left join employees m On e.reports\_to = m.employee\_id;

```

- USING Clause:

- USING keyword works only when the column name across different tables are same.

- Equivalent to ON cond\_1

- Can be used for single and multiple columns.

```sql

SELECT o.order\_id,

c.first\_name,

sh.name AS shipper

FROM orders o

JOIN customers c

USING (customer\_id)

LEFT JOIN shippers sh

USING (shipper\_id)

-- AND

SELECT

FROM order\_items oi

JOIN order\_item\_notes oin USING (order\_id, product\_id)

-- AND

SELECT p.date,

c.name,

p.amount,

pm.name AS payment\_method

FROM payments p

JOIN clients c USING (client\_id)

JOIN payment\_methods pm ON p.payment\_method = pm.payment\_method\_id

```

- NATURAL JOINS:

- Database engine will join the tables using the column name, column having the same name.

- Due to lack of control, it's better to not use it, as it can produce unexpected result.

```sql

SELECT o.order\_id,

c.first\_name

FROM orders o

NATURAL JOIN customers c

```

- CROSS JOIN Clause:

\* Every record of one table combines with every record of another table.

\* n x m records.

\* Implicit Syntax:

```sql

SELECT \*

FROM shippers s, products p

ORDER BY s.name;

```

\* Explicit Syntax:

```sql

SELECT \*

FROM shippers s

CROSS JOIN products p

ORDER BY s.name

```

- UNION:

- Combine multiple rows rather than columns.

- Combine multiple query results.

- These queries can be from the same table or multiple table

```sql

SELECT order\_id, order\_date, 'Active' AS status

FROM orders o

WHERE o.order\_date >= '2019-01-01'

UNION

SELECT order\_id, order\_date, 'Archived' AS status

FROM orders o

WHERE o.order\_date <= '2019-01-01';

```

- Number of columns each query returns should be same or it will returns an error.

```sql

SELECT customer\_id, first\_name, points, 'Bronze' AS type

FROM customers

WHERE points < 2000

UNION

SELECT customer\_id, first\_name, points, 'Silver' AS type

FROM customers

WHERE points BETWEEN 2000 AND 3000

UNION

SELECT customer\_id, first\_name, points, 'Gold' AS type

FROM customers

WHERE points > 3000

ORDER BY first\_name

```

### 3. Inserting Data

```sql

INSERT INTO table\_name,

VALUES(DEFAULT, 'value\_1', 'value\_2', NULL, '2002-01-01');

-- OR

INSERT INTO table\_name (col\_val\_1, col\_val\_2, col\_date)

VALUES (value\_1, value\_2, '2002-01-01');

-- String and Date value should be within ''.

-- Numbers can be without ''.

-- DEFAULT keyword is used for autoincrement column and NULL allowed column.

-- NULL keyword is for null allowed column.

-- OR

INSERT INTO table\_name (name)

VALUES ('value\_1'),

('value\_2'),

('value\_3')

-- Inserting multiple values is allowed in one go.

-- Inserting hierarchical values: This is inserting the values in one table and then use that new record to insert value in other table.

-- LAST\_INSERT\_ID() is the function that gives back the last insert ID.

INSERT INTO orders (customer\_id, order\_date, status)

VALUES (1, '2021-01-01', 1);

-- OR

INSERT INTO order\_items

VALUES (LAST\_INSERT\_ID(), 1, 1, 2.95),

(LAST\_INSERT\_ID(), 2, 1, 2.95);

--

CREATE TABLE new\_table AS SELECT \* FROM orders

-- Copies the table but not the column attributes like Primary Key marking, AI marking, etc.

-- It contains the subquery of SELECT...

TRUNCATE table\_name

-- deletes all the records within the table.

-- We can use subquery inside INSERT statement

CREATE TABLE invoices\_archived AS

SELECT i.invoice\_id,

i.number,

c.name AS client\_name,

i.invoice\_total,

i.payment\_total,

i.invoice\_date,

i.due\_date,

i.payment\_date

FROM invoices i

JOIN clients c USING (client\_id)

WHERE payment\_date IS NOT NULL

```

### 4. Updating Data

- Updating Single Row

```sql

UPDATE table\_name

SET col\_name\_1 = value\_1/expression, col\_name\_2 = value\_2/expression

WHERE cond\_1 (this condition is necessary)

-- OR

UPDATE invoices

SET

payment\_total = invoice\_total \* 0.5

payment\_date = due\_date

WHERE invoice\_id = 3;

```

- Updating Multiple Row

\* Leave the WHERE Clause

\* WHERE client\_id IN (3, 5);

- Updating Subqueries in Updates

```sql

UPDATE orders

SET comments = 'GOLD CUSTOMERS'

WHERE customer\_id IN (SELECT customer\_id

FROM customers

WHERE points > 3000)

```

### 5. Deleting Data

- DELETING ROWS:

```sql

DELETE FROM table\_name

WHERE cond\_1 = value\_1;

-- OR

DELETE FROM table\_name

WHERE cond\_1 = (sub\_query)

```

> ## INTERMEDIATE:

### 5. Summarizing Data (Useful in Report Tools)

- Aggregate Functions:

- MAX()

- MIN()

- AVG()

- SUM()

- COUNT()

```sql

SELECT

MAX(invoice\_total) AS highest,

MIN(invoice\_total) AS lowest,

AVG(invoice\_total) AS average,

SUM(invoice\_total \* 1.1) AS total,

COUNT(DISTINCT invoice\_total) AS number\_of\_invoices,

COUNT(\*) AS total\_records

FROM invoices

WHERE invoice\_date > '2019-07-01';

-- Work on Non-NULL values

-- Work on Date Values and String values as well.

-- Work on Numeric values

-- Work with WHERE clause

-- By Default duplicate values are included. Using DISTINCT can give DISTINCT values.

```

- GROUP BY Clause:

- Group BY include all the SELECT columns except the aggregate one.

```sql

SELECT

client\_id,

SUM(invoice\_total) AS total\_sales

FROM invoices

GROUP BY client\_id

ORDER BY total\_sales DESC

-- by default the result is sorted by column\_name in GROUP BY clause.

-- GROUP BY is always after FROM and WHERE Clause and before ORDER BY clause.

SELECT date,

pm.name AS payment\_method,

SUM(amount) AS total\_payments

FROM payments p

JOIN payment\_methods pm on p.payment\_method = pm.payment\_method\_id

GROUP BY date, payment\_method

ORDER BY date

```

- HAVING CLAUSE:

- Filter data after the rows have been grouped.

```sql

SELECT client\_id,

SUM(invoice\_total) AS total\_sales

FROM invoices

GROUP BY client\_id

HAVING total\_sales > 800;

```

- WITH ROLL UP (only MySQL)

- Sums the above Aggregate column values. Summarize the data.

- Only applies to columns that aggregate.

- It calculates the summary for each group if used with GROUP BY multiple columns.

```sql

SELECT state,

city,

SUM(invoice\_total) AS total\_sales

FROM invoices i

JOIN clients c on c.client\_id = i.client\_id

GROUP BY state, city WITH ROLLUP

-- AND

SELECT pm.name AS payment\_method,

SUM(p.amount) AS total

FROM payments p

JOIN payment\_methods pm ON p.payment\_method = pm.payment\_method\_id

GROUP BY pm.name WITH ROLLUP

ORDER BY total

```

### 6. Writing Complex Queries

- SUBQUERIES

- A subquery is a select query within another query.

- \*\*Question: Find (SQL MOSH Databases) the products where the unit price is greater than the product Lettuce with id = 3\*\*

```sql

SELECT \*

FROM products

WHERE unit\_price > (SELECT unit\_price

FROM products

WHERE product\_id = 3)

```

- \*\*Question: In sql\_hr database, find employees who earn more than average\*\*

```sql

SELECT \*

FROM employees

WHERE salary > (SELECT AVG(salary)

FROM employees)

ORDER BY employee\_id

```

- USING IN OPERATOR TO WRITE SUBQUERIES

- \*\*Ques: Find the products that have never been ordered\*\*

```sql

SELECT \*

FROM products

WHERE product\_id NOT IN (SELECT DISTINCT product\_id

FROM order\_items)

```

- \*\*Ques: Find clients without invoices\*\*

```sql

SELECT \*

FROM clients

WHERE client\_id NOT IN (SELECT DISTINCT client\_id

FROM invoices)

```

- SUBQUERIES vs JOINS

- Quite often, we can write subqueries using JOIN clauses.

- \*\*Ques: same as above\*\*

```sql

SELECT \*

FROM clients c

LEFT JOIN invoices i USING (client\_id)

WHERE i.invoice\_id IS NULL

```

- \*\*Ques: Find customers who have ordered lettuce (id = 3), select customer\_id, first\_name, last\_name\*\*

```sql

SELECT DISTINCT customer\_id,

first\_name,

last\_name

FROM customers c

JOIN orders o USING (customer\_id)

JOIN order\_items oi USING (order\_id)

WHERE oi.product\_id = 3

```

- ALL Keyword:

- \*\*Ques: Select invoices larger than all invoices of client 3\*\*

```sql

SELECT \*

FROM invoices

WHERE invoice\_total > (SELECT MAX(invoice\_total)

FROM invoices

WHERE client\_id = 3)

-- Above is equivalent to

SELECT \*

FROM invoices

WHERE invoice\_total > ALL (SELECT invoice\_total

FROM invoices

WHERE client\_id = 3)

```

- ANY/SOME Keyword

- \*\*Ques: Select clients with at least two invoices\*\*

```sql

SELECT c.client\_id,

c.name

FROM clients c

WHERE c.client\_id IN (SELECT client\_id

FROM invoices

GROUP BY client\_id

HAVING COUNT(\*) >= 2)

-- Equivalent to

SELECT c.client\_id,

c.name

FROM clients c

WHERE c.client\_id = (ANY/SOME) (SELECT client\_id

FROM invoices

GROUP BY client\_id

HAVING COUNT(\*) >= 2)

-- Equivalent to

SELECT c.client\_id,

c.name

FROM invoices i

JOIN clients c USING (client\_id)

GROUP BY client\_id

HAVING COUNT(\*) >= 2

```

- CORRELATED Queries:

- \*\*Ques: Select employees whose salary is above the average in their office\*\*

```sql

SELECT \*

FROM employees e

WHERE salary > (SELECT AVG(salary)

FROM employees

WHERE office\_id = e.office\_id)

-- In the first run, the first employee is selected and then the subquery condition runs for it. The subquery condition is dependent upon the outer query using the WHERE clause. As such, it calculates the average salary of all employees where their office\_id = outer employee's office\_id. In short, it averages the salary office wise in each iteration, rather than just once that happens when there is no correlation.

-- This constant iterative execution causes the query to run slow sometimes depending upon the number of data records.

-- Still quite powerfull having lot of application in real world

```

- \*\*Ques: Get invoices that are larger than the client's average invoice amount\*\*

```sql

SELECT invoice\_id,

invoice\_total

FROM invoices i

WHERE invoice\_total > (SELECT AVG(invoice\_total)

FROM invoices

WHERE i.client\_id = client\_id)

```

- The EXISTS Operator

- When the IN operator subquery returns a large result set of values, it's better to use the EXISTS operator which doesn't return the result set.

- EXISTS works better in above case as its fast in comparison to IN Operator.

```sql

SELECT \*

FROM clients c

WHERE EXISTS(SELECT client\_id

FROM invoices

WHERE c.client\_id = client\_id)

--

SELECT \*

FROM products

WHERE NOT EXISTS(

SELECT product\_id

FROM order\_items

WHERE products.product\_id = order\_items.product\_id

)

```

- Subqueries in SELECT Clause

```sql

SELECT invoice\_id,

invoice\_total,

(SELECT AVG(invoice\_total)

FROM invoices) AS invoice\_average,

invoice\_total - (SELECT invoice\_average) AS difference

FROM invoices

--

SELECT client\_id,

c.name,

(SELECT SUM(invoice\_total)

FROM invoices

WHERE c.client\_id = client\_id) AS total\_sales,

(SELECT AVG(invoice\_total)

FROM invoices) AS invoice\_average,

(SELECT total\_sales - invoice\_average) AS difference

FROM clients c

```

- Subqueries in FROM Clause

```sql

SELECT \*

FROM (SELECT client\_id,

c.name,

(SELECT SUM(invoice\_total)

FROM invoices

WHERE c.client\_id = client\_id) AS total\_sales,

(SELECT AVG(invoice\_total)

FROM invoices) AS invoice\_average,

(SELECT total\_sales - invoice\_average) AS difference

FROM clients c) AS sales\_summary

WHERE total\_sales IS NOT NULL

-- It's better to use VIEWS in its place to store virtual tables obtained from subqueries within FROM Clause.

```

### 7. Built-in Functions

- Numeric Functions

```sql

SELECT ROUND(5.7345, 2) -- keep 2 digits after round off -> 5.73

SELECT CEILING(5.2) -- 6

SELECT CEIL(5.2) -- 6

SELECT FLOOR(5.2) -- 5

SELECT ABS(-5.2) -- 5.2

SELECT RAND() -- generates a random floating point number between 0 and 1

```

- String functions

```sql

SELECT LENGTH('sky') -- 3

SELECT UPPER('sky') -- SKY

SELECT LOWER('SKY') -- sky

SELECT LTRIM(' Sky') -- Sky

SELECT RTRIM('Sky ') -- Sky

SELECT TRIM(' Sky ') -- Sky

SELECT LEFT('Kindergarten', 4) -- Kind

SELECT RIGHT('Kindergarten', 6) -- garten

SELECT SUBSTRING('Kindergarten', 3 (position), 5 (length from position == optional argument)) -- nderg / ndergarten

SELECT LOCATE('n', 'Kindergarten') -- 3 [lowercase/uppercase doesn't matter. If n doesn't exist, it will give 0. Sequence of characters - garden, can be searched as well]

SELECT REPLACE('Kindergarten', 'garten', 'garden') -- Kindergarden

SELECT CONCAT('first', 'last') -- firstlast

SELECT CONCAT(first\_name, ' ', last\_name) AS full\_name

FROM customers;

```

- DATE Functions

```sql

SELECT NOW() -- date and time of now

SELECT CURDATE() -- current date

SELECT CURTIME() -- current time

SELECT YEAR(NOW()) -- 2021

SELECT MONTH(NOW()) -- 2 [Feb]

SELECT DAY(NOW()) -- 26

SELECT HOUR(NOW()) -- current hour

SELECT MINUTE(NOW()) -- current minute

SELECT SECOND(NOW()) -- current day

SELECT DAYNAME(NOW()) -- returns String value - Friday

SELECT MONTHNAME(NOW()) -- returns String value - February

SELECT EXTRACT(DAY FROM NOW()) -- use this standard SQL Function

```

- Formatting Dates and Time

```sql

SELECT DATE\_FORMAT(NOW() [date value], '%M %d, %Y' [format string to format the date])

SELECT TIME\_FORMAT(NOW(), '%H:%i:%s')

```

- Calculating Dates and Times

```sql

SELECT DATE\_ADD(NOW(), INTERVAL 1 DAY) -- one day forward; 1 YEAR -> one year forward; 1 MONTH -> one month forward

SELECT DATE\_ADD(NOW(), INTERVAL -1 DAY) -- yesterday

SELECT DATE\_SUB(NOW(), INTERVAL 1 DAY) -- yesterday

SELECT DATEDIFF('2019-01-05', '2019-01-01') -- only returns difference in days not time

SELECT TIME\_TO\_SEC(NOW()) -- number of seconds elapsed since midnight

```

- IFNULL and COALESCE Functions

```sql

SELECT order\_id,

IFNULL(shipper\_id, 'Not assigned') AS shipper

FROM orders

-- If shipper\_id is NULL, return Not assigned string instead.

SELECT order\_id,

COALESCE(shipper\_id, comments, 'Not assigned') AS shipper

FROM orders

-- This function returns the first non-null value in the list.

SELECT CONCAT(first\_name, ' ', last\_name) AS full\_name,

IFNULL(phone, 'Unknown') AS phone\_number

FROM customers

```

- IF function

- IF(expression, first\_value if expression is true, second\_value if expression is false)

```sql

SELECT order\_id,

order\_date,

IF(

YEAR(order\_date) = YEAR(NOW()),

'Active',

'Archived'

) AS status

FROM orders

```

- CASE operator

- IF case operator works for single condition only. In case of multiple condition, we use CASE Operator.

```sql

SELECT CONCAT(first\_name, ' ', last\_name) AS customer,

points,

CASE

WHEN points > 3000 THEN 'Gold'

WHEN points BETWEEN 2000 AND 3000 THEN 'Silver'

ELSE 'Bronze'

END AS category

FROM customers

ORDER BY points DESC

```

### 8. Views

- VIEWS are for reusing a frequent query in future.

- VIEWS act as a virtual table. They don't store data. They provide a view to the underlying table. The data is stored in the table.

- VIEWS simplify queries.

- VIEWS help reduce the impact of changes.

- VIEWS helps enhance the security by restricting access to data.

- CREATE VIEW:

```sql

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

SELECT \*

FROM sales\_by\_client sc

JOIN clients c on sc. client\_id = c.client\_id

-- AND

CREATE VIEW clients\_balance AS

SELECT c.client\_id,

c.name,

SUM(i.invoice\_total - i.payment\_total) AS balance

FROM clients c

JOIN invoices i on c.client\_id = i.client\_id

GROUP BY c.client\_id, c.name

ORDER BY c.client\_id

```

- DROP VIEW:

```sql

DROP VIEW clients\_balance

```

- REPLACE keyword - to recreate the view

```sql

CREATE OR REPLACE VIEW clients\_balance AS

SELECT c.client\_id,

c.name,

SUM(i.invoice\_total - i.payment\_total) AS balance

FROM clients c

JOIN invoices i on c.client\_id = i.client\_id

GROUP BY c.client\_id, c.name

ORDER BY c.client\_id

```

- UPDATABLE VIEWS:

- We can use the updatable views in insert, update, or delete statements.

- We can basically update data through it.

- Views are updatable if they don't have below listed attributes in them:

- DISTINCT keyword

- Aggregate Functions (MIN, MAX, SUM, etc)

- GROUP BY / HAVING

- UNION

- Sometimes, for security reasons, we don't have access to the table directly so this helps us update data through views.

- Any update in VIEWS updates the table.

```sql

CREATE OR REPLACE VIEW invoices\_with\_balance AS

SELECT i.\*,

i.invoice\_total - i.payment\_total AS balance

FROM invoices i

WHERE i.payment\_total > 0;

-- AND

UPDATE invoices\_with\_balance

SET payment\_total = 10

WHERE invoice\_id = 2

```

- WITH CHECK OPTION: This clause prevents the disappearance of rows from views in some cases where they disappear after update. It gives error if the row could be excluded after the update.

```sql

CREATE OR REPLACE VIEW invoices\_with\_balance AS

SELECT i.\*,

i.invoice\_total - i.payment\_total AS balance

FROM invoices i

WHERE i.payment\_total > 0

WITH CHECK OPTION

```

### 9. Stored Procedures

- Stored procedure is an object in MySQL which are function like in essence, that can be called anytime a complex set of queries are needed to be executed.

- They are used to separate the concerns on application end of programming languages [JAVA, Python, C++, PHP, etc.]. Simply call the stored procedure from application end rather than writing SQL queries.

- Enchance data security as we are able to hide the SQL implementation of table structure. We can then call these stored procedures based on user roles for enchance data security.

- We can also pass PARAMETERS:

```sql

DELIMETER $$

CREATE PROCEDURE get\_clients(client\_id INT)

BEGIN

SELECT \*

FROM clients c

WHERE c.client\_id = client\_id;

END $$

DELIMETER ;

-- AND

CALL get\_clients(1);

-- OR IF NO ARGUMENT

DELIMETER $$

CREATE PROCEDURE get\_clients(client\_id INT)

BEGIN

IF client\_id IS NULL THEN

SELECT \* FROM clients

ELSE

SELECT \*

FROM clients c

WHERE c.client\_id = client\_id;

END IF;

END $$

DELIMETER ;

-- OR

DELIMETER $$

CREATE PROCEDURE get\_clients(client\_id INT)

BEGIN

SELECT \*

FROM clients c

WHERE c.client\_id = IFNULL(c.client\_id, client\_id);

END $$

DELIMETER ;

-- AND

CALL get\_clients(NULL);

```

- \*\*PARAMETER VALIDATION :\*\* is done using SIGNAL SQLSTATE & SET\_MESSAGE clauses/keywords. Useful for validating parameter values before modifying/inserting in table.

```sql

DELIMETER $$

CREATE PROCEDURE update\_payment\_by\_client

(

client\_id INT,

payment\_id INT

)

BEGIN

IF payment <= 0 THEN

SIGNAL SQLSTATE '22003' SET\_MESSAGE 'Invalid type'

END IF;

SELECT \*

FROM clients c

WHERE c.client\_id = IFNULL(c.client\_id, client\_id) AND c.payment\_id = IFNULL(c.payment\_id, payment\_id);

END $$

DELIMETER ;

-- SQLSTATE can be checked on IBM SQL site.

```

- \*\*OUT PARAMETER :\*\* are output parameters that need to be stored in variables. Used mainly for application side value return feature.

```sql

DELIMETER $$

CREATE PROCEDURE get\_unpaid\_invoices\_for\_client

(

client\_id INT,

OUT invoices\_count INT,

OUT invoices\_total DECIMAL(9,2)

)

BEGIN

SELECT COUNT(\*), SUM(invoices\_total)

INTO invoices\_count, invoices\_total

FROM invoices i

WHERE i.client\_id = client\_id

AND payment\_total = 0;

END $$

--

SET @invoices\_count = 0

SET @invoices\_total = 0

CALL sql\_invoicing.get\_unpaid\_invoices\_for\_client

(3, @invoices\_count, @invoices\_total);

SELECT @invoices\_count, @invoices\_total;

```

- \*\*User or Session Variables :\*\* These variables remain in the mysql session and decalred by the user. Hence the name.

```sql

SET @invoices\_total = 0;

```

- \*\*LOCAL VARIABLES :\*\* They remain in stored procedure scope, not in the entire mysql session.

```sql

DELIMITER $$

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(invoices\_total)

INTO invoices\_count, invoices\_total

FROM invoices;

SET risk\_factor = invoices\_total/invoices\_count;

SELECT risk\_factor;

END $$

DELIMITER ;

```

- \*\*FUNCTION:\*\* can only return one single value, unlike stored procedure, where they can return multiple result set.

```sql

CREATE FUNCTION get\_risk\_factor(client\_id INT)

RETURNS INTEGER

READS SQL DATA

BEGIN

-- SAME QUERY AS ABOVE

RETURN IFNULL(risk\_factor, 0);

END;

-- DETERMINISTIC: if we give it the same set of values and it always returns the same output for those values. Same output for same input.

-- READS SQL DATA: If we use select statement here.

-- MODIFIES SQL DATA: if we insert/update data here.

SELECT client\_id, name, get\_risk\_factor(client\_id)

FROM clients;

--

DROP FUNCTION IF EXISTS get\_risk\_factor();

```

> ## ADVANCED:

### 10. Triggers

- A block of code that gets executed automatically, after or before, insert, update, and delete sql query statement.

- Two types of triggers:

- Row Level Trigger (available in MySQL): Executes as many times as rows affected.

- Statement Level Trigger (not available in MySQL): Executes per transaction statement.

```sql

DELIMITER $$

CREATE TRIGGER payments\_after\_insert -- tablename\_event(after/before)\_query(insert/update/delete)

AFTER INSERT ON payments -- (AFTER/BEFORE) (INSERT/UPDATE/DELETE) ON (TABLE\_NAME)

FOR EACH ROW

BEGIN

UPDATE invoices

SET payment\_total = payment\_total + NEW.amount

WHERE invoice\_id = NEW.invoice\_id;

-- NEW -> for insert

-- OLD -> for delete and update

END $$

DELIMITER ;

-- NOW running below will update invoices

INSERT INTO payments

(

payment\_id,

client\_id,

invoice\_id,

date,

amount,

payment\_method

)

VALUES (9, 2, 1, '2021-03-02', 10, 1);

--

SHOW TRIGGERS;

--

DROP TRIGGERS IF EXISTS payments\_after\_insert;

```

- We can use triggers for logging the changes made in database. Who made it, when did they made it, and where did they made it, etc.

### 11. Events

- They exists in SQL to run periodic tasks.

- SQL statements schedule by events can be run on periodic basis - yearly, monthly, weekly, daily, hourly, minutely, or in seconds.

```sql

DELIMITER $$

CREATE EVENT yearly\_delete\_stale\_auditions\_rows

ON SCHEDULE

-- AT '2019-05-01'

EVERY 1 YEAR STARTS '2019-03-02' ENDS '2029-03-02'

DO BEGIN

DELETE FROM payments\_audit

WHERE action\_date < NOW() - INTERVAL 1 YEAR;

END $$

DELIMITER ;

--

SHOW EVENTS LIKE 'yearly%';

--

DROP EVENT IF EXISTS '\_name\_';

--

ALTER EVENT '\_name\_' ENABLE/DISABLE;

```

### 12. Transactions

- A group of SQL statements that represents a single unit of work. All the SQL statements should execute successfully or the transaction will fail. For example: CREDIT &amp; DEBIT mechanism. Every time credit happens, debit must also happen somewhere in the process of transaction. Otherwise roll back all changes.

- They have ACID properties:

- \*\*A for Atomicity :\*\* Each transaction is unbreakable, like atoms [old science]. Either all the statements in a transaction is executed successfully and committed or the transaction is rolled back and all the changes are undone.

- \*\*C for Consistency :\*\* Our database will remain consistent throughout. We won't end up with order without an item.

- \*\*I for Isolation :\*\* These transaction are isolated from each other if they try to modify the same data. Other transaction gets locked unless one is done.

- \*\*D for Durability :\*\* Once a transaction is committed, the changes made by the transaction is permanent. Even after the power shortage, it won't change.

```sql

USE sql\_store;

START TRANSACTION;

INSERT INTO orders (customer\_id, order\_date, status)

VALUES (1, '2019-01-01', 1);

INSERT INTO order\_items

VALUES (LAST\_INSERT\_ID(), 1, 1, 1);

COMMIT;

-- AND

USE sql\_store;

START TRANSACTION;

INSERT INTO orders (customer\_id, order\_date, status)

VALUES (1, '2019-01-01', 1);

INSERT INTO order\_items

VALUES (LAST\_INSERT\_ID(), 1, 1, 1);

ROLLBACK; -- when we want to do error checking and manually rollback the transaction

SHOW VARIABLES LIKE 'autocommit';

```

- MySQL wraps every single statement we write inside the transaction and then it do the commit automatically if no error.

13. Concurrency

- When multiple users try to access the same data in order to change it or use it. Like say, updating it at a same time.

- \*\*Problems of concurrency :\*\*

- \*\*LOST UPDATES :\*\* When the later transaction overwrite the former transaction commit when they are working on the same set of data. To prevent it, use LOCK. MySQL uses it automatically.

- \*\*DIRTY READS :\*\* This happens when a transaction reads data that hasn't been committed yet. The solution is READ COMMITTED Isolation such that other transactions can only read each other after being committed.

- \*\*NON-REPEATABLE READS :\*\* Two transactions are in non-repeatable reads when one is reads a value for some computation for the first time, and the other updates the same value while the first is still utilizing the value for its purpose, making the second time utilization happen with different updated value rather than the previous one as needed. In short, if we read same data twice in a transaction but get different results.

- \*\*PHANTOM READS :\*\* When one transaction is being executed, another transaction updates/removes/inserts data that may or may not be required in the first transaction to get accurate results. A "ghost" data appears suddenly during a transaction, thereby getting missed. This is called phantom reads. It's solution depends on bussiness requirements and the only way to solve it is to have SERIALIZABLE Isolation level.

- \*\*Isolation Levels :\*\*

- \*\*READ UNCOMMITED :\*\* doesn't protect us from any of the problems. Fastest.

- \*\*READ COMMITTED :\*\* protects from Dirty Reads.

- \*\*REPEATABLE READ :\*\* protects from Lost Updates, Dirty Reads, Non-Repeatable Reads. Default in MySQL.

- \*\*SERIALIZABLE :\*\* Prevents all the above + phantom reads. Needs extra resources in terms of memory and cpu. Slowest.

```sql

SHOW VARIABLES LIKE 'transaction\_isolation';

SET [SESSION/GLOBAL] TRANSACTION ISOLATION LEVEL [READ UNCOMMITTED/READ COMMITTED/REPEATABLE READ/SERIALIZABLE]

-- empty -> for current transaction

-- SESSION -> for single mysql session and every future transaction that happens during this session

-- GLOBAL -> for all new transaction for all session

```

- \*\*DEADLOCKS :\*\* When two transaction prevent each other from running further as they are both waiting for the other to finish.

To prevent it, the two transaction should have same order of query:

-> T1 -> A, B -> END

-> T2 -> A, B -> END

and not like:

-> T1 -> A, B -> END

-> T2 -> B, A -> END

Also, small scheduled transaction is better.

- \*\*DATA TYPES:\*\*

- \*\*Strings:\*\* CHAR(X) [255B], VARCHAR(X) [64KB], TINYTEXT [255B], TEXT [64KB], MEDIUMTEXT [16MB], LONGTEXT [4GB]

- \*\*Integers:\*\* TINYINT [1B], UNSIGNEDINT [1B], SMALLINT [2B], MEDIUMINT [3B], INT [4B], BIGINT [8B], ZEROFILL -> to zeropad values to have same number of digits. Only affects how they are displayed not how they are stored.

- \*\*Rationals:\*\*

- DECIMAL/DEC/NUMERIC/FIXED (p, s) -> DECIMAL(9,2) = 1234567.89 [9 DIGITS WITH 2 AFTER DECIMAL POINT]: store fixed value monetory values.

- FLOAT/DOUBLE: Store approx. for scientific calculation.

- \*\*Booleans:\*\* 0 -> FALSE, 1 -> TRUE [BOOL/BOOLEAN]

- \*\*Enums:\*\* ENUM('small', 'medium', 'large')

- Only one of the 3 values can be stored, otherwise error will happen.

- Case-insensitive

- BAD for design

- duplication/no-reuse

- change cost is high

- \*\*SET:\*\* Only difference is that multiple values can be stored.

- \*\*DATE/TIME:\*\*

- DATE

- TIME

- DATETIME

- TIMESTAMP

- YEAR

- \*\*BLOB:\*\* For storing binary data like images, videos, pdfs, etc. Better not to use dbs for storing binary files because RDBs are designed to work with structured relational data and not binary data.

- TINYBLOB -> 255 B

- BLOB -> 65 KB

- MEDIUMBLOB -> 16 MB

- LONGBLOB -> 4 GB

- FEW PROBLEMS:

- Increase database size.

- Slower backups

- Performance problems as pulling files out of DBs is slower than file system.

-> More code to read/write images

- \*\*JSON:\*\* Lightweight format for storing and transfering data over the internet.

```sql

UPDATE products

SET properties = JSON\_OBJECT(

'weight', 10,

'dimensions', JSON\_ARRAY(1, 2, 3),

'manufacturer', JSON\_OBJECT('name', 'sony')

) WHERE product\_id = 1;

SELECT

[product\_id, JSON\_EXTRACT(properties, '$.weight') AS weights]

-- OR

[product\_id, properties -> '$.weight' ] -- -> is column pass operator

-- OR

[product\_id, properties -> '$.dimensions[0]'] -- accessing array values of json\_array

-- OR

[product\_id, properties -> '$.manufacturer.name'] -- accessing json\_object within json\_object. This one will give "sony" with double quotes.

-- OR

[product\_id, properties ->> '$.manufacturer.name'] -- this one will give sony as an output without double quotes.

FROM products

WHERE product\_id = 1;

```

- \*\*JSON\_SET:\*\* update existing json\_object's properties with new ones or add new properties altogether.

```sql

UPDATE products

SET properties = JSON\_SET(

-- Object, update, new

properties,

'$.weight', 20,

'$.age', 30

) WHERE product\_id = 1;

```

- \*\*JSON\_REMOVE:\*\* JSON\_REMOVE(json\_object, property to be removed)

- Both JSON\_SET and JSON\_REMOVE modify the json\_object and return json\_objects.

> ## MASTER:

### 14. Designing Databases

- \*\*Data Modelling:\*\* is the process of modelling a data that we want to store. It involves 4 steps:

- \*\*Understanding and analyzing the business requirements\*\*

- \*\*Build a conceptual model of business requirements :\*\* to get a visual representation in order to communicate better with the stake holders. This is representing entities and their relationship with each other.

- \*\*Build a logical model :\*\* Logical models are independent of database technology. We come up with datastructures required in this step.

- \*\*Build a physical model :\*\* This is database technology level designing specific to one database technology.

### 15. Indexing for High Performance

### 16. Securing Databases