LearnFrenzy

1

SQL

INTRODUCTION & DATABASE BASICS

What is SQL?

- SQL (Structured Query Language) is used to communicate with databases.
- It is used to store, retrieve, update and manage data.

Example Databases

- MySQL
- PostgreSQL
- Oracle
- SQL Server
- SQLite
- MariaDB

Features

- Easy to learn and use
- Standard language for databases
- Powerful for data manipulation
- Supports different types of databases

Basic Structure of a Database

Example Table

Table Name : Student

id	name	age	course
1	Amit	21	BCA
2	Neha	22	B.Sc
3	Rahul	20	BBA

Note

A table is a collection of related data in rows and columns.

Key Terms

- Database : Collection of tables
- Table : Collection of related data in rows and columns
- Row : A single record in a table
- Column : A field in a table

Why SQL is Important?

SQL is the foundation of data analysis, data engineering, data science and all data-driven roles.

LF

SQL

2

SQL COMMANDS

SQL commands are categorized into 5 main groups.

1 DDL (Data Definition Language)

Used to define or modify database structure.

Commands: CREATE, ALTER, DROP, TRUNCATE, RENAME

Examples:

```
CREATE TABLE Student (
 id INT, name VARCHAR(50));
ALTER TABLE Student ADD age INT;
DROP TABLE Student;
```

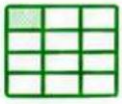


Table Structure

2 DML (Data Manipulation Language)

Used to insert, update, delete and manipulate data in tables.

Commands: INSERT, UPDATE, DELETE, MERGE

Examples:

```
INSERT INTO Student (id, name)
VALUES (1, 'Amit');
UPDATE Student SET age = 21
WHERE id = 1;
DELETE FROM Student WHERE id = 1;
```

Modify Data

3 DCL (Data Control Language)

Used to control access and permissions in the database.

Commands: GRANT, REVOKE

Examples:

```
GRANT SELECT ON Student TO user1;
REVOKE SELECT ON Student FROM user1;
```

Permissions

4 TCL (Transaction Control Language)

Used to manage transactions in the database.

Commands: COMMIT, ROLLBACK, SAVEPOINT

Examples:

```
COMMIT;
ROLLBACK;
SAVEPOINT sp1;
ROLLBACK TO sp1;
```

Manage Transactions

5 DQL (Data Query Language)

Used to retrieve data from the database.

Command: SELECT

Examples:

```
SELECT id, name, age
FROM Student
WHERE age > 18;
```

Retrieve Data

Quick Summary

Type	Full Form	What it does?
DDL	Data Definition Language	Defines structure
DML	Data Manipulation Language	Manipulates data
DCL	Data Control Language	Controls access
TCL	Transaction Control Language	Manages transactions
DQL	Data Query Language	Retrieves data

Tips

- DDL affects structure.
- DML affects data.
- DCL gives/takes permissions.
- TCL ensures safe transactions.
- DQL is what we use most in real-world.

Practice SQL Daily & Become a Pro!

LF

SQL

3

SELECT STATEMENT & WHERE CLAUSE

The SELECT statement is used to retrieve data from one or more tables.
The WHERE clause is used to filter records (rows) based on a condition.

1 SELECT BASICS

- Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

- Example:

```
SELECT id, name, age
FROM Student;
```

Student Table

id	name	age
1	Amit	21
2	Neha	22
3	Rahul	20

* selects all columns from the table. →

```
SELECT * FROM Student;
```

ALIAS (Rename Columns)

- Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

- Example:

```
SELECT name AS student_name, age AS student_age
FROM Student;
```

AS is optional. You can also write without AS.

2 WHERE CLAUSE

- Used to filter records based on a condition.
- Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- Example:

```
SELECT id, name, age
FROM Student
WHERE age > 20;
```

Output

id	name	age
1	Amit	21

Common Operators

Operator	Meaning
=	Equal to
!= or <>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN	Between a range
IN	Match any value in a list
LIKE	Pattern matching
IS NULL	Check for null values

3 MORE EXAMPLES WITH WHERE

1. age >= 20

```
SELECT * FROM Student
WHERE age >= 20;
```

→

id	name	age
1	Amit	21
2	Neha	22

2. name = 'Rahul'

```
SELECT * FROM Student
WHERE name = 'Rahul';
```

→

id	name	age
3	Rahul	20

3. age BETWEEN 20 AND 22

```
SELECT * FROM Student
WHERE age BETWEEN 20 AND 22;
```

→

id	name	age
1	Amit	21
2	Neha	22
3	Rahul	20

4. name IN ('Amit','Rahul')

```
SELECT * FROM Student
WHERE name IN ('Amit','Rahul');
```

→

id	name	age
1	Amit	21
3	Rahul	20

5. name LIKE 'A%'

```
SELECT * FROM Student
WHERE name LIKE 'A%';
```

→

id	name	age
1	Amit	21

6. age IS NULL

```
SELECT * FROM Student
WHERE age IS NULL;
```

→ (returns rows where age is NULL)

TIPS

- ✓ Use WHERE to return only the data you need.
- ✓ Using WHERE improves performance.
- ✓ Combine multiple conditions using AND / OR.

Multiple Conditions

```
SELECT * FROM Student
WHERE age > 18 AND name LIKE 'A%';
```

LF ☆ Master SELECT and WHERE – they are the foundation of SQL! ☺

SQL

4

ORDER BY, GROUP BY & HAVING

- ORDER BY → Used to sort the result set.
- GROUP BY → Used to group rows that have the same values.
- HAVING → Used to filter groups based on a condition.

1 ORDER BY

Used to sort the result set in Ascending (ASC) or Descending (DESC) order.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC | DESC];
```

Example:

```
SELECT id, name, age
FROM Student
ORDER BY age ASC;    -- Ascending
ORDER BY age DESC;   -- Descending
```

Output (ORDER BY age ASC)

id	name	age
3	Rahul	20
1	Amit	21
2	Neha	22

ASC (A to Z)

DESC (Z to A)

Key Points

- Default order is ASC.
- We can sort by multiple columns.

Example (Multiple Columns):

```
SELECT name, age, city
FROM Student
ORDER BY city ASC,  age DESC;
```

First sorted by city (A–Z), then by age (high to low) within each city.

2 GROUP BY

Used to group rows that have the same values summary rows.

Syntax:

```
SELECT column_name,  aggregate_function()
FROM table_name
ORDER BY column_name;
```

Example:

```
SELECT city, COUNT(*) AS total_students
FROM Student
GROUP BY city;
```

Output

city	total_students
Pune	2
Mumbai	1
Delhi	1

When we use GROUP BY, every selected column (in SELECT) must be either in GROUP BY or inside an aggregate function.

Common Aggregate Functions

- COUNT() → Counts rows
- SUM() → Returns total
- AVG() → Returns average
- MIN() → Returns minimum
- MAX() → Returns maximum

3 HAVING

Used to filter groups based on a condition.
(WHERE filters rows, HAVING filters groups)

Syntax:

```
SELECT column_name,  aggregate_function()
FROM table_name
GROUP BY column_name
HAVING condition;
```

Example:

```
SELECT city, COUNT(*) AS total_students
FROM Student
GROUP BY city
HAVING COUNT(*) > 1;
```

Output

city	total_students
Pune	2

This returns only those groups where count is greater than 1.

PRACTICE EXAMPLES

1. List all students ordered by name (A–Z).

```
SELECT * FROM Student ORDER BY name ASC;
```

2. List students ordered by age (high to low).

```
SELECT * FROM Student ORDER BY age DESC;
```

3. Count students in each city.

```
SELECT city, COUNT(*) FROM Student GROUP BY city;
```

4. Show cities having more than 1 student.

```
SELECT city, COUNT(*) FROM Student
GROUP BY city HAVING COUNT(*) > 1;
```

5. Count students in each city having count >= 1 and order by count desc.

```
SELECT city, COUNT(*) AS total FROM Student
GROUP BY city HAVING COUNT(*) >= 1
ORDER BY total DESC;
```

Quick Tips

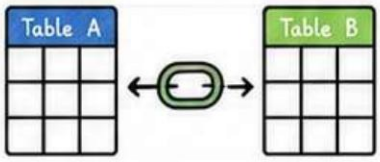
- ✓ Use ORDER BY to sort.
- ✓ Use GROUP BY to create summary.
- ✓ Use HAVING to filter groups.
- ✓ Aggregate functions are very important!

LF Practice daily & become an SQL Expert!

SQL

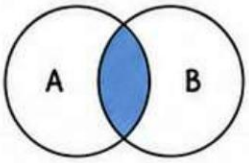
SQL JOINS

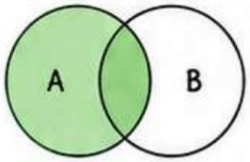
5

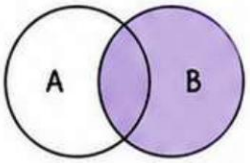


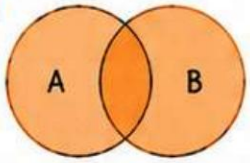
Joins are used to combine rows from two or more tables based on a related column between them.

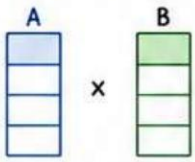
JOIN TYPE	DIAGRAM	SYNTAX	EXAMPLE	DESCRIPTION

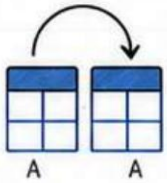
1 INNER JOIN
Returns rows that have matching values in both tables.
		Syntax:
		`SELECT columns`
		`FROM TableA A`
		`INNER JOIN TableB B`
		`ON A.id = B.id;`
			Example:
			`SELECT A.name, B.city`
			`FROM Student A`
			`INNER JOIN Dept B`
			`ON A.dept_id = B.id;`
				• Returns only matching rown
				• Excludes non-matching rows

2 LEFT JOIN
(LEFT OUTER JOIN)
Returns all rows from the left table and matched rows from the right table.
		Syntax:
		`SELECT columns`
		`FROM TableA A`
		`LEFT JOIN TableB B`
		`ON A.id = B.id;`
			Example:
			`SELECT A.name, B.city`
			`FROM Student A`
			`LEFT JOIN Dept B`
			`ON A.dept_id = B.id;`
				• All rows from left table (A)
				• Matching rows from right table (B)
				• Non-matching from B will be NULL

3 RIGHT JOIN
(RIGHT OUTER JOIN)
Returns all rows from the right table and matched rows from the left table.
		Syntax:
		`SELECT columns`
		`FROM TableA A`
		`RIGHT JOIN TableB B`
		`ON A.id = B.id;`
			Example:
			`SELECT A.name, B.city`
			`FROM Student A`
			`RIGHT JOIN Dept B`
			`ON A.dept_id = B.id;`
				• All rows from right table (B)
				• Matching rows from left table (A)
				• Non-matching from A will be NULL

4 FULL OUTER JOIN
Returns all rows when there is a match in either left or right table.
		Syntax:
		`SELECT columns`
		`FROM TableA A`
		`FULL OUTER JOIN TableB B`
		`ON A.id = B.id;`
			Example:
			`SELECT A.name, B.city`
			`FROM Student A`
			`FULL OUTER JOIN Dept B`
			`ON A.dept_id = B.id;`
				• All rows from both tables
				• NULLs for non-matching sides

5 CROSS JOIN
Returns the Cartesian product of both tables (each row of A with each row of B).
		Syntax:
		`SELECT columns`
		`FROM TableA A`
		`CROSS JOIN TableB B;`
			Example:
			`SELECT A.name, B.city`
			`FROM Student A`
			`CROSS JOIN Dept B;`
				• No condition
				• Number of rows = rows in A × rows in B

6 SELF JOIN
A table is joined with itself.
		Syntax:
		`SELECT A.col, B.col`
		`FROM TableA A`
		`JOIN TableA B`
		`ON A.id = B.manager_id;`
			Example:
			`-- Employees & Managers`
			`SELECT A.name AS Employee,`
			`B.name AS Manager`
			`FROM Employee A`
			`JOIN Employee B`
			`ON A.manager_id = B.id;`
				• Useful for hierarchical relationships
				• Manager-Employee mapping

Note: FULL OUTER JOIN is not supported in MySQL. Use UNION of LEFT JOIN and RIGHT JOIN as alternative.

Sample Tables

Student (A)

id	name	dept_id
1	Amit	101
2	Neha	102
3	Rahul	103
4	Pooja	NULL

Dept (B)

id	dept_name
101	IT
102	HR
104	Finance

Interview Tip

- INNER JOIN → Intersection (matches)
- LEFT JOIN → All from Left
- RIGHT JOIN → All from Right
- FULL JOIN → All from Both
- CROSS JOIN → Multiply rows
- SELF JOIN → Table with itself

Quick Note

Always write JOIN with explicit ON condition for better performance and readability.

LF ☆ Understand Joins Perfectly – Write Powerful SQL! ☺

SQL AGGREGATE FUNCTIONS

6

Aggregate functions perform a calculation on a set of values and return a single value.

1 Common Aggregate Functions

Function	Purpose	Example Result
COUNT()	Counts number of rows	10
SUM()	Returns total sum	2500
AVG()	Returns average value	250.5
MIN()	Returns minimum value	100
MAX()	Returns maximum value	500

2 Example Table

Table : Sales

id	product	amount
1	Laptop	500
2	Mobile	300
3	Tablet	200
4	Laptop	400
5	Mobile	600

3 Examples

COUNT() – Count total rows

```
SELECT COUNT(*) AS total_sales
FROM Sales;
```

SUM() – Total amount

```
SELECT SUM(amount) AS total_amount
FROM Sales;
```

AVG() – Average amount

```
SELECT AVG(amount) AS avg_amount
FROM Sales;
```

MIN() – Minimum amount

```
SELECT MIN(amount) AS min_amount
FROM Sales;
```

MAX() – Maximum amount

```
SELECT MAX(amount) AS max_amount
FROM Sales;
```

Note:
Aggregate functions ignore NULL values (except COUNT(*))

4 Aggregate Functions with GROUP BY

Used to get aggregate results for each group.

Example:

```
SELECT product,
       SUM(amount) AS total_amount,
       AVG(amount) AS avg_amount
FROM Sales
GROUP BY product;
```

Output:

product	total_amount	avg_amount
Laptop	900	450
Mobile	900	450
Tablet	200	200

Quick Tips

- ✓ Use COUNT(*) to count all rows.
- ✓ Use GROUP BY with aggregate functions for summary by category.
- ✓ Use HAVING to filter groups (not WHERE).
- ✓ Aggregate functions help in data analysis & reporting.

5 Example with HAVING

Used to filter groups based on aggregate condition.

```
SELECT product, SUM(amount) AS total
FROM Sales
GROUP BY product
HAVING SUM(amount) > 500;
```

Interview Questions

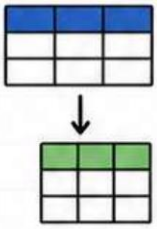
1. What is the difference between WHERE and HAVING?
2. Do aggregate functions consider NULL values?
3. Which aggregate function is used to find total, average?
4. Can we use aggregate functions without GROUP BY?

LF ☆ Master Aggregate Functions – Key for Data Analysis ! ☺

SQL

7

SUBQUERIES & COMMON TABLE EXPRESSIONS (CTEs)



A Subquery is a query inside another query.
CTE (Common Table Expression) is a temporary result set that exists only for the duration of a query.

1 SUBQUERIES

Subqueries can be used in SELECT, FROM, WHERE, HAVING.

a) Subquery in WHERE

Find employees who earn more than the average salary.

```
SELECT name, salary
FROM Employee
WHERE salary > (SELECT AVG(salary)
        FROM Employee);
```

Result

name	salary
Amit	60000
Neha	65000
Rahul	70000

b) Subquery in FROM

Find department-wise average salary and show only departments with avg > 50000.

```
SELECT dept, avg_salary
FROM ( SELECT dept, AVG(salary) AS avg_salary
        FROM Employee
        GROUP BY dept ) AS d
WHERE avg_salary > 50000;
```

c) Subquery in SELECT

Show each employee and the total number of employees in the company.

```
SELECT name, salary,
(SELECT COUNT(*) FROM Employee) AS total_emp
FROM Employee;
```

Result (partial)

name	salary	total_emp
Amit	60000	5
Neha	65000	5
Rahul	70000	5
...	...	...

d) Subquery with EXISTS

Find employees who have completed projects.

```
SELECT name
FROM Employee e
WHERE EXISTS ( SELECT 1
                FROM Project p
                WHERE p.emp_id = e.id );
```

Use subqueries when you need to compare, filter, or derive data based on another query result.

2 COMMON TABLE EXPRESSION (CTE)

CTE improves readability and helps break complex queries into simpler parts.

Syntax:

```
WITH cte_name AS (
  SELECT column1, column2
  FROM table_name
  WHERE condition
)
SELECT * FROM cte_name;
```

Example:

```
-- Find employees with salary > 50000
-- and show department wise count
WITH HighEarners AS (
  SELECT id, name, dept, salary
  FROM Employee
  WHERE salary > 50000
)
SELECT dept, COUNT(*) AS emp_count
FROM HighEarners
GROUP BY dept;
```

Result

dept	emp_count
IT	2
HR	1
Finance	1

Benefits of CTE

- ✓ Improves readability
- ✓ Easier to debug
- ✓ Can be reused
- ✓ Useful for recursive queries
- ✓ Acts like a temporary result set

3 TYPES OF CTE

a) Non-Recursive CTE

Used for simple result sets.

```
WITH cte AS (
  SELECT * FROM Employee
  WHERE salary > 50000
)
SELECT * FROM cte;
```

b) Recursive CTE

Used for hierarchical / tree data.

```
WITH RECURSIVE emp_hierarchy AS (
  SELECT id, name, manager_id
  FROM Employee WHERE manager_id IS NULL
  UNION ALL
  SELECT e.id, e.name, e.manager_id
  FROM Employee e
  JOIN emp_hierarchy eh
    ON e.manager_id = eh.id
)
SELECT * FROM emp_hierarchy;
```

c) Nested CTE

CTE inside another CTE.

```
WITH cte1 AS (
  SELECT dept, AVG(salary) AS avg_sal
  FROM Employee GROUP BY dept
),
cte2 AS (
  SELECT * FROM cte1 WHERE avg_sal > 50000
)
SELECT * FROM cte2;
```

INTERVIEW TIPS

- Subqueries are usually slower on large data.
- CTEs are preferred for complex queries.
- EXISTS is faster than IN in some cases.
- Use CTE for better performance and readability.

Quick Comparison

Feature	Subquery	CTE
Definition	Query inside another query	Temporary named result set
Reusability	Limited	Can be reused
Readability	Harder for complex logic	Easier
Performance	May be slower	Generally better

Remember!

Good SQL is not just about writing queries, but writing smart & readable queries!

LF

Practice Subqueries & CTEs Daily – Think in SQL!

SQL WINDOW FUNCTIONS

8

Window functions perform calculations across a set of table rows that are somehow related to the current row.
They do NOT collapse rows like aggregate functions (GROUP BY).

1 COMMON WINDOW FUNCTIONS

Function	Purpose	Example Use
ROW_NUMBER()	Assigns a unique sequential number to rows.	Top N records, remove duplicates
RANK()	Assigns rank to rows. Same values get same rank, next rank is skipped.	Ranking with gaps
DENSE_RANK()	Assigns rank to rows. Same values get same rank, next rank is not skipped.	Ranking without gaps
LEAD()	Access data from the next row.	Compare with next row
LAG()	Access data from the previous row.	Compare with previous row

KEY POINTS

- ★ OVER() clause defines the window (partition + order).
- ★ PARTITION BY divides rows into groups.
- ★ ORDER BY defines the order within each partition.
- ★ Window functions return a value for EACH ROW.

Window functions are used with OVER() clause.

2 SAMPLE TABLE

Table : Employee

id	name	dept	salary
1	Amit	IT	60000
2	Neha	IT	70000
3	Rahul	IT	70000
4	Pooja	HR	50000
5	Karan	HR	55000

3 EXAMPLES OF WINDOW FUNCTIONS

a) ROW_NUMBER()

```
SELECT id, name, dept, salary,
    ROW_NUMBER() OVER
    (PARTITION BY dept
    ORDER BY salary DESC) AS rn
FROM Employee;
```

id	name	dept	salary	rn
2	Neha	IT	70000	1
3	Rahul	IT	70000	2
1	Amit	IT	60000	3
5	Karan	HR	50000	1
4	Pooja	HR	50000	2

Gives unique number to each row within a department by salary.

b) RANK()

```
SELECT id, name, dept, salary,
    RANK() OVER
    (PARTITION BY dept
    ORDER BY salary DESC) AS rnk
FROM Employee;
```

id	name	dept	salary	rnk
2	Neha	IT	70000	1
3	Rahul	IT	70000	1
1	Amit	IT	60000	3
5	Karan	HR	55000	1
4	Pooja	HR	50000	2

Same salary gets same rank. Next rank is skipped.

c) DENSE_RANK()

```
SELECT id, name, dept, salary,
    DENSE_RANK() OVER
    (PARTITION BY dept
    ORDER BY salary DESC) AS drnk
FROM Employee;
```

id	name	dept	salary	drnk
2	Neha	IT	70000	1
3	Rahul	IT	70000	1
1	Amit	IT	60000	2
5	Karan	HR	55000	1
4	Pooja	HR	50000	2

Same salary gets same rank. Next rank is NOT skipped.

d) LEAD() - Next Row Value

```
SELECT id, name, salary,
    LEAD(salary) OVER
    (PARTITION BY dept ORDER BY salary) AS next_salary
FROM Employee;
```

id	name	salary	next_salary
1	Amit	60000	70000
2	Neha	70000	70000
3	Rahul	70000	NULL
4	Pooja	50000	55000
5	Karan	55000	NULL

Returns value from the next row within the partition.

e) LAG() - Previous Row Value

```
SELECT id, name, salary,
    LAG(salary) OVER
    (PARTITION BY dept ORDER BY salary) AS prev_salary
FROM Employee;
```

id	name	salary	prev_salary
1	Amit	60000	NULL
2	Neha	70000	60000
3	Rahul	70000	70000
4	Pooja	50000	NULL
5	Karan	55000	50000

Returns value from the previous row within the partition.

QUICK TIPS

- ✓ Always use OVER() with window functions.
- ✓ Use PARTITION BY to divide data into groups.
- ✓ Use ORDER BY to define row sequence.
- ✓ Window functions keep the number of rows same.
- ✓ Great for ranking, comparisons & running totals.

INTERVIEW QUESTIONS

1. What is the difference between RANK() and DENSE_RANK()?
2. How does ROW_NUMBER() differ from RANK()?
3. How will you get the second highest salary in each department?
4. How to compare current row with previous and next row values?
5. Can we use window functions without PARTITION BY?

LF ☆ Practice window functions daily - very important for real-world queries! ☺

SQL
VIEWS, INDEXES & STORED PROCEDURES

9

These database objects help improve security, performance and reusability of SQL code.

1 VIEWS

A VIEW is a virtual table based on the result set of a SELECT query.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
CREATE VIEW emp_info AS
SELECT id, name, dept, salary
FROM Employee
WHERE salary > 50000;
```

Benefits

- ✓ Simplifies complex queries
- ✓ Enhances security (hides data)
- ✓ Provides data abstraction
- ✓ Acts like a virtual table

Using View: `SELECT * FROM emp_info;`

Note: Changes in base table will reflect in the view.

2 INDEXES

An INDEX is used to speed up data retrieval operations on a table.

Syntax:

```
CREATE INDEX index_name
ON table_name(column_name);
```

Example:

```
CREATE INDEX idx_emp_name
ON Employee(name);
```

Benefits

- ✓ Improves query performance
- ✓ Faster search and sorting
- ✓ Useful for large tables
- ✗ Slows down INSERT/UPDATE/DELETE (extra maintenance)

Types of Indexes

- Primary Index (default on Primary Key)
- Unique Index (values are unique)
- Composite Index (on multiple columns)
- Clustered Index (sorts the data)
- Non-Clustered Index (separate structure)

Drop Index:

```
DROP INDEX index_name
ON table_name;
```

Interview Tip

Indexes are like the index page of a book. They help the database find data faster without scanning the full table.

3 STORED PROCEDURES

A STORED PROCEDURE is a precompiled collection of one or more SQL statements that can be stored and reused.

Syntax (Create):

```
CREATE PROCEDURE procedure_name
@apcam1 datatype,
@apcam2 datatype
AS
BEGIN
    -- SQL statements
END;
```

Example:

```
CREATE PROCEDURE GetEmpByDept
@dept_id INT
AS
BEGIN
  SELECT id, name, salary
  FROM Employee
  WHERE dept_id = @dept_id;
END;
```

Execute Procedure:

```
EXEC GetEmpByDept @dept_id = 10;
```

Benefits

- ✓ Reusability of code
- ✓ Improves performance
- ✓ Better security
- ✓ Easy maintenance

Types of Stored Procedures

- System Stored Procedures – Built-in by DBMS
- User Defined Stored Procedures – Created by users

Drop Procedure:

```
DROP PROCEDURE procedure_name;
```

INTERVIEW QUESTIONS

1. What is the difference between VIEW and TABLE?
2. How does an INDEX improve performance?
3. What are the disadvantages of too many indexes?
4. What is the difference between Clustered and Non-Clustered Index?
5. What is the use of Stored Procedures?
6. How do you pass parameters to a Stored Procedure?
7. What is the difference between Stored Procedure and Function?

QUICK SUMMARY

Feature	Use
VIEW	Virtual table for simplicity & security
INDEX	Speeds up search and retrieval
STORED PROCEDURE	Reusable SQL code for efficiency

LF ☆ Good SQL = Better Performance + Secure Data + Clean Code! ☺

SQL INTERVIEW CHEAT SHEET

10

Quick Revision Guide for SQL Interview Questions

1 SQL COMMAND CATEGORIES

Category	Commands	Purpose
DDL	CREATE, ALTER, DROP, TRUNCATE	Define and modify database structure
DML	INSERT, UPDATE, DELETE, MERGE	Manipulate data
DQL	SELECT	Query data
DCL	GRANT, REVOKE	Control access
TCL	COMMIT, ROLLBACK, SAVEPOINT	Manage transactions

2 COMMON AGGREGATE FUNCTIONS

Function	Purpose
COUNT()	Counts rows
SUM()	Total sum
AVG()	Average value
MIN()	Minimum value
MAX()	Maximum value

Use with GROUP BY to get aggregated results.

3 COMMON JOINS

Join Type	Returns
INNER JOIN	Matching rows in both tables
LEFT JOIN	All from left, matched from right
RIGHT JOIN	All from right, matched from left
FULL OUTER JOIN	All rows from both tables
CROSS JOIN	Cartesian product
SELF JOIN	Table joined with itself

4 WHERE vs HAVING

WHERE
- Filters individual rows
- Used before GROUP BY
- Cannot use aggregate functions

HAVING
- Filters grouped results
- Used after GROUP BY
- Can use aggregate functions

Example:

```
SELECT dept, COUNT(*) AS total
FROM Employee
WHERE salary > 50000
GROUP BY dept
HAVING COUNT(*) > 2;
```

5 SUBQUERY vs CTE

SUBQUERY
- Query inside another query
- Temporary result
- Used in WHERE, FROM, SELECT
- Can be correlated

CTE (WITH)
- Named temporary result set
- Improves readability
- Can be recursive
- Reusable in query

Exapuery:

```
SELECT name
FROM Employee
WHERE salary > (SELECT AVG(salary)
                FROM Employee);
```

CTE:

```
WITH HighEarners AS (
    SELECT * FROM Employee
    WHERE salary > 50000
)
SELECT COUNT(*) FROM HighEarners;
```

6 WINDOW FUNCTIONS

- Perform calculations across related rows
- Do NOT collapse rows like GROUP BY
- Used with OVER() clause

Function	Purpose
ROW_NUMBER()	Unique row number
RANK()	Rank with gaps
DENSE_RANK()	Rank without gaps
LEAD()	Next row value
LAG()	Previous row value
NTILE(n)	Divide into n groups

Example:

```
SELECT name, salary,
       ROW_NUMBER() OVER (PARTITION BY dept
                          ORDER BY salary DESC) AS rn
FROM Employee;
```

7 INDEXES

- Speed up data retrieval
- Reduce full table scans
- Improves query performance

Types:
- Primary (default, unique)
- Unique (no duplicate)
- Clustered (sorts data)
- Non-Clustered (separate structure)

Example:

```
CREATE INDEX idx_emp_name
ON Employee(name);
```

8 VIEWS

- Virtual table based on a query
- Does not store data
- Improves security & simplicity
- Hides complex logic

Example:

```
CREATE VIEW emp_info AS
SELECT id, name, dept, salary
FROM Employee
WHERE salary > 50000;
```

Usage:

```
SELECT * FROM emp_info;
```

9 STORED PROCEDURES

- Precompiled collection of SQL statements
- Stored in database
- Reusable & secure
- Improves performance

Example:

```
CREATE PROCEDURE GetEmpByDept
@dept_id INT
AS
BEGIN
  SELECT * FROM Employee
  WHERE dept_id = @dept_id;
END;
```

Execute:

```
EXEC GetEmpByDept @dept_id = 10;
```

10 TRIGGERS & FUNCTIONS

Trigger:
- Automatically executes on events (INSERT, UPDATE, DELETE)
- Used for audit, validation, logging

Example:

```
CREATE TRIGGER trg_audit
AFTER INSERT ON Employee
FOR EACH ROW
BEGIN
    -- audit logic
END;
```

Function:
- Returns a single value
- Can be used in SELECT, WHERE
- Reusable business logic

Example:

```
CREATE FUNCTION GetTotal(id INT)
RETURNS DECIMAL(10,2)
BEGIN
  RETURR (SELECT SUM(amount)
          FROM Orders WHERE emp_id = id);
END;
```

QUICK TIPS

- ✓ Use meaningful table and column names.
- ✓ Avoid SELECT * in production.
- ✓ Use indexes on frequently filtered/joined columns.
- ✓ Write readable and well-formatted queries.
- ✓ Understand execution plan for slow queries.
- ✓ Practice with real-world datasets.

COMMON INTERVIEW QUESTIONS

1. What is the difference between WHERE and HAVING?
2. Explain INNER JOIN vs LEFT JOIN.
3. What are window functions? Give use cases.
4. How do indexes improve performance?
5. What is a view? When would you use it?
6. Difference between Stored Procedure and Function?
7. What is a trigger? Give an example use case.

KEY TAKEAWAY

Good SQL is not just about writing queries, but about writing correct, efficient and maintainable queries!

LearnFrenzy ☆ Practice Today = Better Opportunities Tomorrow! ☺