How to Learn in SQL in 2025 – Step by Step

Tip 1: Start with the Basics
Learn fundamental SQL concepts:
- `SELECT`, `FROM`, `WHERE`
- `INSERT`, `UPDATE`, `DELETE`
- Filtering, sorting, and simple aggregations (`COUNT`, `SUM`, `AVG`)

Tip 2: Understand Joins
Joins are essential for combining tables:
- `INNER JOIN` – Only matching rows
- `LEFT JOIN` – All from left table + matches from right
- `RIGHT JOIN` – All from right table + matches from left
- `FULL OUTER JOIN` – Everything

Tip 3: Practice Aggregations & Grouping
- `GROUP BY` and `HAVING`
- Aggregate functions: `SUM()`, `COUNT()`, `AVG()`, `MIN()`, `MAX()`

Tip 4: Work with Subqueries
- Nested queries for advanced filtering
- `EXISTS`, `IN`, `ANY`, `ALL`

Tip 5: Learn Window Functions
- `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`
- `LEAD()` / `LAG()` for analyzing trends and sequences

Tip 6: Practice Data Manipulation & Transactions
- `COMMIT`, `ROLLBACK`, `SAVEPOINT`
- Understand how to maintain data integrity

Tip 7: Explore Indexes & Optimization
- Learn how indexes speed up queries
- Use `EXPLAIN` to analyze query plans

Tip 8: Build Mini Projects
- Employee database with departments
- Sales and inventory tracking
- Customer orders and reporting dashboard

Tip 9: Solve SQL Challenges
- Platforms: LeetCode, HackerRank, Mode Analytics
- Practice joins, aggregations, and nested queries

Tip 10: Be Consistent
- Write SQL daily
- Review queries you wrote before
- Read others' solutions to improve efficiency

SQL Beginner Roadmap

Start Here
Install SQL Server / MySQL / SQLite
Learn How to Run SQL Queries

SQL Basics

- What is SQL?
- Basic SELECT Statements
- Filtering with WHERE Clause
- Sorting with ORDER BY
- Using LIMIT / TOP

Data Manipulation

- INSERT INTO
- UPDATE
- DELETE

Table Management

- CREATE TABLE
- ALTER TABLE
- DROP TABLE

SQL Joins

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL OUTER JOIN

Advanced Queries

- GROUP BY & HAVING
- Subqueries
- Aggregate Functions (COUNT, SUM, AVG)

Practice Projects

- Build a Simple Library DB
- Employee Management System
- Sales Report Analysis

Move to Next Level (Only After Basics)

- Learn Indexing & Performance Tuning
- Stored Procedures & Triggers
- Database Design & Normalization

SQL Basics You Should Know

1. What is SQL?*
***SQL* (Structured Query Language) is used to *communicate with databases*.**
You can use it to *store*, *find*, *update*, or *delete* data in tables.
Think of a *table* like an Excel sheet: rows = data, columns = categories (name, age, etc.)

2. SELECT Statement*
Used to *view data* from a table.

***Example Table – students*:**

| id | name | age |
|----|---------|-----|
| 1 | Anya | 19 |
| 2 | Rahul | 17 |
| 3 | Simran | 20 |

Query:
```sql
SELECT * FROM students;
```

Output:
Returns *all rows and columns* from the table.

3. WHERE Clause*
Used to *filter* data based on a condition.

Query:
```sql
SELECT * FROM students WHERE age > 18;
```

Output:

| id | name | age |
|----|--------|-----|
| 1 | Anya | 19 |
| 3 | Simran | 20 |

4. ORDER BY*
Used to *sort* results by a column.

Query:
```sql
SELECT * FROM students ORDER BY age DESC;
```

Output (sorted by age descending):

| id | name | age |
|----|--------|-----|
| 3 | Simran | 20 |
| 1 | Anya | 19 |
| 2 | Rahul | 17 |

5. LIMIT or TOP
Used to *show limited number* of rows.

MySQL Example:
```sql
SELECT * FROM students LIMIT 2;
```

SQL Server Example:
```sql
SELECT TOP 2 * FROM students;
```

Output:
Returns only the *first 2 rows*.

These are the *core commands* to *view and search* data in SQL.
Practice them on sample tables to get comfortable.

SQL Data Manipulation

Learn how to add, update, and delete data in your tables.

1. INSERT INTO – Add Data

Used to insert new rows into a table.

Syntax:
```
INSERT INTO table_name (column1, column2)
VALUES (value1, value2);
```

Example:
```
INSERT INTO students (name, age)
VALUES ('Alice', 20);
```

2. UPDATE – Modify Existing Data

Used to update existing records in a table.

Syntax:
```
UPDATE table_name
SET column1 = value1
WHERE condition;
```

Example:
```
UPDATE students
SET age = 21
WHERE name = 'Alice';
```

Always use WHERE with UPDATE to avoid changing all rows!

3. DELETE – Remove Data
Used to delete rows from a table.

Syntax:
```
DELETE FROM table_name
WHERE condition;
```

Example:
```
DELETE FROM students
WHERE name = 'Alice';
```

Use WHERE carefully — without it, all data will be deleted!

4. TRUNCATE vs DELETE
- `DELETE`: Removes rows *with* conditions, logs each row
- `TRUNCATE`: Removes *all* rows quickly, no WHERE, cannot be rolled back in some systems

5. INSERT INTO SELECT – Copy Data
```
INSERT INTO new_table (col1, col2)
SELECT col1, col2 FROM old_table
WHERE condition;
```

Tip: Always take a backup before using UPDATE or DELETE.

SQL Table Management

* CREATE TABLE* – *Used to create a new table in the database.*
```sql
CREATE TABLE Students (
 ID INT PRIMARY KEY,
 Name VARCHAR(50),
 Age INT,
 Grade VARCHAR(10)
);
```

* ALTER TABLE* – *Used to modify an existing table structure.*

➢ *Add a new column*
```sql
ALTER TABLE Students ADD Email VARCHAR(100);
```

➢ *Rename a column* *(syntax may vary by SQL version)*
```sql
ALTER TABLE Students RENAME COLUMN Name TO FullName;
```

➢ *Change data type*
```sql
ALTER TABLE Students MODIFY Age SMALLINT;
```

➢ *Drop a column*
```sql
ALTER TABLE Students DROP COLUMN Grade;
```

* DROP TABLE* – *Permanently deletes the entire table and its data.*
```sql
DROP TABLE Students;
```
Caution: This action cannot be undone.

TRUNCATE TABLE – *Removes all rows but keeps the table structure.*
```sql
TRUNCATE TABLE Students;
```

* RENAME TABLE* – *Change the table name.*
```sql
RENAME TABLE Students TO Alumni;
```

DESCRIBE Table – *View the structure of a table.*
```sql
DESCRIBE Students;
```

Pro Tips:
- Always back up data before making structural changes.
- Use `IF EXISTS` or `IF NOT EXISTS` to avoid errors.
```sql
DROP TABLE IF EXISTS Students;
CREATE TABLE IF NOT EXISTS Teachers (...);
```

Today, let's understand SQL JOINS in detail:

SQL JOINs are used to combine rows from two or more tables based on related columns.

* 1. INNER JOIN*
Returns only the matching rows from both tables.

Example:
```sql
SELECT Employees.name, Departments.dept_name
FROM Employees
INNER JOIN Departments
ON Employees.dept_id = Departments.id;
```

Use Case: Employees with assigned departments only.

*** 2. LEFT JOIN (LEFT OUTER JOIN)***
Returns all rows from the *left* table, and matching rows from the *right* table. If no match, returns NULL.

Example:
```sql
SELECT Employees.name, Departments.dept_name
FROM Employees
LEFT JOIN Departments
ON Employees.dept_id = Departments.id;
```

***Use Case:* All employees, even those without a department.**

*** 3. RIGHT JOIN (RIGHT OUTER JOIN)***
Returns all rows from the *right* table, and matching rows from the *left* table. If no match, returns NULL.

Example:
```sql
SELECT Employees.name, Departments.dept_name
FROM Employees
RIGHT JOIN Departments
ON Employees.dept_id = Departments.id;
```

***Use Case:* All departments, even those without employees.**

*** 4. FULL OUTER JOIN***
Returns all rows from both tables. Non-matching rows show NULL.

Example:
```sql
SELECT Employees.name, Departments.dept_name
FROM Employees
FULL OUTER JOIN Departments
ON Employees.dept_id = Departments.id;
```

***Use Case:* See all employees and departments, matched or not.**

Tips:
- Always specify the join condition (`ON`)
- Use table aliases to simplify long queries
- NULLs can appear if there's no match in a join

Advanced SQL Queries

GROUP BY & HAVING
- *GROUP BY* groups rows sharing a value to perform aggregate calculations.
- *HAVING* filters groups based on conditions (like WHERE but for groups).

Example:
Find total sales per product with sales > 1000:
```sql
SELECT product_id, SUM(sales) AS total_sales

FROM sales_data
GROUP BY product_id
HAVING SUM(sales) > 1000;
```

Subqueries
- A query inside another query. Useful for filtering or calculating values dynamically.

Example:
Get customers who placed orders over 500:
```sql
SELECT customer_id, order_id, amount
FROM orders
WHERE amount > (SELECT AVG(amount) FROM orders);
```

Aggregate Functions
- Perform calculations on sets of rows:
- `COUNT()` counts rows
- `SUM()` adds numeric values
- `AVG()` calculates average
- `MAX()` and `MIN()` find extremes

Example:
Find average order amount per customer:
```sql
SELECT customer_id, AVG(amount) AS avg_order
FROM orders
GROUP BY customer_id;
```

Complex Joins with Filtering
- Join tables and filter results in one query.

Example:
List customers with orders over100:
```sql
SELECT c.customer_name, o.order_id, o.amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.amount > 100;
```

SQL Interviews LOVE to test you on Window Functions. Here's the list of 7 most popular window functions

*** RANK() - gives a rank to each row in a partition based on a specified column or value**

*** DENSE_RANK() - gives a rank to each row, but DOESN'T skip rank values**

*** ROW_NUMBER() - gives a unique integer to each row in a partition based on the order of the rows**

*** LEAD() - retrieves a value from a subsequent row in a partition based on a specified column or expression**

*** LAG() - retrieves a value from a previous row in a partition based on a specified column or expression**

*** NTH_VALUE() - retrieves the nth value in a partition**

SQL Window Functions – Part 1:

What Are Window Functions?
They perform calculations across rows related to the current row without reducing the result set. Common for rankings, comparisons, and totals.

1. RANK()
Assigns a rank based on order. Ties get the same rank, but next rank is skipped.

Syntax:
```sql
RANK() OVER (
  PARTITION BY column
  ORDER BY column
)
```

Example Table: Sales

Employee	Region	Sales
A	East	500
B	East	600
C	East	600
D	East	400

Query:
```sql
SELECT Employee, Sales,
RANK() OVER (PARTITION BY Region ORDER BY Sales DESC) AS Rank
FROM Sales;
```

Result:

Employee	Sales	Rank
B	600	1
C	600	1
A	500	3
D	400	4

2. DENSE_RANK()
Same logic as RANK but does not skip ranks.

Query:
```sql
SELECT Employee, Sales,
DENSE_RANK() OVER (PARTITION BY Region ORDER BY Sales DESC) AS DenseRank
FROM Sales;
```

Result:
Employee	Sales	DenseRank
B	600	1
C	600	1
A	500	2
D	400	3

RANK vs DENSE_RANK
- RANK skips ranks after ties. Tie at 1 means next is 3
- DENSE_RANK does not skip. Tie at 1 means next is 2

Use RANK when position gaps matter
Use DENSE_RANK for continuous ranking

SQL Window Functions – Part 2: ROW_NUMBER()

What Is ROW_NUMBER()?
It assigns a unique sequential number to each row within a partition, based on the specified order.
Unlike `RANK()` or `DENSE_RANK()`, it doesn't give equal ranks to ties.

Syntax:
```sql
ROW_NUMBER() OVER (
  PARTITION BY column
  ORDER BY column
)
```

Example Table: Sales
Employee	Region	Sales
A	East	500
B	East	600
C	East	600
D	East	400

Query:
```sql
SELECT Employee, Sales,
ROW_NUMBER() OVER (PARTITION BY Region ORDER BY Sales DESC) AS RowNum
FROM Sales;
```

Result:
Employee	Sales	RowNum
B	600	1
C	600	2
A	500	3
D	400	4

Key Points:
- ROW_NUMBER gives a unique number to each row, even if values are the same

- Useful for pagination, selecting top N per group, or deduplicating rows

Use Case Example: Get Top Seller Per Region
```sql
SELECT *
FROM (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY Region ORDER BY Sales DESC) AS rn
  FROM Sales
) AS ranked
WHERE rn = 1;
```
Use ROW_NUMBER when every row needs a unique rank

SQL Window Functions – Part 3: LEAD & LAG

What Are LEAD and LAG?
They let you access data from the next or previous row without joins or subqueries. Great for comparisons over a sequence.

1. LAG()
Fetches data from the *previous* row in the window.

Syntax:
```sql
LAG(column, offset, default) OVER (
  PARTITION BY column
  ORDER BY column
)
```

Example:
```sql
SELECT Employee, Sales,
LAG(Sales) OVER (ORDER BY Sales DESC) AS Prev_Sales
FROM Sales;
```

Result:
Employee | Sales | Prev_Sales
B | 600 | NULL
A | 500 | 600
D | 400 | 500

2. LEAD()
Fetches data from the *next* row in the window.

Syntax:
```sql
LEAD(column, offset, default) OVER (
  PARTITION BY column
  ORDER BY column
)
```

Example:

```sql
SELECT Employee, Sales,
LEAD(Sales) OVER (ORDER BY Sales DESC) AS Next_Sales
FROM Sales;
```

Result:
Employee | Sales | Next_Sales
B | 600 | 500
A | 500 | 400
D | 400 | NULL

Use Cases:
• Compare current row with previous or next
• Track changes or trends
• Detect outliers or drops

SQL Window Functions – Part 4: NTH_VALUE()

What Is NTH_VALUE()?
Returns the N-th row's value *within a window frame*. Useful for comparing current rows with a specific ranked row.

Syntax:
```
NTH_VALUE(column, N) OVER (
  PARTITION BY column
  ORDER BY column
  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
```

Example Table: Sales
Employee	Region	Sales
A	East	500
B	East	600
C	East	550
D	East	400

SQL Query: Get 2nd highest Sales per Region
```
SELECT Employee, Sales,
NTH_VALUE(Sales, 2) OVER (
  PARTITION BY Region
  ORDER BY Sales DESC
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
) AS Second_Highest
FROM Sales;
```

Result:
Employee	Sales	Second_Highest
B	600	550
C	550	550

| A | 500 | 550 |
| D | 400 | 550 |

Tips:
- *Use ORDER BY DESC* for N-th highest
- Must set *ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING* to access full partition
- Unlike `FIRST_VALUE` or `LAST_VALUE`, `NTH_VALUE` picks a specific rank

Combine with `PARTITION BY` for group-wise calculations