

TRACK 2

SQL for Data Analysis

Relational databases, ER diagrams and keys, DDL/DML, filtering, aggregation, joins, set operators, subqueries, window functions, CTEs, views, procedures and performance.

15 lessons

Version 2026.10

Updated October 2026

Created by **Varun Kumar**
www.skillnavora.in

Contents

Every entry is a clickable link. The PDF also has bookmarks for each lesson and section.

- [01 Databases, RDBMS & SQL Basics](#)
- [02 ER Diagrams, Relationships, Keys & Normalisation](#)
- [03 Data Types, Operators, NULLs, Variables & Comments](#)
- [04 DDL, DML & Constraints: CREATE, ALTER, INSERT, UPDATE, DELETE](#)
- [05 SELECT, WHERE, ORDER BY & Filtering Operators](#)
- [06 SQL Functions: Aggregate, Text, Date & Number](#)
- [07 GROUP BY, HAVING, CASE WHEN, ROLLUP, CUBE & COALESCE](#)
- [08 Joins and a Retail Case Study](#)
- [09 Set Operators: UNION, UNION ALL, INTERSECT & EXCEPT](#)
- [10 Subqueries, Derived Tables & EXISTS](#)
- [11 OFFSET-FETCH and Window Functions I: OVER, PARTITION BY & Ranking](#)
- [12 Window Functions II: Running Totals, LEAD, LAG & Frames](#)
- [13 CTEs, Views, Temporary Tables, Stored Procedures & Functions](#)
- [14 Transactions, Indexes, Performance & Security](#)
- [15 End-to-End SQL Case Study & Interview Practice](#)

Databases, RDBMS & SQL Basics

Understand what a database is, how relational databases organise data into tables, the vocabulary of SQL, the four families of SQL statements (DDL, DML, DCL, TCL), how SQL compares with Excel, and how to install SQL Server and SSMS and run your first query.

● SQL class 1

✦ SQL Server 2022 / SSMS 21, 2026

Beginner

⌚ 100 min

By the end of this lesson you can

- Explain what a database, a DBMS and an RDBMS are and why companies use them instead of spreadsheets
- Use the core vocabulary (table, row/record, column/field, schema, query, result set)
- Classify SQL statements into DDL, DML, DQL, DCL and TCL
- Compare Excel and SQL and know when to use each
- Install SQL Server Developer edition and SSMS, connect, create a database and run a first SELECT

What is a database?

A **database** is an organised collection of data stored so that it can be found, updated and analysed efficiently and safely by many people at the same time. A **DBMS** (database management system) is the software that manages it. Your bank balance, every Swiggy order, every Aadhaar-linked record and every hospital appointment lives in a database.

Why companies use databases

Spreadsheet (Excel)

- One user edits at a time (mostly)
- ≈1 million rows per sheet
- Easy to break: typing over formulas
- Little control over who sees what
- Great for quick analysis and reports

Database (SQL Server, PostgreSQL...)

- Thousands of users at the same time
- Billions of rows
- Rules (constraints) protect data quality
- Fine-grained permissions and audit
- Backups, recovery and transactions

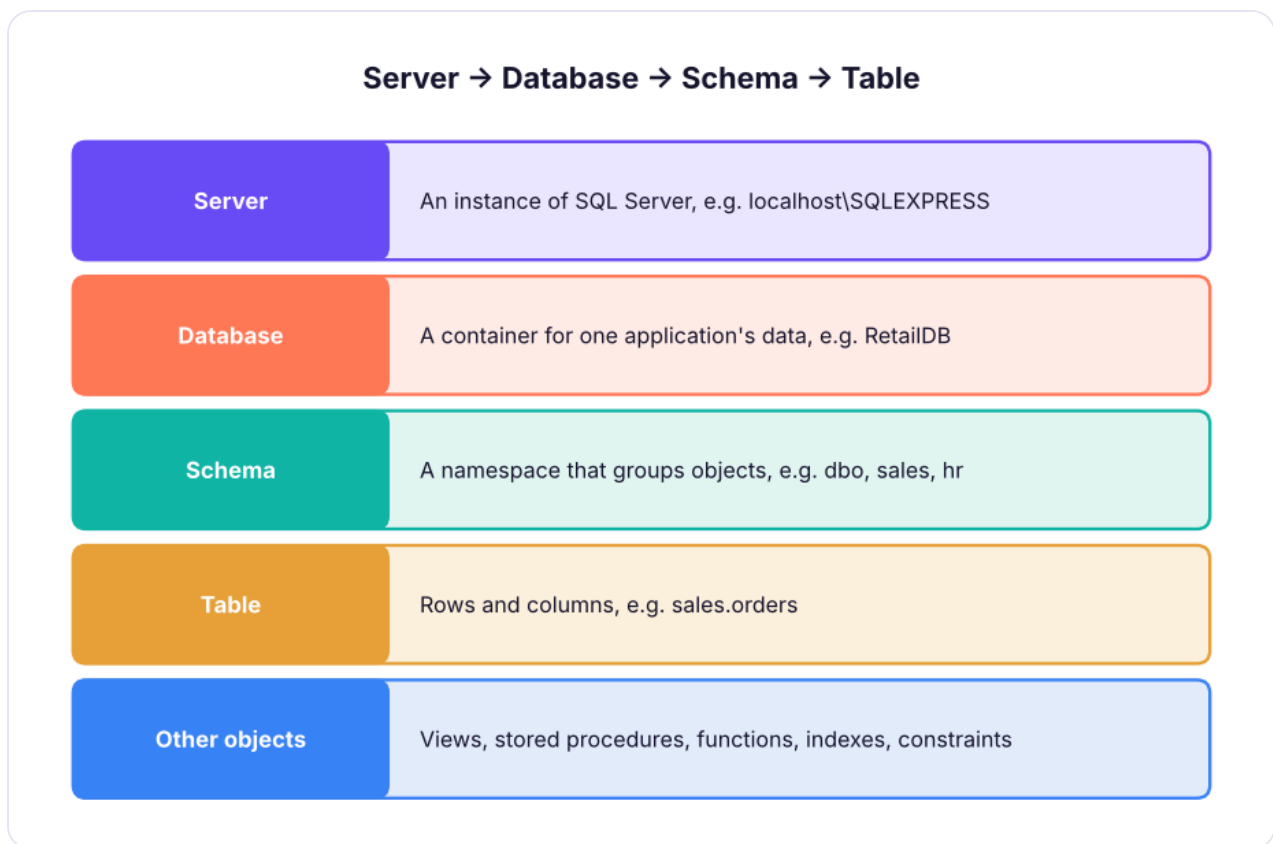
Spreadsheets are personal tools; databases are shared systems built for scale, safety and many users.

Types of databases

Type	Organises data as	Examples	Typical use
Relational (RDBMS)	Tables with rows and columns, linked by keys	SQL Server, PostgreSQL, MySQL, Oracle, SQLite	Orders, payments, HR, finance: most business data
Document	JSON-like documents	MongoDB, Cosmos DB	Product catalogues, app content
Key-value	Key → value pairs	Redis, DynamoDB	Caching, sessions
Column-family / warehouse	Columns, built for analytics	Snowflake, BigQuery, Redshift, Databricks SQL	Analytics on huge data
Graph	Nodes and edges	Neo4j	Social networks, fraud rings
Vector	Number vectors (embeddings)	pgvector, Pinecone	AI search and RAG (GenAI track)

This track focuses on **relational databases** and **SQL**, the language almost every data role uses daily. You will practise in **Microsoft SQL Server** with **SSMS**, as in class. The SQL you learn transfers to PostgreSQL, MySQL, Snowflake and BigQuery with small differences in functions.

How a relational database organises data



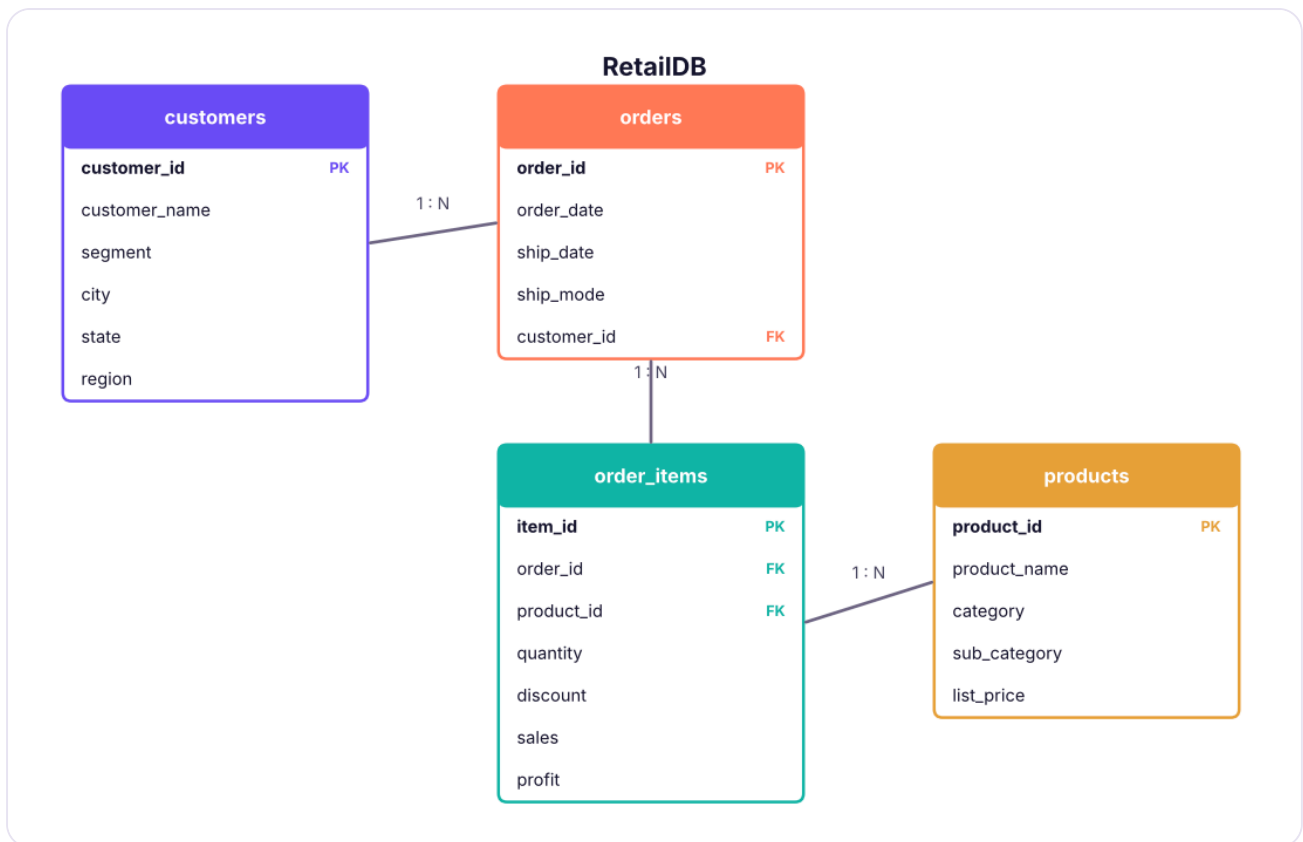
The hierarchy inside SQL Server: a server holds databases, a database holds schemas, a schema holds tables and other objects.

The vocabulary

DEFINITION Terms you will use every day

Table (relation): a set of rows about one kind of thing, such as customers or orders. **Row (record, tuple):** one item, such as one customer. **Column (field, attribute):** one property, such as `city`. Every value in a column has the same data type. **Primary key:** the column (or columns) that uniquely identifies each row, such as `customer_id`. **Foreign key:** a column that refers to the primary key of another table, linking them. **Schema:** (1) the structure of the database (tables, columns, types, relationships); (2) in SQL Server, a named container of objects such as `dbo`. **Query:** a request for data written in SQL. Its output is a **result set**. **SQL:** Structured Query Language, the standard language for relational databases (ANSI/ISO standard since 1986–87). Pronounced "S-Q-L" or "sequel". **T-SQL:** Transact-SQL, Microsoft SQL Server's dialect with extra features (variables, TOP, procedures).

Here is the course's retail database. The order data is split into related tables instead of one wide sheet:



The course retail database. Each order belongs to one customer; each order has many order items; each item refers to one product.

Your first queries

A `SELECT` statement asks for columns `FROM` a table. Here are the first rows of the `customers` table:

SQL

```
SELECT TOP 6 customer_id, customer_name, segment, city, region
FROM customers;
```

RESULT

customer_id	customer_name	segment	city	region
C0001	Saanvi Khan	Corporate	Chandigarh	North
C0002	Meera Bose	Corporate	Delhi	North
C0003	Riya Shah	Consumer	Kolkata	East
C0004	Diya Fernandes	Corporate	Indore	West
C0005	Neha Sharma	Corporate	Hyderabad	South
C0006	Vikram Mehta	Consumer	Chandigarh	North

6 rows × 5 columns

How many rows are in each table?

SQL

```
SELECT 'customers' AS table_name, COUNT(*) AS row_count FROM customers
UNION ALL SELECT 'orders', COUNT(*) FROM orders
UNION ALL SELECT 'order_items', COUNT(*) FROM order_items
UNION ALL SELECT 'products', COUNT(*) FROM products;
```

RESULT

table_name	row_count
customers	800
orders	3200
order_items	6323
products	78

4 rows × 2 columns

A taste of what is coming: total sales by region in one statement (Lesson 7 explains GROUP BY):

SQL

```
SELECT c.region,  
       COUNT(DISTINCT o.order_id) AS orders,  
       ROUND(SUM(oi.sales), 0)    AS total_sales  
FROM customers AS c  
JOIN orders AS o      ON o.customer_id = c.customer_id  
JOIN order_items AS oi ON oi.order_id = o.order_id  
GROUP BY c.region  
ORDER BY total_sales DESC;
```

RESULT

region	orders	total_sales
South	831	53,952,769
West	811	51,958,521
North	792	49,018,161
East	766	46,484,119

4 rows × 3 columns

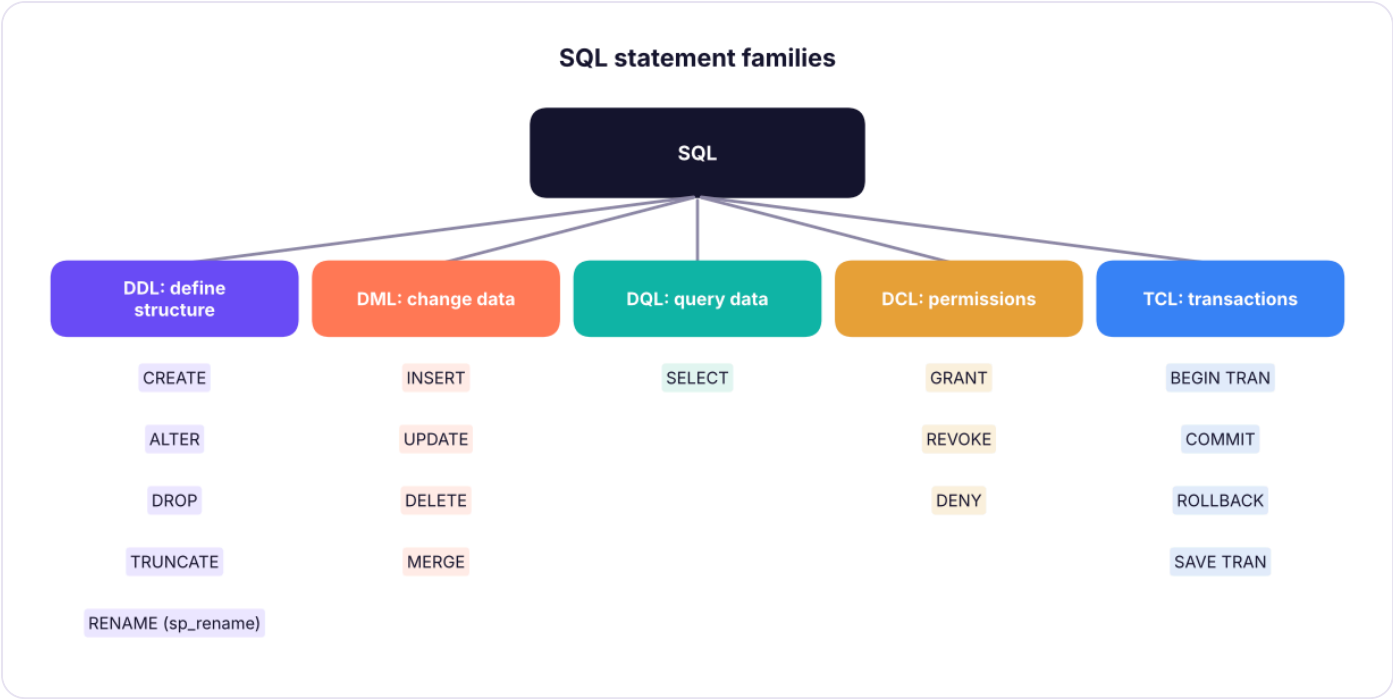
TIP SQL reads almost like English

"Select the region, the number of distinct orders and the total sales, from customers joined to orders and order items, grouped by region, ordered by total sales descending." That readability is why SQL has lasted 50 years.

Properties of SQL

1. **Declarative:** you say *what* you want, not *how* to get it. The database's **query optimiser** decides the fastest way (which index, which join order).
2. **Set-based:** statements work on whole sets of rows at once, not one row at a time like a loop.
3. **Not case-sensitive for keywords:** select, SELECT and Select are the same. By convention, keywords are written in UPPERCASE. In SQL Server, text comparisons are usually case-insensitive too (default collation).
4. **Whitespace does not matter:** you can split a query over many lines; indent it for readability.
5. **Statements end with a semicolon ;** (optional in SQL Server for most statements, but a good habit).
6. **Standard, with dialects:** the core is an ANSI/ISO standard; each product adds its own functions (T-SQL, PL/pgSQL, MySQL).

The families of SQL statements



SQL statements fall into five families.

Family	Full name	What it does	Example
DDL	Data Definition Language	Creates and changes the structure : databases, tables, columns	CREATE TABLE customers (...)
DML	Data Manipulation Language	Adds, changes and removes rows	INSERT INTO customers VALUES (...)
DQL	Data Query Language	Reads data	SELECT * FROM customers
DCL	Data Control Language	Gives and removes permissions	GRANT SELECT ON customers TO analyst
TCL	Transaction Control Language	Groups changes so they succeed or fail together	BEGIN TRAN ... COMMIT

Many books put SELECT inside DML; others separate it as DQL. Both are fine in interviews if you explain them.

Excel vs SQL

Task	Excel	SQL
See the data	Open the sheet	SELECT * FROM table
Filter rows	AutoFilter	WHERE
Sort	Sort dialog	ORDER BY
New calculated column	Formula in a new column	Expression in SELECT
Totals by group	Pivot table / SUMIFS	GROUP BY with SUM
Filter groups	Pivot value filter	HAVING
Lookup from another table	VLOOKUP / XLOOKUP	JOIN
Stack two tables	Copy-paste / Power Query Append	UNION ALL
Remove duplicates	Remove Duplicates	DISTINCT
Rank / running total	RANK, SUM(\$B\$2:B2)	Window functions RANK() OVER, SUM() OVER
Data size	~1 million rows per sheet	Billions of rows
Repeatability	Manual steps (unless Power Query)	Saved queries, views, procedures
Sharing	Files get copied	One source of truth with permissions

Use Excel for quick, small, ad-hoc analysis and presentation; use SQL to get, combine and summarise data at source. In practice analysts use both: SQL to extract the right data, Excel or Power BI to present it.

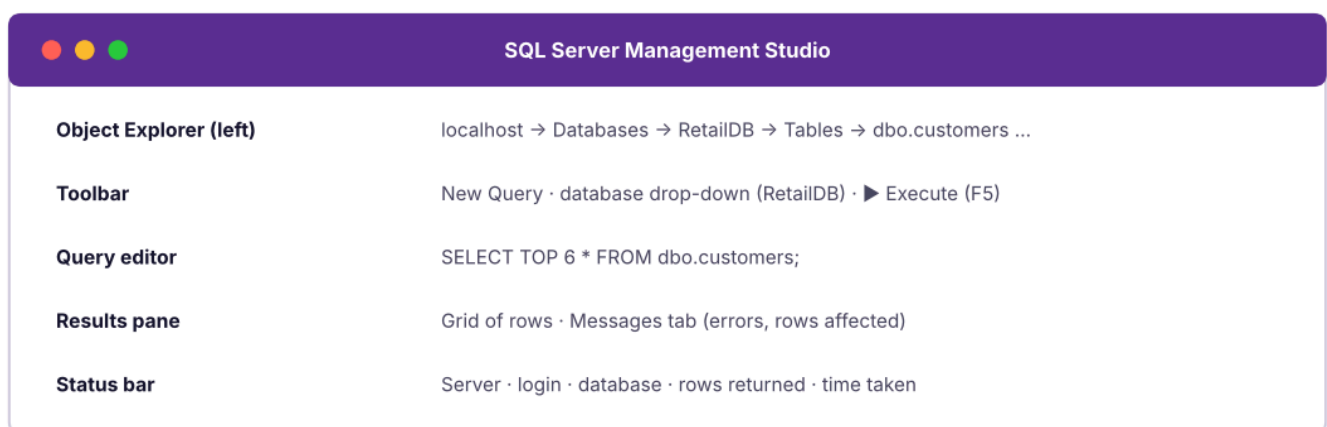
Installing SQL Server and SSMS

STEP BY STEP Install on Windows (free)

1. Download **SQL Server 2022 Developer** edition (free, full-featured, for development and learning) from Microsoft's SQL Server downloads page. Choose **Basic** installation. Express edition is also free but limited to 10 GB per database.
2. Note the instance name shown at the end (for example MSSQLSERVER = default, or SQLEXPRESS).
3. Download and install **SQL Server Management Studio (SSMS)** from Microsoft Learn ("Download SSMS"). SSMS is the editor you write queries in.
4. Open SSMS → **Connect to Server**: Server type *Database Engine*; Server name *localhost* (or *localhost\SQLEXPRESS*); Authentication *Windows Authentication*; tick **Trust server certificate** → Connect.
5. In Object Explorer you will see Databases, Security and more. Click **New Query** (Ctrl + N).

NOTE Mac or Linux?

SQL Server runs in **Docker** on Mac and Linux (mcr.microsoft.com/mssql/server image). Connect with **VS Code + the MSSQL extension** or Azure Data Studio's successor tools. Alternatively, practise the same SQL in **PostgreSQL** (pgAdmin) or **SQLite** (DB Browser); only a few functions differ.



The SSMS window: Object Explorer on the left, the query editor in the middle, results below.

Create a database and load data

SQL

```
-- 1. Create the database and switch to it
CREATE DATABASE RetailDB;
GO
USE RetailDB;
GO
-- 2. Check which database you are in
SELECT DB_NAME() AS current_database;
```

To load the course CSV files: right-click the database → **Tasks** → **Import Flat File** → choose `customers.csv` → check the column names and data types in the preview (make IDs `nvarchar`, dates `date`, amounts `decimal(12,2)`) → Finish. Repeat for the other files. Lesson 4 shows how to create tables with SQL and load them with `BULK INSERT`.

Running queries in SSMS

Action	Shortcut
Execute the whole window (or the selected text only)	F5 or Ctrl + E
New query window	Ctrl + N
Comment / uncomment selected lines	Ctrl + K, Ctrl + C / Ctrl + K, Ctrl + U
Show/hide results pane	Ctrl + R
IntelliSense refresh (after creating tables)	Ctrl + Shift + R
Results to grid / text / file	Ctrl + D / Ctrl + T / Ctrl + Shift + F
Estimated execution plan	Ctrl + L

TIP Select a part, run a part

Highlight just one statement and press F5 to run only that statement. This is how you test a long script step by step.

Summary

KEY TAKEAWAYS

- A **database** stores shared data safely; an **RDBMS** stores it in related **tables** of **rows** and **columns**.
- **Primary keys** identify rows; **foreign keys** link tables.
- SQL is declarative and set-based; keywords are not case-sensitive; statements end with `;`.
- Statement families: **DDL** (structure), **DML** (rows), **DQL** (SELECT), **DCL** (permissions), **TCL** (transactions).
- Excel is for quick analysis and presentation; SQL is for getting, combining and summarising data at scale.
- Practise in SQL Server Developer + SSMS (or Docker / PostgreSQL on Mac).

Quick quiz

6 questions · tap an answer to check it

1. Which statement family does CREATE TABLE belong to?

A DDL

B DML

C DCL

D TCL

Answer: A. DDL defines structure: CREATE, ALTER, DROP, TRUNCATE.

2. In a relational database, what links the orders table to the customers table?

A A merged cell

B A foreign key in orders that refers to the customer's primary key

C A pivot cache

D A shared file name

Answer: B. orders.customer_id is a foreign key that refers to customers.customer_id.

3. What does "SQL is declarative" mean?

- A SQL must declare every variable
- B You describe what result you want and the database decides how to get it
- C SQL statements must be in capitals
- D SQL can only define tables

Answer: B. The optimiser chooses the execution plan; you write the logic.

4. Which is NOT an advantage of a database over a spreadsheet?

- A Many users at the same time
- B Permissions on who sees what
- C Constraints that protect data quality
- D It is always faster to build a quick one-off chart

Answer: D. For quick one-off charts, Excel is usually faster. Databases win on scale, safety and sharing.

5. GRANT SELECT ON orders TO analyst_role is an example of:

- A DDL
- B DML
- C DCL
- D TCL

Answer: C. DCL controls permissions.

6. In SSMS, what does pressing F5 with part of the script highlighted do?

A Runs only the highlighted statement(s)

B Runs the whole script

C Formats the code

D Opens the execution plan

Answer: A. Highlighting limits execution to the selected text.

EXERCISE Map the vocabulary

For the orders table in RetailDB, name: (a) one row; (b) one column; (c) the primary key; (d) the foreign key and the table it refers to; (e) the statement family of `INSERT INTO orders ...`

Show solution (worked answer)

(a) One order, for example the row for IN-2023-00001. (b) Any field, for example ship_mode.
(c) order_id. (d) customer_id, which refers to customers.customer_id. (e) DML.

INTERVIEW CORNER Questions you may be asked

Q: What is the difference between DBMS and RDBMS? A DBMS manages data in any form; an RDBMS stores data in tables with relationships defined by keys, enforces integrity with constraints and uses SQL. Examples: SQL Server, PostgreSQL, MySQL, Oracle.

Q: Explain DDL, DML, DCL and TCL with examples. DDL changes structure (CREATE, ALTER, DROP, TRUNCATE); DML changes rows (INSERT, UPDATE, DELETE); DCL manages permissions (GRANT, REVOKE); TCL manages transactions (COMMIT, ROLLBACK, SAVE TRANSACTION). SELECT is sometimes called DQL.

Q: Why would a company keep data in SQL Server instead of Excel files? Concurrency, scale, data integrity (constraints, transactions), security and auditing, backup and recovery, and a single source of truth for reports and applications.

ER Diagrams, Relationships, Keys & Normalisation

Design databases the way professionals do. Draw ER diagrams, read crow's-foot notation, model one-to-one, one-to-many and many-to-many relationships, master every type of key (super, candidate, primary, alternate, composite, foreign, surrogate) and normalise tables to 3NF.

● SQL class 1

★ 2026

Beginner

🕒 110 min

By the end of this lesson you can

- Read and draw an ER diagram with entities, attributes and relationships
- Identify 1:1, 1:N and M:N relationships and resolve M:N with a junction table
- Define super, candidate, primary, alternate, composite, foreign, surrogate and unique keys, with examples
- Explain referential integrity and what happens on delete/update of a parent row
- Normalise a flat table into 1NF, 2NF and 3NF, and know when to denormalise

Why design matters

A database that is designed badly is slow, wastes space and, worst of all, lets data contradict itself. Imagine the customer's city stored in every order row: if the customer moves, you must update thousands of rows, and missing one creates two different "truths". Good design stores **each fact once** and connects tables with **keys**.

Entity–Relationship (ER) diagrams

An **ER diagram** is a picture of the data model before any table is created.

DEFINITION Building blocks

Entity: a real-world thing you store data about: Customer, Order, Product, Employee. Becomes a **table**. **Attribute:** a property of an entity: name, city, price. Becomes a **column**. **Relationship:** how entities are associated: a Customer *places* Orders. Implemented with **foreign keys**.

Cardinality: how many of one entity relate to another: one-to-one, one-to-many, many-to-many. **Participation (optionality):** whether the relationship is mandatory (every order must have a customer) or optional (a customer may have zero orders).

Crow's-foot notation

The most common notation in industry draws symbols at the ends of each relationship line:

Exactly one



min 1, max 1

Zero or one



min 0, max 1

One or many



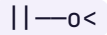
min 1, max many

Zero or many



min 0, max many

Crow's-foot symbols. Read each end as "minimum ... maximum".

Reading example: Customer  Order means "each order belongs to **exactly one** customer; each customer places **zero or many** orders".

Types of relationships

One-to-one (1:1)

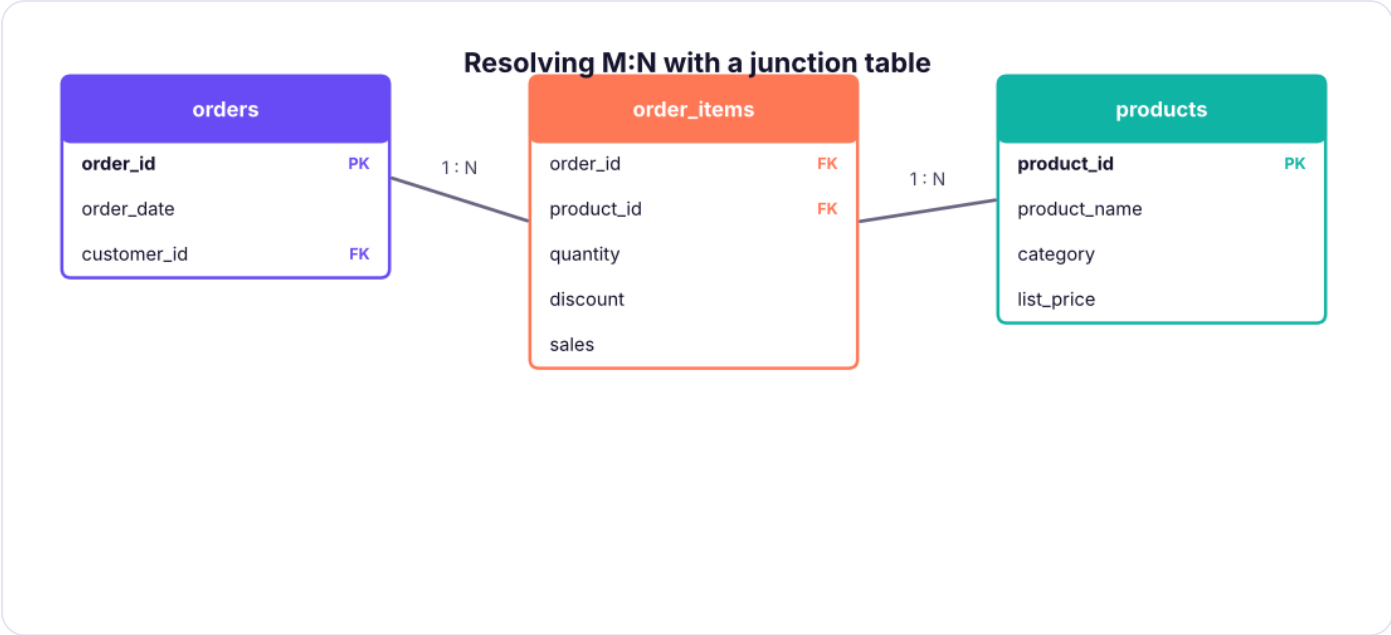
Each row in A matches at most one row in B. Example: Employee ↔ Employee_Bank_Details. Often used to split sensitive or rarely used columns into a separate, more protected table. Implemented with a foreign key that is also **unique** (often the same as the primary key).

One-to-many (1:N) — the most common

One row in A matches many rows in B; each B row matches one A row. Examples: Customer → Orders, Department → Employees, Category → Products. The **foreign key goes on the "many" side** (`orders.customer_id`).

Many-to-many (M:N)

Many rows in A match many in B: Students ↔ Courses, Orders ↔ Products, Actors ↔ Movies. Relational tables cannot store this directly, so we add a **junction (bridge, associative) table** that holds a pair of foreign keys, turning one M:N into two 1:N relationships.



An order contains many products and a product appears in many orders. The junction table `order_items` resolves the many-to-many relationship and stores facts about each pair (quantity, discount).

Relationship	Example	Where the foreign key goes
1 : 1	Employee – Passport	Either side, marked UNIQUE
1 : N	Department – Employees	On the N side (<code>employees.dept_id</code>)
M : N	Students – Courses	In a junction table <code>enrolments(student_id, course_id)</code>
Self (recursive)	Employee – Manager	In the same table: <code>employees.manager_id</code> → <code>employees.emp_id</code>

Let's check the real cardinalities in our retail data with SQL. How many orders does each customer have, and how many items does each order have?

SQL

```
SELECT MIN(n_orders) AS min_orders_per_customer,  
       MAX(n_orders) AS max_orders_per_customer,  
       ROUND(AVG(n_orders * 1.0), 2) AS avg_orders_per_customer  
FROM (SELECT customer_id, COUNT(*) AS n_orders FROM orders GROUP BY customer_id) AS t;
```

RESULT

min_orders_per_customer	max_orders_per_customer	avg_orders_per_customer
1	10	4.08

1 rows × 3 columns

SQL

```
SELECT items_in_order, COUNT(*) AS number_of_orders  
FROM (SELECT order_id, COUNT(*) AS items_in_order FROM order_items GROUP BY order_id) AS t  
GROUP BY items_in_order  
ORDER BY items_in_order;
```

RESULT

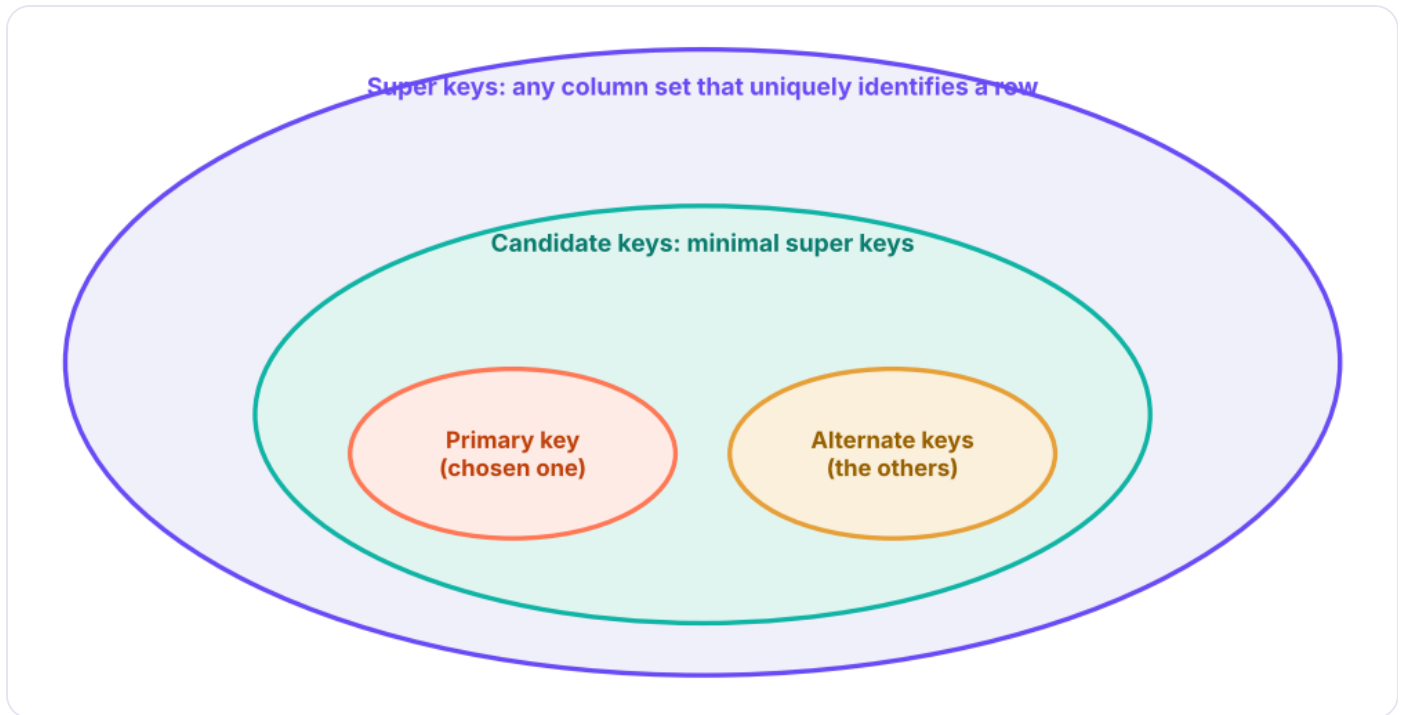
items_in_order	number_of_orders
1	1394
2	920
3	455
4	431

4 rows × 2 columns

So Customer → Orders is 1:N, and Order → Order items is 1:N (1 to 4 items per order here).

Keys

Keys are columns (or sets of columns) with special meaning. They are among the most frequently asked interview topics.



How the main key types relate: every candidate key is a minimal super key; one candidate is chosen as the primary key and the rest become alternate keys.

Example table `employees(emp_id, email, pan_number, name, dept_id, phone)`, where `emp_id`, `email` and `pan_number` are each unique:

Key type	Definition	Example
Super key	Any set of columns that uniquely identifies a row (may include extra columns)	{emp_id}, {emp_id, name}, {email, dept_id}
Candidate key	A minimal super key: remove any column and it stops being unique	{emp_id}, {email}, {pan_number}
Primary key (PK)	The candidate key chosen to identify rows. Unique, not NULL , one per table	emp_id
Alternate key	Candidate keys not chosen as primary; enforce them with UNIQUE	email, pan_number
Composite key	A key made of two or more columns	(order_id, product_id) in order_items; (student_id, course_id) in enrolments
Foreign key (FK)	A column that refers to a PK (or unique key) in another table	employees.dept_id → departments.dept_id
Unique key	A UNIQUE constraint: no duplicates, but (in SQL Server) one NULL is allowed	phone
Surrogate key	An artificial key with no business meaning, often an auto-number	customer_key INT IDENTITY(1,1)
Natural key	A key with business meaning	PAN number, GSTIN, ISBN, email

TIP Surrogate or natural?

Natural keys can change (people change email, products get recoded) and can be long. Most warehouses and Power BI models use **surrogate integer keys** for joins, and keep natural keys as alternate (unique) keys.

Primary key vs unique key

	Primary key	Unique key
Duplicates allowed	No	No
NULL allowed	No	Yes (SQL Server allows one NULL; PostgreSQL allows many)
Per table	Exactly one	Many
Default index (SQL Server)	Clustered	Non-clustered

Referential integrity

A foreign key guarantees that every child row points to a parent that exists: you cannot insert an order for customer C9999 if that customer does not exist. What happens when a **parent** row is deleted or its key is updated is controlled by the FK's actions:

Action	On DELETE of the parent	Use when
NO ACTION (default)	The delete is blocked if children exist	Most business data
CASCADE	Children are deleted too	Order → order items
SET NULL	Children's FK becomes NULL	Employee → manager who leaves
SET DEFAULT	Children get the column's default	Rare

SQL

```
CREATE TABLE departments (
  dept_id INT PRIMARY KEY,
  dept_name VARCHAR(50) NOT NULL UNIQUE
);
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  email VARCHAR(100) NOT NULL UNIQUE,           -- alternate key
  name VARCHAR(100) NOT NULL,
  dept_id INT NOT NULL REFERENCES departments(dept_id), -- foreign key
  manager_id INT NULL REFERENCES employees(emp_id)      -- self-referencing FK
);
```

Lesson 4 shows these constraints in action.

Normalisation

Normalisation organises columns and tables to reduce redundancy and avoid **anomalies**:

- **Update anomaly**: the same fact stored in many rows gets changed in some but not others.

- **Insert anomaly:** you cannot add a fact without another fact (cannot add a new product until someone orders it).
- **Delete anomaly:** deleting one fact removes another (deleting the only order of a product deletes the product's information).

Start with the flat Superstore file, which repeats customer and product details on every order line:

SQL

```
SELECT TOP 5 order_id, order_date, customer_id, customer_name, city, product_id,
product_name, category, quantity, sales
FROM superstore;
```

RESULT

order_id	order_date	customer_id	customer_name	city	product_id	product_name
IN-2023-00001	2023-01-01	C0765	Ishaan Gupta	Bengaluru	P0059	Nova Phone 223
IN-2023-00001	2023-01-01	C0765	Ishaan Gupta	Bengaluru	P0050	Orbit Appliance
IN-2023-00001	2023-01-01	C0765	Ishaan Gupta	Bengaluru	P0011	Orbit Tablet 248
IN-2023-00002	2023-01-03	C0791	Kabir Singh	Ahmedabad	P0037	Crest Storage 946
IN-2023-00003	2023-01-03	C0049	Sneha Khan	Ahmedabad	P0018	Crest Bookcase

5 rows × 7 columns

First normal form (1NF)

- Each cell holds **one atomic value** (no lists like "Chair, Table").
- No repeating groups of columns (product1, product2, product3).
- Each row is unique (a key exists).

The flat file is already in 1NF: one product per row, and every row has a unique row_id. The natural key would be (order_id, product_id), but 70 lines repeat the same product within the same

order, which is exactly why real systems add a surrogate key such as `row_id` or `item_id`.

Second normal form (2NF)

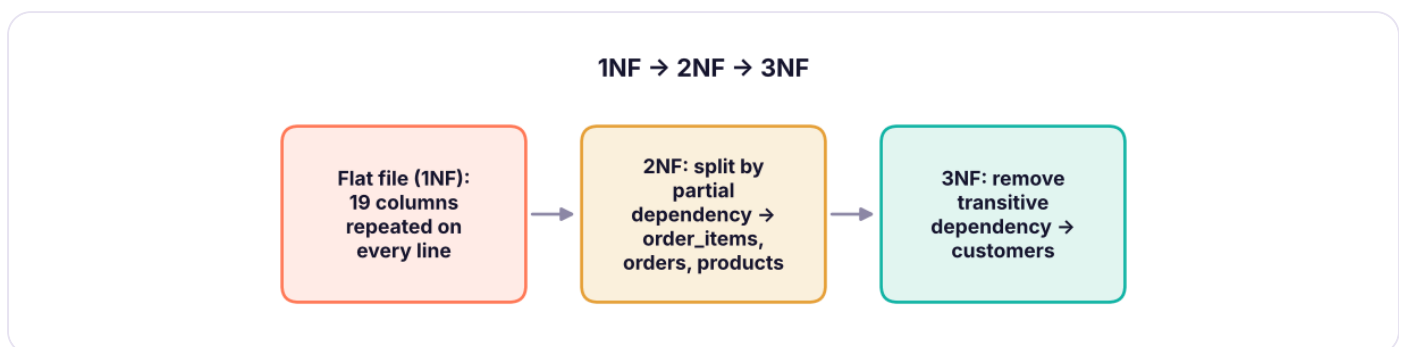
1NF **and** every non-key column depends on the **whole** key, not part of it (this matters only with composite keys). With key (`order_id`, `product_id`):

- `quantity`, `discount`, `sales` depend on both: they stay.
- `order_date`, `customer_id`, `ship_mode` depend only on `order_id` → move to an `orders` table.
- `product_name`, `category` depend only on `product_id` → move to a `products` table.

Third normal form (3NF)

2NF **and** no **transitive** dependency: non-key columns must depend only on the key, not on other non-key columns. In `orders`, `customer_name`, `city` and `region` depend on `customer_id`, not on `order_id` → move them to a `customers` table.

"Every non-key attribute must provide a fact about the key, the whole key, and nothing but the key." (Bill Kent)



Normalising the flat Superstore file to 3NF gives the four-table RetailDB design.

How much repetition did normalisation remove? Count the distinct customers and products versus rows:

SQL

```
SELECT COUNT(*)           AS order_lines,  
       COUNT(DISTINCT order_id) AS orders,  
       COUNT(DISTINCT customer_id) AS customers,  
       COUNT(DISTINCT product_id) AS products  
FROM superstore;
```

RESULT

order_lines	orders	customers	products
6323	3200	785	78

1 rows × 4 columns

The flat file stores customer details 6,323 times; the normalised design stores them once per customer (fewer than 800 rows).

Normal form	Rule in plain words
1NF	One value per cell; no repeating columns; a key exists
2NF	No column depends on only part of a composite key
3NF	No column depends on another non-key column
BCNF	Every determinant is a candidate key (stricter 3NF)
4NF / 5NF	Remove multi-valued and join dependencies (rare in practice)

Denormalisation and the star schema

Transactional systems (OLTP) are normalised for safe, fast writes. Analytical systems (OLAP, data warehouses, Power BI) are often **denormalised** into a **star schema**: one central **fact table** (sales events with numbers) surrounded by **dimension tables** (customer, product, date, region) to make reading and aggregating fast and simple. You will build one in the Power BI track (Lesson 3).

	OLTP (normalised)	OLAP / warehouse (star schema)
Purpose	Run the business: place orders, update stock	Analyse the business
Operations	Many small inserts and updates	Large reads and aggregations
Design	3NF, many tables	Fact + dimension tables
Examples	App database in SQL Server/PostgreSQL	Snowflake, BigQuery, Fabric, Power BI model

Summary

KEY TAKEAWAYS

- ER diagrams show **entities** (tables), **attributes** (columns) and **relationships** with **cardinality** (1:1, 1:N, M:N).
- The foreign key sits on the **many** side; M:N needs a **junction table** with a composite key.
- Super key $\supseteq$ candidate keys (minimal); one candidate becomes the **primary key** (unique, not NULL), the others are **alternate keys** (UNIQUE).
- Composite keys combine columns; surrogate keys are artificial integers; natural keys carry business meaning.
- Foreign keys enforce **referential integrity**; ON DELETE CASCADE / SET NULL / NO ACTION decide what happens to children.
- Normalise to **3NF** to remove redundancy and anomalies; denormalise into a **star schema** for analytics.

Quick quiz

6 questions · tap an answer to check it

1. Students can enrol in many courses and courses have many students. How do you model this?

- A Put course_id in the students table
- B Put student_id in the courses table
- C Create a junction table enrolments(student_id, course_id)
- D Use one table with a column per course

Answer: C. A junction table resolves M:N into two 1:N relationships.

2. In a 1:N relationship between departments and employees, where does the foreign key go?

A In departments

B In employees

C In both tables

D In a third table

Answer: B. The FK goes on the "many" side: employees.dept_id.

3. Which statement about a primary key is FALSE?

A It must be unique

B It cannot be NULL

C A table can have several primary keys

D It can consist of more than one column

Answer: C. A table has one primary key, which may be composite.

4. email is unique in the employees table but emp_id was chosen as the primary key. What is email?

A A foreign key

B An alternate (candidate) key

C A surrogate key

D A super key only

Answer: B. Unchosen candidate keys are alternate keys, enforced with UNIQUE.

5. In orders(order_id, customer_id, customer_city), customer_city depends on customer_id. Which normal form is violated?

A 1NF

B 2NF

C 3NF

D None

Answer: C. A non-key column depending on another non-key column is a transitive dependency, violating 3NF.

6. What does ON DELETE CASCADE on order_items.order_id do?

A Deleting an order also deletes its order items

B Deleting an order is blocked if it has items

C Order items get a NULL order_id

D Nothing

Answer: A. CASCADE propagates the delete to child rows.

EXERCISE Design a library database

A library has members, books and loans. A member can borrow many books over time; a book (copy) can be borrowed many times; each loan has a borrow date, due date and return date. Books have an ISBN, title and author; authors can write many books and a book can have several authors. Draw the entities, keys and relationships.

Show solution (worked answer)

- members(member_id PK, name, email UNIQUE, phone)
- books(book_id PK, isbn UNIQUE, title, publisher, year)
- authors(author_id PK, name)
- book_authors(book_id FK, author_id FK, PRIMARY KEY (book_id, author_id)): junction for M:N.
- copies(copy_id PK, book_id FK, shelf): one book has many physical copies.
- loans(loan_id PK, copy_id FK, member_id FK, borrow_date, due_date, return_date NULL). Relationships: members 1:N loans; copies 1:N loans; books 1:N copies; books M:N authors (via book_authors).

INTERVIEW CORNER Questions you may be asked

Q: Difference between primary key, unique key and foreign key? PK uniquely identifies rows, no NULLs, one per table. A unique key also prevents duplicates but allows a NULL (one in SQL Server) and a table can have many. A foreign key references a PK/unique key in another table to enforce relationships.

Q: What is a candidate key? A composite key? A candidate key is a minimal set of columns that uniquely identifies rows; one is chosen as the PK. A composite key uses two or more columns together, such as (order_id, product_id).

Q: What is normalisation and why denormalise? Normalisation removes redundancy and anomalies (1NF–3NF). Data warehouses denormalise into star schemas to make analytical queries simpler and faster.

Data Types, Operators, NULLs, Variables & Comments

Choose the right SQL Server data type for every column, use arithmetic, comparison, logical and string operators correctly, understand NULL and three-valued logic, declare variables, follow naming rules and write readable, commented SQL.

● SQL class 2

✦ SQL Server 2022, 2026

Beginner

🕒 100 min

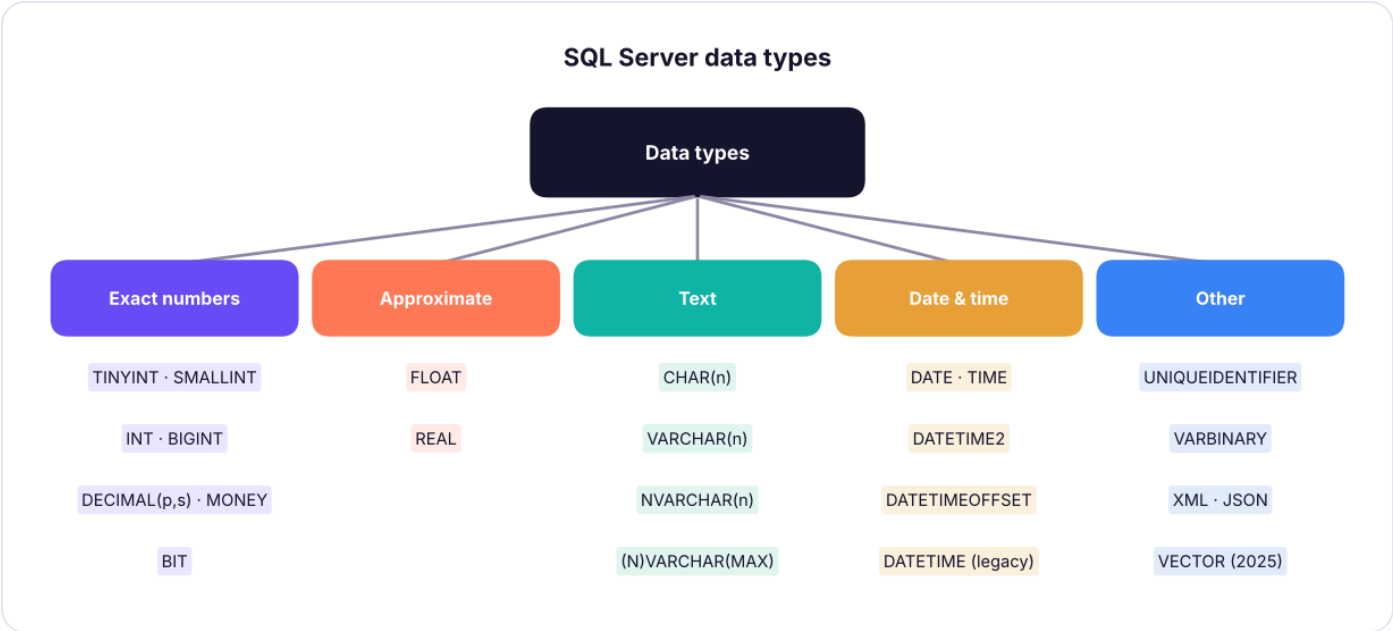
By the end of this lesson you can

- Pick suitable numeric, text, date/time and other data types, and know why the choice matters
- Use arithmetic, comparison, logical and concatenation operators, including integer division and modulo
- Explain NULL, three-valued logic, IS NULL and COALESCE
- Declare and use T-SQL variables with DECLARE, SET and SELECT
- Follow identifier naming rules and conventions, and comment code

Why data types matter

Every column has exactly one **data type**. The type decides what values are allowed, how much storage they use, how they sort and compare, and which functions work on them. A column of dates stored as text sorts "01-02-2026" before "31-01-2025" and cannot be used for date arithmetic, the same problem you met in Excel.

SQL Server data types



The main families of SQL Server data types.

Numbers

Type	Range / precision	Storage	Use for
TINYINT	0 to 255	1 byte	Ratings, small codes
SMALLINT	−32,768 to 32,767	2 bytes	Small counts
INT	about ±2.1 billion	4 bytes	IDs, quantities (default choice)
BIGINT	about ±9.2 × 10 ¹⁸	8 bytes	Very large IDs, event logs
DECIMAL(p, s) / NUMERIC	exact, p digits with s after the point	5–17 bytes	Money , rates: DECIMAL(12,2) holds up to 9,999,999,999.99
MONEY	exact, 4 decimals	8 bytes	Currency (DECIMAL is more portable)
FLOAT , REAL	approximate binary floating point	8 / 4 bytes	Scientific measurements, ML scores
BIT	0, 1 or NULL	1 bit	Yes/No flags

WATCH OUT Never store money in FLOAT

FLOAT cannot represent many decimal fractions exactly (0.1 + 0.2 is not exactly 0.3), so totals drift by tiny amounts. Use DECIMAL(p, s) for currency.

Text

Type	Description	Example
CHAR(n)	Fixed length, padded with spaces	CHAR(2) for state codes MH
VARCHAR(n)	Variable length, up to n bytes (n ≤ 8000)	VARCHAR(100) for names in English
NVARCHAR(n)	Variable length Unicode (Hindi, Tamil, emoji)	NVARCHAR(100) for names in any language
VARCHAR(MAX) , NVARCHAR(MAX)	Up to 2 GB	Long descriptions, JSON

Prefix Unicode string literals with N : N 'नमस्ते' . Without N , non-Latin characters can turn into ? .

Dates and times

Type	Stores	Example
DATE	Date only	2026-04-27
TIME	Time only	14:30:00
DATETIME2(p)	Date + time, precise (preferred)	2026-04-27 14:30:15.1234567
DATETIME	Older date + time (3.33 ms precision)	Legacy tables
DATETIMEOFFSET	Date + time + time zone	2026-04-27 14:30 +05:30
SMALLDATETIME	Date + time to the minute	Low-precision logs

Always write date literals in the unambiguous **ISO format** 'YYYY-MM-DD' ('2026-04-27').
'04/05/2026' means April 5 or May 4 depending on the server's language settings.

Others

UNIQUEIDENTIFIER (GUIDs), VARBINARY (files, images), XML , JSON stored in NVARCHAR(MAX) with JSON functions, GEOGRAPHY (maps), and from SQL Server 2025 a native VECTOR type for AI embeddings.

Choosing types for the retail tables

Column	Good type	Why
customer_id	VARCHAR(10)	Codes like C0001 keep leading letters/zeros
customer_name	NVARCHAR(100)	Names in any language
order_date	DATE	No time needed
quantity	INT (or SMALLINT)	Whole numbers
discount	DECIMAL(4,2)	0.00 to 0.99 exactly
sales , profit	DECIMAL(12,2)	Money
is_returned	BIT	Yes/No

Here are the types DuckDB inferred when loading the CSV (SSMS's Import Flat File wizard does the same kind of guessing; always review its suggestions):

OUTPUT

	column_name	column_type
0	row_id	BIGINT
1	order_id	VARCHAR
2	order_date	DATE
3	ship_date	DATE
4	ship_mode	VARCHAR
5	customer_id	VARCHAR
6	customer_name	VARCHAR
7	segment	VARCHAR
8	city	VARCHAR
9	state	VARCHAR
10	region	VARCHAR
11	product_id	VARCHAR

19 rows × 2 columns (first 12 shown)

Operators

Arithmetic

Operator	Meaning	Example	Result in SQL Server
+	Add	SELECT 7 + 2	9
-	Subtract	SELECT 7 - 2	5
*	Multiply	SELECT 7 * 2	14
/	Divide	SELECT 7 / 2	3 (integer ÷ integer = integer)
/	Divide	SELECT 7 / 2.0	3.500000
%	Modulo (remainder)	SELECT 7 % 2	1

COMMON MISTAKES

Integer division

In SQL Server, dividing two integers drops the decimals: `SELECT 1/3` returns 0, and `COUNT(...)/COUNT(...)` returns 0 or 1 instead of a percentage. Multiply by 1.0 or CAST one side: `SELECT 100.0 * returned / total` or `CAST(returned AS DECIMAL(10,2)) / total`. (PostgreSQL behaves the same way; MySQL and DuckDB return decimals.)

Calculated columns are written right in the SELECT list. Here is unit price and margin for a few order lines:

SQL

```
SELECT TOP 6
    order_id,
    quantity,
    sales,
    profit,
    ROUND(sales / quantity, 2)      AS unit_price,
    ROUND(100.0 * profit / sales, 1) AS margin_pct
FROM superstore
ORDER BY row_id;
```

RESULT

order_id	quantity	sales	profit	unit_price	margin_pct
IN-2023-00001	2	36,000	3,472.55	18,000	9.6
IN-2023-00001	1	3,850	1,169.54	3,850	30.4
IN-2023-00001	4	62,080	6,044.47	15,520	9.7
IN-2023-00002	2	5,216	450.16	2,608	8.6
IN-2023-00003	3	24,591	-5,512.33	8,197	-22.4
IN-2023-00003	1	8,288	-1,598.43	8,288	-19.3

6 rows × 6 columns

Comparison

Operator	Meaning
=	Equal
◇ or ≠	Not equal
> < ≥ ≤	Greater / less (than or equal)
BETWEEN a AND b	Inclusive range
IN (...)	Matches any value in a list
LIKE	Pattern match with % and _
IS NULL / IS NOT NULL	Tests for missing values

Logical

AND (both true), OR (at least one true), NOT (reverses). **Precedence:** NOT first, then AND, then OR. Use brackets to make intent obvious.

SQL

```
-- Without brackets, AND is evaluated first:
SELECT COUNT(*) AS rows_without_brackets
FROM superstore
WHERE region = 'North' OR region = 'South' AND category = 'Technology';
```

RESULT

rows_without_brackets

2025

1 rows × 1 columns

SQL

```
-- With brackets: (North or South) and Technology
SELECT COUNT(*) AS rows_with_brackets
FROM superstore
WHERE (region = 'North' OR region = 'South') AND category = 'Technology';
```

RESULT

rows_with_brackets

952

1 rows × 1 columns

The first query returns all North rows plus South Technology rows: probably not what was meant. Brackets fix it.

String concatenation

Method	Example	NULL behaviour
+	first_name + ' ' + last_name	Any NULL makes the whole result NULL
CONCAT()	CONCAT(first_name, ' ', last_name)	Treats NULL as empty text
CONCAT_WS()	CONCAT_WS(' ', city, state, 'India')	Separator; skips NULLs

SQL

```
SELECT TOP 4 customer_name,
      CONCAT(city, ' ', state) AS city_state,
      CONCAT_WS(' | ', segment, region) AS segment_region
FROM customers;
```

RESULT

customer_name	city_state	segment_region
Saanvi Khan	Chandigarh, Punjab	Corporate North
Meera Bose	Delhi, Delhi	Corporate North
Riya Shah	Kolkata, West Bengal	Consumer East
Diya Fernandes	Indore, Madhya Pradesh	Corporate West

4 rows × 3 columns

NULL: the missing value

NULL means **unknown or missing**. It is not zero and not an empty string. Any comparison with NULL returns **UNKNOWN**, not TRUE or FALSE. SQL uses **three-valued logic**:

Expression	Result
NULL = NULL	UNKNOWN
NULL <> 5	UNKNOWN
5 + NULL	NULL
TRUE AND UNKNOWN	UNKNOWN
FALSE AND UNKNOWN	FALSE
TRUE OR UNKNOWN	TRUE
NOT UNKNOWN	UNKNOWN

AND				OR			
	TRUE	FALSE	UNKNOWN		TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN	TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	FALSE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Three-valued logic. Any comparison involving NULL is UNKNOWN; WHERE keeps only TRUE rows.

WHERE keeps only rows where the condition is TRUE, so rows where it is UNKNOWN disappear silently. That is why you must test with IS NULL:

SQL

```
WITH t AS (  
  SELECT 'A' AS item, 10 AS score UNION ALL  
  SELECT 'B', NULL UNION ALL  
  SELECT 'C', 30  
)  
SELECT  
  (SELECT COUNT(*) FROM t WHERE score = NULL) AS equals_null_rows, -- always 0!  
  (SELECT COUNT(*) FROM t WHERE score IS NULL) AS is_null_rows,  
  (SELECT COUNT(score) FROM t) AS count_score, -- ignores NULL  
  (SELECT COUNT(*) FROM t) AS count_star,  
  (SELECT AVG(score * 1.0) FROM t) AS avg_ignores_null, -- (10+30)/2  
  (SELECT AVG(COALESCE(score, 0) * 1.0) FROM t) AS avg_null_as_zero; -- (10+0+30)/3
```

RESULT

equals_null_rows	is_null_rows	count_score	count_star	avg_ignores_null	avg_null_as_;
0	1	2	3	20	13.33

1 rows × 6 columns

Function	What it does
ISNULL(x, y)	T-SQL: return y when x is NULL
COALESCE(x, y, z, ...)	ANSI: the first non-NULL value; works in every database
NULLIF(a, b)	NULL if a = b, otherwise a. Classic use: sales / NULLIF(qty, 0) avoids divide-by-zero errors

Variables in T-SQL

T-SQL lets you store a value in a variable for the length of a batch. Variable names start with @.

SQL

```
DECLARE @target_region VARCHAR(20) = 'West';      -- declare and assign
DECLARE @min_sales      DECIMAL(12,2);
SET @min_sales = 50000;                          -- assign with SET

SELECT order_id, city, sales
FROM superstore
WHERE region = @target_region AND sales ≥ @min_sales;

DECLARE @total DECIMAL(14,2);
SELECT @total = SUM(sales) FROM superstore;      -- assign from a query
PRINT CONCAT('Total sales: ', @total);          -- message in the Messages tab
```

Variables make scripts reusable (change 'West' once) and are the basis of stored procedures (Lesson 13). Other databases use different syntax (PostgreSQL DO blocks or \set in psql).

Identifiers: naming rules and conventions

Rules (SQL Server): - Start with a letter, _, @ (variables) or # (temporary tables). - Up to 128 characters; letters, digits, _, @, \$, #. - Reserved words (ORDER, GROUP, USER) and names with spaces must be wrapped in [square brackets] (or "double quotes"): [Order Details], [order].

Conventions used in this handbook (pick one style and be consistent): - snake_case, lowercase table and column names: order_items, customer_id. - Plural table names (customers) or singular (customer): be consistent. - Primary key id or <table>_id; foreign keys named after the referenced key (customer_id). - Prefix views vw_, procedures usp_, indexes ix_. - Avoid spaces, special characters and reserved words, so you never need brackets.

Comments and formatting

SQL

```
-- A single-line comment: everything after -- is ignored

/* A multi-line comment:
   Purpose : Top 10 customers by 2025 sales
   Author  : Varun Kumar
   Updated : 27-Apr-2026                               */

SELECT TOP 10
    customer_id,
    SUM(sales) AS total_sales      -- revenue in rupees
FROM superstore
WHERE order_date ≥ '2025-01-01'
    AND order_date < '2026-01-01' -- half-open range: safe with times
GROUP BY customer_id
ORDER BY total_sales DESC;
```

TIP Readable SQL checklist

One clause per line (SELECT, FROM, WHERE...), one column per line in long queries, consistent indentation, keywords in capitals, meaningful aliases, and a header comment explaining the purpose. Tools such as the "Format SQL" extension or SQLFluff format code automatically.

Summary

KEY TAKEAWAYS

- Choose types deliberately: INT for IDs and counts, DECIMAL(p, s) for money, NVARCHAR for names, DATE/DATETIME2 for dates; never FLOAT for money.
- Write dates as 'YYYY-MM-DD'.
- $7/2 = 3$ in SQL Server: use `1.0 *` or `CAST` for decimal division; `%` gives the remainder.
- Precedence: NOT, AND, OR; use brackets.
- NULL is unknown: compare with `IS NULL`; aggregates ignore NULLs; `COALESCE`, `ISNULL` and `NULLIF` handle them.
- `CONCAT` treats NULL as empty; `+` with a NULL returns NULL.
- Variables: `DECLARE @x type = value`; name things consistently and comment your code.

Quick quiz

6 questions · tap an answer to check it

1. Which data type should store a product price in rupees and paise?

A FLOAT

B INT

C DECIMAL(10,2)

D VARCHAR(10)

Answer: C. DECIMAL stores exact decimal values, which money needs.

2. In SQL Server, what does `SELECT 9 / 4` return?

A 2

B 2.25

C 2.0

D 3

Answer: A. Integer divided by integer gives an integer (the fraction is dropped).

3. What does `WHERE discount = NULL` return?

A Rows where discount is missing

B No rows

C All rows

D An error

Answer: B. Comparisons with NULL are UNKNOWN, so WHERE keeps nothing. Use IS NULL.

4. Which expression returns 'Pune' when city is NULL?

A `city + 'Pune'`

B `COALESCE(city, 'Pune')`

C `NULLIF(city, 'Pune')`

D `city = 'Pune'`

Answer: B. COALESCE returns the first non-NULL value.

5. WHERE $a = 1$ OR $b = 2$ AND $c = 3$ is evaluated as:

A $(a = 1 \text{ OR } b = 2) \text{ AND } c = 3$

B $a = 1 \text{ OR } (b = 2 \text{ AND } c = 3)$

C Left to right with equal precedence

D An error

Answer: B. AND has higher precedence than OR.

6. Why use NVARCHAR instead of VARCHAR for customer names?

A It is faster

B It stores Unicode characters such as Devanagari or Tamil correctly

C It allows numbers

D It is required for primary keys

Answer: B. NVARCHAR stores Unicode text; VARCHAR may lose characters outside the code page.

EXERCISE Fix the query

A colleague wrote: `SELECT order_id, profit/sales*100 AS margin FROM superstore WHERE category = 'Furniture' OR category = 'Technology' AND discount > 0.2 AND ship_mode = NULL;` List every problem and rewrite it.

Show solution (worked answer)

1. AND binds before OR, so the discount and ship-mode filters apply only to Technology: add brackets.
2. `ship_mode = NULL` never matches: use `IS NULL` (or remove it if not needed).
3. If profit and sales were integers, `profit/sales` would be integer division; use `100.0 * profit / sales` and guard against zero with `NULLIF(sales, 0)`. Rewritten: `SELECT order_id, ROUND(100.0 * profit / NULLIF(sales, 0), 1) AS margin_pct FROM superstore WHERE category IN ('Furniture', 'Technology') AND discount > 0.2;`

INTERVIEW CORNER Questions you may be asked

Q: What is the difference between CHAR, VARCHAR and NVARCHAR? CHAR is fixed length (padded), VARCHAR is variable length non-Unicode, NVARCHAR is variable length Unicode (two bytes per character for most scripts).

Q: How does SQL treat NULL? As unknown: comparisons return UNKNOWN, aggregates (except `COUNT(*)`) ignore NULLs, arithmetic with NULL returns NULL. Use `IS NULL`, `COALESCE/ISNULL` and `NULLIF`.

Q: Why does `SELECT 100 * returned / total` give 0? Integer division when both values are integers; cast to decimal or multiply by `100.0`.

Q: DECIMAL vs FLOAT? DECIMAL is exact (money, rates); FLOAT is approximate binary (scientific values) and can introduce rounding errors.

DDL, DML & Constraints: CREATE, ALTER, INSERT, UPDATE, DELETE

Create and change tables with DDL (CREATE, ALTER, DROP, TRUNCATE), add, change and remove rows with DML (INSERT, UPDATE, DELETE, MERGE), protect data quality with constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, DEFAULT, IDENTITY) and understand DELETE vs TRUNCATE vs DROP.

● SQL classes 2-3

✦ SQL Server 2022, 2026

Beginner

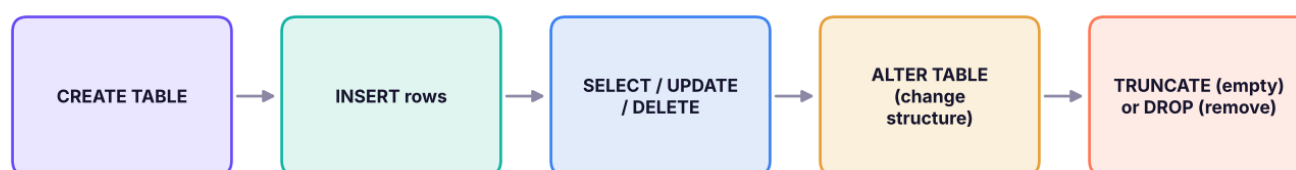
⌚ 120 min

By the end of this lesson you can

- Create databases and tables with appropriate data types and constraints
- Modify table structure with ALTER TABLE (add, change, drop columns and constraints)
- Insert single and multiple rows, insert from a query, update and delete rows safely
- Explain every constraint type and read the error each one raises
- Compare DELETE, TRUNCATE and DROP, and use MERGE for upserts

The life cycle of a table

DDL and DML together



DDL creates and changes structure; DML fills and changes the rows inside it.

CREATE TABLE

SQL

```
CREATE TABLE departments (  
    dept_id    INT            PRIMARY KEY,  
    dept_name  VARCHAR(50)    NOT NULL UNIQUE,  
    location   VARCHAR(50)    DEFAULT 'Bengaluru'  
);  
  
CREATE TABLE staff (  
    emp_id     INT            PRIMARY KEY,  
    full_name  VARCHAR(100)   NOT NULL,  
    email      VARCHAR(100)   NOT NULL UNIQUE,  
    dept_id    INT            NOT NULL REFERENCES departments(dept_id),  
    salary     DECIMAL(10,2)   CHECK (salary > 0),  
    join_date  DATE            DEFAULT CURRENT_DATE,  
    is_active  BOOLEAN         DEFAULT TRUE  
);
```

RESULT

Commands completed successfully.

Commands completed successfully.

In SQL Server you would usually write `IDENTITY(1,1)` for an auto-numbered key, `BIT` for the flag and `DEFAULT GETDATE()` for today. Here is the same `staff` table in full T-SQL style, with named constraints (names make error messages and later changes much easier):

SQL

```
CREATE TABLE dbo.staff (  
    emp_id     INT IDENTITY(1,1) NOT NULL,  
    full_name  NVARCHAR(100)     NOT NULL,  
    email      VARCHAR(100)      NOT NULL,  
    dept_id    INT               NOT NULL,  
    salary     DECIMAL(10,2)      NULL,  
    join_date  DATE              NOT NULL CONSTRAINT df_staff_join  DEFAULT  
    (CAST(GETDATE() AS DATE)),  
    is_active  BIT               NOT NULL CONSTRAINT df_staff_active DEFAULT (1),  
    CONSTRAINT pk_staff          PRIMARY KEY (emp_id),  
    CONSTRAINT uq_staff_email     UNIQUE (email),  
    CONSTRAINT fk_staff_dept      FOREIGN KEY (dept_id) REFERENCES dbo.departments(dept_id),  
    CONSTRAINT ck_staff_salary    CHECK (salary > 0)  
);
```

Constraints at a glance

Constraint	Guarantees	Example
PRIMARY KEY	Unique and not NULL; identifies each row	emp_id
FOREIGN KEY / REFERENCES	Value must exist in the parent table	dept_id → departments
UNIQUE	No duplicate values (NULL allowed once in SQL Server)	email
NOT NULL	A value is required	full_name
CHECK	Value must satisfy a condition	salary > 0 , discount BETWEEN 0 AND 0.9
DEFAULT	Value used when none is supplied	is_active = 1
IDENTITY(seed, step)	Auto-numbering (SQL Server)	IDENTITY(1,1)

INSERT: adding rows

SQL

```
INSERT INTO departments (dept_id, dept_name, location) VALUES
  (10, 'Finance', 'Mumbai'),
  (20, 'IT', 'Pune'),
  (30, 'HR', DEFAULT),
  (40, 'Sales', 'Delhi');

INSERT INTO staff (emp_id, full_name, email, dept_id, salary, join_date) VALUES
  (1, 'Kavya Iyer', 'kavya@abc.in', 30, 97000, '2016-02-28'),
  (2, 'Imran Reddy', 'imran@abc.in', 10, 157000, '2019-12-14'),
  (3, 'Harpreet Kulkarni', 'harpreet@abc.in', 10, 131000, '2024-10-05'),
  (4, 'Rahul Iyer', 'rahul@abc.in', 20, 112000, '2017-02-06'),
  (5, 'Isha Mehta', 'isha@abc.in', 40, 57000, '2021-09-07');

SELECT * FROM departments;
```

RESULT

(4 rows affected)
(5 rows affected)

dept_id	dept_name	location
10	Finance	Mumbai
20	IT	Pune
30	HR	Bengaluru
40	Sales	Delhi

4 rows × 3 columns

SQL

```
SELECT emp_id, full_name, dept_id, salary, join_date, is_active FROM staff;
```

RESULT

emp_id	full_name	dept_id	salary	join_date	is_active
1	Kavya Iyer	30	97,000	2016-02-28	True
2	Imran Reddy	10	157,000	2019-12-14	True
3	Harpreet Kulkarni	10	131,000	2024-10-05	True
4	Rahul Iyer	20	112,000	2017-02-06	True
5	Isha Mehta	40	57,000	2021-09-07	True

5 rows × 6 columns

DEFAULT filled HR's location with Bengaluru and every employee's `is_active` with TRUE.

TIP Always list the columns

`INSERT INTO staff VALUES (...)` without a column list depends on the column order and breaks the day someone adds a column. Always write `INSERT INTO staff (col1, col2, ...) VALUES (...)`.

INSERT ... SELECT and SELECT ... INTO

Copy rows from a query into a table:

SQL

```
-- Into an existing table
INSERT INTO staff_archive (emp_id, full_name, dept_id, salary)
SELECT emp_id, full_name, dept_id, salary
FROM staff
WHERE is_active = 0;

-- SQL Server: create a NEW table from a query (structure + data, no constraints)
SELECT emp_id, full_name, salary
INTO staff_backup_2026
FROM staff;
```

The portable equivalent of `SELECT ... INTO` is `CREATE TABLE new_table AS SELECT ...` (PostgreSQL, MySQL, Snowflake, DuckDB).

What constraints do when rules are broken

Constraints reject bad data **at the database**, whatever app or person sends it. These statements would all fail in SQL Server with the messages shown (abridged):

Statement	Broken rule	SQL Server message
INSERT INTO departments VALUES (10, 'Ops', 'Pune')	PRIMARY KEY	Violation of PRIMARY KEY constraint 'pk_departments'. Cannot insert duplicate key ... The duplicate key value is (10).
INSERT INTO staff (... , email, ...) VALUES (... , 'kavya@abc.in', ...)	UNIQUE	Violation of UNIQUE KEY constraint 'uq_staff_email'. Cannot insert duplicate key ...
INSERT INTO staff (... , dept_id, ...) VALUES (... , 99, ...)	FOREIGN KEY	The INSERT statement conflicted with the FOREIGN KEY constraint "fk_staff_dept".
INSERT INTO staff (emp_id, email, dept_id) VALUES (9, 'x@abc.in', 10)	NOT NULL	Cannot insert the value NULL into column 'full_name' ... column does not allow nulls. INSERT fails.
UPDATE staff SET salary = -500 WHERE emp_id = 1	CHECK	The UPDATE statement conflicted with the CHECK constraint "ck_staff_salary".
DELETE FROM departments WHERE dept_id = 10	FOREIGN KEY (children exist)	The DELETE statement conflicted with the REFERENCE constraint "fk_staff_dept".

Our practice engine raises the same kind of error. For example, trying to add an employee to a department that does not exist:

OUTPUT

Error: Constraint Error: Violates foreign key constraint because key "dept_id: 99" does not exist in the referenced table

UPDATE: changing rows

SQL

```
-- Give everyone in Finance a 5% raise
UPDATE staff
SET    salary = salary * 1.05
WHERE  dept_id = 10;

-- Change two columns for one person
UPDATE staff
SET    dept_id = 20, salary = 60000
WHERE  emp_id = 5;

SELECT emp_id, full_name, dept_id, salary FROM staff ORDER BY emp_id;
```

RESULT

(2 rows affected)

(1 row affected)

emp_id	full_name	dept_id	salary
1	Kavya Iyer	30	97,000
2	Imran Reddy	10	164,850
3	Harpreet Kulkarni	10	137,550
4	Rahul Iyer	20	112,000
5	Isha Mehta	20	60,000

5 rows × 4 columns

WATCH OUT UPDATE or DELETE without WHERE changes EVERY row

UPDATE staff SET salary = 0; sets every salary to zero. Before any UPDATE or DELETE:

1. Write the WHERE first and run it as a SELECT to see exactly which rows will change.
2. Wrap the change in a transaction (BEGIN TRAN ... then check, then COMMIT or ROLLBACK) (Lesson 14).
3. On production data, take a backup or work in a copy.

UPDATE with a join (SQL Server)

SQL

```
-- Mark staff in Mumbai departments as 'on-site'
UPDATE s
SET     s.work_mode = 'on-site'
FROM    staff AS s
JOIN    departments AS d ON d.dept_id = s.dept_id
WHERE   d.location = 'Mumbai';
```

DELETE: removing rows

SQL

```
DELETE FROM staff
WHERE emp_id = 5;

SELECT COUNT(*) AS staff_left FROM staff;
```

RESULT

(1 row affected)

staff_left

4

1 rows × 1 columns

ALTER TABLE: changing structure

SQL

```
ALTER TABLE staff ADD COLUMN phone VARCHAR(15);
ALTER TABLE staff ADD COLUMN work_mode VARCHAR(10) DEFAULT 'hybrid';
SELECT emp_id, full_name, phone, work_mode FROM staff;
```

RESULT

emp_id	full_name	phone	work_mode
1	Kavya Iyer	NULL	hybrid
2	Imran Reddy	NULL	hybrid
3	Harpreet Kulkarni	NULL	hybrid
4	Rahul Iyer	NULL	hybrid

4 rows × 4 columns

The SQL Server forms of the common changes:

SQL

```

ALTER TABLE staff ADD phone VARCHAR(15) NULL;           -- add a column (no
COLUMN keyword in T-SQL)
ALTER TABLE staff ALTER COLUMN full_name NVARCHAR(150) NOT NULL; -- change type /
nullability
ALTER TABLE staff DROP COLUMN phone;                     -- remove a column
ALTER TABLE staff ADD CONSTRAINT ck_staff_mode CHECK (work_mode IN ('remote','hybrid','on-
site'));
ALTER TABLE staff DROP CONSTRAINT ck_staff_mode;         -- remove a constraint
by name
EXEC sp_rename 'staff.full_name', 'employee_name', 'COLUMN'; -- rename a column
EXEC sp_rename 'staff', 'employees_master';               -- rename a table

```

NOTE Changing a column that has data

Making a column smaller (VARCHAR(100) → VARCHAR(10)) fails if existing values do not fit; adding NOT NULL fails if NULLs exist. Clean the data first, or add a DEFAULT.

DELETE vs TRUNCATE vs DROP

This is one of the most common SQL interview questions.

	DELETE	TRUNCATE TABLE	DROP TABLE
Family	DML	DDL	DDL
Removes	Some or all rows	All rows	The whole table : rows, structure, indexes, constraints, permissions
WHERE allowed	Yes	No	No
Speed on large tables	Slower: logs each row	Very fast: deallocates pages	Fast
Logging	Every row logged	Minimal (page deallocations)	Minimal
Fires DELETE triggers	Yes	No	No
Resets IDENTITY	No	Yes , back to the seed	Table is gone
Works when referenced by a FK	Yes (if no child rows violate it)	No	No (drop the FK first)
Can be rolled back in a transaction	Yes	Yes in SQL Server (inside an explicit transaction)	Yes in SQL Server (inside an explicit transaction)

SQL

```
CREATE TABLE temp_log (id INT, msg VARCHAR(50));
INSERT INTO temp_log VALUES (1, 'start'), (2, 'load'), (3, 'end');
DELETE FROM temp_log WHERE id = 2;           -- removes one row
SELECT COUNT(*) AS after_delete FROM temp_log;
```

RESULT

Commands completed successfully.
 (3 rows affected)
 (1 row affected)

after_delete

2

1 rows × 1 columns

SQL

```
TRUNCATE TABLE temp_log; -- removes all rows, keeps the table
SELECT COUNT(*) AS after_truncate FROM temp_log;
```

RESULT

after_truncate

0

1 rows × 1 columns

SQL

```
DROP TABLE temp_log; -- removes the table itself
SELECT COUNT(*) AS tables_named_temp_log FROM information_schema.tables WHERE table_name =
'temp_log';
```

RESULT

tables_named_temp_log

0

1 rows × 1 columns

In SQL Server, `DROP TABLE IF EXISTS temp_log;` avoids an error when the table is already gone.

MERGE: insert or update in one statement (upsert)

Data pipelines often receive a file of changes: some rows are new, others update existing rows.

MERGE handles both:

SQL

```
MERGE dbo.staff AS target
USING dbo.staff_updates AS src
    ON target.emp_id = src.emp_id
WHEN MATCHED THEN
    UPDATE SET target.salary = src.salary, target.dept_id = src.dept_id
WHEN NOT MATCHED BY TARGET THEN
    INSERT (emp_id, full_name, email, dept_id, salary)
    VALUES (src.emp_id, src.full_name, src.email, src.dept_id, src.salary)
WHEN NOT MATCHED BY SOURCE THEN
    UPDATE SET target.is_active = 0; -- in staff but not in the file
```

Loading CSV files with SQL

SQL

```
CREATE TABLE dbo.employees_small (  
    emp_id VARCHAR(10) PRIMARY KEY, name NVARCHAR(100), department VARCHAR(30),  
    city VARCHAR(30), join_date DATE, salary DECIMAL(10,2), rating TINYINT  
);  
  
BULK INSERT dbo.employees_small  
FROM 'C:\Data\employees_small.csv'  
WITH (FORMAT = 'CSV', FIRSTROW = 2, FIELDTERMINATOR = ',', ROWTERMINATOR = '0x0a');
```

The SSMS **Import Flat File** wizard is easier for one-off loads; BULK INSERT, bcp or an ETL tool (SSIS, Azure Data Factory, Fabric pipelines) are used for repeatable loads.

DO IT WITH AI Do it with AI

Ask: "Write SQL Server CREATE TABLE statements with constraints for a hospital database: patients, doctors, appointments (a patient can see many doctors; a doctor sees many patients). Include primary keys, foreign keys, CHECK constraints for appointment status, and sensible data types." Then review it: are the keys right, is money DECIMAL, are dates DATE/DATETIME2, are the constraints named?

Summary

KEY TAKEAWAYS

- **DDL**: CREATE, ALTER, DROP, TRUNCATE change structure. **DML**: INSERT, UPDATE, DELETE, MERGE change rows.
- Constraints (PK, FK, UNIQUE, NOT NULL, CHECK, DEFAULT, IDENTITY) protect data quality at the source; name them.
- Always list columns in INSERT; always test the WHERE of an UPDATE/DELETE with a SELECT first, inside a transaction.
- DELETE removes chosen rows (logged, triggers fire); TRUNCATE empties the table fast and resets IDENTITY; DROP removes the table.
- MERGE performs upserts; BULK INSERT and the Import wizard load files.

Quick quiz

6 questions · tap an answer to check it

1. Which statement removes all rows but keeps the table structure and resets the IDENTITY counter?

A DELETE FROM t

B TRUNCATE TABLE t

C DROP TABLE t

D ALTER TABLE t

Answer: B. TRUNCATE empties the table quickly and resets IDENTITY to its seed.

2. You run UPDATE staff SET salary = 50000; by mistake. What happened?

A Only the first row changed

B Every row's salary is now 50,000

C Nothing, because there is no WHERE

D An error is raised

Answer: B. Without WHERE, UPDATE changes every row. Use transactions and test with SELECT first.

3. Which constraint stops a salary of -500?

A UNIQUE

B DEFAULT

C CHECK (salary > 0)

D FOREIGN KEY

Answer: C. CHECK validates values against a condition.

4. Inserting an employee with dept_id 99 fails with a FOREIGN KEY error. Why?

- A** dept_id must be unique
- B** No department with id 99 exists in the parent table
- C** dept_id cannot be a number
- D** The employee table is full

Answer: B. A foreign key value must exist in the referenced table.

5. Which is TRUE about DELETE?

- A** It can remove selected rows with WHERE and fires DELETE triggers
- B** It is DDL
- C** It resets IDENTITY
- D** It removes the table structure

Answer: A. DELETE is DML, works row by row with a WHERE, and fires triggers.

6. How do you add a column phone to staff in SQL Server?

- A** ALTER TABLE staff ADD COLUMN phone VARCHAR(15) only
- B** ALTER TABLE staff ADD phone VARCHAR(15)
- C** UPDATE staff ADD phone
- D** INSERT COLUMN phone INTO staff

Answer: B. T-SQL uses ADD without the COLUMN keyword (PostgreSQL/MySQL accept ADD COLUMN).

EXERCISE Build and populate a small schema

Create `projects`(`project_id` PK, `name` UNIQUE NOT NULL, `budget` DECIMAL(12,2) CHECK `budget` ≥ 0, `start_date` DATE, `status` with CHECK IN ('planned', 'active', 'closed') DEFAULT 'planned') and `assignments`(`project_id` FK, `emp_id` FK, `role`, `hours`, PRIMARY KEY (`project_id`, `emp_id`)). Insert 3 projects and 4 assignments, raise the budget of active projects by 10%, close one project, and delete assignments of closed projects.

Show solution (worked answer)

SQL

```
CREATE TABLE projects (  
  project_id INT PRIMARY KEY, name VARCHAR(80) NOT NULL UNIQUE,  
  budget DECIMAL(12,2) CHECK (budget ≥ 0), start_date DATE,  
  status VARCHAR(10) DEFAULT 'planned' CHECK (status IN  
  ('planned', 'active', 'closed')));  
CREATE TABLE assignments (  
  project_id INT REFERENCES projects(project_id), emp_id INT REFERENCES staff(emp_id),  
  role VARCHAR(30), hours INT CHECK (hours > 0), PRIMARY KEY (project_id, emp_id));  
INSERT INTO projects (project_id, name, budget, start_date, status) VALUES  
  (1, 'CRM revamp', 2500000, '2026-01-10', 'active'), (2, 'Payroll audit', 600000, '2026-03-  
01', 'active'), (3, 'Data lake', 4000000, '2026-06-01', 'planned');  
INSERT INTO assignments VALUES (1, 1, 'Analyst', 120), (1, 4, 'Developer', 300),  
  (2, 2, 'Lead', 80), (3, 3, 'Architect', 40);  
UPDATE projects SET budget = budget * 1.10 WHERE status = 'active';  
UPDATE projects SET status = 'closed' WHERE project_id = 2;  
DELETE FROM assignments WHERE project_id IN (SELECT project_id FROM projects WHERE  
status = 'closed');
```

INTERVIEW CORNER**Questions you may be asked**

Q: DELETE vs TRUNCATE vs DROP? DELETE (DML) removes chosen rows with WHERE, logs each row, fires triggers and keeps IDENTITY. TRUNCATE (DDL) removes all rows quickly, minimal logging, resets IDENTITY, no WHERE, not allowed when referenced by a FK. DROP removes the whole table and its definition.

Q: What are constraints and why use them instead of app validation? Rules enforced by the database (PK, FK, UNIQUE, NOT NULL, CHECK, DEFAULT). They protect data no matter which app, script or person writes to the table.

Q: How do you safely run an UPDATE on production? Run the WHERE as a SELECT first, wrap in BEGIN TRAN, check @@ROWCOUNT and the result, then COMMIT or ROLLBACK; have a backup.

SELECT, WHERE, ORDER BY & Filtering Operators

Retrieve exactly the rows and columns you need. Master SELECT, aliases with AS, DISTINCT, TOP, WHERE with comparison and logical operators, IN, BETWEEN, LIKE wildcards, IS NULL, ORDER BY, and the logical order in which SQL processes a query.

● SQL class 3

✦ SQL Server 2022, 2026

Beginner

🕒 120 min

By the end of this lesson you can

- Select columns, calculated expressions and aliases, and remove duplicates with DISTINCT
- Filter rows with comparison, logical, IN, NOT IN, BETWEEN, LIKE, NOT LIKE, IS NULL and IS NOT NULL
- Sort results by one or more columns, expressions and positions; limit rows with TOP
- Explain the logical processing order of a query and why it matters for aliases
- Translate business questions into correct SELECT statements

The shape of a SELECT statement

SQL

```
SELECT    column_list           -- what to show
FROM      table_name           -- where it comes from
WHERE     row_conditions        -- which rows to keep
GROUP BY  grouping_columns     -- (Lesson 7)
HAVING    group_conditions      -- (Lesson 7)
ORDER BY  sort_columns;        -- how to sort the result
```

Only `SELECT` is required (`SELECT 1 + 1;` works). Everything else is optional but must appear in this order.

SELECT: columns, expressions and aliases

SQL

```
SELECT TOP 5
    order_id,
    order_date,
    customer_name,
    sales,
    profit,
    sales - profit           AS cost,
    ROUND(100.0 * profit / sales, 1) AS margin_pct,
    'INR'                   AS currency
FROM superstore;
```

RESULT

order_id	order_date	customer_name	sales	profit	cost	margin_pct	currency
IN-2023-00001	2023-01-01	Ishaan Gupta	36,000	3,472.55	32,527.45	9.6	INR
IN-2023-00001	2023-01-01	Ishaan Gupta	3,850	1,169.54	2,680.46	30.4	INR
IN-2023-00001	2023-01-01	Ishaan Gupta	62,080	6,044.47	56,035.53	9.7	INR
IN-2023-00002	2023-01-03	Kabir Singh	5,216	450.16	4,765.84	8.6	INR
IN-2023-00003	2023-01-03	Sneha Khan	24,591	-5,512.33	30,103.33	-22.4	INR

5 rows × 8 columns

- `SELECT *` returns every column. Use it to explore, never in production code (it breaks when columns change and reads more data than needed).
- An **alias** renames a column or table in the result: `AS margin_pct`. `AS` is optional (`sales - profit cost`) but makes code clearer. Aliases with spaces need brackets or quotes: `AS [Margin %]`.
- Expressions can use operators, functions and literals.

Table aliases

SQL

```
SELECT s.order_id, s.sales
FROM superstore AS s;      -- 's' is a short name for the table
```

Table aliases become essential with joins (Lesson 8).

DISTINCT: unique values

SQL

```
SELECT DISTINCT region
FROM superstore
ORDER BY region;
```

RESULT

region
East
North
South
West

4 rows × 1 columns

SQL

```
SELECT DISTINCT category, sub_category
FROM superstore
ORDER BY category, sub_category;
```

RESULT

category	sub_category
Furniture	Bookcases
Furniture	Chairs
Furniture	Furnishings
Furniture	Tables
Office Supplies	Appliances
Office Supplies	Art
Office Supplies	Binders
Office Supplies	Paper
Office Supplies	Storage
Technology	Accessories
Technology	Copiers
Technology	Machines

13 rows × 2 columns (first 12 shown)

DISTINCT applies to the **whole row** of selected columns: the second query returns each category-sub-category combination once. To count unique values: COUNT(DISTINCT customer_id).

TOP: limit the rows

SQL

```
SELECT TOP 5 order_id, customer_name, city, sales
FROM superstore
ORDER BY sales DESC;
```

RESULT

order_id	customer_name	city	sales
IN-2024-01857	Arjun Fernandes	Delhi	619,290
IN-2023-00564	Krishna Reddy	Hyderabad	551,530
IN-2024-01040	Aditya Kulkarni	Chennai	551,530
IN-2025-02880	Farhan Khan	Kochi	530,820
IN-2023-00809	Rohan Singh	Jaipur	526,396

5 rows × 4 columns

Dialect	Syntax for "first 5 rows"
SQL Server	SELECT TOP 5 ... ORDER BY ... (also TOP 10 PERCENT, TOP 5 WITH TIES)
SQL Server 2012+, ANSI	ORDER BY ... OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY (Lesson 11)
PostgreSQL, MySQL, SQLite, DuckDB	... ORDER BY ... LIMIT 5
Oracle	FETCH FIRST 5 ROWS ONLY

WATCH OUT

TOP without ORDER BY

Without ORDER BY, the database returns rows in no guaranteed order, so "top 5" may differ between runs. Always sort when you use TOP.

WHERE: filtering rows

Comparison operators

SQL
<pre>SELECT COUNT(*) AS lines_over_50k FROM superstore WHERE sales > 50000;</pre>
RESULT

lines_over_50k

1126

1 rows × 1 columns

SQL

```
SELECT order_id, city, category, sales, discount
FROM superstore
WHERE region = 'West'
      AND category = 'Furniture'
      AND discount ≥ 0.3
ORDER BY sales DESC;
```

RESULT

order_id	city	category	sales	discount
IN-2025-02904	Ahmedabad	Furniture	76,048	0.3
IN-2025-02783	Pune	Furniture	69,762	0.3
IN-2024-01002	Indore	Furniture	58,135	0.3
IN-2023-00431	Ahmedabad	Furniture	54,474	0.3
IN-2025-02016	Ahmedabad	Furniture	54,138	0.3
IN-2024-01490	Indore	Furniture	49,728	0.3

139 rows × 5 columns (first 6 shown)

Text values go in **single quotes** ('West'). Double quotes are for identifiers in standard SQL. Dates are written as text in ISO format: order_date ≥ '2025-01-01'.

IN and NOT IN

IN is shorthand for several OR conditions on the same column.

SQL

```
SELECT city, COUNT(*) AS order_lines
FROM superstore
WHERE city IN ('Pune', 'Mumbai', 'Ahmedabad')
GROUP BY city
ORDER BY city;
```

RESULT

city	order_lines
Ahmedabad	373
Mumbai	283
Pune	529

3 rows × 2 columns

SQL

```
SELECT COUNT(*) AS non_metro_lines
FROM superstore
WHERE city NOT IN ('Delhi', 'Mumbai', 'Bengaluru', 'Chennai', 'Kolkata', 'Hyderabad');
```

RESULT

non_metro_lines
3918

1 rows × 1 columns

COMMON MISTAKES NOT IN with a NULL

If the list (or subquery) inside `NOT IN` contains a `NULL`, `NOT IN` returns **no rows at all**, because `x <> NULL` is `UNKNOWN`. Use `NOT EXISTS` (Lesson 10) or filter `NULL`s out of the subquery.

BETWEEN

`BETWEEN a AND b` is **inclusive** of both ends.

SQL

```
SELECT COUNT(*) AS lines_between_10k_and_20k
FROM superstore
WHERE sales BETWEEN 10000 AND 20000;
```

RESULT

lines_between_10k_and_20k

896

1 rows × 1 columns

For dates, prefer a **half-open range**: `order_date ≥ '2025-01-01' AND order_date < '2026-01-01'`. It works whether the column is `DATE` or `DATETIME2`; `BETWEEN '2025-01-01' AND '2025-12-31'` silently misses orders placed after midnight on 31-Dec when the column has times.

SQL

```
SELECT COUNT(DISTINCT order_id) AS orders_in_2025
FROM superstore
WHERE order_date ≥ '2025-01-01' AND order_date < '2026-01-01';
```

RESULT

orders_in_2025

1232

1 rows × 1 columns

LIKE: pattern matching

Wildcard	Meaning	Example	Matches
%	Any sequence of characters (including none)	'Apex%'	Apex Chair 412, Apex Phone 118
_	Exactly one character	'C00_1'	C0011, C0021 ...
[] (SQL Server)	One character from a set or range	'[A-C]%'	starts with A, B or C
[^] (SQL Server)	One character not in the set	'[^A-M]%'	starts with N-Z

SQL

```
SELECT DISTINCT product_name
FROM superstore
WHERE product_name LIKE 'Apex Ch%'
ORDER BY product_name;
```

RESULT

product_name
Apex Chair 969

1 rows × 1 columns

SQL

```
SELECT COUNT(DISTINCT customer_id) AS customers_named_with_khan
FROM superstore
WHERE customer_name LIKE '%Khan%';
```

RESULT

customers_named_with_khan
44

1 rows × 1 columns

SQL

```
SELECT DISTINCT city
FROM superstore
WHERE city NOT LIKE '%a%'      -- city names without a lowercase 'a'
ORDER BY city;
```

RESULT

city
Delhi
Indore
Kochi
Lucknow
Pune

5 rows × 1 columns

NOTE Case sensitivity

SQL Server's default collation is **case-insensitive**, so LIKE 'apex%' also matches "Apex...". PostgreSQL is case-sensitive (use ILIKE), and so is the practice engine used for the outputs in this handbook. Write patterns in the same case as the data to be safe everywhere.

Leading wildcards ('%Khan') cannot use an index and scan the whole table; fine for analysis, slow for apps on big tables.

IS NULL and IS NOT NULL

SQL

```
SELECT COUNT(*) AS homes_missing_area
FROM houses
WHERE area_sqft IS NULL;
```

RESULT

homes_missing_area

25

1 rows × 1 columns

(The houses table is loaded from house_prices.csv; 25 houses have a missing area, which you will clean in the Python track.)

Combining conditions

SQL

```
SELECT order_id, region, category, ship_mode, sales, profit
FROM superstore
WHERE (region = 'North' OR region = 'East')
      AND category <> 'Office Supplies'
      AND ship_mode IN ('First Class', 'Same Day')
      AND profit < 0
ORDER BY profit;
```

RESULT

order_id	region	category	ship_mode	sales	profit
IN-2023-00320	North	Technology	Same Day	337,316	-67,392.8
IN-2024-01213	East	Technology	Same Day	317,128	-35,733.2
IN-2024-01808	North	Technology	First Class	155,328	-31,737.4
IN-2023-00406	North	Technology	First Class	159,390	-27,928.8
IN-2023-00358	North	Technology	Same Day	77,664	-18,678.2
IN-2024-01289	East	Technology	First Class	149,088	-18,075.8
IN-2025-02849	East	Technology	First Class	81,522	-17,598.3
IN-2024-01055	East	Technology	First Class	86,968	-16,505.7

128 rows × 6 columns (first 8 shown)

ORDER BY: sorting

SQL

```
SELECT region, city, COUNT(*) AS lines, ROUND(SUM(sales), 0) AS sales
FROM superstore
GROUP BY region, city
ORDER BY region ASC, sales DESC;
```

RESULT

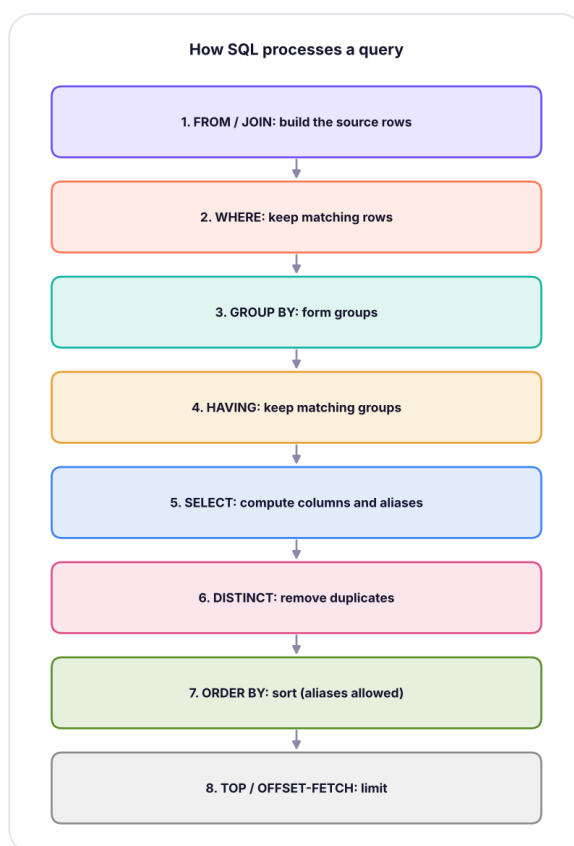
region	city	lines	sales
East	Kolkata	430	13,888,466
East	Patna	375	12,444,393
East	Bhubaneswar	383	10,482,685
East	Guwahati	335	9,668,576
North	Delhi	453	15,657,610
North	Chandigarh	361	11,541,792
North	Lucknow	422	11,328,882
North	Jaipur	294	10,489,878

16 rows × 4 columns (first 8 shown)

- ASC (default) or DESC for each column separately.
- Sort by an expression or alias: `ORDER BY sales - profit DESC, ORDER BY margin_pct.`
- Sort by column position (`ORDER BY 2 DESC`) works but is fragile; avoid it in saved code.
- NULLs sort **first** in ascending order in SQL Server (last in PostgreSQL/Oracle by default).
- Custom order with CASE: `ORDER BY CASE ship_mode WHEN 'Same Day' THEN 1 WHEN 'First Class' THEN 2 ELSE 3 END.`

The logical processing order

You *write* SELECT first, but SQL *processes* the clauses in a different order. This explains several common errors.



Logical processing order of a SELECT statement. Each step works on the output of the previous one.

Consequences:

- You **cannot** use a column alias in WHERE, because WHERE runs before SELECT: `WHERE margin_pct > 10` fails. Repeat the expression, or use a subquery/CTE.
- You **can** use an alias in ORDER BY, because it runs after SELECT.
- WHERE cannot contain aggregate functions (`WHERE SUM(sales) > 100`); use HAVING (Lesson 7).

SQL

```
-- Fails: Invalid column name 'margin_pct'
SELECT order_id, 100.0 * profit / sales AS margin_pct
FROM superstore
WHERE margin_pct > 10;

-- Works: repeat the expression in WHERE, use the alias in ORDER BY
SELECT order_id, 100.0 * profit / sales AS margin_pct
FROM superstore
WHERE 100.0 * profit / sales > 10
ORDER BY margin_pct DESC;
```

From business question to query

STEP BY STEP Translate a question

Question: "Which Technology orders in the South had a discount but still made more than ₹10,000 profit in 2025? Show the biggest first." 1. **Table:** superstore. 2. **Rows (WHERE):** category = Technology, region = South, discount > 0, profit > 10000, order date in 2025. 3. **Columns (SELECT):** order_id, order_date, city, sub_category, discount, profit. 4. **Order:** profit descending.

SQL

```
SELECT order_id, order_date, city, sub_category, discount, profit
FROM superstore
WHERE category = 'Technology'
  AND region = 'South'
  AND discount > 0
  AND profit > 10000
  AND order_date ≥ '2025-01-01' AND order_date < '2026-01-01'
ORDER BY profit DESC;
```

RESULT

order_id	order_date	city	sub_category	discount	profit
IN-2025-02318	2025-04-26	Kochi	Machines	0.1	21,209.4
IN-2025-02908	2025-10-29	Bengaluru	Phones	0.1	16,734.2
IN-2025-02157	2025-03-08	Bengaluru	Copiers	0.1	12,678.4
IN-2025-03094	2025-12-08	Bengaluru	Machines	0.1	11,292.0
IN-2025-02514	2025-07-09	Bengaluru	Machines	0.1	10,891.9
IN-2025-03146	2025-12-20	Bengaluru	Copiers	0.1	10,679.1
IN-2025-02547	2025-07-20	Kochi	Phones	0.1	10,558.6

7 rows × 6 columns

DO IT WITH AI Do it with AI

Give an assistant your table's column list and a question, and ask for SQL Server syntax:
"Table superstore(order_id, order_date, region, city, category, sub_category, sales, discount, profit). Write a T-SQL query for: Furniture orders in Kolkata or Patna in 2024 with negative profit, sorted by profit." Read the query before running it: check the date range, the brackets around OR, and that it uses the column names you gave.

Summary

KEY TAKEAWAYS

- SELECT columns and expressions; name them with AS; avoid SELECT * in saved queries.
- DISTINCT removes duplicate rows of the selected columns; TOP n with ORDER BY limits results.
- Filter with =, <, >, <=, >=, AND/OR/NOT (use brackets), IN, BETWEEN (inclusive), LIKE with % and _, and IS NULL.
- For dates use half-open ranges: >= start AND < next_start.
- ORDER BY sorts ascending by default; can use aliases.
- Logical order: FROM → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → TOP. That is why aliases fail in WHERE.

Quick quiz

6 questions · tap an answer to check it

1. Which query returns each distinct region exactly once?

- A `SELECT region FROM superstore`
- B `SELECT DISTINCT region FROM superstore`
- C `SELECT UNIQUE(region) FROM superstore`
- D `SELECT TOP 1 region FROM superstore`

Answer: B. `DISTINCT` removes duplicate rows from the result.

2. `WHERE sales BETWEEN 100 AND 200` includes which values?

- A 101 to 199
- B 100 to 200, both included
- C 100 to 199
- D 101 to 200

Answer: B. `BETWEEN` is inclusive of both boundaries.

3. Which pattern finds product names that contain "Desk" anywhere?

- A `LIKE 'Desk%'`
- B `LIKE '%Desk'`
- C `LIKE '%Desk%'`
- D `LIKE '_Desk_'`

Answer: C. `%` on both sides matches any characters before and after.

4. Why does WHERE `margin_pct > 10` fail when `margin_pct` is an alias defined in SELECT?

- A** WHERE is processed before SELECT, so the alias does not exist yet
- B Aliases can only be used with GROUP BY
- C `margin_pct` must be in brackets
- D You must use HAVING for all aliases

Answer: A. Logical order: WHERE comes before SELECT.

5. What is wrong with WHERE `order_date BETWEEN '2025-01-01' AND '2025-12-31'` on a DATETIME2 column?

- A Nothing
- B** It misses rows on 31-Dec after 00:00:00
- C It includes 2026 dates
- D BETWEEN does not work with dates

Answer: B. '2025-12-31' means midnight; later times that day are excluded. Use `< '2026-01-01'`.

6. SELECT TOP 3 * FROM orders without ORDER BY returns:

- A The 3 newest orders
- B The 3 largest orders
- C** Any 3 rows, in no guaranteed order
- D An error

Answer: C. Without ORDER BY, row order is not defined.

EXERCISE Superstore filters

Write queries for: (1) all Same Day orders in the West with sales over ₹50,000; (2) customers whose names start with A, B or C (distinct, sorted); (3) order lines with a 40% discount in the Furniture category, showing the 10 biggest losses; (4) the number of order lines in 2024 for the Corporate segment in Pune or Indore.

Show solution (worked answer)

SQL

```
-- (1)
SELECT order_id, order_date, city, sales FROM superstore
WHERE ship_mode = 'Same Day' AND region = 'West' AND sales > 50000 ORDER BY sales
DESC;
-- (2) SQL Server
SELECT DISTINCT customer_name FROM superstore WHERE customer_name LIKE '[A-C]%' ORDER
BY customer_name;
-- (3)
SELECT TOP 10 order_id, product_name, sales, profit FROM superstore
WHERE discount = 0.4 AND category = 'Furniture' ORDER BY profit ASC;
-- (4)
SELECT COUNT(*) AS lines FROM superstore
WHERE segment = 'Corporate' AND city IN ('Pune', 'Indore')
AND order_date ≥ '2024-01-01' AND order_date < '2025-01-01';
```

INTERVIEW CORNER Questions you may be asked

Q: What is the order of execution of a SQL query? FROM/JOIN, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY, then TOP/OFFSET-FETCH.

Q: Difference between WHERE and HAVING? WHERE filters rows before grouping and cannot use aggregates; HAVING filters groups after aggregation.

Q: How do you find employees whose name's second letter is "a"? WHERE name LIKE '_a%'.

Q: IN vs EXISTS? IN compares a value with a list or subquery result; EXISTS checks whether a correlated subquery returns any row. EXISTS is safer with NULLs and often efficient for large subqueries (Lesson 10).

SQL Functions: Aggregate, Text, Date & Number

Use SQL Server's built-in functions to summarise, clean and reshape data. Aggregate functions (COUNT, SUM, AVG, MIN, MAX, STRING_AGG), text functions (LEN, LEFT, RIGHT, SUBSTRING, CHARINDEX, REPLACE, TRIM, UPPER, CONCAT), date functions (GETDATE, YEAR, MONTH, DATEPART, DATENAME, DATEADD, DATEDIFF, EOMONTH) and number functions (ROUND, CEILING, FLOOR, ABS, POWER, SQRT), plus CAST and CONVERT.

● SQL class 4

✦ SQL Server 2022, 2026

Beginner–Intermediate

🕒 120 min

By the end of this lesson you can

- Distinguish scalar functions (one value per row) from aggregate functions (one value per group)
- Summarise data with COUNT(*), COUNT(column), COUNT(DISTINCT), SUM, AVG, MIN, MAX and STRING_AGG
- Clean and extract text with LEN, LEFT, RIGHT, SUBSTRING, CHARINDEX, REPLACE, TRIM, UPPER, LOWER and CONCAT
- Work with dates using YEAR, MONTH, DATEPART, DATENAME, DATEADD, DATEDIFF, EOMONTH and DATEFROMPARTS
- Round and convert numbers with ROUND, CEILING, FLOOR, ABS, POWER, SQRT, CAST, CONVERT and TRY_CAST

Two kinds of functions

Scalar vs aggregate

Scalar functions

- Work on each row separately
- UPPER(city), YEAR(order_date)
- ROUND(sales, 0), LEN(name)
- Output: same number of rows
- Can be used in SELECT, WHERE, ORDER BY

Aggregate functions

- Work on a set of rows
- SUM(sales), COUNT(*), AVG(profit)
- MIN(order_date), MAX(sales)
- Output: one row (or one per group)
- Used in SELECT and HAVING, not WHERE

Scalar functions return one value per row. Aggregate functions collapse many rows into one value (per group).

Aggregate functions

Function	Returns	NULL handling
COUNT(*)	Number of rows	Counts every row
COUNT(col)	Number of non-NULL values	Skips NULLs
COUNT(DISTINCT col)	Number of distinct non-NULL values	Skips NULLs
SUM(col)	Total	Skips NULLs
AVG(col)	Average of non-NULL values	Skips NULLs (they are not treated as 0)
MIN(col) , MAX(col)	Smallest / largest (numbers, text, dates)	Skips NULLs
STDEV(col) , VAR(col)	Sample standard deviation / variance	Skips NULLs
STRING_AGG(col, sep)	Values joined into one string	Skips NULLs

SQL

```

SELECT COUNT(*)                AS order_lines,
       COUNT(DISTINCT order_id) AS orders,
       COUNT(DISTINCT customer_id) AS customers,
       ROUND(SUM(sales), 0)      AS total_sales,
       ROUND(AVG(sales), 0)      AS avg_line_sales,
       MIN(order_date)          AS first_order,
       MAX(order_date)          AS last_order,
       ROUND(STDEV(sales), 0)    AS sd_sales
FROM superstore;

```

RESULT

order_lines	orders	customers	total_sales	avg_line_sales	first_order	last_order
6323	3200	785	201,413,569	31,854	2023-01-01	2025-12-31

1 rows × 8 columns

SQL

```

SELECT category,
       STRING_AGG(sub_category, ', ') AS sub_categories
FROM (SELECT DISTINCT category, sub_category FROM superstore) AS t
GROUP BY category
ORDER BY category;

```

RESULT

category	sub_categories
Furniture	Tables, Furnishings, Bookcases, Chairs
Office Supplies	Storage, Appliances, Art, Binders, Paper
Technology	Copiers, Phones, Machines, Accessories

3 rows × 2 columns

In SQL Server add `WITHIN GROUP (ORDER BY sub_category)` after `STRING_AGG(...)` to control the order of the joined values.

WATCH OUT AVG of integers

AVG on an INT column returns an integer in SQL Server (AVG of 1, 2 = 1). Use `AVG(quantity * 1.0)` or `AVG(CAST(quantity AS DECIMAL(10,2)))` for decimals.

Text (string) functions

Function	Example	Result
<code>LEN(s)</code>	<code>LEN('Pune ')</code>	4 (trailing spaces not counted)
<code>UPPER(s), LOWER(s)</code>	<code>UPPER('pune')</code>	PUNE
<code>LEFT(s, n), RIGHT(s, n)</code>	<code>LEFT('IN-2025-00042', 2)</code>	IN
<code>SUBSTRING(s, start, n)</code>	<code>SUBSTRING('IN-2025-00042', 4, 4)</code>	2025
<code>CHARINDEX(find, s)</code>	<code>CHARINDEX('-', 'IN-2025-00042')</code>	3
<code>REPLACE(s, old, new)</code>	<code>REPLACE('2025-04-27', '-', '/')</code>	2025/04/27
<code>TRIM(s), LTRIM, RTRIM</code>	<code>TRIM(' Delhi ')</code>	Delhi
<code>CONCAT(a, b, ...)</code>	<code>CONCAT(city, ', ', state)</code>	Pune, Maharashtra
<code>REVERSE(s)</code>	<code>REVERSE('abc')</code>	cba
<code>REPLICATE(s, n)</code>	<code>REPLICATE('*', 3)</code>	***
<code>STRING_SPLIT(s, sep)</code>	<code>SELECT value FROM STRING_SPLIT('a,b,c', ',')</code>	3 rows
<code>FORMAT(value, fmt)</code>	<code>FORMAT(1234567, 'N0', 'en-IN')</code>	12,34,567

SQL

```
SELECT TOP 6
  order_id,
  LEFT(order_id, 2) AS country,
  SUBSTRING(order_id, 4, 4) AS order_year,
  RIGHT(order_id, 5) AS order_no,
  product_name,
  LEFT(product_name, CHARINDEX(' ', product_name) - 1) AS brand,
  UPPER(city) AS city_upper,
  LEN(customer_name) AS name_length
FROM superstore
ORDER BY row_id;
```

RESULT

order_id	country	order_year	order_no	product_name	brand	city_upper	name_
IN-2023-00001	IN	2023	00001	Nova Phone 223	Nova	BENGALURU	12
IN-2023-00001	IN	2023	00001	Orbit Appliance 798	Orbit	BENGALURU	12
IN-2023-00001	IN	2023	00001	Orbit Table 248	Orbit	BENGALURU	12
IN-2023-00002	IN	2023	00002	Crest Storage 946	Crest	AHMEDABAD	11
IN-2023-00003	IN	2023	00003	Crest Bookcase 170	Crest	AHMEDABAD	10
IN-2023-00003	IN	2023	00003	Zenith Table 111	Zenith	AHMEDABAD	10

6 rows × 8 columns

The brand is everything before the first space: CHARINDEX(' ', product_name) finds the space position, and LEFT(..., position - 1) cuts there. This is the SQL version of Excel's LEFT(A2, FIND(" ", A2) - 1) from Excel Lesson 5.

Cleaning text

SQL

```
WITH raw AS (  
    SELECT ' bengaluru ' AS city UNION ALL  
    SELECT 'BENGALURU'      UNION ALL  
    SELECT 'Bangalore'      UNION ALL  
    SELECT 'Bengaluru'  
)  
SELECT city                                AS raw_value,  
    CONCAT(UPPER(LEFT(TRIM(city), 1)), LOWER(SUBSTRING(TRIM(city), 2, 100))) AS  
proper_case,  
    CASE WHEN LOWER(TRIM(city)) IN ('bangalore', 'bengaluru') THEN 'Bengaluru' ELSE  
TRIM(city) END AS standardised  
FROM raw;
```

RESULT

raw_value	proper_case	standardised
bengaluru	Bengaluru	Bengaluru
BENGALURU	Bengaluru	Bengaluru
Bangalore	Bangalore	Bengaluru
Bengaluru	Bengaluru	Bengaluru

4 rows × 3 columns

(SQL Server has no PROPER function; joining UPPER(LEFT(...,1)) and LOWER(SUBSTRING(...,2,...)) capitalises the first letter of a single word. Standardising spellings is best done with CASE or a mapping table joined in.)

NOTE '+' vs CONCAT for text

In SQL Server 'a' + 'b' joins text, but 'a' + NULL is NULL and 'Q' + 2 fails (it tries to convert 'Q' to a number). CONCAT('Q', 2) gives Q2 and treats NULL as empty. Prefer CONCAT.

Date and time functions

Function	Example (d = '2026-04-27')	Result
GETDATE(), SYSDATETIME()		Current date and time
CAST(GETDATE() AS DATE)		Today's date
YEAR(d), MONTH(d), DAY(d)	MONTH(d)	4
DATEPART(part, d)	DATEPART(quarter, d)	2
DATENAME(part, d)	DATENAME(month, d), DATENAME(weekday, d)	April, Monday
DATEADD(part, n, d)	DATEADD(month, 3, d)	2026-07-27
DATEDIFF(part, start, end)	DATEDIFF(day, '2026-04-01', d)	26
EOMONTH(d, [n])	EOMONTH(d)	2026-04-30
DATEFROMPARTS(y, m, d)	DATEFROMPARTS(2026, 4, 1)	2026-04-01
DATETRUNC(part, d) (2022+)	DATETRUNC(month, d)	2026-04-01
FORMAT(d, 'MMM-yyyy')		Apr-2026

Date parts you can use: year, quarter, month, dayofyear, day, week, weekday, hour, minute, second.

SQL

```
SELECT TOP 6
    order_id,
    order_date,
    ship_date,
    YEAR(order_date)           AS yr,
    DATEPART(quarter, order_date) AS qtr,
    DATENAME(month, order_date)  AS month_name,
    DATENAME(weekday, order_date) AS weekday_name,
    DATEDIFF(day, order_date, ship_date) AS days_to_ship,
    EOMONTH(order_date)         AS month_end
FROM orders
ORDER BY order_date DESC;
```

RESULT

order_id	order_date	ship_date	yr	qtr	month_name	weekday_name	days_to_ship
IN-2025-03197	2025-12-31	2026-01-04	2025	4	December	Wednesday	4
IN-2025-03198	2025-12-31	2026-01-04	2025	4	December	Wednesday	4
IN-2025-03199	2025-12-31	2026-01-02	2025	4	December	Wednesday	2
IN-2025-03200	2025-12-31	2026-01-06	2025	4	December	Wednesday	6
IN-2025-03194	2025-12-30	2026-01-03	2025	4	December	Tuesday	4
IN-2025-03195	2025-12-30	2026-01-06	2025	4	December	Tuesday	7

6 rows × 9 columns

SQL

```
SELECT ship_mode,
       COUNT(*) AS orders,
       AVG(DATEDIFF(day, order_date, ship_date) * 1.0) AS avg_days_to_ship,
       MAX(DATEDIFF(day, order_date, ship_date)) AS max_days
FROM orders
GROUP BY ship_mode
ORDER BY avg_days_to_ship;
```

RESULT

ship_mode	orders	avg_days_to_ship	max_days
Same Day	176	0.00	0
First Class	467	1.53	2
Second Class	621	3.00	4
Standard Class	1936	5.54	7

4 rows × 4 columns

WATCH OUT DATEDIFF counts boundaries

DATEDIFF(year, '2025-12-31', '2026-01-01') returns **1**, because one year boundary is crossed, even though only one day passed. For exact age use DATEDIFF(day, ...)/365.25 or compare month and day as well.

Useful date patterns

SQL

```
-- Orders in the last 30 days
WHERE order_date ≥ DATEADD(day, -30, CAST(GETDATE() AS DATE))

-- First day of the month
SELECT DATEFROMPARTS(YEAR(order_date), MONTH(order_date), 1) -- or DATETRUNC(month,
order_date)

-- Indian financial year (April-March)
SELECT CASE WHEN MONTH(order_date) ≥ 4 THEN YEAR(order_date) + 1 ELSE YEAR(order_date) END
AS fy

-- Month label for reports
SELECT FORMAT(order_date, 'MMM-yyyy') AS month_label
```

Number functions

Function	Example	Result
ROUND(n, d)	ROUND(1234.567, 1)	1234.600
ROUND(n, -2)	ROUND(1234.567, -2)	1200.000
CEILING(n)	CEILING(4.1)	5
FLOOR(n)	FLOOR(4.9)	4
ABS(n)	ABS(-250)	250
POWER(n, p)	POWER(2, 10)	1024
SQRT(n), SQUARE(n)	SQRT(144)	12
SIGN(n)	SIGN(-3)	-1
n % m	17 % 5	2
RAND()		Random float 0-1

SQL

```
SELECT sub_category,
       ROUND(SUM(sales), -3) AS sales_to_thousand,
       CEILING(SUM(quantity) / 12.0) AS boxes_of_12,
       ROUND(100.0 * SUM(profit) / SUM(sales), 2) AS margin_pct,
       ABS(SUM(profit)) AS abs_profit,
       SIGN(SUM(profit)) AS profit_sign
FROM superstore
GROUP BY sub_category
ORDER BY margin_pct;
```

RESULT

sub_category	sales_to_thousand	boxes_of_12	margin_pct	abs_profit	profit_sign
Tables	14,046,000	103	-5.81	816,042	-1
Bookcases	11,275,000	101	-3.75	422,786	-1
Chairs	8,091,000	110	4.23	342,043	1
Furnishings	1,429,000	94	5.50	78,632	1
Copiers	78,750,000	99	10.71	8,436,141	1
Phones	31,919,000	110	11.48	3,663,117	1
Accessories	3,179,000	98	11.63	369,554	1
Machines	42,590,000	106	12.18	5,186,711	1
Art	246,000	103	14.85	36,526	1
Appliances	6,334,000	106	14.92	944,927	1
Storage	2,659,000	107	14.94	397,228	1
Paper	493,000	124	15.21	75,040	1

13 rows × 6 columns (first 12 shown)

Converting types: CAST, CONVERT, TRY_CAST

Function	Example	Notes
CAST(x AS type)	CAST('2026-04-27' AS DATE)	ANSI standard
CONVERT(type, x, style)	CONVERT(VARCHAR(10), order_date, 103) → 27/04/2026	SQL Server; style codes format dates (103 = dd/mm/yyyy, 23 = yyyy-mm-dd)
TRY_CAST(x AS type)	TRY_CAST('abc' AS INT) → NULL	Returns NULL instead of an error: great for dirty data
TRY_CONVERT, TRY_PARSE		Same idea

SQL

```
-- Find rows where a text column does not contain a valid number
SELECT * FROM staging_sales WHERE amount_text IS NOT NULL AND TRY_CAST(amount_text AS DECIMAL(12,2)) IS NULL;
```

Conditional functions (preview)

Function	Example
COALESCE(a, b, ...)	COALESCE(phone, email, 'no contact')
ISNULL(a, b)	ISNULL(discount, 0)
NULLIF(a, b)	profit / NULLIF(sales, 0)
IIF(cond, a, b)	IIF(profit < 0, 'Loss', 'Profit')
CASE WHEN ... END	Multi-branch logic (Lesson 7)

SQL

```
SELECT TOP 6 order_id, sales, profit,
      IIF(profit < 0, 'Loss', 'Profit') AS result,
      ROUND(profit / NULLIF(sales, 0) * 100, 1) AS margin_pct
FROM superstore
ORDER BY profit;
```

RESULT

order_id	sales	profit	result	margin_pct
IN-2025-03027	318,492	-75,874	Loss	-23.8
IN-2023-00320	337,316	-67,393	Loss	-20.0
IN-2023-00372	378,192	-59,796	Loss	-15.8
IN-2024-01335	194,160	-56,392	Loss	-29.0
IN-2025-02299	202,475	-46,375	Loss	-22.9
IN-2023-00077	240,940	-45,712	Loss	-19.0

6 rows × 5 columns

DO IT WITH AI

Do it with AI

Prompt: "In SQL Server, write a query on superstore(order_id, order_date, ship_date, ship_mode) that returns the average and 90th percentile of days to ship by ship mode and quarter." Expect it to use DATEDIFF, DATEPART(quarter, ...) and PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY ...) OVER (PARTITION BY ...), which you will meet with window functions in Lesson 12.

Summary

KEY TAKEAWAYS

- **Scalar** functions work per row; **aggregate** functions summarise many rows.
- `COUNT(*)` counts rows; `COUNT(col)` skips NULLs; `COUNT(DISTINCT col)` counts unique values; `AVG` ignores NULLs.
- Text: `LEN`, `LEFT/RIGHT/SUBSTRING`, `CHARINDEX`, `REPLACE`, `TRIM`, `UPPER/LOWER`, `CONCAT`, `STRING_AGG`, `STRING_SPLIT`.
- Dates: `YEAR/MONTH/DAY`, `DATEPART`, `DATENAME`, `DATEADD`, `DATEDIFF` (counts boundaries), `EOMONTH`, `DATEFROMPARTS`, `DATETRUNC`.
- Numbers: `ROUND` (negative digits round to tens/hundreds), `CEILING`, `FLOOR`, `ABS`, `POWER`, `SQRT`, `%`.
- Convert with `CAST/CONVERT`; use `TRY_CAST` on dirty data.

Quick quiz

6 questions · tap an answer to check it

1. A column has values 10, NULL, 20. What does `AVG(col)` return?

A 10

B 15

C 30

D NULL

Answer: B. `AVG` ignores NULLs: $(10 + 20) / 2 = 15$.

2. What does `SUBSTRING('IN-2025-00042', 4, 4)` return?

A IN-2

B 2025

C 0004

D 5-00

Answer: B. Starting at position 4, take 4 characters.

3. Which returns the number of days between order and ship dates?

A `DATEPART(day, ship_date - order_date)`

B `DATEDIFF(day, order_date, ship_date)`

C `DATEADD(day, order_date, ship_date)`

D `DAY(ship_date) - DAY(order_date)`

Answer: B. `DATEDIFF(datepart, start, end)` counts the boundaries between the two dates.

4. What does `ROUND(45678.9, -3)` return?

A 45678.000

B 45679

C 46000.0

D 45000

Answer: C. -3 rounds to the nearest thousand.

5. Which function returns NULL instead of an error when text cannot be converted?

A CAST

B CONVERT

C TRY_CAST

D ISNULL

Answer: C. TRY_CAST and TRY_CONVERT return NULL on conversion failure.

6. Which counts unique customers?

A COUNT(customer_id)

B COUNT(DISTINCT customer_id)

C SUM(DISTINCT customer_id)

D DISTINCT COUNT(customer_id)

Answer: B. COUNT(DISTINCT ...) counts distinct non-NULL values.

EXERCISE Function practice on Superstore

Write one query that returns, for each order year: the number of orders, the number of distinct customers, total sales rounded to the nearest lakh (100,000), the average days to ship, and a comma-separated list of the ship modes used.

Show solution (worked answer)

SQL

```
WITH modes AS (                                     -- one row per year and ship mode
    SELECT DISTINCT YEAR(order_date) AS order_year, ship_mode FROM orders
),
mode_list AS (
    SELECT order_year, STRING_AGG(ship_mode, ', ') AS ship_modes FROM modes GROUP BY
    order_year
)
SELECT YEAR(o.order_date)                           AS order_year,
       COUNT(DISTINCT o.order_id)                   AS orders,
       COUNT(DISTINCT o.customer_id)                 AS customers,
       ROUND(SUM(oi.sales), -5)                      AS sales_nearest_lakh,
       AVG(DATEDIFF(day, o.order_date, o.ship_date) * 1.0) AS avg_days_to_ship,
       MAX(m.ship_modes)                            AS ship_modes
FROM orders AS o
JOIN order_items AS oi ON oi.order_id = o.order_id
JOIN mode_list AS m ON m.order_year = YEAR(o.order_date)
GROUP BY YEAR(o.order_date)
ORDER BY order_year;
```

STRING_AGG(DISTINCT ...) is not allowed in SQL Server, so the distinct ship modes are collected in a CTE first (CTEs are covered in Lesson 13). Note that averaging days to ship over order *lines* weights orders with more items more heavily; average over orders alone if each order should count once.

INTERVIEW CORNER**Questions you may be asked**

Q: *COUNT()* vs *COUNT(column)* vs *COUNT(DISTINCT column)*? *COUNT()* counts rows; *COUNT(column)* counts non-NULL values; *COUNT(DISTINCT column)* counts unique non-NULL values.

Q: **How do you extract the domain from an email in SQL Server?** `SUBSTRING(email, CHARINDEX('@', email) + 1, LEN(email))` or `RIGHT(email, LEN(email) - CHARINDEX('@', email))`.

Q: **How do you get the last day of the previous month?** `EOMONTH(GETDATE(), -1)`.

Q: **Difference between CAST and CONVERT?** `CAST` is ANSI standard; `CONVERT` is SQL Server-specific and accepts a style argument for date/number formatting.

GROUP BY, HAVING, CASE WHEN, ROLLUP, CUBE & COALESCE

Turn rows into business summaries. Group with GROUP BY, filter groups with HAVING, create conditional columns and buckets with CASE WHEN, pivot data with conditional aggregation, and add subtotals and grand totals with ROLLUP, CUBE and GROUPING SETS, cleaning the labels with COALESCE and GROUPING().

● SQL class 4 · SQL–Power BI bridge

✦ SQL Server 2022, 2026

Intermediate

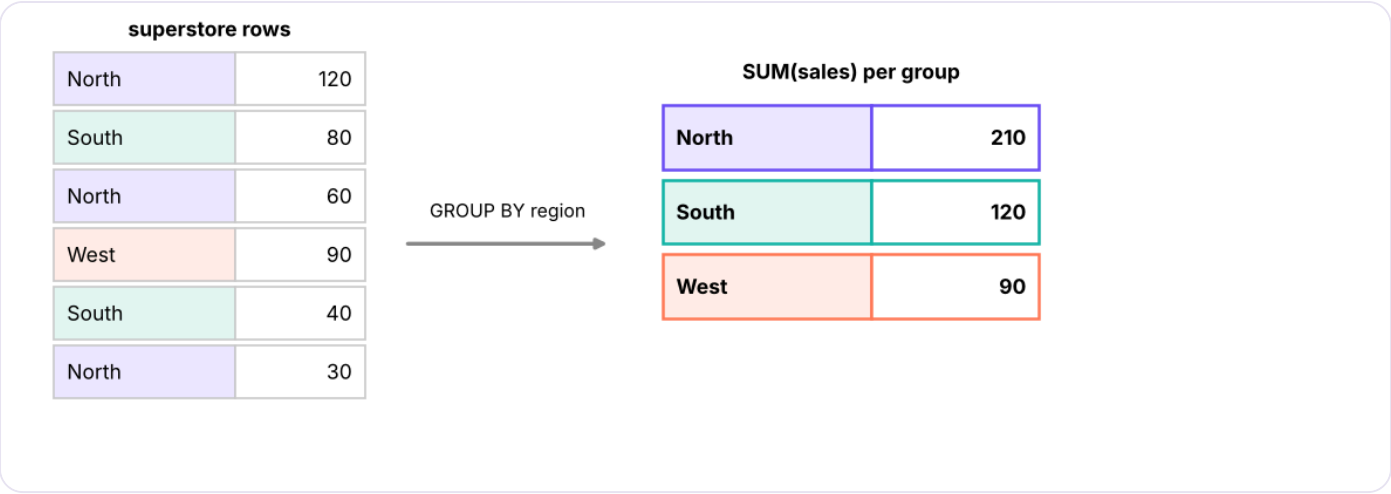
🕒 130 min

By the end of this lesson you can

- Summarise data by one or more columns with GROUP BY and aggregate functions
- Filter groups with HAVING and explain WHERE vs HAVING
- Create conditional columns and buckets with CASE WHEN ... THEN ... ELSE ... END
- Build pivot-style reports with conditional aggregation
- Add subtotals and grand totals with ROLLUP, CUBE and GROUPING SETS, and label them with COALESCE and GROUPING()

GROUP BY: one row per group

`GROUP BY` collapses rows that share the same value(s) into one row, and aggregate functions summarise each group. It is the SQL version of a pivot table (Excel Lesson 8) and of `groupby()` in pandas.



GROUP BY region: rows with the same region are collected into one group, then SUM is computed for each group.

SQL

```
SELECT region,
       COUNT(DISTINCT order_id) AS orders,
       ROUND(SUM(sales), 0) AS sales,
       ROUND(SUM(profit), 0) AS profit,
       ROUND(100.0 * SUM(profit) / SUM(sales), 1) AS margin_pct
FROM superstore
GROUP BY region
ORDER BY sales DESC;
```

RESULT

region	orders	sales	profit	margin_pct
South	831	53,952,769	4,702,012	8.7
West	811	51,958,521	4,719,833	9.1
North	792	49,018,161	4,555,830	9.3
East	766	46,484,119	4,374,490	9.4

4 rows × 5 columns

Grouping by several columns

SQL

```
SELECT category,
       sub_category,
       COUNT(*)           AS lines,
       ROUND(SUM(sales), 0) AS sales,
       ROUND(SUM(profit), 0) AS profit
FROM superstore
GROUP BY category, sub_category
ORDER BY category, sales DESC;
```

RESULT

category	sub_category	lines	sales	profit
Furniture	Tables	505	14,046,224	-816,042
Furniture	Bookcases	477	11,274,849	-422,786
Furniture	Chairs	493	8,091,474	342,043
Furniture	Furnishings	442	1,429,307	78,632
Office Supplies	Appliances	492	6,333,529	944,927
Office Supplies	Storage	507	2,659,340	397,228
Office Supplies	Paper	559	493,232	75,040
Office Supplies	Binders	494	401,219	61,074
Office Supplies	Art	462	245,985	36,526
Technology	Copiers	468	78,749,926	8,436,141
Technology	Machines	491	42,590,335	5,186,711
Technology	Phones	488	31,919,223	3,663,117

13 rows × 5 columns (first 12 shown)

Grouping by an expression

SQL

```
SELECT YEAR(order_date) AS order_year,
       COUNT(DISTINCT order_id) AS orders,
       ROUND(SUM(sales), 0) AS sales
FROM superstore
GROUP BY YEAR(order_date)
ORDER BY order_year;
```

RESULT

order_year	orders	sales
2023	883	57,640,217
2024	1085	65,396,556
2025	1232	78,376,797

3 rows × 3 columns

WATCH OUT The golden rule of GROUP BY

Every column in SELECT must either be **in the GROUP BY** or **inside an aggregate function**.
SELECT region, city, SUM(sales) FROM superstore GROUP BY region fails in SQL Server: "Column 'city' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause." Which city should it show for the North group? SQL refuses to guess.

HAVING: filtering groups

WHERE filters **rows before** grouping; HAVING filters **groups after** aggregation.

SQL

```
SELECT city,
       COUNT(DISTINCT order_id) AS orders,
       ROUND(SUM(profit), 0) AS profit
FROM superstore
WHERE category = 'Furniture'           -- 1. keep only furniture rows
GROUP BY city                          -- 2. one row per city
HAVING SUM(profit) < 0                 -- 3. keep only loss-making cities
ORDER BY profit;
```

RESULT

city	orders	profit
Chennai	100	-122,258
Delhi	101	-106,170
Chandigarh	92	-92,165
Lucknow	100	-87,616
Guwahati	77	-80,036
Kochi	93	-71,758
Bengaluru	101	-52,961
Bhubaneswar	80	-50,956
Ahmedabad	96	-50,770
Kolkata	111	-39,811
Patna	86	-38,624
Indore	102	-30,482

15 rows × 3 columns (first 12 shown)

	WHERE	HAVING
Runs	Before GROUP BY	After GROUP BY
Filters	Individual rows	Groups
Can use aggregates	No	Yes
Example	WHERE region = 'West'	HAVING COUNT(*) > 100
Performance	Reduces rows early (faster)	Applied to fewer, summarised rows

TIP Filter as early as possible

Put conditions that do not need aggregates in WHERE, not HAVING: HAVING region = 'West' works but makes the database group every region first and discard most of them.

CASE WHEN: conditional columns

CASE is SQL's IF/IFS (Excel Lesson 5). It returns a value for the first condition that is true.

SQL

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ELSE default_result      -- optional; without ELSE, unmatched rows get NULL
END
```

SQL

```
SELECT order_id, sales, discount, profit,
       CASE
         WHEN discount = 0    THEN 'No discount'
         WHEN discount ≤ 0.2 THEN 'Up to 20%'
         ELSE 'Above 20%'
       END AS discount_band,
       CASE WHEN profit < 0 THEN 'Loss' ELSE 'Profit' END AS result
FROM superstore
ORDER BY row_id;
```

RESULT

order_id	sales	discount	profit	discount_band	result
IN-2023-00001	36,000	0.0	3,472.55	No discount	Profit
IN-2023-00001	3,850	0.0	1,169.54	No discount	Profit
IN-2023-00001	62,080	0.0	6,044.47	No discount	Profit
IN-2023-00002	5,216	0.2	450.16	Up to 20%	Profit
IN-2023-00003	24,591	0.3	-5,512.33	Above 20%	Loss
IN-2023-00003	8,288	0.3	-1,598.43	Above 20%	Loss
IN-2023-00003	2,100	0.0	510.43	No discount	Profit
IN-2023-00003	69,280	0.0	15,852.15	No discount	Profit

6,323 rows × 6 columns (first 8 shown)

The **simple** form compares one expression with values:

SQL

```
CASE ship_mode
  WHEN 'Same Day' THEN 1
  WHEN 'First Class' THEN 2
  ELSE 3
END
```

CASE inside GROUP BY: buckets

SQL

```
SELECT CASE
  WHEN discount = 0 THEN '0%'
  WHEN discount ≤ 0.2 THEN '1-20%'
  ELSE '>20%'
END AS discount_band,
COUNT(*) AS lines,
ROUND(SUM(sales), 0) AS sales,
ROUND(SUM(profit), 0) AS profit,
ROUND(100.0 * SUM(profit) / SUM(sales), 1) AS margin_pct
FROM superstore
GROUP BY CASE
  WHEN discount = 0 THEN '0%'
  WHEN discount ≤ 0.2 THEN '1-20%'
  ELSE '>20%'
END
ORDER BY discount_band;
```

RESULT

discount_band	lines	sales	profit	margin_pct
0%	3480	123,936,290	21,718,963	17.5
1-20%	1943	57,729,097	697,141	1.2
>20%	900	19,748,182	-4,063,940	-20.6

3 rows × 5 columns

The same insight as the Excel crosstab and dashboard: discounts above 20% turn the margin sharply negative.

Conditional aggregation: pivoting with CASE

Put CASE inside an aggregate to turn row values into columns, which is how analysts build pivot-style reports in SQL:

SQL

```
SELECT region,
       ROUND(SUM(CASE WHEN category = 'Furniture' THEN sales ELSE 0 END), 0) AS
furniture,
       ROUND(SUM(CASE WHEN category = 'Office Supplies' THEN sales ELSE 0 END), 0) AS
office_supplies,
       ROUND(SUM(CASE WHEN category = 'Technology' THEN sales ELSE 0 END), 0) AS
technology,
       ROUND(SUM(sales), 0) AS total
FROM superstore
GROUP BY region
ORDER BY region;
```

RESULT

region	furniture	office_supplies	technology	total
East	7,942,093	2,719,513	35,822,514	46,484,119
North	7,975,869	2,219,543	38,822,750	49,018,161
South	9,593,302	2,468,888	41,890,579	53,952,769
West	9,330,590	2,725,361	39,902,570	51,958,521

4 rows × 5 columns

SQL

```
SELECT department,
       COUNT(*) AS employees,
       SUM(CASE WHEN attrition = 'Yes' THEN 1 ELSE 0 END) AS leavers,
       ROUND(100.0 * SUM(CASE WHEN attrition = 'Yes' THEN 1 ELSE 0 END) / COUNT(*), 1) AS
attrition_pct,
       ROUND(AVG(CASE WHEN overtime = 'Yes' THEN 1.0 ELSE 0 END) * 100, 1) AS
overtime_pct
FROM employees
GROUP BY department
ORDER BY attrition_pct DESC;
```

RESULT

department	employees	leavers	attrition_pct	overtime_pct
Sales	414	126	30.4	26.6
R&D	389	101	26.0	32.1
IT	309	75	24.3	29.8
HR	83	19	22.9	31.3
Finance	154	35	22.7	20.1
Operations	151	33	21.9	32.5

6 rows × 5 columns

AVG(CASE WHEN ... THEN 1.0 ELSE 0 END) is a neat way to get a percentage of rows meeting a condition.

NOTE The PIVOT operator

SQL Server also has a PIVOT operator:

SQL

```
SELECT region, [Furniture], [Office Supplies], [Technology]
FROM (SELECT region, category, sales FROM superstore) AS src
PIVOT (SUM(sales) FOR category IN ([Furniture], [Office Supplies], [Technology])) AS
p;
```

Conditional aggregation is more flexible (several measures, any conditions) and works in every database, so most analysts prefer it. UNPIVOT does the reverse, like Power Query's Unpivot (Excel Lesson 12).

Subtotals and grand totals: ROLLUP, CUBE, GROUPING SETS

ROLLUP

GROUP BY ROLLUP (a, b) returns the groups for (a, b), **subtotals** for each a, and a **grand total**. Subtotal rows show NULL in the rolled-up columns.

SQL

```
SELECT region,  
       category,  
       ROUND(SUM(sales), 0) AS sales  
FROM superstore  
GROUP BY ROLLUP (region, category)  
ORDER BY region, category;
```

RESULT

region	category	sales
East	Furniture	7,942,093
East	Office Supplies	2,719,513
East	Technology	35,822,514
East	NULL	46,484,119
North	Furniture	7,975,869
North	Office Supplies	2,219,543
North	Technology	38,822,750
North	NULL	49,018,161
South	Furniture	9,593,302
South	Office Supplies	2,468,888
South	Technology	41,890,579
South	NULL	53,952,769
West	Furniture	9,330,590
West	Office Supplies	2,725,361
West	Technology	39,902,570
West	NULL	51,958,521
NULL	NULL	201,413,569

17 rows × 3 columns

Cleaning the labels with COALESCE and GROUPING()

NULLs in subtotal rows are confusing. COALESCE replaces them with a label. If the data itself could contain NULL categories, use GROUPING(col), which returns 1 only on rows where the column was rolled up:

SQL

```
SELECT CASE WHEN GROUPING(region) = 1 THEN 'All regions' ELSE region END      AS
region,
       CASE WHEN GROUPING(category) = 1 THEN 'Subtotal' ELSE category END    AS
category,
       ROUND(SUM(sales), 0)                                                  AS
sales
FROM superstore
GROUP BY ROLLUP (region, category)
ORDER BY GROUPING(region), region, GROUPING(category), category;
```

RESULT

region	category	sales
East	Furniture	7,942,093
East	Office Supplies	2,719,513
East	Technology	35,822,514
East	Subtotal	46,484,119
North	Furniture	7,975,869
North	Office Supplies	2,219,543
North	Technology	38,822,750
North	Subtotal	49,018,161
South	Furniture	9,593,302
South	Office Supplies	2,468,888
South	Technology	41,890,579
South	Subtotal	53,952,769
West	Furniture	9,330,590
West	Office Supplies	2,725,361
West	Technology	39,902,570
West	Subtotal	51,958,521
All regions	Subtotal	201,413,569

17 rows × 3 columns

The simple version used in class, `COALESCE(region, 'All regions')`, gives the same labels when the data has no real NULLs.

CUBE

`CUBE (a, b)` returns **every combination** of subtotals: by (a, b), by a, by b, and the grand total. Use it when you need totals in both directions, like a pivot table with row and column grand totals.

SQL

```
SELECT COALESCE(segment, 'All segments') AS segment,  
       COALESCE(region, 'All regions')   AS region,  
       COUNT(DISTINCT order_id)          AS orders  
FROM superstore  
GROUP BY CUBE (segment, region)  
ORDER BY GROUPING(segment), segment, GROUPING(region), region;
```

RESULT

segment	region	orders
Consumer	East	355
Consumer	North	402
Consumer	South	385
Consumer	West	412
Consumer	All regions	1554
Corporate	East	245
Corporate	North	224
Corporate	South	269
Corporate	West	217
Corporate	All regions	955
Home Office	East	166
Home Office	North	166
Home Office	South	177
Home Office	West	182
Home Office	All regions	691
All segments	East	766
All segments	North	792
All segments	South	831
All segments	West	811
All segments	All regions	3200

20 rows × 3 columns

GROUPING SETS

GROUPING SETS lists exactly the groupings you want in one query:

SQL

```
SELECT COALESCE(category, '-') AS category,
       COALESCE(ship_mode, '-') AS ship_mode,
       ROUND(SUM(sales), 0) AS sales
FROM superstore
GROUP BY GROUPING SETS ((category), (ship_mode), ())
ORDER BY category, ship_mode;
```

RESULT

category	ship_mode	sales
Furniture	—	34,841,853
Office Supplies	—	10,133,304
Technology	—	156,438,413
—	First Class	31,455,543
—	Same Day	10,082,430
—	Second Class	40,858,316
—	Standard Class	119,017,281
—	—	201,413,569

8 rows × 3 columns

This returns totals by category, totals by ship mode and the grand total, without the category × ship mode detail.

Syntax	Groupings produced for (a, b)
GROUP BY a, b	(a, b)
ROLLUP (a, b)	(a, b), (a), ()
CUBE (a, b)	(a, b), (a), (b), ()
GROUPING SETS ((a), (b), ())	exactly (a), (b), ()

TIP Bridge to Power BI

Power BI matrices compute these subtotals automatically, but ROLLUP/CUBE are perfect when you must deliver a fixed report table from SQL, feed a paginated report, or check that a dashboard's totals are right.

A complete analysis query

Question: "For 2025, show each sub-category's sales, profit and margin, a performance label, and only sub-categories with at least ₹10 lakh sales, best margin first."

SQL

```
SELECT sub_category,
       ROUND(SUM(sales), 0)           AS sales,
       ROUND(SUM(profit), 0)          AS profit,
       ROUND(100.0 * SUM(profit) / SUM(sales), 1) AS margin_pct,
       CASE
         WHEN SUM(profit) < 0          THEN 'Loss-making'
         WHEN 100.0 * SUM(profit) / SUM(sales) < 8 THEN 'Low margin'
         ELSE 'Healthy'
       END                           AS status
FROM superstore
WHERE order_date ≥ '2025-01-01' AND order_date < '2026-01-01'
GROUP BY sub_category
HAVING SUM(sales) ≥ 1000000
ORDER BY margin_pct DESC;
```

RESULT

sub_category	sales	profit	margin_pct	status
Appliances	2,539,536	379,442	14.9	Healthy
Phones	12,935,910	1,708,007	13.2	Healthy
Machines	16,502,707	2,000,026	12.1	Healthy
Accessories	1,221,677	145,007	11.9	Healthy
Copiers	30,984,937	3,349,006	10.8	Healthy
Chairs	3,053,530	132,519	4.3	Low margin
Bookcases	3,918,269	-169,018	-4.3	Loss-making
Tables	5,297,064	-286,188	-5.4	Loss-making

Summary

KEY TAKEAWAYS

- `GROUP BY` makes one row per group; every selected column must be grouped or aggregated.
- `WHERE` filters rows before grouping; `HAVING` filters groups after (can use aggregates).
- `CASE WHEN ... THEN ... ELSE ... END` creates conditional columns and buckets; without `ELSE`, unmatched rows are `NULL`.
- Conditional aggregation (`SUM(CASE WHEN ... THEN x ELSE 0 END)`) pivots rows into columns and computes percentages.
- `ROLLUP` adds hierarchical subtotals + grand total; `CUBE` adds all combinations; `GROUPING SETS` picks exactly the groupings you need.
- Label subtotal rows with `COALESCE` or, more robustly, `GROUPING()`.

Quick quiz

6 questions · tap an answer to check it

1. Why does `SELECT region, city, SUM(sales) FROM superstore GROUP BY region` fail?

- A SUM needs a WHERE clause
- B city is neither grouped nor aggregated**
- C region must be aggregated
- D GROUP BY needs ORDER BY

Answer: B. Every selected column must appear in `GROUP BY` or in an aggregate.

2. Which clause keeps only customers with more than 10 orders?

- A WHERE COUNT(order_id) > 10
- B HAVING COUNT(DISTINCT order_id) > 10**
- C ORDER BY COUNT(*) > 10
- D GROUP BY COUNT(*) > 10

Answer: B. Conditions on aggregates go in HAVING.

3. What does a CASE expression return when no WHEN matches and there is no ELSE?

- A 0
- B An empty string
- C NULL**
- D An error

Answer: C. Without ELSE, unmatched rows return NULL.

4. GROUP BY ROLLUP (region, category) produces which groupings?

- A (region, category), (region), and the grand total**
- B Every combination including (category) alone
- C Only the grand total
- D Only (region, category)

Answer: A. ROLLUP is hierarchical; CUBE would also include (category).

5. What does `AVG(CASE WHEN attrition = 'Yes' THEN 1.0 ELSE 0 END)` compute?

- A The number of leavers
- B The share (proportion) of employees who left
- C The average salary of leavers
- D NULL

Answer: B. The average of 1s and 0s is the proportion of 1s.

6. Which function tells you that a row is a subtotal created by ROLLUP?

- A COALESCE
- B ISNULL
- C GROUPING
- D NULLIF

Answer: C. `GROUPING(col)` returns 1 when col was aggregated away in that row.

EXERCISE HR summary

Using the employees table (HR data): (1) headcount, average monthly income and attrition % by department and gender, only for groups with at least 30 employees; (2) a column per education level showing headcount, one row per department; (3) average income by job satisfaction (1–4) with a grand total row using ROLLUP.

Show solution (worked answer)

SQL

```
-- (1)
SELECT department, gender, COUNT(*) AS headcount,
       ROUND(AVG(monthly_income * 1.0), 0) AS avg_income,
       ROUND(100.0 * SUM(CASE WHEN attrition = 'Yes' THEN 1 ELSE 0 END) / COUNT(*), 1)
AS attrition_pct
FROM employees GROUP BY department, gender HAVING COUNT(*) ≥ 30 ORDER BY department,
gender;
-- (2)
SELECT department,
       SUM(CASE WHEN education = 'Diploma' THEN 1 ELSE 0 END) AS diploma,
       SUM(CASE WHEN education = 'Bachelor's' THEN 1 ELSE 0 END) AS bachelors,
       SUM(CASE WHEN education = 'Master's' THEN 1 ELSE 0 END) AS masters,
       SUM(CASE WHEN education = 'PhD' THEN 1 ELSE 0 END) AS phd
FROM employees GROUP BY department;
-- (3)
SELECT COALESCE(CAST(job_satisfaction AS VARCHAR(5)), 'All') AS satisfaction,
       ROUND(AVG(monthly_income * 1.0), 0) AS avg_income
FROM employees GROUP BY ROLLUP (job_satisfaction) ORDER BY GROUPING(job_satisfaction),
job_satisfaction;
```

Note the doubled apostrophe in 'Bachelor's': that is how you write a quote inside a SQL string.

INTERVIEW CORNER**Questions you may be asked**

Q: WHERE vs HAVING? WHERE filters rows before aggregation and cannot use aggregate functions; HAVING filters groups after GROUP BY and can.

Q: How do you pivot rows to columns in SQL? Conditional aggregation `SUM(CASE WHEN col = 'X' THEN value END)` per output column, or SQL Server's PIVOT operator.

Q: What are ROLLUP and CUBE? GROUP BY extensions that add subtotal and grand-total rows: ROLLUP follows the column hierarchy; CUBE produces all combinations. GROUPING() identifies the subtotal rows.

Q: Write a query to find the second-highest salary per department. Use DENSE_RANK() in a subquery/CTE (Lesson 11): `WHERE rnk = 2`.

Joins and a Retail Case Study

Combine tables with INNER, LEFT, RIGHT, FULL OUTER, CROSS and SELF joins, find missing matches with anti-joins, avoid the classic join mistakes (row explosion, filters in ON vs WHERE), and answer a full set of business questions on the retail database.

● SQL class 4

✦ SQL Server 2022, 2026

Intermediate

🕒 150 min

By the end of this lesson you can

- Explain each join type with a Venn picture and predict how many rows it returns
- Write INNER, LEFT, RIGHT, FULL OUTER, CROSS and SELF joins with table aliases
- Find unmatched rows with LEFT JOIN ... IS NULL (anti-join)
- Avoid row explosion, ambiguous columns, and the ON-vs-WHERE trap in outer joins
- Join three or four tables to answer business questions

Why joins?

In a normalised database (Lesson 2) customer details live in `customers`, products in `products`, and orders in `orders` and `order_items`. To answer "sales by customer segment and product category" you must **join** them back together on their keys. A join is SQL's version of XLOOKUP (Excel Lesson 7) and pandas `merge()`.

A tiny example first

Two small tables make every join type easy to see:

SQL

```
CREATE TABLE students (student_id INT PRIMARY KEY, name VARCHAR(20));
INSERT INTO students VALUES (1, 'Asha'), (2, 'Ravi'), (3, 'Meera'), (4, 'Kabir');

CREATE TABLE enrolments (student_id INT, course VARCHAR(20));
INSERT INTO enrolments VALUES (1, 'SQL'), (1, 'Python'), (2, 'Excel'), (5, 'Power BI');

SELECT 'students' AS t, COUNT(*) AS n FROM students UNION ALL SELECT 'enrolments', COUNT(*)
FROM enrolments;
```

RESULT

Commands completed successfully.

(4 rows affected)

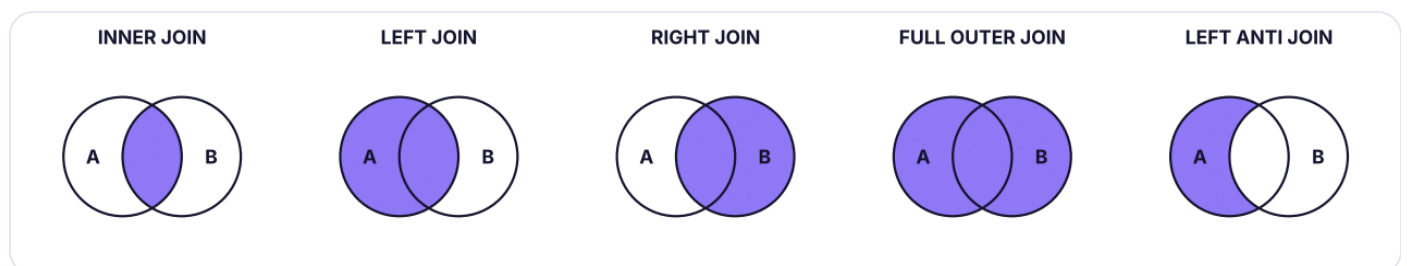
Commands completed successfully.

(4 rows affected)

t	n
students	4
enrolments	4

2 rows × 2 columns

Asha (1) has two enrolments, Ravi (2) has one, Meera (3) and Kabir (4) have none, and enrolment for student 5 has no matching student (an "orphan" row).



The join types as Venn diagrams. Shaded areas are the rows returned (A = left table, B = right table).

INNER JOIN: only matches

SQL

```
SELECT s.student_id, s.name, e.course
FROM students AS s
INNER JOIN enrolments AS e
      ON e.student_id = s.student_id
ORDER BY s.student_id, e.course;
```

RESULT

student_id	name	course
1	Asha	Python
1	Asha	SQL
2	Ravi	Excel

3 rows × 3 columns

Asha appears **twice** because she has two matching enrolments. Meera, Kabir and the orphan enrolment disappear.

LEFT (OUTER) JOIN: everything from the left

SQL

```
SELECT s.student_id, s.name, e.course
FROM students AS s
LEFT JOIN enrolments AS e
      ON e.student_id = s.student_id
ORDER BY s.student_id, e.course;
```

RESULT

student_id	name	course
1	Asha	Python
1	Asha	SQL
2	Ravi	Excel
3	Meera	NULL
4	Kabir	NULL

5 rows × 3 columns

Every student is kept; where there is no match, the right-hand columns are **NULL**.

RIGHT JOIN: everything from the right

SQL

```
SELECT e.student_id, s.name, e.course
FROM students AS s
RIGHT JOIN enrolments AS e
      ON e.student_id = s.student_id
ORDER BY e.student_id;
```

RESULT

student_id	name	course
1	Asha	SQL
1	Asha	Python
2	Ravi	Excel
5	NULL	Power BI

4 rows × 3 columns

Every enrolment is kept, including student 5 who has no name. A RIGHT JOIN is just a LEFT JOIN with the tables swapped; most people write LEFT JOINS for consistency.

FULL OUTER JOIN: everything from both

SQL

```
SELECT s.student_id AS s_id, s.name, e.student_id AS e_id, e.course
FROM students AS s
FULL OUTER JOIN enrolments AS e
      ON e.student_id = s.student_id
ORDER BY COALESCE(s.student_id, e.student_id), e.course;
```

RESULT

s_id	name	e_id	course
1	Asha	1	Python
1	Asha	1	SQL
2	Ravi	2	Excel
3	Meera	NULL	NULL
4	Kabir	NULL	NULL
NULL	NULL	5	Power BI

6 rows × 4 columns

Matched rows, plus unmatched students, plus unmatched enrolments. Useful for **reconciliation**: comparing two systems' lists to see what is missing on each side.

Anti-join: rows with no match

SQL

```
SELECT s.student_id, s.name
FROM students AS s
LEFT JOIN enrolments AS e ON e.student_id = s.student_id
WHERE e.student_id IS NULL;
```

RESULT

student_id	name
3	Meera
4	Kabir

2 rows × 2 columns

"Students not enrolled in anything." The same result with NOT EXISTS is shown in Lesson 10.

CROSS JOIN: every combination

SQL

```
CREATE TABLE sizes (size VARCHAR(5));
INSERT INTO sizes VALUES ('S'), ('M'), ('L');
CREATE TABLE colours (colour VARCHAR(10));
INSERT INTO colours VALUES ('Black'), ('White');

SELECT c.colour, z.size
FROM colours AS c
CROSS JOIN sizes AS z
ORDER BY c.colour, z.size;
```

RESULT

Commands completed successfully.

(3 rows affected)

Commands completed successfully.

(2 rows affected)

colour	size
Black	L
Black	M
Black	S
White	L
White	M
White	S

6 rows × 2 columns

2 colours × 3 sizes = 6 rows: a **Cartesian product**. Use it on purpose (all product-month combinations for a report grid, a calendar × store table); never by accident.

SELF JOIN: a table joined to itself

SQL

```
CREATE TABLE team (emp_id INT, name VARCHAR(20), manager_id INT);
INSERT INTO team VALUES (1, 'Varun', NULL), (2, 'Asha', 1), (3, 'Ravi', 1), (4, 'Meera', 2), (5, 'Kabir', 2);

SELECT e.name AS employee, m.name AS manager
FROM team AS e
LEFT JOIN team AS m ON m.emp_id = e.manager_id
ORDER BY e.emp_id;
```

RESULT

Commands completed successfully.
(5 rows affected)

employee	manager
Varun	NULL
Asha	Varun
Ravi	Varun
Meera	Asha
Kabir	Asha

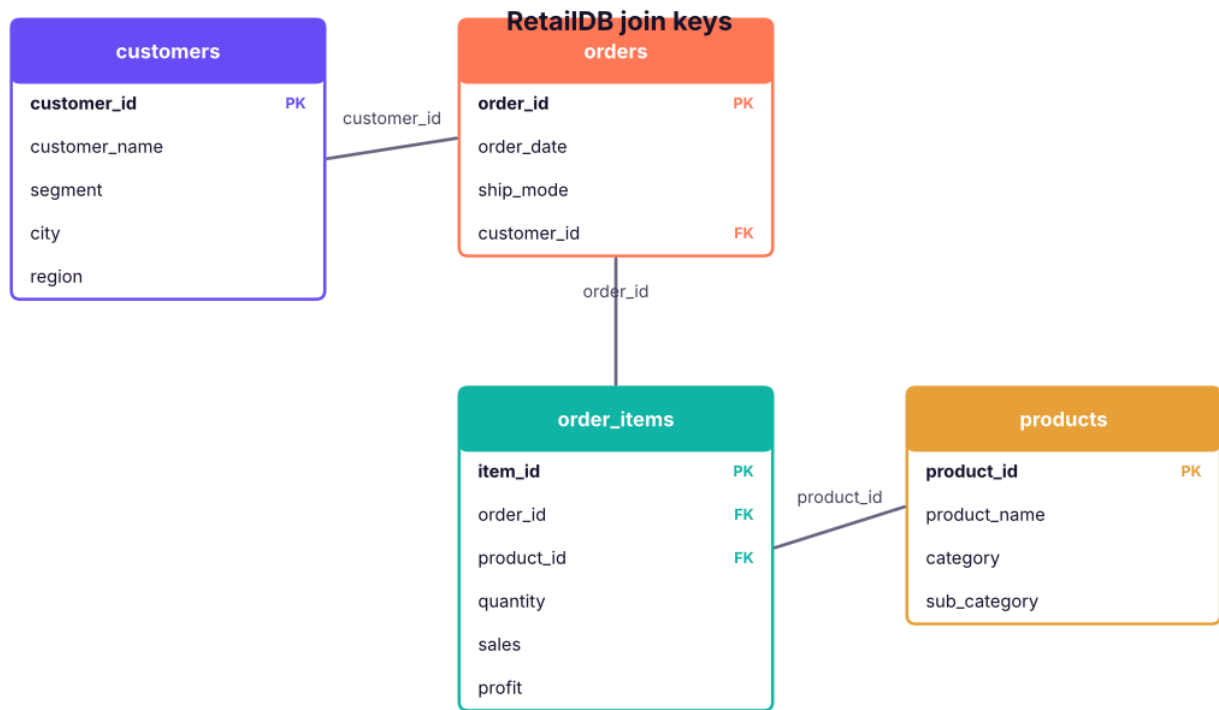
5 rows × 2 columns

The same table appears twice with different aliases (e for employee, m for manager). LEFT JOIN keeps Varun, who has no manager.

How many rows will a join return?

Join	Rows returned
INNER	Sum over matches; a key matching m rows on the left and n on the right produces $m \times n$ rows
LEFT	Every left row at least once (more if it matches several right rows)
RIGHT	Every right row at least once
FULL	Matches + unmatched left + unmatched right
CROSS	Left rows $\times$ right rows

Joins on the retail database



Join paths in RetailDB. Follow the keys: customers → orders → order_items → products.

Two tables

SQL

```
SELECT o.order_id, o.order_date, c.customer_name, c.segment, c.city
FROM orders AS o
INNER JOIN customers AS c ON c.customer_id = o.customer_id
ORDER BY o.order_date DESC, o.order_id;
```

RESULT

order_id	order_date	customer_name	segment	city
IN-2025-03197	2025-12-31	Vivaan Patel	Home Office	Bengaluru
IN-2025-03198	2025-12-31	Neha Fernandes	Consumer	Bhubaneswar
IN-2025-03199	2025-12-31	Aarav Patel	Home Office	Jaipur
IN-2025-03200	2025-12-31	Kabir Rao	Home Office	Pune
IN-2025-03192	2025-12-30	Ananya Bose	Consumer	Bengaluru
IN-2025-03193	2025-12-30	Ananya Nair	Consumer	Ahmedabad
IN-2025-03194	2025-12-30	Krishna Iyer	Consumer	Bengaluru
IN-2025-03195	2025-12-30	Ishaan Shah	Corporate	Bhubaneswar

3,200 rows × 5 columns (first 8 shown)

Four tables

SQL

```
SELECT c.segment,
       p.category,
       COUNT(DISTINCT o.order_id) AS orders,
       ROUND(SUM(oi.sales), 0)   AS sales,
       ROUND(SUM(oi.profit), 0)  AS profit
FROM order_items AS oi
JOIN orders      AS o ON o.order_id = oi.order_id
JOIN customers   AS c ON c.customer_id = o.customer_id
JOIN products    AS p ON p.product_id = oi.product_id
GROUP BY c.segment, p.category
ORDER BY c.segment, sales DESC;
```

RESULT

segment	category	orders	sales	profit
Consumer	Technology	727	76,891,232	9,074,795
Consumer	Furniture	749	17,031,585	-279,832
Consumer	Office Supplies	909	4,680,475	704,297
Corporate	Technology	452	45,607,235	5,223,370
Corporate	Furniture	443	10,323,095	-285,209
Corporate	Office Supplies	570	3,183,289	488,994
Home Office	Technology	339	33,939,946	3,357,358
Home Office	Furniture	319	7,487,173	-253,113
Home Office	Office Supplies	391	2,269,540	321,503

9 rows × 5 columns

JOIN on its own means INNER JOIN. Start from the table at the most detailed level (the **grain**: here, one row per order item) and join outward to the lookup tables.

Case study: questions for the Head of Sales

CASE STUDY Q1. Which customers have never placed an order?

Marketing wants to send them a welcome-back offer.

SQL

```
SELECT c.customer_id, c.customer_name, c.city, c.segment
FROM customers AS c
LEFT JOIN orders AS o ON o.customer_id = c.customer_id
WHERE o.order_id IS NULL
ORDER BY c.customer_id;
```

RESULT

customer_id	customer_name	city	segment
C0057	Meera Patel	Bhubaneswar	Consumer
C0113	Sneha Shah	Guwahati	Home Office
C0153	Nikhil Das	Delhi	Consumer
C0316	Joseph Mehta	Patna	Corporate
C0376	Sneha Kulkarni	Kolkata	Consumer
C0401	Neha Kulkarni	Kolkata	Corporate
C0404	Nikhil Fernandes	Lucknow	Corporate
C0507	Ananya Singh	Hyderabad	Home Office
C0524	Sai Mehta	Pune	Consumer
C0539	Harpreet Sharma	Jaipur	Consumer
C0704	Zoya Patel	Mumbai	Consumer
C0711	Vikram Reddy	Bhubaneswar	Consumer

15 rows × 4 columns (first 12 shown)

CASE STUDY

Q2. Which products have never been sold?

SQL

```
SELECT COUNT(*) AS products_never_sold
FROM products AS p
LEFT JOIN order_items AS oi ON oi.product_id = p.product_id
WHERE oi.item_id IS NULL;
```

RESULT

products_never_sold

0

1 rows × 1 columns

CASE STUDY

Q3. Top 10 customers by 2025 sales, with their city and segment

SQL

```
SELECT TOP 10
    c.customer_name, c.city, c.segment,
    COUNT(DISTINCT o.order_id) AS orders_2025,
    ROUND(SUM(oi.sales), 0) AS sales_2025
FROM customers AS c
JOIN orders AS o      ON o.customer_id = c.customer_id
JOIN order_items AS oi ON oi.order_id   = o.order_id
WHERE o.order_date ≥ '2025-01-01' AND o.order_date < '2026-01-01'
GROUP BY c.customer_id, c.customer_name, c.city, c.segment
ORDER BY sales_2025 DESC;
```

RESULT

customer_name	city	segment	orders_2025	sales_2025
Joseph Shah	Pune	Consumer	5	872,896
Aarav Mehta	Bengaluru	Consumer	6	835,933
Kavya Iyer	Delhi	Consumer	3	821,696
Vivaan Verma	Mumbai	Home Office	2	757,989
Sneha Khan	Ahmedabad	Consumer	3	756,586
Pooja Nair	Pune	Consumer	4	700,790
Farhan Khan	Kochi	Corporate	2	648,834
Ananya Iyer	Chennai	Consumer	4	636,502
Meera Fernandes	Chennai	Consumer	4	614,306
Neha Fernandes	Pune	Consumer	2	588,041

10 rows × 5 columns

CASE STUDY

Q4. Which region × category combinations lose money on discounted Furniture?

SQL

```
SELECT c.region,
       p.sub_category,
       COUNT(*)           AS lines,
       ROUND(AVG(oi.discount), 2) AS avg_discount,
       ROUND(SUM(oi.profit), 0)  AS profit
FROM   order_items AS oi
JOIN   orders      AS o ON o.order_id   = oi.order_id
JOIN   customers   AS c ON c.customer_id = o.customer_id
JOIN   products    AS p ON p.product_id = oi.product_id
WHERE  p.category = 'Furniture' AND oi.discount > 0
GROUP BY c.region, p.sub_category
HAVING SUM(oi.profit) < 0
ORDER BY profit;
```

RESULT

region	sub_category	lines	avg_discount	profit
South	Tables	100	0.25	-407,431
West	Tables	90	0.26	-369,074
East	Tables	73	0.27	-318,641
North	Tables	74	0.28	-301,656
North	Bookcases	82	0.28	-283,651
South	Bookcases	81	0.27	-263,943
East	Bookcases	80	0.27	-248,275
West	Bookcases	80	0.26	-216,090
North	Chairs	53	0.21	-76,380
West	Chairs	60	0.19	-71,006
South	Chairs	58	0.19	-60,298
East	Chairs	46	0.17	-31,667

16 rows × 5 columns (first 12 shown)

CASE STUDY Q5. How does actual selling price compare with the list price?

SQL

```
SELECT p.category,
       ROUND(SUM(oi.sales), 0) AS actual_sales,
       ROUND(SUM(oi.quantity * p.list_price), 0) AS list_value,
       ROUND(100.0 * (1 - SUM(oi.sales) / SUM(oi.quantity * p.list_price)), 1) AS
avg_discount_given_pct
FROM order_items AS oi
JOIN products AS p ON p.product_id = oi.product_id
GROUP BY p.category
ORDER BY avg_discount_given_pct DESC;
```

RESULT

category	actual_sales	list_value	avg_discount_given_pct
Furniture	34,841,853	41,094,510	15.2
Office Supplies	10,133,304	10,963,700	7.6
Technology	156,438,413	168,811,900	7.3

3 rows × 4 columns

Common join mistakes

1. Row explosion (fan-out)

Joining two "many" tables on a shared key multiplies rows. If you join `order_items` to another table with several rows per order (for example a payments table with instalments), each item is repeated for each payment and `SUM(sales)` is inflated. Always ask: *what is one row of each table, and is the key unique on one side?* Check with `COUNT(*)` before and after the join, or aggregate each table to the same grain first (Lesson 10).

2. Ambiguous column names

`SELECT order_id FROM orders JOIN order_items ON ...` fails: "Ambiguous column name 'order_id'". Prefix columns with the table alias (`o.order_id`). Do it for every column in multi-table queries.

3. Filtering an outer join in WHERE

A condition on the **right** table placed in `WHERE` removes the `NULL` rows and silently turns a `LEFT JOIN` into an `INNER JOIN`:

SQL

```
-- Intended: all students, with their SQL course if any
SELECT s.name, e.course
FROM students AS s
LEFT JOIN enrolments AS e ON e.student_id = s.student_id
WHERE e.course = 'SQL';           -- wrong: drops students without SQL
```

RESULT

name	course
Asha	SQL

1 rows × 2 columns

SQL

```
SELECT s.name, e.course
FROM students AS s
LEFT JOIN enrolments AS e
      ON e.student_id = s.student_id
      AND e.course = 'SQL'      -- right: condition in ON keeps every student
ORDER BY s.student_id;
```

RESULT

name	course
Asha	SQL
Ravi	NULL
Meera	NULL
Kabir	NULL

4 rows × 2 columns

4. Joining on the wrong or non-unique key

Joining customers to orders on `city` instead of `customer_id` creates nonsense matches. Join on keys, and check the relationship (1:1, 1:N) you expect.

5. NULL keys never match

NULL = NULL is UNKNOWN, so rows with a NULL key never match in any join. Clean or default keys first if that matters.

TIP Join check-list

1. What is the grain (one row = ?) of each table?
2. Which key connects them, and is it unique on one side?
3. Do I need unmatched rows (LEFT/FULL) or only matches (INNER)?
4. Did the row count change as expected?
5. Are filters on outer-joined tables in ON or WHERE on purpose?

Older join syntax you may see

SQL

```
-- Implicit join (comma join), pre-1992 style: works but avoid
SELECT o.order_id, c.customer_name
FROM orders o, customers c
WHERE o.customer_id = c.customer_id;
```

Forgetting the WHERE turns it into a CROSS JOIN. Use explicit JOIN ... ON.

Summary

KEY TAKEAWAYS

- **INNER**: only matching rows. **LEFT**: all left rows + matches (NULLs where none). **RIGHT**: the mirror. **FULL**: everything. **CROSS**: every combination. **SELF**: a table with itself (aliases).
- **Anti-join**: LEFT JOIN ... WHERE right.key IS NULL finds rows with no match.
- Join from the most detailed table outward, on keys, with aliases on every column.
- Watch for row explosion, ambiguous columns, and filters on the right table in WHERE (move them to ON).
- Joins = XLOOKUP / Power Query Merge / pandas merge / Power BI relationships.

Quick quiz

6 questions · tap an answer to check it

1. Table A has 4 rows, B has 3. A CROSS JOIN returns:

A 3 rows

B 4 rows

C 7 rows

D 12 rows

Answer: D. A Cartesian product returns 4×3 rows.

2. Which join lists every customer, including those with no orders?

A customers INNER JOIN orders

B customers LEFT JOIN orders

C customers RIGHT JOIN orders with orders on the right

D customers CROSS JOIN orders

Answer: B. LEFT JOIN keeps every row from the left table.

3. How do you find products that were never ordered?

A products INNER JOIN order_items WHERE item_id IS NULL

B products LEFT JOIN order_items ... WHERE order_items.item_id IS NULL

C products FULL JOIN order_items

D SELECT DISTINCT product_id FROM order_items

Answer: B. A left anti-join returns left rows with no match.

4. FROM students s LEFT JOIN enrolments e ON ... WHERE e.course = 'SQL' behaves like:

- A A LEFT JOIN
- B An INNER JOIN, because rows with NULL course are removed**
- C A FULL JOIN
- D A CROSS JOIN

Answer: B. The WHERE condition on the right table discards the unmatched (NULL) rows.

5. What is a self join used for?

- A Relating rows of the same table, such as employees and their managers**
- B Joining a table to a copy in another database
- C Removing duplicates
- D Joining without a key

Answer: A. The same table is used twice with different aliases.

6. After joining order_items to a payments table with 2 instalments per order, SUM(sales) doubled. What happened?

- A A NULL key
- B Row explosion: each item was repeated once per payment row**
- C Integer division
- D A wrong data type

Answer: B. Joining two many-sided tables multiplies rows; aggregate to the same grain first.

EXERCISE Join practice

(1) List each order with customer name, city and the number of items in the order. (2) For each customer segment, show the share of sales from each category (%). (3) Find customers who bought Technology but never Furniture. (4) Using team, show each manager with the number of direct reports.

Show solution (worked answer)

SQL

```
-- (1)
SELECT o.order_id, c.customer_name, c.city, COUNT(oi.item_id) AS items
FROM orders o JOIN customers c ON c.customer_id = o.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY o.order_id, c.customer_name, c.city;

-- (2)
SELECT c.segment, p.category,
       ROUND(100.0 * SUM(oi.sales) / SUM(SUM(oi.sales)) OVER (PARTITION BY c.segment),
1) AS pct_of_segment
FROM order_items oi JOIN orders o ON o.order_id = oi.order_id
JOIN customers c ON c.customer_id = o.customer_id JOIN products p ON p.product_id =
oi.product_id
GROUP BY c.segment, p.category;           -- the window function is explained in
Lesson 12

-- (3)
SELECT DISTINCT c.customer_id, c.customer_name
FROM customers c JOIN orders o ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id JOIN products p ON p.product_id =
oi.product_id
WHERE p.category = 'Technology'
      AND c.customer_id NOT IN (SELECT o2.customer_id FROM orders o2 JOIN order_items oi2
ON oi2.order_id = o2.order_id
                                JOIN products p2 ON p2.product_id = oi2.product_id WHERE
p2.category = 'Furniture');

-- (4)
SELECT m.name AS manager, COUNT(e.emp_id) AS direct_reports
FROM team m LEFT JOIN team e ON e.manager_id = m.emp_id
GROUP BY m.emp_id, m.name HAVING COUNT(e.emp_id) > 0;
```

INTERVIEW CORNER**Questions you may be asked**

Q: Explain the different types of joins. INNER returns matches; LEFT/RIGHT keep all rows from one side with NULLs where unmatched; FULL keeps all rows from both; CROSS returns every combination; SELF joins a table to itself.

Q: Table A has 5 rows with id 1 repeated twice, B has 3 rows with id 1 repeated three times. How many rows does an inner join on id return (other ids unmatched)? $2 \times 3 = 6$ rows.

Q: How do you find records in one table but not another? LEFT JOIN ... WHERE right.key IS NULL, NOT EXISTS, or EXCEPT (Lesson 9).

Q: What is the difference between putting a condition in ON vs WHERE for a LEFT JOIN? In ON it only limits which right rows match (all left rows stay); in WHERE it filters the final result and removes unmatched left rows.

Set Operators: UNION, UNION ALL, INTERSECT & EXCEPT

Stack and compare query results. Append data with UNION ALL, remove duplicates with UNION, find common rows with INTERSECT and differences with EXCEPT, follow the rules for compatible columns, and know when to use a set operator instead of a join.

● SQL class 4

✦ SQL Server 2022, 2026

Intermediate

🕒 80 min

By the end of this lesson you can

- Explain the difference between joins (add columns) and set operators (add or compare rows)
- Append results with UNION ALL and de-duplicate with UNION
- Find common rows with INTERSECT and missing rows with EXCEPT
- Apply the rules: same number of columns, compatible types, names from the first query, one ORDER BY at the end
- Use set operators for reconciliation and data-quality checks

Joins add columns; set operators add rows

Join vs set operator

JOIN

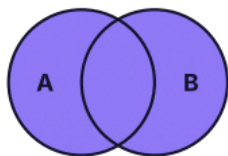
- Combines tables horizontally
- Matches rows on a key
- Result has columns from both tables
- Example: orders + customer details

Set operator

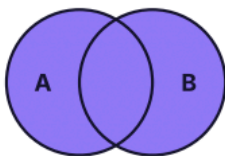
- Combines results vertically
- Compares whole rows
- Same columns in both queries
- Example: 2024 customers ☒ 2025 customers

A join puts tables side by side (more columns). A set operator stacks or compares the rows of two query results with the same columns.

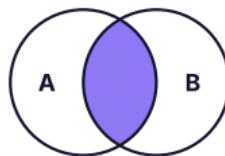
UNION ALL (keeps duplicates)



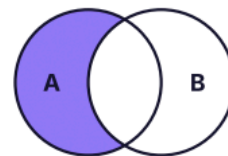
UNION (distinct)



INTERSECT



EXCEPT (A minus B)



The four set operators. A and B are the result sets of two queries.

Rules for set operators

1. Both queries must return the **same number of columns**.
2. Corresponding columns must have **compatible data types** (text with text, numbers with numbers).
3. Column **names come from the first query**.
4. Only **one ORDER BY**, at the very end, applies to the combined result.
5. `UNION`, `INTERSECT` and `EXCEPT` remove duplicates; `UNION ALL` does not.

UNION ALL: append

UNION ALL stacks results exactly as they are, like Power Query's Append or pasting one table under another. It is the fastest set operator because it does not check for duplicates.

SQL

```
SELECT 'Furniture' AS source, COUNT(*) AS lines, ROUND(SUM(sales), 0) AS sales FROM
superstore WHERE category = 'Furniture'
UNION ALL
SELECT 'Technology', COUNT(*), ROUND(SUM(sales), 0) FROM superstore WHERE category =
'Technology'
UNION ALL
SELECT 'Office Supplies', COUNT(*), ROUND(SUM(sales), 0) FROM superstore WHERE category =
'Office Supplies'
ORDER BY sales DESC;
```

RESULT

source	lines	sales
Technology	1892	156,438,413
Furniture	1917	34,841,853
Office Supplies	2514	10,133,304

3 rows × 3 columns

A very common real use: combining monthly or regional tables with the same structure.

SQL

```
SELECT * FROM sales_jan_2026
UNION ALL
SELECT * FROM sales_feb_2026
UNION ALL
SELECT * FROM sales_mar_2026;
```

UNION: append and remove duplicates

SQL

```
SELECT COUNT(*) AS cities_union_all
FROM (SELECT city FROM customers WHERE segment = 'Corporate'
      UNION ALL
      SELECT city FROM customers WHERE segment = 'Home Office') AS t;
```

RESULT

cities_union_all

407

1 rows × 1 columns

SQL

```
SELECT COUNT(*) AS cities_union
FROM (SELECT city FROM customers WHERE segment = 'Corporate'
      UNION
      SELECT city FROM customers WHERE segment = 'Home Office') AS t;
```

RESULT

cities_union

16

1 rows × 1 columns

UNION ALL returned one row per customer; UNION returned each city once.

TIP UNION or UNION ALL?

Use UNION ALL unless you **need** duplicates removed. UNION has to sort or hash the whole result to find duplicates, which is slow on large data, and it can silently remove rows that are genuinely repeated (two identical order lines).

INTERSECT: rows in both

"Which customers ordered in **both** 2024 and 2025?" (retained customers)

SQL

```
SELECT COUNT(*) AS customers_in_both_years
FROM (
  SELECT customer_id FROM orders WHERE YEAR(order_date) = 2024
  INTERSECT
  SELECT customer_id FROM orders WHERE YEAR(order_date) = 2025
) AS t;
```

RESULT

customers_in_both_years

467

1 rows × 1 columns

EXCEPT: rows in the first but not the second

"Which 2024 customers did **not** come back in 2025?" (churned customers)

SQL

```
SELECT COUNT(*) AS customers_lost_after_2024
FROM (
    SELECT customer_id FROM orders WHERE YEAR(order_date) = 2024
    EXCEPT
    SELECT customer_id FROM orders WHERE YEAR(order_date) = 2025
) AS t;
```

RESULT

customers_lost_after_2024

123

1 rows × 1 columns

"Which 2025 customers are **new** (did not order in 2024 or 2023)?"

SQL

```
SELECT COUNT(*) AS new_customers_2025
FROM (
    SELECT customer_id FROM orders WHERE YEAR(order_date) = 2025
    EXCEPT
    SELECT customer_id FROM orders WHERE YEAR(order_date) < 2025
) AS t;
```

RESULT

new_customers_2025

52

1 rows × 1 columns

Order matters for EXCEPT: A EXCEPT B ≠ B EXCEPT A. Oracle calls it MINUS.

Let's put the customer movement together in one table:

SQL

```
SELECT 'Ordered in 2024' AS measure, COUNT(DISTINCT customer_id) AS customers FROM orders
WHERE YEAR(order_date) = 2024
UNION ALL
SELECT 'Ordered in 2025', COUNT(DISTINCT customer_id) FROM orders WHERE YEAR(order_date) =
2025
UNION ALL
SELECT 'Retained (2024 and 2025)', COUNT(*) FROM (SELECT customer_id FROM orders WHERE
YEAR(order_date) = 2024 INTERSECT SELECT customer_id FROM orders WHERE YEAR(order_date) =
2025) AS r
UNION ALL
SELECT 'Lost (2024 only)', COUNT(*) FROM (SELECT customer_id FROM orders WHERE
YEAR(order_date) = 2024 EXCEPT SELECT customer_id FROM orders WHERE YEAR(order_date) =
2025) AS l;
```

RESULT

measure	customers
Ordered in 2024	590
Ordered in 2025	623
Retained (2024 and 2025)	467
Lost (2024 only)	123

4 rows × 2 columns

Retention rate 2024 → 2025 = retained ÷ customers in 2024. This is the core of customer-retention analysis.

Reconciliation and data-quality checks

Set operators are perfect for comparing two versions of the same data, for example a source system extract and the warehouse table:

SQL

```
-- Rows in the source extract that are missing or different in the warehouse
SELECT order_id, order_date, customer_id, total FROM staging.orders_extract
EXCEPT
SELECT order_id, order_date, customer_id, total FROM dw.fact_orders;

-- And the reverse: rows in the warehouse not in the source
SELECT order_id, order_date, customer_id, total FROM dw.fact_orders
EXCEPT
SELECT order_id, order_date, customer_id, total FROM staging.orders_extract;
```

If both queries return no rows, the two tables match exactly. Because set operators compare **whole rows**, any difference in any column shows up. (Set operators treat two NULLs as equal, unlike = in a join.)

Set operator or join?

Question	Use
Add columns from another table	JOIN
Stack tables with the same columns	UNION ALL
Distinct list from several sources	UNION
Which keys are in both lists?	INTERSECT, or INNER JOIN / EXISTS
Which keys are in A but not B?	EXCEPT, or LEFT JOIN ... IS NULL / NOT EXISTS
Compare entire rows column by column	EXCEPT (handles NULLs as equal)

Summary

KEY TAKEAWAYS

- Joins add **columns**; set operators add or compare **rows**.
- Rules: same number of columns, compatible types, names from the first query, one ORDER BY at the end.
- UNION ALL appends everything (fast); UNION removes duplicates.
- INTERSECT returns rows in both results; EXCEPT returns rows in the first but not the second (order matters).
- Use them for combining periods, retention/churn analysis and reconciling systems.

Quick quiz

6 questions · tap an answer to check it

1. Which operator keeps duplicate rows?

A UNION ALL

B UNION

C INTERSECT

D EXCEPT

Answer: A. UNION ALL does not remove duplicates; the others return distinct rows.

2. Where can ORDER BY appear in a UNION query?

A After each SELECT

B Once, at the end of the whole statement

C Only in the first SELECT

D ORDER BY is not allowed

Answer: B. One ORDER BY applies to the combined result.

3. SELECT a, b FROM t1 UNION SELECT c FROM t2 fails because:

A UNION needs WHERE

B The queries return different numbers of columns

C Column names differ

D t2 must be first

Answer: B. Both queries must return the same number of columns.

4. To find customers who ordered in 2024 but not in 2025, use:

A 2024 UNION 2025

B 2024 INTERSECT 2025

C 2024 EXCEPT 2025

D 2025 EXCEPT 2024

Answer: C. EXCEPT returns rows in the first set that are not in the second.

5. Which column names appear in the result of a UNION?

A Those of the first query

B Those of the last query

C Both, combined

D Generic names like col1

Answer: A. Names (and aliases) come from the first SELECT.

6. Why is UNION ALL usually faster than UNION?

A It uses an index

B It does not need to sort or hash rows to remove duplicates

C It returns fewer rows

D It runs queries in parallel

Answer: B. Duplicate removal is extra work that UNION ALL skips.

EXERCISE Customer movement by region

For each region, find the number of customers who ordered in 2024, the number retained in 2025, and the retention rate. Hint: build the retained list with INTERSECT, then join to customers for the region.

Show solution (worked answer)

SQL

```
WITH c24 AS (SELECT DISTINCT customer_id FROM orders WHERE YEAR(order_date) = 2024),
     kept AS (SELECT customer_id FROM c24
              INTERSECT
              SELECT customer_id FROM orders WHERE YEAR(order_date) = 2025)
SELECT c.region,
       COUNT(DISTINCT c24.customer_id) AS customers_2024,
       COUNT(DISTINCT kept.customer_id) AS retained_2025,
       ROUND(100.0 * COUNT(DISTINCT kept.customer_id) / COUNT(DISTINCT
c24.customer_id), 1) AS retention_pct
FROM c24
JOIN customers AS c ON c.customer_id = c24.customer_id
LEFT JOIN kept ON kept.customer_id = c24.customer_id
GROUP BY c.region
ORDER BY retention_pct DESC;
```

(CTEs with WITH are explained in Lesson 13.)

INTERVIEW CORNER Questions you may be asked

Q: UNION vs UNION ALL? UNION removes duplicate rows (slower); UNION ALL keeps all rows (faster). Use UNION ALL unless de-duplication is required.

Q: JOIN vs UNION? JOIN combines columns from related tables on a key; UNION stacks rows from queries with the same column structure.

Q: How would you check that two tables contain exactly the same data? Run A EXCEPT B and B EXCEPT A; both must return zero rows (also compare row counts to catch duplicates).

Subqueries, Derived Tables & EXISTS

Put queries inside queries. Use scalar subqueries, multi-row subqueries with IN, ANY and ALL, correlated subqueries, EXISTS and NOT EXISTS, and derived tables (inline views) to answer questions that one level of SELECT cannot, and learn when a join, CTE or window function is better.

● SQL classes 4–5

✦ SQL Server 2022, 2026

Intermediate

⌚ 120 min

By the end of this lesson you can

- Identify scalar, multi-row, multi-column and table subqueries and where each can be used
- Filter with subqueries using comparison operators, IN, NOT IN, ANY and ALL
- Write correlated subqueries and explain how they are evaluated
- Use EXISTS and NOT EXISTS safely, and know why NOT IN fails with NULLs
- Build derived tables (inline views) to aggregate before joining

What is a subquery?

A **subquery** (inner query, nested query) is a `SELECT` written inside another statement, in brackets. The outer query uses its result.

Subquery types and places

In WHERE

Filter with a value or list:
WHERE sales > (SELECT
AVG(sales) ...)

In FROM (derived table / inline view)

Treat a query result as a
table: FROM (SELECT ...) AS t

In SELECT

Add a calculated value per
row: (SELECT COUNT(*) ...) AS
n

In HAVING

Compare groups with a value:
HAVING SUM(x) > (SELECT ...)

Correlated

Refers to the outer row;
runs once per outer row
(logically)

EXISTS / NOT EXISTS

Tests whether any matching
row exists

Where subqueries can appear and what they return.

By what it returns	Example	Used with
Scalar (one value)	(SELECT AVG(sales) FROM superstore)	=, >, < ... or as a column
Multi-row (one column, many rows)	(SELECT customer_id FROM orders WHERE ...)	IN , NOT IN , ANY , ALL , EXISTS
Table (many columns and rows)	(SELECT region, SUM(sales) ... GROUP BY region)	FROM as a derived table

Scalar subqueries

"Which order lines are above the average line value?"

SQL

```
SELECT COUNT(*) AS lines_above_average,  
       (SELECT ROUND(AVG(sales), 0) FROM superstore) AS average_line_sales  
FROM superstore  
WHERE sales > (SELECT AVG(sales) FROM superstore);
```

RESULT

lines_above_average	average_line_sales
1603	31,854

1 rows × 2 columns

The inner query runs first and returns one number; the outer query compares each row with it. You cannot write `WHERE sales > AVG(sales)` directly, because aggregates are not allowed in `WHERE` (Lesson 5's processing order).

"Show each region's sales and its share of the grand total":

SQL

```
SELECT region,
       ROUND(SUM(sales), 0) AS sales,
       ROUND(100.0 * SUM(sales) / (SELECT SUM(sales) FROM superstore), 1) AS
pct_of_total
FROM superstore
GROUP BY region
ORDER BY sales DESC;
```

RESULT

region	sales	pct_of_total
South	53,952,769	26.8
West	51,958,521	25.8
North	49,018,161	24.3
East	46,484,119	23.1

4 rows × 3 columns

Multi-row subqueries: IN, NOT IN, ANY, ALL

"Orders from customers in the Corporate segment" (without a join):

SQL

```
SELECT COUNT(*) AS corporate_orders
FROM orders
WHERE customer_id IN (SELECT customer_id FROM customers WHERE segment = 'Corporate');
```

RESULT

corporate_orders
955

1 rows × 1 columns

"Employees earning more than **every** employee in HR" and "more than **at least one** Sales manager":

SQL

```
SELECT COUNT(*) AS earn_more_than_all_hr
FROM employees
WHERE monthly_income > ALL (SELECT monthly_income FROM employees WHERE department = 'HR');
```

RESULT

earn_more_than_all_hr
4

1 rows × 1 columns

SQL

```
SELECT COUNT(*) AS earn_more_than_some_sales_manager
FROM employees
WHERE monthly_income > ANY (SELECT monthly_income FROM employees WHERE job_title = 'Sales Manager');
```

RESULT

earn_more_than_some_sales_manager
745

1 rows × 1 columns

Operator	Meaning	Equivalent
> ALL (...)	Greater than every value	> (SELECT MAX(...))
> ANY (...) / > SOME (...)	Greater than at least one value	> (SELECT MIN(...))
= ANY (...)	Equal to one of the values	IN (...)
<> ALL (...)	Different from every value	NOT IN (...)

Correlated subqueries

A **correlated** subquery refers to a column of the outer query, so its result changes for every outer row.

"Which employees earn more than the average of **their own** department?"

SQL

```
SELECT e.emp_id, e.department, e.job_title, e.monthly_income
FROM employees AS e
WHERE e.monthly_income > (SELECT AVG(e2.monthly_income)
                          FROM employees AS e2
                          WHERE e2.department = e.department)    -- refers to the outer
--
ORDER BY e.monthly_income DESC;
```

RESULT

emp_id	department	job_title	monthly_income
E00210	R&D	Research Director	218417
E00519	Finance	Finance Manager	209833
E00328	Operations	Operations Manager	202750
E01187	IT	Support Engineer	196167
E01094	HR	HR Manager	194917
E01473	IT	Tech Lead	194500
E00439	IT	Tech Lead	193917
E00580	R&D	Research Director	188083

638 rows × 4 columns (first 8 shown)

Conceptually, for each employee *e*, SQL computes the average of *e*'s department and compares. Optimisers usually rewrite this efficiently, but on large tables a window function (Lesson 12: `AVG(...) OVER (PARTITION BY department)`) or a join to a derived table is often clearer and faster.

"Latest order of each customer" (a classic):

SQL

```
SELECT o.customer_id, o.order_id, o.order_date
FROM orders AS o
WHERE o.order_date = (SELECT MAX(o2.order_date)
                     FROM orders AS o2
                     WHERE o2.customer_id = o.customer_id)
ORDER BY o.customer_id;
```

RESULT

customer_id	order_id	order_date
C0001	IN-2025-02263	2025-04-09
C0002	IN-2025-02871	2025-10-21
C0003	IN-2025-02642	2025-08-18
C0004	IN-2025-02344	2025-05-10
C0005	IN-2025-02982	2025-11-12
C0006	IN-2025-02960	2025-11-08

793 rows × 3 columns (first 6 shown)

EXISTS and NOT EXISTS

EXISTS (subquery) is TRUE if the subquery returns **at least one row**. It stops at the first match, and the columns it selects do not matter (SELECT 1 is conventional).

"Customers who bought at least one Copier":

SQL

```
SELECT COUNT(*) AS customers_who_bought_copiers
FROM customers AS c
WHERE EXISTS (SELECT 1
              FROM orders AS o
              JOIN order_items AS oi ON oi.order_id = o.order_id
              JOIN products AS p     ON p.product_id = oi.product_id
              WHERE o.customer_id = c.customer_id
              AND p.sub_category = 'Copiers');
```

RESULT

customers_who_bought_copiers

343

1 rows × 1 columns

"Customers who have never ordered" (anti-join, compare with Lesson 8):

SQL

```
SELECT COUNT(*) AS customers_without_orders
FROM customers AS c
WHERE NOT EXISTS (SELECT 1 FROM orders AS o WHERE o.customer_id = c.customer_id);
```

RESULT

customers_without_orders

15

1 rows × 1 columns

Why NOT EXISTS is safer than NOT IN

SQL

```
WITH a AS (SELECT 1 AS id UNION ALL SELECT 2 UNION ALL SELECT 3),
      b AS (SELECT 2 AS id UNION ALL SELECT NULL)
SELECT
  (SELECT COUNT(*) FROM a WHERE id NOT IN (SELECT id FROM b)) AS
not_in_rows,
  (SELECT COUNT(*) FROM a WHERE NOT EXISTS (SELECT 1 FROM b WHERE b.id = a.id)) AS
not_exists_rows;
```

RESULT

not_in_rows	not_exists_rows
-------------	-----------------

0	2
---	---

1 rows × 2 columns

`NOT IN` returns **0 rows** because `b` contains a `NULL`: `3 NOT IN (2, NULL)` means `3 <> 2 AND 3 <> NULL`, which is `UNKNOWN`. `NOT EXISTS` correctly returns 2 rows (1 and 3). Prefer `NOT EXISTS`, or add `WHERE id IS NOT NULL` inside the `NOT IN` subquery.

Derived tables (inline views)

A subquery in `FROM` acts as a temporary table for the rest of the query. It **must have an alias**. Derived tables are the standard way to **aggregate first, then filter, join or aggregate again**.

"Average order value by segment" requires two levels: first total each order, then average the orders.

SQL

```
SELECT segment,
       COUNT(*)           AS orders,
       ROUND(AVG(order_total), 0) AS avg_order_value
FROM (
  SELECT o.order_id, c.segment, SUM(oi.sales) AS order_total  -- level 1: one row per
  order
  FROM orders AS o
  JOIN customers AS c ON c.customer_id = o.customer_id
  JOIN order_items AS oi ON oi.order_id = o.order_id
  GROUP BY o.order_id, c.segment
) AS order_totals  -- level 2: average of
orders
GROUP BY segment
ORDER BY avg_order_value DESC;
```

RESULT

segment	orders	avg_order_value
Consumer	1554	63,451
Home Office	691	63,237
Corporate	955	61,899

3 rows × 3 columns

If you averaged `oi.sales` directly, you would get the average **line** value, not the average **order** value: a frequent interview trap.

"Cities whose 2025 sales beat their 2024 sales", using two derived tables joined together:

SQL

```
SELECT a.city, ROUND(a.sales, 0) AS sales_2024, ROUND(b.sales, 0) AS sales_2025,
       ROUND(100.0 * (b.sales - a.sales) / a.sales, 1) AS growth_pct
FROM (SELECT city, SUM(sales) AS sales FROM superstore WHERE YEAR(order_date) = 2024 GROUP
BY city) AS a
JOIN (SELECT city, SUM(sales) AS sales FROM superstore WHERE YEAR(order_date) = 2025 GROUP
BY city) AS b
  ON b.city = a.city
WHERE b.sales > a.sales
ORDER BY growth_pct DESC;
```

RESULT

city	sales_2024	sales_2025	growth_pct
Pune	4,469,290	8,808,243	97.1
Jaipur	2,381,435	4,387,920	84.3
Bengaluru	3,955,891	5,909,257	49.4
Bhubaneswar	3,241,529	4,715,977	45.5
Chennai	4,306,703	6,192,082	43.8
Guwahati	3,063,023	4,156,269	35.7
Kolkata	4,204,948	5,388,829	28.2
Kochi	4,419,418	5,550,394	25.6
Indore	4,663,519	5,766,524	23.7
Ahmedabad	3,655,966	4,499,715	23.1
Chandigarh	3,384,407	3,711,620	9.7
Patna	4,459,339	4,649,126	4.3

12 rows × 4 columns

NOTE Nested (ordinary) subqueries vs inline views

A subquery in WHERE that does not refer to the outer query is an **ordinary (nested) subquery**: it runs once. A subquery in FROM is an **inline view** (derived table). One that refers to the outer row is **correlated**. In class these were introduced as inline views and ordinary subqueries; the next step is the **CTE** (Lesson 13), which is a named, more readable inline view.

Subqueries in SELECT

A scalar subquery in the column list adds a per-row value:

SQL

```
SELECT c.customer_id, c.customer_name,
       (SELECT COUNT(*) FROM orders AS o WHERE o.customer_id = c.customer_id) AS
order_count,
       (SELECT MAX(o.order_date) FROM orders AS o WHERE o.customer_id = c.customer_id) AS
last_order
FROM customers AS c
ORDER BY order_count DESC, c.customer_id;
```

RESULT

customer_id	customer_name	order_count	last_order
C0219	Reyansh Verma	10	2025-04-10
C0396	Saanvi Patel	10	2025-07-18
C0069	Isha Verma	9	2025-10-04
C0110	Riya Das	9	2025-12-07
C0271	Meera Khan	9	2025-12-26
C0291	Meera Rao	9	2025-02-04

800 rows × 4 columns (first 6 shown)

Each correlated scalar subquery runs per customer. For many such columns, a single `LEFT JOIN` to an aggregated derived table is usually faster.

Choosing the right tool

Need	Good options
Compare with an overall aggregate	Scalar subquery, or window function
Filter by membership in another table	<code>IN</code> , <code>EXISTS</code> , or <code>JOIN</code>
Rows with no match	<code>NOT EXISTS</code> (safest), <code>LEFT JOIN ... IS NULL</code> , <code>EXCEPT</code>
Compare with the row's own group	Correlated subquery, or <code>AVG() OVER (PARTITION BY ...)</code>
Aggregate in two steps	Derived table or CTE
Reuse the same subquery several times	CTE (Lesson 13)

Summary

KEY TAKEAWAYS

- Subqueries are queries in brackets inside another query; they can return a value, a list or a table.
- Scalar subqueries compare with or display a single value (e.g. the overall average).
- IN, ANY, ALL work with multi-row subqueries; > ALL = greater than the max.
- Correlated subqueries reference the outer row (e.g. above the employee's department average).
- EXISTS / NOT EXISTS test for matching rows; prefer NOT EXISTS over NOT IN when NULLs are possible.
- Derived tables (inline views) aggregate first, then filter/join; they need an alias.

Quick quiz

6 questions · tap an answer to check it

1. Why does `WHERE sales > AVG(sales)` fail?

- A** AVG needs GROUP BY
- B** Aggregate functions are not allowed in WHERE; use a subquery
- C** sales must be in brackets
- D** The comparison must use ALL

Answer: B. WHERE runs before aggregation, so compute the average in a subquery.

2. A subquery in FROM must have:

- A** An ORDER BY
- B** An alias
- C** A WHERE clause
- D** A GROUP BY

Answer: B. Derived tables must be named.

3. WHERE x NOT IN (SELECT y FROM t) returns no rows even though some x values are missing from t. Most likely cause?

A t is empty

B t.y contains a NULL

C x has duplicates

D NOT IN cannot be used with subqueries

Answer: B. One NULL in the list makes every NOT IN comparison UNKNOWN.

4. What is a correlated subquery?

A A subquery in FROM

B A subquery that references a column from the outer query

C A subquery with a JOIN

D A subquery that returns one value

Answer: B. It is evaluated in the context of each outer row.

5. salary > ALL (SELECT salary FROM emp WHERE dept = 'HR') is equivalent to:

A salary > the maximum HR salary

B salary > the minimum HR salary

C salary > the average HR salary

D salary IN the HR salaries

Answer: A. Greater than all values means greater than the largest one.

6. To compute average order value from order lines, you should:

- A** AVG(sales) on order_items
- B** Sum sales per order in a derived table, then average those totals
- C** Divide total sales by the number of lines
- D** Use COUNT(DISTINCT sales)

Answer: B. Average order value is the mean of order totals, which needs two aggregation levels.

EXERCISE Subquery practice

(1) List products whose total sales are above the average product's total sales. (2) For each department, show employees whose income is the highest in that department. (3) Find cities where every order line in 2025 had a discount of at most 20%. (4) Show customers whose first order was in 2025.

Show solution (worked answer)

SQL

```
-- (1)
SELECT product_id, SUM(sales) AS sales FROM order_items GROUP BY product_id
HAVING SUM(sales) > (SELECT AVG(t.s) FROM (SELECT SUM(sales) AS s FROM order_items
GROUP BY product_id) AS t);
-- (2)
SELECT e.department, e.emp_id, e.monthly_income FROM employees e
WHERE e.monthly_income = (SELECT MAX(e2.monthly_income) FROM employees e2 WHERE
e2.department = e.department);
-- (3)
SELECT DISTINCT city FROM superstore s
WHERE YEAR(order_date) = 2025
AND NOT EXISTS (SELECT 1 FROM superstore s2 WHERE s2.city = s.city AND
YEAR(s2.order_date) = 2025 AND s2.discount > 0.2);
-- (4)
SELECT customer_id FROM orders GROUP BY customer_id HAVING MIN(order_date) ≥ '2025-
01-01';
```

INTERVIEW CORNER**Questions you may be asked**

Q: Difference between a correlated and a non-correlated subquery? A non-correlated subquery runs independently once; a correlated subquery references the outer query and is evaluated per outer row.

Q: EXISTS vs IN? IN compares a value with a list; EXISTS checks for the existence of matching rows and stops at the first one. EXISTS handles NULLs safely in the negative form (NOT EXISTS).

Q: Find employees who earn more than their department's average. Correlated subquery on department, or `AVG(income) OVER (PARTITION BY department)` in a CTE and filter.

Q: Subquery vs JOIN: which is faster? It depends; modern optimisers often produce the same plan. Joins are usually clearer for combining columns; EXISTS for existence tests; derived tables/CTEs for multi-step aggregation. Check the execution plan.

OFFSET-FETCH and Window Functions I: OVER, PARTITION BY & Ranking

Page through results with OFFSET and FETCH, then learn window functions, the most powerful analytics feature in SQL. Understand OVER and PARTITION BY, rank rows with ROW_NUMBER, RANK, DENSE_RANK and NTILE, solve top-N-per-group and de-duplication problems, and compare aggregate windows with GROUP BY.

● SQL class 5

✦ SQL Server 2022, 2026

Intermediate–Advanced

🕒 130 min

By the end of this lesson you can

- Paginate with ORDER BY ... OFFSET ... ROWS FETCH NEXT ... ROWS ONLY
- Explain what a window function is and how OVER, PARTITION BY and ORDER BY define the window
- Use ROW_NUMBER, RANK, DENSE_RANK and NTILE and explain how ties behave
- Solve top-N per group, Nth highest value and duplicate-removal problems
- Use aggregate window functions to show totals and shares beside detail rows

OFFSET and FETCH: pagination

`OFFSET n ROWS FETCH NEXT m ROWS ONLY` skips `n` rows and returns the next `m`. It requires `ORDER BY`. It is the standard (ANSI) way to page results, used for "page 3 of results" in applications and for "rows 11 to 20" questions.

SQL

```
SELECT customer_name, city, segment
FROM customers
ORDER BY customer_id
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
```

RESULT

customer_name	city	segment
Sai Iyer	Indore	Consumer
Neha Kulkarni	Bengaluru	Consumer
Diya Khan	Bhubaneswar	Home Office
Kabir Das	Kolkata	Consumer
Reyansh Joshi	Mumbai	Corporate

5 rows × 3 columns

Page	OFFSET	FETCH
1	0	page size
2	page size	page size
page p	$(p - 1) \times \text{page size}$	page size

TOP n cannot skip rows; OFFSET-FETCH can. PostgreSQL/MySQL write `LIMIT m OFFSET n`. On very deep pages (OFFSET 1,000,000) databases still read and discard all skipped rows, so large systems use **keyset pagination** (`WHERE id > last_seen_id ORDER BY id`).

The "Nth highest value" question can be answered with it too: the second-highest line value is `ORDER BY sales DESC OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY` on distinct values:

SQL

```
SELECT DISTINCT sales AS second_highest_sales
FROM superstore
ORDER BY sales DESC
OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY;
```

RESULT

second_highest_sales

551,530

1 rows × 1 columns

What is a window function?

A **window function** computes a value for **each row** using a set of related rows (its **window**), **without collapsing the rows** like GROUP BY does.

Collapse vs keep

GROUP BY region

- North · 120
- North · 60 → North · 180 (one row)
- South · 80
- South · 40 → South · 120 (one row)
- Detail rows are lost

SUM(sales) OVER (PARTITION BY region)

- North · 120 · total 180
- North · 60 · total 180
- South · 80 · total 120
- South · 40 · total 120
- Every detail row is kept

GROUP BY collapses rows into one per group. A window function keeps every row and adds the group's result beside it.

The OVER clause

SQL

```
function_name(expression) OVER (  
    PARTITION BY column_list    -- split rows into groups (optional)  
    ORDER BY column_list       -- order inside each group (needed for ranking, running  
    totals)  
    ROWS BETWEEN ... AND ...   -- optional frame (Lesson 12)  
)
```

- **PARTITION BY** is like GROUP BY for the window: the function restarts for every partition. Without it, the whole result is one partition.
- **ORDER BY** inside OVER orders rows *within the partition* (it does not sort the final output; use the normal ORDER BY for that).

SQL

```
SELECT region, city, order_id, sales,  
       SUM(sales) OVER (PARTITION BY region)           AS region_total,  
       ROUND(100.0 * sales / SUM(sales) OVER (PARTITION BY region), 4) AS pct_of_region,  
       AVG(sales) OVER ()                             AS overall_avg  
FROM superstore  
ORDER BY region, sales DESC;
```

RESULT

region	city	order_id	sales	region_total	pct_of_region	overall_avg
East	Kolkata	IN-2025-03160	472,740	46,484,119	1.02	31,854
East	Kolkata	IN-2025-02134	442,350	46,484,119	0.95	31,854
East	Kolkata	IN-2025-02822	422,640	46,484,119	0.91	31,854
East	Guwahati	IN-2025-02562	413,040	46,484,119	0.89	31,854
East	Patna	IN-2023-00171	393,950	46,484,119	0.85	31,854
East	Patna	IN-2025-02317	388,320	46,484,119	0.84	31,854
East	Bhubaneswar	IN-2025-02981	372,720	46,484,119	0.80	31,854
East	Patna	IN-2024-01792	372,720	46,484,119	0.80	31,854

6,323 rows × 7 columns (first 8 shown)

Each row keeps its own detail but now also shows its region total, its share of the region, and the overall average. With GROUP BY you would need a join to get this.

Ranking functions

All four need ORDER BY inside OVER.

Function	Behaviour with ties	Example values 90, 80, 80, 70
ROW_NUMBER()	Unique numbers; ties broken arbitrarily	1, 2, 3, 4
RANK()	Ties share a rank; gaps follow	1, 2, 2, 4
DENSE_RANK()	Ties share a rank; no gaps	1, 2, 2, 3
NTILE(n)	Splits rows into n equal buckets	4 rows, NTILE(2): 1, 1, 2, 2

SQL

```
WITH scores AS (  
    SELECT 'Asha' AS name, 90 AS score UNION ALL SELECT 'Ravi', 80 UNION ALL SELECT  
    'Meera', 80 UNION ALL SELECT 'Kabir', 70  
)  
SELECT name, score,  
    ROW_NUMBER() OVER (ORDER BY score DESC) AS row_number,  
    RANK() OVER (ORDER BY score DESC) AS rank,  
    DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rank,  
    NTILE(2) OVER (ORDER BY score DESC) AS half  
FROM scores;
```

RESULT

name	score	row_number	rank	dense_rank	half
Asha	90	1	1	1	1
Meera	80	2	2	2	1
Ravi	80	3	2	2	2
Kabir	70	4	4	3	2

4 rows × 6 columns

Ravi and Meera tie at 80: RANK gives both 2 and skips 3; DENSE_RANK gives both 2 and continues with 3; ROW_NUMBER gives 2 and 3 in arbitrary order (add a tie-breaker column to make it deterministic).

Ranking within groups

"Rank each sub-category by sales within its category":

SQL

```
SELECT category, sub_category,  
    ROUND(SUM(sales), 0) AS sales,  
    RANK() OVER (PARTITION BY category ORDER BY SUM(sales) DESC) AS  
rank_in_category  
FROM superstore  
GROUP BY category, sub_category  
ORDER BY category, rank_in_category;
```

RESULT

category	sub_category	sales	rank_in_category
Furniture	Tables	14,046,224	1
Furniture	Bookcases	11,274,849	2
Furniture	Chairs	8,091,474	3
Furniture	Furnishings	1,429,307	4
Office Supplies	Appliances	6,333,529	1
Office Supplies	Storage	2,659,340	2
Office Supplies	Paper	493,232	3
Office Supplies	Binders	401,219	4
Office Supplies	Art	245,985	5
Technology	Copiers	78,749,926	1
Technology	Machines	42,590,335	2
Technology	Phones	31,919,223	3

13 rows × 4 columns (first 12 shown)

Note the order: GROUP BY runs first, then the window function ranks the grouped rows. That is why SUM(sales) can appear inside OVER.

Top-N per group

You cannot filter on a window function in WHERE (windows are computed in the SELECT step, after WHERE). Wrap the query in a CTE or derived table, then filter the rank.

"Top 2 cities by sales in each region":

SQL

```
WITH city_sales AS (  
    SELECT region, city, SUM(sales) AS sales  
    FROM superstore  
    GROUP BY region, city  
) , ranked AS (  
    SELECT region, city, sales,  
           DENSE_RANK() OVER (PARTITION BY region ORDER BY sales DESC) AS rnk  
    FROM city_sales  
)  
SELECT region, city, ROUND(sales, 0) AS sales, rnk  
FROM ranked  
WHERE rnk ≤ 2  
ORDER BY region, rnk;
```

RESULT

region	city	sales	rnk
East	Kolkata	13,888,466	1
East	Patna	12,444,393	2
North	Delhi	15,657,610	1
North	Chandigarh	11,541,792	2
South	Kochi	14,713,126	1
South	Bengaluru	13,625,216	2
West	Pune	17,357,038	1
West	Indore	14,408,485	2

8 rows × 4 columns

"Each customer's most recent order" (use ROW_NUMBER to get exactly one row per customer even with ties):

SQL

```
WITH ranked AS (  
    SELECT customer_id, order_id, order_date,  
           ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC, order_id  
DESC) AS rn  
    FROM orders  
)  
SELECT customer_id, order_id, order_date  
FROM ranked  
WHERE rn = 1  
ORDER BY customer_id;
```

RESULT

customer_id	order_id	order_date
C0001	IN-2025-02263	2025-04-09
C0002	IN-2025-02871	2025-10-21
C0003	IN-2025-02642	2025-08-18
C0004	IN-2025-02344	2025-05-10
C0005	IN-2025-02982	2025-11-12
C0006	IN-2025-02960	2025-11-08

785 rows × 3 columns (first 6 shown)

Nth highest per group

"Second-highest paid employee in each department" (a top interview question):

SQL

```
WITH r AS (  
    SELECT department, emp_id, monthly_income,  
           DENSE_RANK() OVER (PARTITION BY department ORDER BY monthly_income DESC) AS rnk  
    FROM employees  
)  
SELECT department, emp_id, monthly_income  
FROM r  
WHERE rnk = 2  
ORDER BY department;
```

RESULT

department	emp_id	monthly_income
Finance	E00145	185500
HR	E00450	175750
IT	E01473	194500
Operations	E00378	184167
R&D	E00580	188083
Sales	E01218	184833

6 rows × 3 columns

Use DENSE_RANK for "Nth highest **value**" (ties share a position, no gaps) and ROW_NUMBER for "Nth **row**".

Removing duplicates with ROW_NUMBER

SQL

```
WITH d AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY customer_id, order_date, product_id ORDER BY
item_id) AS rn
    FROM order_items_raw
)
DELETE FROM d WHERE rn > 1;           -- SQL Server lets you DELETE through a CTE
```

Rows with the same business key get numbers 1, 2, 3...; keeping rn = 1 keeps one copy. Always check with a SELECT ... WHERE rn > 1 first.

Let's count how many lines in our data repeat the same product in the same order (what a de-duplication would touch):

SQL

```
WITH d AS (
    SELECT order_id, product_id,
        ROW_NUMBER() OVER (PARTITION BY order_id, product_id ORDER BY row_id) AS rn
    FROM superstore
)
SELECT COUNT(*) AS repeated_lines FROM d WHERE rn > 1;
```

RESULT

repeated_lines

70

1 rows × 1 columns

(These are legitimate repeat lines, not errors; the point is the pattern.)

NTILE: buckets and percentiles

SQL

```
WITH cust AS (  
    SELECT customer_id, SUM(sales) AS sales FROM superstore GROUP BY customer_id  
) , q AS (  
    SELECT customer_id, sales, NTILE(4) OVER (ORDER BY sales DESC) AS quartile FROM cust  
)  
SELECT quartile,  
       COUNT(*)           AS customers,  
       ROUND(SUM(sales), 0) AS sales,  
       ROUND(100.0 * SUM(sales) / SUM(SUM(sales)) OVER (), 1) AS pct_of_total_sales  
FROM q  
GROUP BY quartile  
ORDER BY quartile;
```

RESULT

quartile	customers	sales	pct_of_total_sales
1	197	111,100,409	55.2
2	196	54,386,341	27.0
3	196	28,080,117	13.9
4	196	7,846,704	3.9

4 rows × 4 columns

Quartile 1 (the top 25% of customers) generates a disproportionate share of sales: the **Pareto principle** from the Excel charts lesson, now computed in SQL. NTILE is also how RFM scoring is done (ML track).

Aggregate window functions

Any aggregate (SUM, AVG, COUNT, MIN, MAX) becomes a window function with OVER. Without ORDER BY in the window, it uses the whole partition: perfect for totals, shares and comparisons with the

group average.

SQL

```
SELECT department, emp_id, monthly_income,
       ROUND(AVG(monthly_income) OVER (PARTITION BY department), 0) AS dept_avg,
       monthly_income - ROUND(AVG(monthly_income) OVER (PARTITION BY department), 0) AS
vs_dept_avg,
       COUNT(*) OVER (PARTITION BY department) AS
dept_headcount,
       MAX(monthly_income) OVER (PARTITION BY department) AS dept_max
FROM employees
ORDER BY department, monthly_income DESC;
```

RESULT

department	emp_id	monthly_income	dept_avg	vs_dept_avg	dept_headcount	dept_
Finance	E00519	209833	83,681	126,152	154	2098
Finance	E00145	185500	83,681	101,819	154	2098
Finance	E00336	174417	83,681	90,736	154	2098
Finance	E00201	169500	83,681	85,819	154	2098
Finance	E01172	168167	83,681	84,486	154	2098
Finance	E01151	155167	83,681	71,486	154	2098
Finance	E00759	152083	83,681	68,402	154	2098
Finance	E01004	144667	83,681	60,986	154	2098

1,500 rows × 7 columns (first 8 shown)

This replaces the correlated subquery from Lesson 10 ("earns more than the department average") with one pass over the data.

GROUP BY vs window functions

	GROUP BY	Window function
Rows returned	One per group	Same as input
Detail columns	Only grouped columns	All columns
Use for	Summary reports	Rankings, running totals, shares, comparisons with group values
Filter the result	HAVING	Wrap in CTE/subquery and WHERE
Can combine	Yes: group first, then window over the groups	

Summary

KEY TAKEAWAYS

- `OFFSET n ROWS FETCH NEXT m ROWS ONLY` (with `ORDER BY`) paginates; it can skip rows, unlike `TOP`.
- A **window function** adds a calculation over a set of related rows to each row, without collapsing them. `OVER (PARTITION BY ... ORDER BY ...)` defines the window.
- `ROW_NUMBER` (unique), `RANK` (ties, gaps), `DENSE_RANK` (ties, no gaps), `NTILE(n)` (equal buckets).
- To filter on a window result, compute it in a CTE or derived table, then use `WHERE`.
- Top-N per group and Nth highest per group: `DENSE_RANK/ROW_NUMBER` partitioned by group; de-duplicate with `ROW_NUMBER ... WHERE rn > 1`.
- Aggregate windows (`SUM()` `OVER (PARTITION BY ...)`) show totals, shares and group averages beside each row.

Quick quiz

6 questions · tap an answer to check it

1. Scores 100, 90, 90, 80. What does `DENSE_RANK() OVER (ORDER BY score DESC)` return?

A 1, 2, 3, 4

B 1, 2, 2, 4

C 1, 2, 2, 3

D 1, 1, 2, 3

Answer: C. `DENSE_RANK` gives tied rows the same rank and continues without gaps.

2. Why can't you write `WHERE ROW_NUMBER() OVER (...) = 1`?

A `ROW_NUMBER` needs `GROUP BY`

B Window functions are evaluated after `WHERE`, so you must filter in an outer query or CTE

C `OVER` cannot contain `ORDER BY`

D It is allowed in SQL Server

Answer: B. Logical processing order: `WHERE` runs before `SELECT`, where windows are computed.

3. What is the difference between `PARTITION BY` and `GROUP BY`?

A None

B `PARTITION BY` defines the window groups but keeps every row; `GROUP BY` collapses rows

C `PARTITION BY` only works with `RANK`

D `GROUP BY` keeps all rows

Answer: B. Partitions restart the calculation for each group without reducing the number of rows.

4. Which returns the 3rd highest distinct salary overall?

- A** DENSE_RANK() OVER (ORDER BY salary DESC) = 3
- B ROW_NUMBER() OVER (ORDER BY salary DESC) = 3
- C TOP 3 salary
- D RANK() OVER (ORDER BY salary) = 3

Answer: A. DENSE_RANK counts distinct values; ROW_NUMBER would count rows even if values repeat.

5. SUM(sales) OVER (PARTITION BY region) returns:

- A One row per region
- B** On every row, the total sales of that row's region
- C The overall total on every row
- D A running total

Answer: B. Without ORDER BY in the window, it sums the whole partition.

6. To fetch rows 21–30 of a sorted result you write:

- A TOP 30
- B** ORDER BY ... OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY
- C LIMIT 10, 20 in T-SQL
- D WHERE ROW 21 TO 30

Answer: B. Skip 20, take 10.

EXERCISE Window practice

(1) For each customer segment, rank customers by total sales and show the top 3. (2) For each order line, show its sales as a percentage of its order's total. (3) For each department, list employees whose income is within the top 5% of their department (use NTILE(20)). (4) Show products whose total sales rank in the top 10% of their category.

Show solution (worked answer)

SQL

```
-- (1)
WITH t AS (SELECT c.segment, c.customer_name, SUM(oi.sales) AS sales
            FROM order_items oi JOIN orders o ON o.order_id = oi.order_id JOIN
customers c ON c.customer_id = o.customer_id
            GROUP BY c.segment, c.customer_id, c.customer_name),
      r AS (SELECT *, DENSE_RANK() OVER (PARTITION BY segment ORDER BY sales DESC) AS
rnk FROM t)
SELECT * FROM r WHERE rnk ≤ 3 ORDER BY segment, rnk;

-- (2)
SELECT item_id, order_id, sales, ROUND(100.0 * sales / SUM(sales) OVER (PARTITION BY
order_id), 1) AS pct_of_order FROM order_items;

-- (3)
WITH x AS (SELECT department, emp_id, monthly_income, NTILE(20) OVER (PARTITION BY
department ORDER BY monthly_income DESC) AS bucket FROM employees)
SELECT * FROM x WHERE bucket = 1;

-- (4)
WITH p AS (SELECT pr.category, pr.product_id, SUM(oi.sales) AS sales FROM order_items
oi JOIN products pr ON pr.product_id = oi.product_id GROUP BY pr.category,
pr.product_id),
      n AS (SELECT *, NTILE(10) OVER (PARTITION BY category ORDER BY sales DESC) AS
decile FROM p)
SELECT * FROM n WHERE decile = 1;
```

INTERVIEW CORNER**Questions you may be asked**

Q: RANK vs DENSE_RANK vs ROW_NUMBER? ROW_NUMBER numbers rows uniquely; RANK gives ties the same rank and skips following ranks; DENSE_RANK gives ties the same rank without gaps.

Q: Find the second-highest salary in each department. CTE with DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC), filter = 2.

Q: How do you remove duplicate rows keeping one? ROW_NUMBER() OVER (PARTITION BY business_key ORDER BY ...) and delete rows where it is greater than 1.

Q: Window function vs GROUP BY? Windows keep detail rows and add group-level values; GROUP BY returns one row per group.

Window Functions II: Running Totals, LEAD, LAG & Frames

Build time-series analytics in SQL. Use window frames (ROWS BETWEEN) for running totals and moving averages, LAG and LEAD for month-on-month and year-on-year growth, FIRST_VALUE and LAST_VALUE, cumulative percentages, PERCENT_RANK, CUME_DIST and PERCENTILE_CONT, and gaps-and-islands patterns.

● SQL class 5–6

✦ SQL Server 2022, 2026

Advanced

🕒 130 min

By the end of this lesson you can

- Define a window frame with ROWS BETWEEN and explain the default frame when ORDER BY is present
- Calculate running totals, moving averages and cumulative shares
- Compare each row with the previous or next row using LAG and LEAD (MoM, YoY, gaps)
- Use FIRST_VALUE, LAST_VALUE and NTH_VALUE, PERCENT_RANK, CUME_DIST and PERCENTILE_CONT
- Solve streak and gap problems with ROW_NUMBER differences

Window frames

Inside `OVER`, `ORDER BY` defines the order and an optional **frame** defines *which rows around the current row* the function uses.

SQL

```
SUM(x) OVER (PARTITION BY p ORDER BY d
             ROWS BETWEEN <start> AND <end>)
```

Frame boundary	Meaning
UNBOUNDED PRECEDING	From the first row of the partition
n PRECEDING	n rows before the current row
CURRENT ROW	The current row
n FOLLOWING	n rows after
UNBOUNDED FOLLOWING	To the last row of the partition

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (running total)



ROWS BETWEEN 2 PRECEDING AND CURRENT ROW (3-row moving average)



ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING (centred window)



Frames relative to the current row (highlighted). A running total uses everything up to the current row; a 3-row moving average uses the previous two rows plus the current one.

COMMON MISTAKES **The default frame trap**

When `ORDER BY` is present and no frame is written, SQL Server uses `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. `RANGE` treats rows with the **same `ORDER BY` value** as peers and includes them all, which gives surprising results with ties, and `LAST_VALUE` then returns the current row's value. For running totals always write `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` explicitly; it is also much faster.

Running totals

First build the monthly sales table, then add a running (cumulative) total and a share:

SQL

```
WITH monthly AS (  
    SELECT YEAR(order_date) AS yr, MONTH(order_date) AS mth, SUM(sales) AS sales  
    FROM superstore  
    GROUP BY YEAR(order_date), MONTH(order_date)  
)  
SELECT yr, mth,  
    ROUND(sales, 0) AS sales,  
    ROUND(SUM(sales) OVER (PARTITION BY yr ORDER BY mth  
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW), 0) AS  
ytd_sales,  
    ROUND(100.0 * SUM(sales) OVER (PARTITION BY yr ORDER BY mth  
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)  
        / SUM(sales) OVER (PARTITION BY yr), 1) AS  
ytd_pct_of_year  
FROM monthly  
ORDER BY yr, mth;
```

RESULT

yr	mth	sales	ytd_sales	ytd_pct_of_year
2023	1	2,447,151	2,447,151	4.2
2023	2	3,861,852	6,309,002	10.9
2023	3	3,441,306	9,750,308	16.9
2023	4	3,927,557	13,677,865	23.7
2023	5	4,840,130	18,517,994	32.1
2023	6	5,785,712	24,303,706	42.2
2023	7	5,390,409	29,694,115	51.5
2023	8	4,129,386	33,823,500	58.7
2023	9	5,714,731	39,538,231	68.6
2023	10	5,134,331	44,672,561	77.5
2023	11	5,815,308	50,487,869	87.6
2023	12	7,152,348	57,640,217	100.0
2024	1	6,022,738	6,022,738	9.2
2024	2	2,463,147	8,485,885	13.0

36 rows × 5 columns (first 14 shown)

PARTITION BY yr restarts the running total every January (year-to-date). The percentage uses a second window (the whole year) as the denominator.

Moving averages

A moving average smooths noisy data and shows the trend. Here is the 7-day average of daily units sold:

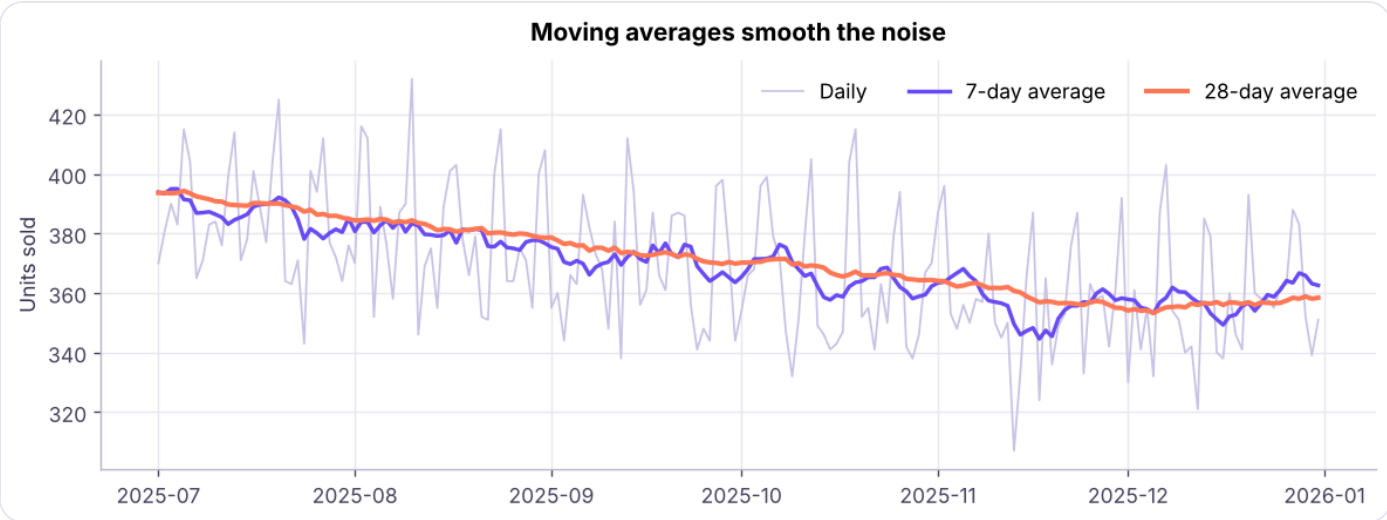
SQL

```
SELECT date, units_sold,
       ROUND(AVG(units_sold * 1.0) OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT
ROW), 1) AS avg_7_day
FROM daily_sales
WHERE date ≥ '2025-11-01'
ORDER BY date;
```

RESULT

date	units_sold	avg_7_day
2025-11-01	387	387.0
2025-11-02	396	391.5
2025-11-03	353	378.7
2025-11-04	348	371.0
2025-11-05	356	368.0
2025-11-06	350	365.0
2025-11-07	358	364.0
2025-11-08	357	359.7

61 rows × 3 columns (first 8 shown)



Daily units sold and the 7-day and 28-day moving averages (last 6 months of 2025). The weekly pattern (weekend peaks) disappears in the averages; the 28-day line shows the trend.

The first six rows of a 7-row window have fewer than seven rows available: SQL averages whatever exists (partial windows). Check `COUNT(*) OVER (...)` if you need to exclude them.

LAG and LEAD: compare with previous and next rows

Function	Returns
<code>LAG(col, n, default)</code>	The value <i>n</i> rows before the current row (default <i>n</i> = 1)
<code>LEAD(col, n, default)</code>	The value <i>n</i> rows after

SQL

```
WITH monthly AS (  
  SELECT YEAR(order_date) * 100 + MONTH(order_date) AS ym, SUM(sales) AS sales  
  FROM superstore  
  GROUP BY YEAR(order_date) * 100 + MONTH(order_date)  
)  
SELECT ym,  
  ROUND(sales, 0) AS sales,  
  ROUND(LAG(sales) OVER (ORDER BY ym), 0) AS prev_month,  
  ROUND(100.0 * (sales - LAG(sales) OVER (ORDER BY ym)) / LAG(sales) OVER (ORDER BY  
ym), 1) AS mom_pct,  
  ROUND(LAG(sales, 12) OVER (ORDER BY ym), 0) AS  
same_month_last_year,  
  ROUND(100.0 * (sales - LAG(sales, 12) OVER (ORDER BY ym)) / LAG(sales, 12) OVER  
(ORDER BY ym), 1) AS yoy_pct,  
  ROUND(LEAD(sales) OVER (ORDER BY ym), 0) AS next_month  
FROM monthly  
ORDER BY ym;
```

RESULT

ym	sales	prev_month	mom_pct	same_month_last_year	yoy_pct	next_mor
202301	2,447,151	NULL	NULL	NULL	NULL	3,861,85
202302	3,861,852	2,447,151	57.8	NULL	NULL	3,441,30
202303	3,441,306	3,861,852	-10.9	NULL	NULL	3,927,55
202304	3,927,557	3,441,306	14.1	NULL	NULL	4,840,13
202305	4,840,130	3,927,557	23.2	NULL	NULL	5,785,71
202306	5,785,712	4,840,130	19.5	NULL	NULL	5,390,40
202307	5,390,409	5,785,712	-6.8	NULL	NULL	4,129,38
202308	4,129,386	5,390,409	-23.4	NULL	NULL	5,714,73
202309	5,714,731	4,129,386	38.4	NULL	NULL	5,134,33
202310	5,134,331	5,714,731	-10.2	NULL	NULL	5,815,30
202311	5,815,308	5,134,331	13.3	NULL	NULL	7,152,34
202312	7,152,348	5,815,308	23.0	NULL	NULL	6,022,73

36 rows × 7 columns (first 12 shown)

- **MoM** compares with `LAG(sales, 1)`; **YoY** compares with `LAG(sales, 12)` when there is exactly one row per month. This is the SQL equivalent of the pivot table's "Difference From" settings (Excel Lesson 8).
- The first row has no previous value, so `LAG` returns `NULL` (or the default you supply: `LAG(sales, 1, 0)`).

WATCH OUT Gaps in the data

`LAG(sales, 12)` assumes every month exists. If a month has no sales it also has no row, so `LAG(12)` points at the wrong month. Build a calendar table (a list of every month) and `LEFT JOIN` the sales onto it first.

Time between events

"How many days passed between each customer's consecutive orders?"

SQL

```
WITH o AS (  
    SELECT customer_id, order_date,  
           LAG(order_date) OVER (PARTITION BY customer_id ORDER BY order_date) AS  
prev_order  
    FROM orders  
)  
SELECT COUNT(*) AS repeat_orders,  
       ROUND(AVG(DATEDIFF(day, prev_order, order_date) * 1.0), 1) AS  
avg_days_between_orders,  
       MIN(DATEDIFF(day, prev_order, order_date)) AS shortest_gap,  
       MAX(DATEDIFF(day, prev_order, order_date)) AS longest_gap  
FROM o  
WHERE prev_order IS NOT NULL;
```

RESULT

repeat_orders	avg_days_between_orders	shortest_gap	longest_gap
2415	186.6	0	1001

1 rows × 4 columns

FIRST_VALUE, LAST_VALUE, NTH_VALUE

SQL

```
SELECT department, emp_id, monthly_income,  
       FIRST_VALUE(emp_id) OVER (PARTITION BY department ORDER BY monthly_income DESC)  
AS top_earner,  
       LAST_VALUE(emp_id) OVER (PARTITION BY department ORDER BY monthly_income DESC  
                               ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  
AS lowest_earner,  
       monthly_income - FIRST_VALUE(monthly_income) OVER (PARTITION BY department ORDER BY  
monthly_income DESC) AS gap_to_top  
FROM employees  
ORDER BY department, monthly_income DESC;
```

RESULT

department	emp_id	monthly_income	top_earner	lowest_earner	gap_to_top
Finance	E00519	209833	E00519	E00322	0
Finance	E00145	185500	E00519	E00322	-24333
Finance	E00336	174417	E00519	E00322	-35416
Finance	E00201	169500	E00519	E00322	-40333
Finance	E01172	168167	E00519	E00322	-41666
Finance	E01151	155167	E00519	E00322	-54666
Finance	E00759	152083	E00519	E00322	-57750
Finance	E01004	144667	E00519	E00322	-65166

1,500 rows × 6 columns (first 8 shown)

LAST_VALUE needs the explicit full frame (UNBOUNDED FOLLOWING); with the default frame it just returns the current row.

Distribution functions

Function	Meaning
PERCENT_RANK()	$(\text{rank} - 1) \div (\text{rows} - 1)$: 0 for the lowest, 1 for the highest
CUME_DIST()	Share of rows with a value $\leq$ the current one
PERCENTILE_CONT(p) WITHIN GROUP (ORDER BY x) OVER (...)	Interpolated percentile (e.g. median at $p = 0.5$)
PERCENTILE_DISC(p) ...	An actual value from the data

SQL

```
SELECT DISTINCT department,
       ROUND(PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY monthly_income) OVER (PARTITION BY
department), 0) AS median_income,
       ROUND(AVG(monthly_income) OVER (PARTITION BY department), 0)
AS mean_income
FROM employees
ORDER BY department;
```

RESULT

department	median_income	mean_income
Finance	77,709	83,681
HR	86,833	87,805
IT	68,750	79,051
Operations	92,667	92,461
R&D	70,250	80,255
Sales	74,709	82,448

6 rows × 3 columns

In SQL Server, PERCENTILE_CONT is a window function (needs OVER); PostgreSQL uses it as an aggregate (GROUP BY). Mean above median is the signature of a right-skewed distribution (Statistics, ML track).

Cumulative share: the Pareto in SQL

Which products produce the first 80% of sales?

SQL

```
WITH p AS (
    SELECT product_id, SUM(sales) AS sales FROM order_items GROUP BY product_id
), c AS (
    SELECT product_id, sales,
           SUM(sales) OVER (ORDER BY sales DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) * 1.0 / SUM(sales) OVER () AS cum_share,
           ROW_NUMBER() OVER (ORDER BY sales DESC) AS rnk
    FROM p
)
SELECT COUNT(*) AS products_for_80pct_of_sales,
       (SELECT COUNT(*) FROM p) AS total_products
FROM c
WHERE cum_share ≤ 0.8;
```

RESULT

products_for_80pct_of_sales	total_products
20	78

1 rows × 2 columns

Gaps and islands: streaks

A classic: find runs of consecutive days. The trick is that `date - ROW_NUMBER()` is **constant within a streak**.

SQL

```
WITH days AS (  
    SELECT DISTINCT order_date AS d FROM orders WHERE order_date ≥ '2025-12-01'  
) , g AS (  
    SELECT d, DATEADD(day, -ROW_NUMBER() OVER (ORDER BY d), d) AS grp FROM days  
)  
SELECT MIN(d) AS streak_start, MAX(d) AS streak_end, COUNT(*) AS days_in_streak  
FROM g  
GROUP BY grp  
ORDER BY streak_start;
```

RESULT

streak_start	streak_end	days_in_streak
2025-12-01	2025-12-31	31

1 rows × 3 columns

Each group of consecutive order days becomes one row: all of December 2025 had orders every day, so there is a single streak. The same pattern finds consecutive login days, uninterrupted employment spells and missing-sequence ranges.

Summary

KEY TAKEAWAYS

- **Frames** (`ROWS BETWEEN ... AND ...`) choose the rows around the current row; always write `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` for running totals (the default `RANGE` frame includes ties).
- Running totals, YTD, cumulative % and moving averages use aggregate functions with frames.
- `LAG/LEAD` read the previous/next row (MoM, YoY, time between events); fill calendar gaps first.
- `FIRST_VALUE/LAST_VALUE` need an explicit full frame for `LAST_VALUE`.
- `PERCENT_RANK`, `CUME_DIST`, `PERCENTILE_CONT` describe position and median within a distribution.
- Gaps and islands: `value - ROW_NUMBER()` is constant within a consecutive run.

Quick quiz

6 questions · tap an answer to check it

1. Which frame gives a 3-month moving average (current and two previous rows)?

A ROWS BETWEEN 3 PRECEDING AND CURRENT ROW

B ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

C ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING

D RANGE 3

Answer: B. Two preceding rows plus the current row make three rows.

2. What does `LAG(sales, 12) OVER (ORDER BY ym)` return for monthly data?

A Sales 12 months in the future

B Sales for the same month last year (if no month is missing)

C The 12th highest sales

D The average of 12 months

Answer: B. LAG reads n rows back; with one row per month that is 12 months earlier.

3. Why does LAST_VALUE often return the current row's value?

- A** The default frame ends at the current row, so you must extend it to UNBOUNDED FOLLOWING
- B** LAST_VALUE is broken in SQL Server
- C** It only works with PARTITION BY
- D** ORDER BY is missing

Answer: A. Specify ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

4. How do you restart a running total every year?

- A** ORDER BY year
- B** PARTITION BY year in the OVER clause
- C** GROUP BY year only
- D** Use LAG

Answer: B. Each partition has its own running total.

5. In the gaps-and-islands trick, what stays constant inside a streak of consecutive dates?

- A** The date
- B** The row number
- C** The date minus its ROW_NUMBER
- D** The count

Answer: C. Both increase by 1 per row, so their difference is constant.

6. Mean income is much higher than median income. What does that suggest?

A Symmetric data

B A right-skewed distribution with a few very high values

C Missing data

D A data error

Answer: B. A long right tail pulls the mean above the median.

EXERCISE Time-series practice

(1) Monthly sales per region with MoM %; (2) for each customer, the days since their previous order and a flag for "returned within 30 days"; (3) a 3-month moving average of sales by category; (4) the month when cumulative 2025 sales first exceeded ₹5 crore.

Show solution (worked answer)

SQL

```
-- (1)
WITH m AS (SELECT region, YEAR(order_date)*100+MONTH(order_date) AS ym, SUM(sales) AS
sales FROM superstore GROUP BY region, YEAR(order_date)*100+MONTH(order_date))
SELECT region, ym, ROUND(sales,0) AS sales,
      ROUND(100.0*(sales - LAG(sales) OVER (PARTITION BY region ORDER BY ym)) /
LAG(sales) OVER (PARTITION BY region ORDER BY ym), 1) AS mom_pct FROM m ORDER BY
region, ym;

-- (2)
WITH o AS (SELECT customer_id, order_date, LAG(order_date) OVER (PARTITION BY
customer_id ORDER BY order_date) AS prev FROM orders)
SELECT customer_id, order_date, DATEDIFF(day, prev, order_date) AS days_since_prev,
      CASE WHEN DATEDIFF(day, prev, order_date) ≤ 30 THEN 1 ELSE 0 END AS
within_30_days FROM o WHERE prev IS NOT NULL;

-- (3)
WITH m AS (SELECT category, YEAR(order_date)*100+MONTH(order_date) AS ym, SUM(sales)
AS sales FROM superstore GROUP BY category, YEAR(order_date)*100+MONTH(order_date))
SELECT category, ym, AVG(sales) OVER (PARTITION BY category ORDER BY ym ROWS BETWEEN 2
PRECEDING AND CURRENT ROW) AS ma3 FROM m;

-- (4)
WITH m AS (SELECT MONTH(order_date) AS mth, SUM(sales) AS sales FROM superstore WHERE
YEAR(order_date) = 2025 GROUP BY MONTH(order_date)),
      c AS (SELECT mth, SUM(sales) OVER (ORDER BY mth ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS cum FROM m)
SELECT TOP 1 mth FROM c WHERE cum > 50000000 ORDER BY mth;
```

INTERVIEW CORNER**Questions you may be asked**

Q: How do you calculate a running total in SQL? `SUM(x) OVER (ORDER BY d ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`, optionally partitioned.

Q: How do you calculate month-on-month growth? Aggregate to months, then `(sales - LAG(sales) OVER (ORDER BY month)) / LAG(sales) OVER (ORDER BY month)`; use `LAG(..., 12)` for YoY.

Q: ROWS vs RANGE in a window frame? ROWS counts physical rows; RANGE includes all rows with the same ORDER BY value as the current row (peers). ROWS is predictable and faster.

Q: How would you find consecutive login days? Gaps and islands: `date - ROW_NUMBER() OVER (PARTITION BY user ORDER BY date)` as a group key, then GROUP BY it.

CTEs, Views, Temporary Tables, Stored Procedures & Functions

Write queries other people can read and reuse. Use CTEs (including recursive CTEs for hierarchies and calendars), create views for reusable logic and security, use temp tables and table variables, and build stored procedures, user-defined functions, parameters and basic error handling in T-SQL.

● SQL class 6

✦ SQL Server 2022, 2026

Advanced

🕒 130 min

By the end of this lesson you can

- Structure complex queries with one or more CTEs and explain CTE vs subquery vs temp table
- Write recursive CTEs for hierarchies (org charts) and number or date series
- Create, alter and use views, and explain what views are good for
- Choose between temp tables, table variables and CTEs
- Write stored procedures with parameters, scalar and table-valued functions, and TRY...CATCH

CTE: a named, readable subquery

A **Common Table Expression** is a named result set defined with `WITH` that exists for **one statement**. It makes complex logic read top to bottom, like paragraphs, instead of inside-out.

SQL

```
WITH name1 AS (  
    SELECT ...  
)  
name2 AS (  
    SELECT ... FROM name1 ...      -- later CTEs can use earlier ones  
)  
SELECT ... FROM name2 ...;
```

From nested subqueries to CTEs

Question: "Which regions had above-average 2025 sales per customer?"

SQL

```
WITH customer_sales AS (                                -- step 1: sales per customer in 2025
    SELECT c.region, c.customer_id, SUM(oi.sales) AS sales
    FROM order_items AS oi
    JOIN orders AS o ON o.order_id = oi.order_id
    JOIN customers AS c ON c.customer_id = o.customer_id
    WHERE o.order_date ≥ '2025-01-01' AND o.order_date < '2026-01-01'
    GROUP BY c.region, c.customer_id
),
region_avg AS (                                         -- step 2: average per region
    SELECT region, COUNT(*) AS customers, AVG(sales) AS avg_sales_per_customer
    FROM customer_sales
    GROUP BY region
)
SELECT region, customers,
       ROUND(avg_sales_per_customer, 0) AS avg_sales_per_customer,
       ROUND((SELECT AVG(sales) FROM customer_sales), 0) AS overall_avg
FROM region_avg
WHERE avg_sales_per_customer > (SELECT AVG(sales) FROM customer_sales) -- step 3: compare
ORDER BY avg_sales_per_customer DESC;
```

RESULT

region	customers	avg_sales_per_customer	overall_avg
West	156	142,513	125,805
South	160	130,010	125,805

2 rows × 4 columns

customer_sales is used **twice**, which with plain subqueries would mean copy-pasting the same block. CTEs also make window-function filters (Lesson 11) clean.

TIP CTE style

Name CTEs for what they contain (customer_sales, not cte1); keep each CTE doing one job; comment each step. If a CTE needs a long explanation, it probably should be two CTEs.

CTE, subquery, view and temp table

	Subquery / derived table	CTE	View	Temp table
Lives for	The statement	The statement	Permanently (a saved query)	The session (#t)
Stores data?	No	No (re-evaluated)	No (unless indexed/materialised)	Yes
Reusable in other queries	No	No	Yes	In the same session
Can be referenced several times	Only by repeating	Yes , in the same statement	Yes	Yes
Recursive	No	Yes	No	No
Can be indexed	No	No	Indexed views only	Yes
Best for	Simple one-off logic	Readability, multi-step logic, recursion	Sharing logic, hiding complexity, security	Large intermediate results used many times

NOTE A CTE is not a cache

In SQL Server the CTE's query may run each time it is referenced. If an expensive CTE is used many times, load it into a **temp table** once.

Recursive CTEs

A recursive CTE has an **anchor** (starting rows) and a **recursive member** that refers to the CTE itself, joined with UNION ALL. It repeats until no new rows are produced.

Number and date series

SQL

```
WITH numbers AS (
    SELECT 1 AS n                -- anchor
    UNION ALL
    SELECT n + 1 FROM numbers WHERE n < 7    -- recursive member with a stop condition
)
SELECT n FROM numbers;
```

RESULT

n
1
2
3
4
5
6
7

7 rows × 1 columns

A **calendar table** of every day in 2026, handy for filling gaps before LAG/LEAD (Lesson 12):

SQL

```
WITH calendar AS (  
    SELECT CAST('2026-01-01' AS DATE) AS d  
    UNION ALL  
    SELECT CAST(DATEADD(day, 1, d) AS DATE) FROM calendar WHERE d < CAST('2026-12-31' AS  
DATE)  
)  
SELECT COUNT(*) AS days, MIN(d) AS first_day, MAX(d) AS last_day,  
       SUM(CASE WHEN DATENAME(weekday, d) IN ('Saturday', 'Sunday') THEN 1 ELSE 0 END) AS  
weekend_days  
FROM calendar;
```

RESULT

days	first_day	last_day	weekend_days
365	2026-01-01	2026-12-31	104

1 rows × 4 columns

In SQL Server, the default limit is 100 recursions, so a year-long calendar needs `OPTION (MAXRECURSION 400)` after the final `SELECT` (use 0 for no limit, with care).

Hierarchies: org chart

SQL

```
CREATE TABLE org (emp_id INT, name VARCHAR(20), manager_id INT);
INSERT INTO org VALUES (1,'Varun',NULL), (2,'Asha',1), (3,'Ravi',1), (4,'Meera',2),
(5,'Kabir',2), (6,'Zoya',4), (7,'Farhan',3);

WITH tree AS (
    SELECT emp_id, name, manager_id, 0 AS level, CAST(name AS VARCHAR(200)) AS path
    FROM org WHERE manager_id IS NULL -- anchor: the top
    UNION ALL
    SELECT o.emp_id, o.name, o.manager_id, t.level + 1, CAST(CONCAT(t.path, ' > ', o.name)
AS VARCHAR(200))
    FROM org AS o
    JOIN tree AS t ON o.manager_id = t.emp_id -- walk down one level
)
SELECT level, CONCAT(REPLICATE(' ', level), name) AS org_chart, path
FROM tree
ORDER BY path;
```

RESULT

Commands completed successfully.
(7 rows affected)

level	org_chart	path
0	Varun	Varun
1	Asha	Varun > Asha
2	Kabir	Varun > Asha > Kabir
2	Meera	Varun > Asha > Meera
3	Zoya	Varun > Asha > Meera > Zoya
1	Ravi	Varun > Ravi
2	Farhan	Varun > Ravi > Farhan

7 rows × 3 columns

Recursive CTEs also solve bill-of-materials, category trees and graph-path problems.

Views

A **view** is a saved SELECT with a name. Querying it runs the stored query; it holds no data of its own.

SQL

```
CREATE VIEW vw_order_detail AS
SELECT o.order_id, o.order_date, o.ship_mode,
       c.customer_id, c.customer_name, c.segment, c.city, c.region,
       p.product_id, p.product_name, p.category, p.sub_category,
       oi.quantity, oi.discount, oi.sales, oi.profit
FROM order_items AS oi
JOIN orders      AS o ON o.order_id    = oi.order_id
JOIN customers   AS c ON c.customer_id = o.customer_id
JOIN products    AS p ON p.product_id  = oi.product_id;

SELECT category, region, ROUND(SUM(sales), 0) AS sales
FROM vw_order_detail
WHERE order_date ≥ '2025-01-01'
GROUP BY category, region
ORDER BY category, region;
```

RESULT

category	region	sales
Furniture	East	2,875,816
Furniture	North	2,688,426
Furniture	South	3,759,688
Furniture	West	3,436,316
Office Supplies	East	1,075,076
Office Supplies	North	833,204
Office Supplies	South	1,018,343
Office Supplies	West	1,044,699
Technology	East	14,959,309
Technology	North	12,911,446
Technology	South	16,023,523
Technology	West	17,750,953

12 rows × 3 columns

Now analysts query one friendly object instead of repeating a four-table join. Power BI and Excel can connect to the view too.

Use views to	Notes
Hide complexity	One name for a multi-join query
Provide one definition of a metric	"Net sales" defined once, used everywhere
Secure data	GRANT SELECT ON vw_sales_summary TO analysts without granting access to base tables; expose only some columns/rows
Keep apps working when tables change	Change the view, not every report

SQL Server syntax: CREATE VIEW dbo.vw_x AS ..., ALTER VIEW, DROP VIEW IF EXISTS, CREATE OR ALTER VIEW. Rules: no ORDER BY inside a view (unless with TOP); views can usually be INSERT/UPDATED only if they map simply to one table. **Indexed (materialised) views** store the results physically for speed (WITH SCHEMABINDING + a unique clustered index).

Temporary tables and table variables

SQL

```
-- Temp table: lives for the session, stored in tempdb, can be indexed, has statistics
SELECT customer_id, SUM(sales) AS sales
INTO #cust_sales -- the # prefix makes it local temporary
FROM superstore
GROUP BY customer_id;

CREATE INDEX ix_cust ON #cust_sales (customer_id);
SELECT TOP 10 * FROM #cust_sales ORDER BY sales DESC;
DROP TABLE IF EXISTS #cust_sales;

-- Table variable: scoped to the batch; for small row counts
DECLARE @top TABLE (customer_id VARCHAR(10), sales DECIMAL(14,2));
INSERT INTO @top SELECT TOP 5 customer_id, SUM(sales) FROM superstore GROUP BY customer_id
ORDER BY SUM(sales) DESC;
SELECT * FROM @top;
```

##name is a **global** temp table visible to other sessions. Temp tables vanish when the session ends.

Stored procedures

A **stored procedure** is a named, saved block of T-SQL that can take parameters, run many statements, and return results. It is compiled and its plan cached, and permissions can be granted on the procedure instead of the tables.

SQL

```
CREATE OR ALTER PROCEDURE dbo.usp_sales_by_region
    @from_date DATE,
    @to_date DATE,
    @category VARCHAR(30) = NULL           -- optional parameter with a default
AS
BEGIN
    SET NOCOUNT ON;                      -- suppress "(n rows affected)" messages

    SELECT c.region,
           COUNT(DISTINCT o.order_id) AS orders,
           SUM(oi.sales) AS sales,
           SUM(oi.profit) AS profit
    FROM dbo.order_items AS oi
    JOIN dbo.orders AS o ON o.order_id = oi.order_id
    JOIN dbo.customers AS c ON c.customer_id = o.customer_id
    JOIN dbo.products AS p ON p.product_id = oi.product_id
    WHERE o.order_date ≥ @from_date
           AND o.order_date < DATEADD(day, 1, @to_date)
           AND (@category IS NULL OR p.category = @category) -- optional filter
    GROUP BY c.region
    ORDER BY sales DESC;
END;
GO

EXEC dbo.usp_sales_by_region @from_date = '2025-01-01', @to_date = '2025-12-31';
EXEC dbo.usp_sales_by_region '2025-01-01', '2025-12-31', 'Technology';
```

Output parameters, variables, IF and loops

SQL

```
CREATE OR ALTER PROCEDURE dbo.usp_add_order_note
    @order_id VARCHAR(20),
    @note      NVARCHAR(200),
    @rows      INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    IF NOT EXISTS (SELECT 1 FROM dbo.orders WHERE order_id = @order_id)
    BEGIN
        RAISERROR('Order %s does not exist.', 16, 1, @order_id); -- or THROW 50001,
        'Order does not exist.', 1;
        RETURN;
    END;

    UPDATE dbo.orders SET note = @note WHERE order_id = @order_id;
    SET @rows = @@ROWCOUNT; -- rows changed by the
    last statement
END;
GO

DECLARE @n INT;
EXEC dbo.usp_add_order_note 'IN-2025-00042', N'Gift wrap', @rows = @n OUTPUT;
SELECT @n AS rows_updated;
```

Control flow	Syntax
Condition	IF ... BEGIN ... END ELSE BEGIN ... END
Loop	WHILE @i ≤ 10 BEGIN ... SET @i += 1; END
Exit / return	RETURN
Row count	@@ROWCOUNT
Dynamic SQL	EXEC sp_executesql @sql, N'@x INT', @x = 5 (use parameters; never concatenate user input: SQL injection)

Error handling: TRY...CATCH

SQL

```
BEGIN TRY
    BEGIN TRAN;
        UPDATE dbo.accounts SET balance = balance - 5000 WHERE acct_id = 1;
        UPDATE dbo.accounts SET balance = balance + 5000 WHERE acct_id = 2;
    COMMIT;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0 ROLLBACK;
    SELECT ERROR_NUMBER() AS err_no, ERROR_MESSAGE() AS err_msg, ERROR_LINE() AS err_line;
    THROW;
    -- re-raise to the caller
END CATCH;
```

User-defined functions (UDFs)

Type	Returns	Example use
Scalar	One value	dbo.fn_margin(profit, sales)
Inline table-valued	A table, from one SELECT	dbo.fn_orders_by_city('Pune'): like a parameterised view
Multi-statement table-valued	A table built in a body	Complex logic (usually slower)

SQL

```
CREATE OR ALTER FUNCTION dbo.fn_margin (@profit DECIMAL(12,2), @sales DECIMAL(12,2))
RETURNS DECIMAL(6,2)
AS
BEGIN
    RETURN CASE WHEN @sales = 0 THEN NULL ELSE 100.0 * @profit / @sales END;
END;
GO

CREATE OR ALTER FUNCTION dbo.fn_sales_by_city (@city VARCHAR(30))
RETURNS TABLE
AS RETURN (
    SELECT order_id, order_date, sales, profit FROM dbo.vw_order_detail WHERE city = @city
);
GO

SELECT order_id, dbo.fn_margin(profit, sales) AS margin_pct FROM
dbo.fn_sales_by_city('Pune');
```

WATCH OUT Scalar UDFs can be slow

Older SQL Server versions call a scalar UDF once per row, which can make a query many times slower than writing the expression inline (SQL Server 2019+ can inline some). Prefer inline table-valued functions or views for set-based logic.

Procedure vs function vs view

	View	Stored procedure	Function
Parameters	No	Yes	Yes
Can change data (INSERT/UPDATE)	Limited	Yes	No (side-effect free)
Used in SELECT / FROM	Yes	No (EXEC)	Yes
Returns	A table	Result sets, output parameters, a status code	A value or a table

Triggers (awareness)

A **trigger** runs automatically on INSERT, UPDATE or DELETE: audit logs, maintaining summary tables. Use sparingly: they hide logic and can slow writes. Modern designs prefer explicit procedures, constraints, temporal tables or change data capture.

DO IT WITH AI Do it with AI

Prompt: "Convert this nested-subquery SQL into readable CTEs with clear names and comments, keep the result identical, and explain each step." AI is good at refactoring; verify equality by running both versions and comparing with EXCEPT in both directions (Lesson 9).

Summary

KEY TAKEAWAYS

- **CTE** (`WITH name AS (...)`): a named, readable subquery for one statement; can be referenced many times; **recursive CTEs** generate series and walk hierarchies (anchor + `UNION ALL` + recursive member).
- **View**: a saved query for reuse, one definition of a metric, and security. **Temp tables** (`#t`) store intermediate data and can be indexed; table variables suit small sets.
- **Stored procedures** take parameters, run multiple statements, change data and handle errors with `TRY...CATCH`; **functions** return values or tables and can be used inside queries.
- Prefer set-based SQL; avoid row-by-row loops and scalar UDFs on big tables; never concatenate user input into dynamic SQL.

Quick quiz

6 questions · tap an answer to check it

1. A CTE exists for:

A The whole session

B A single statement

C Until the server restarts

D As long as the database

Answer: B. A CTE is defined and used within one statement.

2. What are the two parts of a recursive CTE?

- A SELECT and WHERE
- B An anchor member and a recursive member joined with UNION ALL**
- C INSERT and UPDATE
- D A view and a function

Answer: B. The anchor gives starting rows; the recursive member repeats until nothing new appears.

3. Which object stores data physically for the session and can be indexed?

- A CTE
- B View
- C Temporary table**
- D Subquery

Answer: C.

temp tables live in tempdb for the session.

4. Which can modify data and return an OUTPUT parameter?

- A View
- B Stored procedure**
- C Scalar function
- D CTE

Answer: B. Procedures can run INSERT/UPDATE/DELETE and return output parameters.

5. Why grant access to a view instead of the base table?

- A** Views are faster
- B** You can expose only selected columns and rows and hide the underlying tables
- C** Views cannot be dropped
- D** Tables cannot be queried

Answer: B. Views are a security and simplification layer.

6. A recursive CTE stops with an error after 100 levels. Fix?

- A** Add DISTINCT
- B** Add OPTION (MAXRECURSION n) (and a proper stop condition)
- C** Use a view
- D** Use TOP

Answer: B. SQL Server's default recursion limit is 100.

EXERCISE Build reusable objects

(1) A view `vw_monthly_sales` with year, month, category and total sales. (2) A CTE query that lists, for each region, the best-selling sub-category and its share of the region's sales. (3) A stored procedure `usp_top_customers` (@year INT, @n INT) that returns the top n customers by sales for a year. (4) A recursive CTE that lists all employees under manager 2 in the `org` table with their level.

Show solution (worked answer)

SQL

```
-- (1)
CREATE VIEW vw_monthly_sales AS
SELECT YEAR(order_date) AS yr, MONTH(order_date) AS mth, category, SUM(sales) AS sales
FROM vw_order_detail GROUP BY YEAR(order_date), MONTH(order_date), category;

-- (2)
WITH rs AS (SELECT region, sub_category, SUM(sales) AS sales FROM superstore GROUP BY
region, sub_category),
    r AS (SELECT *, RANK() OVER (PARTITION BY region ORDER BY sales DESC) AS rnk,
100.0*sales/SUM(sales) OVER (PARTITION BY region) AS pct FROM rs)
SELECT region, sub_category, ROUND(pct,1) AS pct_of_region FROM r WHERE rnk = 1;

-- (3) T-SQL
CREATE OR ALTER PROCEDURE usp_top_customers @year INT, @n INT AS
BEGIN
    SELECT TOP (@n) customer_id, customer_name, SUM(sales) AS sales FROM vw_order_detail
WHERE YEAR(order_date) = @year
    GROUP BY customer_id, customer_name ORDER BY sales DESC;
END;

-- (4)
WITH t AS (SELECT emp_id, name, 1 AS level FROM org WHERE manager_id = 2
    UNION ALL SELECT o.emp_id, o.name, t.level + 1 FROM org o JOIN t ON
o.manager_id = t.emp_id)
SELECT * FROM t;
```

INTERVIEW CORNER**Questions you may be asked**

Q: CTE vs temp table vs subquery? Subquery: inline, single use. CTE: named, readable, statement-scoped, supports recursion, may be re-evaluated. Temp table: materialised data for the session, indexable: use for large intermediate results reused many times.

Q: View vs stored procedure? A view is a saved SELECT you query like a table; a procedure is a program with parameters and multiple statements that can change data.

Q: What is a recursive CTE and where do you use it? A CTE that references itself, for hierarchies (org charts, categories), series generation and graph traversal.

Q: How do you prevent SQL injection in dynamic SQL? Use parameterised `sp_executesql` and `QUOTENAME` for identifiers; never concatenate raw user input.

Transactions, Indexes, Performance & Security

Make SQL safe and fast. Understand transactions and ACID (BEGIN, COMMIT, ROLLBACK, SAVE), isolation and locking, indexes (clustered, non-clustered, covering), how to read an execution plan, a checklist for writing fast queries, and permissions with GRANT, REVOKE and roles (DCL).

● SQL class 6 (extended)

✦ SQL Server 2022, 2026

Advanced

🕒 110 min

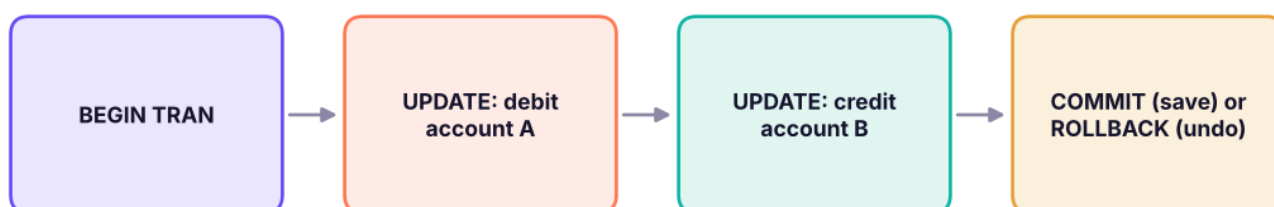
By the end of this lesson you can

- Explain ACID and group statements into transactions that commit or roll back together
- Describe isolation levels, locks, blocking and deadlocks at an analyst level
- Choose between clustered and non-clustered indexes and recognise queries that can or cannot use them
- Read the basics of an execution plan and apply a performance checklist
- Grant and revoke permissions with roles, and apply least privilege

Transactions and ACID

A **transaction** is a group of statements treated as one unit of work: **all succeed (commit) or none do (rollback)**. Moving ₹5,000 between two accounts is two updates; if the server crashes between them, money would vanish without a transaction.

One unit of work



A bank transfer as a transaction. Either both updates are saved, or both are undone.

Property	Meaning	Example
Atomicity	All or nothing	Both account updates happen, or neither
Consistency	Data stays valid (constraints hold)	Total money before = total after; no negative balance if a CHECK forbids it
Isolation	Concurrent transactions do not see each other's half-finished work	Another user does not see the debit without the credit
Durability	Once committed, it survives a crash	Transaction log is written to disk first

Transaction commands

Command	Purpose
<code>BEGIN TRANSACTION</code> (<code>BEGIN TRAN</code>)	Start a transaction
<code>COMMIT</code>	Make all changes permanent
<code>ROLLBACK</code>	Undo everything since <code>BEGIN</code> (or since a savepoint)
<code>SAVE TRANSACTION name</code>	Create a savepoint inside the transaction
<code>ROLLBACK TRANSACTION name</code>	Undo back to the savepoint only
<code>@@TRANCOUNT</code>	How many transactions are open (should be 0 when you leave)

By default SQL Server runs in **autocommit** mode: every single statement is its own transaction. Wrap several statements in `BEGIN TRAN ... COMMIT` to group them.

Seeing a rollback

SQL

```
CREATE TABLE accounts (acct_id INT PRIMARY KEY, holder VARCHAR(20), balance DECIMAL(12,2));
INSERT INTO accounts VALUES (1, 'Asha', 20000), (2, 'Ravi', 5000);

BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 5000 WHERE acct_id = 1;
UPDATE accounts SET balance = balance + 5000 WHERE acct_id = 2;
SELECT * FROM accounts ORDER BY acct_id;      -- inside the transaction: changes visible to us
ROLLBACK;                                     -- something went wrong: undo both updates

SELECT * FROM accounts ORDER BY acct_id;      -- balances are back to the originals
```

RESULT

Commands completed successfully.
(2 rows affected)
Commands completed successfully.
(1 row affected)
(1 row affected)
Commands completed successfully.

RESULT SET 1

acct_id	holder	balance
1	Asha	15,000
2	Ravi	10,000

2 rows × 3 columns

RESULT SET 2

acct_id	holder	balance
1	Asha	20,000
2	Ravi	5,000

2 rows × 3 columns

The safe-update pattern

The pattern that protects you from the "UPDATE without WHERE" disaster (Lesson 4):

SQL

```
BEGIN TRAN;

UPDATE dbo.staff SET salary = salary * 1.05 WHERE dept_id = 10;

SELECT @@ROWCOUNT AS rows_changed;           -- expected: 2?
SELECT emp_id, salary FROM dbo.staff WHERE dept_id = 10;    -- sanity-check

-- If everything looks right:
COMMIT;
-- Otherwise:
-- ROLLBACK;
```

WATCH OUT Never leave a transaction open

An open transaction holds locks that block other users and keeps the log from truncating. Always end with COMMIT or ROLLBACK; check SELECT @@TRANCOUNT; if a query window seems "stuck".

Concurrency: isolation, locks and deadlocks

When many users work at once, the database uses **locks** to keep transactions isolated. **Isolation levels** trade consistency for concurrency:

Level	Prevents	Notes
READ UNCOMMITTED	Nothing (allows dirty reads; NOLOCK hint)	Can read data that is later rolled back; avoid for reports needing accuracy
READ COMMITTED (default)	Dirty reads	A row you read can change before you read it again
REPEATABLE READ	Dirty and non-repeatable reads	Holds read locks longer
SERIALIZABLE	Also phantom rows	Safest, least concurrent
SNAPSHOT / RCSI	Readers do not block writers (row versions)	Common in modern systems, uses tempdb

- **Blocking:** one session waits for a lock held by another. Fix: shorter transactions, indexes, row versioning.
- **Deadlock:** two sessions each hold a lock the other needs; SQL Server kills one (the "victim", error 1205). Fix: access tables in the same order, keep transactions short, retry on error 1205.

Indexes: how databases find rows fast

Without an index, a query like WHERE customer_id = 'C0042' scans **every row** (table scan). An **index** is a sorted structure (a B-tree) that lets the database jump straight to matching rows, like the index of a book.

Scan vs seek

Table scan (no index)

- Read all 5,000,000 rows
- Check each against the filter
- Time grows with table size
- Fine for tiny tables and full reports

Index seek

- Walk root → branch → leaf (3–4 steps)
- Read only matching rows
- Time stays almost flat
- Great for lookups, joins and sorting

A table scan reads every row. An index seek follows a short path through a B-tree.

Index type	What it is	How many per table
Clustered	Defines the physical order of the table's rows; the table <i>is</i> the index. Created by default with a PRIMARY KEY	One
Non-clustered	A separate sorted copy of selected columns with pointers to the rows	Many (up to 999)
Unique	Enforces uniqueness (created for UNIQUE constraints)	Many
Covering (INCLUDE)	A non-clustered index that contains every column a query needs, so the table is never touched	
Filtered	Indexes only rows matching a condition	
Columnstore	Column-wise compressed storage for analytics on huge tables	

SQL

```
CREATE CLUSTERED INDEX cx_orders_date ON dbo.orders (order_date);           -- if not
already clustered by the PK

CREATE NONCLUSTERED INDEX ix_orders_customer
  ON dbo.orders (customer_id)
  INCLUDE (order_date, ship_mode);                                         -- covering
index for "orders per customer" queries

CREATE NONCLUSTERED INDEX ix_items_product ON dbo.order_items (product_id); -- foreign
keys deserve indexes

DROP INDEX ix_orders_customer ON dbo.orders;
```

SQL

```
CREATE INDEX ix_super_city ON superstore (city);  
SELECT index_name, table_name FROM duckdb_indexes() WHERE table_name = 'superstore';
```

RESULT

index_name	table_name
ix_super_city	superstore

1 rows × 2 columns

Which queries can use an index?

Pattern	Uses an index?	Why
WHERE customer_id = 'C0042'	Yes (seek)	Equality on the leading column
WHERE order_date ≥ '2025-01-01' AND order_date < '2026-01-01'	Yes	Range on an indexed column
WHERE YEAR(order_date) = 2025	No (scan)	The function must run on every row: non-SARGable
WHERE customer_name LIKE 'Ra%'	Yes	Prefix match
WHERE customer_name LIKE '%Rao'	No	Leading wildcard
WHERE UPPER(city) = 'PUNE'	No	Function on the column (the default collation is already case-insensitive)
WHERE city = 'Pune' OR product_id = 'P0012'	Often no	OR across different columns
WHERE CAST(order_id AS VARCHAR) = '1001'	No	Type conversion on the column

TIP SARGable predicates

SARG = Search ARGument able. Keep the **column bare** on one side of the comparison and do the maths on the other side: use `order_date ≥ '2025-01-01' AND order_date < '2026-01-01'` instead of `YEAR(order_date) = 2025`, and `sales > 100 / 1.18` instead of `sales * 1.18 > 100`.

Index trade-offs

Indexes speed up **reads** but slow down **writes** (every INSERT/UPDATE/DELETE must also update each index) and use disk space. Index columns used in WHERE, JOIN, ORDER BY and GROUP BY, and foreign keys; do not index everything. Analytic warehouses (Snowflake, BigQuery) use different techniques (partitioning, clustering, columnar storage) instead of manual indexes.

Execution plans

In SSMS press `Ctrl + L` (estimated plan) or `Ctrl + M` then run (actual plan). The plan is a diagram of operators read **right to left**. Things to look for:

Operator	Meaning	Concern
Index Seek	Direct lookup	Good
Index / Clustered Index Scan	Reads many rows	Fine for big reports, suspicious for single-row lookups
Table Scan	Reads the heap	Usually add an index
Key Lookup	Fetches extra columns from the table per row	Consider a covering index (INCLUDE)
Hash / Merge / Nested Loops Join	Join algorithms	Hash for big unsorted inputs; nested loops for small; merge for sorted
Sort	Sorting rows	Expensive; an index in the right order avoids it
Thick arrows, big estimated vs actual row differences	Bad estimates	Update statistics

`SET STATISTICS IO, TIME ON`; shows logical reads (the best measure of work) and CPU time for each query.

Performance checklist for analysts

1. **Select only the columns and rows you need**: avoid `SELECT *`; filter early.
2. **Make filters SARGable** (no functions on columns; use ranges for dates).
3. **Join on indexed keys** with matching data types (an INT joined to a VARCHAR forces conversions).
4. **Aggregate before joining** when joining big tables (reduce rows first, avoids row explosion).
5. **UNION ALL** instead of UNION; **EXISTS** instead of IN (huge list); avoid DISTINCT as a bandage for join duplicates.
6. **Avoid row-by-row logic** (cursors, scalar UDF in WHERE); think in sets.
7. **Test on realistic data**; look at the actual plan; keep statistics updated.
8. **Pull less data into BI tools**: summarise in SQL, or create views and aggregate tables.
9. **Limit while developing**: `SELECT TOP 100` or `WHERE` on a date range, then remove it.

Security: DCL

Data Control Language gives and removes permissions. Follow the **principle of least privilege**: grant only what each person or application needs.

SQL

```
-- Logins (server level) and users (database level)
CREATE LOGIN priya_analyst WITH PASSWORD = 'Use-A-Strong-Passphrase-1!';
CREATE USER  priya_analyst FOR LOGIN priya_analyst;

-- Roles group permissions
CREATE ROLE analyst_role;
GRANT SELECT ON SCHEMA::dbo          TO analyst_role;           -- read all tables in dbo
DENY  SELECT ON dbo.customers (phone, email) TO analyst_role;    -- but hide personal
columns
GRANT EXECUTE ON dbo.usp_sales_by_region TO analyst_role;

ALTER ROLE analyst_role ADD MEMBER priya_analyst;              -- SQL Server 2012+: ALTER
ROLE ... ADD MEMBER

REVOKE SELECT ON dbo.orders FROM analyst_role;                 -- take back a permission
```

Statement	Effect
GRANT	Gives a permission
REVOKE	Removes a previously granted (or denied) permission
DENY	Explicitly blocks a permission (overrides grants)
Fixed roles	db_datareader (read all), db_datawriter, db_owner (full control: avoid)

Other protections: **Row-Level Security** (a region manager sees only their rows), **Dynamic Data Masking** (show XXXX-1234), **Always Encrypted** and **TDE** (encryption), **auditing**, and never embedding passwords in scripts (use Windows/Entra ID authentication or a secret store). In India, personal data handling also falls under the **DPDP Act 2023**: collect only what you need and restrict who can see it.

DO IT WITH AI

Do it with AI

Paste a slow query and its actual execution plan (as XML or text) into an assistant: *"Explain what this plan is doing and suggest indexes or rewrites. Do not invent columns."* AI can spot scans, key lookups and non-SARGable filters quickly, but test every suggestion; an index that helps one query can hurt a hundred writes.

Summary

KEY TAKEAWAYS

- A **transaction** groups statements to commit or roll back together; **ACID** = atomicity, consistency, isolation, durability. Use `BEGIN TRAN ... COMMIT/ROLLBACK`, check `@@ROWCOUNT`, never leave transactions open.
- Isolation levels and locks keep concurrent users consistent; blocking and deadlocks are managed with short transactions and consistent access order.
- **Indexes** (clustered = row order, one per table; non-clustered = separate structure; covering = `INCLUDE`) speed up reads and slow writes. **SARGable** filters keep the column bare.
- Read execution plans for scans, key lookups and bad estimates; use `SET STATISTICS IO, TIME ON`.
- **DCL**: `GRANT`, `REVOKE`, `DENY`, roles, least privilege; protect personal data.

Quick quiz

6 questions · tap an answer to check it

1. Which ACID property says that a committed transaction survives a power failure?

A Atomicity

B Consistency

C Isolation

D Durability

Answer: D. Durability: committed changes are written to the log/disk.

2. Which filter can use an index on order_date?

- A WHERE YEAR(order_date) = 2025
- B WHERE order_date ≥ '2025-01-01' AND order_date < '2026-01-01'
- C WHERE CONVERT(VARCHAR, order_date, 23) LIKE '2025%'
- D WHERE DATEPART(year, order_date) = 2025

Answer: B. The column must be bare (SARGable); functions on the column force a scan.

3. How many clustered indexes can a table have?

- A None
- B One
- C One per column
- D Unlimited

Answer: B. The clustered index defines the physical order, so there can be only one.

4. You ran BEGIN TRAN; UPDATE ...; and closed the window. What is the risk?

- A The transaction stays open, holding locks and blocking other users
- B The update is committed automatically
- C The table is dropped
- D Nothing

Answer: A. Open transactions keep locks until committed or rolled back.

5. What does DENY do?

- A Removes a permission like REVOKE
- B Explicitly blocks a permission, overriding grants**
- C Deletes a login
- D Grants read-only access

Answer: B. DENY takes precedence over GRANT.

6. In an execution plan, a "Key Lookup" repeated for many rows suggests:

- A The table is empty
- B Adding the needed columns to the index with INCLUDE (a covering index)**
- C Using UNION instead of UNION ALL
- D Dropping the clustered index

Answer: B. Key lookups fetch missing columns from the table row by row; covering the query avoids them.

EXERCISE Make it faster

This query is slow on 50 million order lines: `SELECT * FROM order_items WHERE YEAR(order_date) = 2025 AND UPPER(region) = 'WEST' ORDER BY sales DESC;`
Rewrite it and suggest an index.

Show solution (worked answer)

SQL

```
SELECT item_id, order_id, product_id, quantity, sales, profit      -- only the needed
columns
FROM dbo.order_items
WHERE order_date ≥ '2025-01-01' AND order_date < '2026-01-01'    -- SARGable range
instead of YEAR()
    AND region = 'West'                                           -- default
collation is case-insensitive; no UPPER needed
ORDER BY sales DESC;

CREATE NONCLUSTERED INDEX ix_items_date_region
ON dbo.order_items (order_date, region) INCLUDE (order_id, product_id, quantity,
sales, profit);
```

Check the actual plan for a seek, compare logical reads with `SET STATISTICS IO ON`, and consider whether the extra index slows nightly loads.

INTERVIEW CORNER Questions you may be asked

Q: What is ACID? Atomicity (all or nothing), Consistency (valid state), Isolation (concurrent transactions do not interfere), Durability (committed data persists).

Q: Clustered vs non-clustered index? A clustered index sorts and stores the table rows in index order (one per table). A non-clustered index is a separate structure with pointers to rows (many per table).

Q: Why is WHERE YEAR(date) = 2025 slow? It applies a function to every row, so the index on date cannot be used (non-SARGable). Use a date range.

Q: How do you find and fix a slow query? Look at the actual execution plan and I/O statistics, find scans/lookups/bad estimates, make predicates SARGable, add or adjust indexes, reduce columns/rows early, and re-test.

End-to-End SQL Case Study & Interview Practice

Put everything together. Work through a complete analysis of the retail business (data quality, KPIs, trends, customer segmentation, retention, product and discount analysis) using joins, CTEs and window functions, then practise 25 classic SQL interview questions with model answers.

● SQL class 5 (end-to-end case study)

✦ 2026

Advanced

🕒 180 min

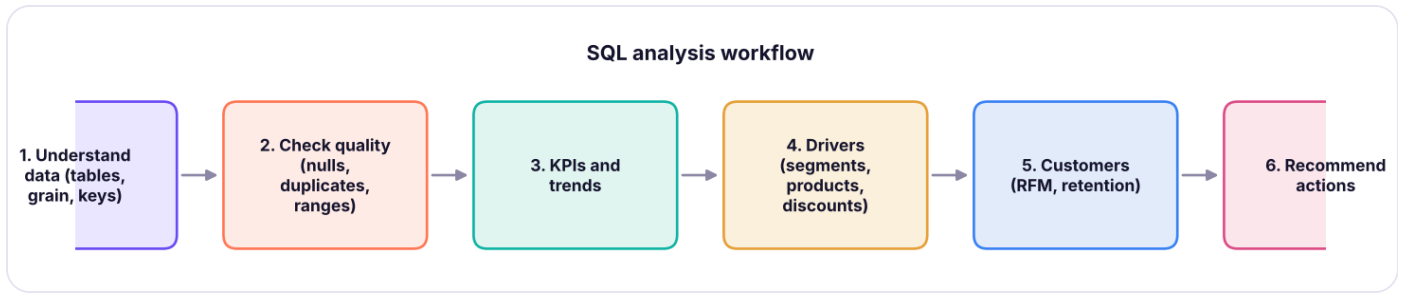
By the end of this lesson you can

- Run a structured SQL analysis: understand the data, check quality, calculate KPIs, find drivers, and recommend actions
- Combine joins, CTEs, aggregation and window functions in multi-step queries
- Build customer segments (RFM), cohort and retention views in SQL
- Answer common SQL interview questions with clear, correct queries
- Explain results in business language

The business problem

CASE STUDY Retail growth review

You are the data analyst at an Indian retailer selling Furniture, Office Supplies and Technology through four regions. Sales grew from 2023 to 2025, but the CFO suspects profit is not keeping pace. Management wants answers to five questions: **(1)** Is the data trustworthy? **(2)** How are sales and profit really trending? **(3)** Where are we losing money and why? **(4)** Who are our best customers and are they staying? **(5)** What should we do next quarter?



A repeatable SQL analysis workflow.

Step 1: understand the data

SQL

```
SELECT 'orders' AS tbl, COUNT(*) AS n, MIN(order_date) AS first_date, MAX(order_date) AS last_date FROM orders
UNION ALL SELECT 'order_items', COUNT(*), NULL, NULL FROM order_items
UNION ALL SELECT 'customers', COUNT(*), NULL, NULL FROM customers
UNION ALL SELECT 'products', COUNT(*), NULL, NULL FROM products;
```

RESULT

tbl	n	first_date	last_date
orders	3200	2023-01-01	2025-12-31
order_items	6323	NULL	NULL
customers	800	NULL	NULL
products	78	NULL	NULL

4 rows × 4 columns

Grain: order_items has one row per product line in an order; orders one row per order; customers one row per customer. KPIs like "orders" must use COUNT(DISTINCT order_id), and "average order value" needs order-level totals (Lesson 10).

Step 2: data-quality checks

Always check before analysing. A compact data-quality report:

SQL

```
SELECT
  (SELECT COUNT(*) FROM order_items WHERE sales IS NULL OR profit IS NULL OR quantity IS
  NULL) AS null_measures,
  (SELECT COUNT(*) FROM order_items WHERE sales ≤ 0 OR quantity ≤ 0)
  AS non_positive,
  (SELECT COUNT(*) FROM order_items WHERE discount < 0 OR discount > 1)
  AS bad_discount,
  (SELECT COUNT(*) FROM orders WHERE ship_date < order_date)
  AS ship_before_order,
  (SELECT COUNT(*) FROM order_items oi LEFT JOIN orders o ON o.order_id = oi.order_id
  WHERE o.order_id IS NULL) AS orphan_items,
  (SELECT COUNT(*) FROM orders o LEFT JOIN customers c ON c.customer_id = o.customer_id
  WHERE c.customer_id IS NULL) AS orders_without_customer,
  (SELECT COUNT(*) - COUNT(DISTINCT item_id) FROM order_items)
  AS duplicate_item_ids;
```

RESULT

null_measures	non_positive	bad_discount	ship_before_order	orphan_items	orders_wi
0	0	0	0	0	0

1 rows × 7 columns

All zeros: the data is clean, so the numbers can be trusted. In real projects every non-zero here becomes a ticket to fix or a documented exclusion. Also check **business-rule** outliers:

SQL

```
SELECT ROUND(MIN(sales), 0) AS min_sales, ROUND(MAX(sales), 0) AS max_sales,
  ROUND(AVG(sales), 0) AS avg_sales,
  MIN(discount) AS min_discount, MAX(discount) AS max_discount,
  MIN(quantity) AS min_qty, MAX(quantity) AS max_qty
FROM order_items;
```

RESULT

min_sales	max_sales	avg_sales	min_discount	max_discount	min_qty	max_qty
42	619,290	31,854	0	0.4	1	7

1 rows × 7 columns

Step 3: KPIs and trends

SQL

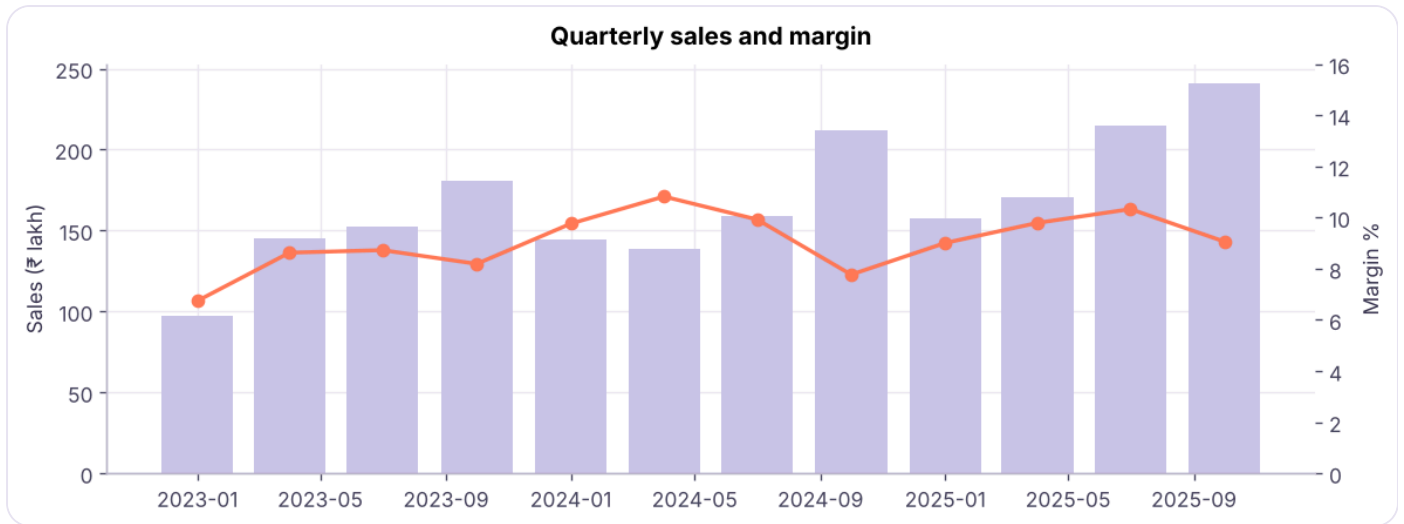
```
WITH yearly AS (  
    SELECT YEAR(o.order_date)           AS yr,  
           COUNT(DISTINCT o.order_id)   AS orders,  
           COUNT(DISTINCT o.customer_id) AS customers,  
           SUM(oi.sales)                 AS sales,  
           SUM(oi.profit)                AS profit  
    FROM orders o JOIN order_items oi ON oi.order_id = o.order_id  
    GROUP BY YEAR(o.order_date)  
)  
SELECT yr, orders, customers,  
       ROUND(sales, 0) AS sales,  
       ROUND(100.0 * (sales - LAG(sales) OVER (ORDER BY yr)) / LAG(sales) OVER (ORDER BY  
yr), 1) AS sales_growth_pct,  
       ROUND(profit, 0) AS profit,  
       ROUND(100.0 * (profit - LAG(profit) OVER (ORDER BY yr)) / LAG(profit) OVER (ORDER BY  
yr), 1) AS profit_growth_pct,  
       ROUND(100.0 * profit / sales, 1) AS margin_pct,  
       ROUND(sales / orders, 0) AS avg_order_value  
FROM yearly  
ORDER BY yr;
```

RESULT

yr	orders	customers	sales	sales_growth_pct	profit	profit_growth_pc
2023	883	540	57,640,217	NULL	4,725,378	NULL
2024	1085	590	65,396,556	13.5	6,135,992	29.9
2025	1232	623	78,376,797	19.8	7,490,793	22.1

3 rows × 9 columns

Finding 1. Sales grew 13.5% in 2024 and 19.8% in 2025, and profit grew even faster (29.9% and 22.1%), so the overall margin edged up from 8.2% to 9.6%. The growth is real, but a 9.6% margin is low compared with the 17–18% earned on lines sold **without** a discount (you will see this in Step 4). The CFO's concern is about the *quality* of profit: let's look at where the money is made and lost.



Quarterly sales and profit margin from the same data. Sales climb steadily and peak each Q4 (festive season); the margin moves in a narrow band below 12%.

Step 4: where do we lose money?

SQL

```
SELECT p.category, p.sub_category,
       COUNT(*)                      AS lines,
       ROUND(SUM(oi.sales), 0)       AS sales,
       ROUND(SUM(oi.profit), 0)      AS profit,
       ROUND(100.0 * SUM(oi.profit) / SUM(oi.sales), 1) AS margin_pct,
       ROUND(AVG(oi.discount) * 100, 1) AS avg_discount_pct
FROM order_items oi JOIN products p ON p.product_id = oi.product_id
GROUP BY p.category, p.sub_category
ORDER BY profit;
```

RESULT

category	sub_category	lines	sales	profit	margin_pct	avg_discount_pct
Furniture	Tables	505	14,046,224	-816,042	-5.8	17.5
Furniture	Bookcases	477	11,274,849	-422,786	-3.7	18.3
Office Supplies	Art	462	245,985	36,526	14.8	7.6
Office Supplies	Binders	494	401,219	61,074	15.2	7.4
Office Supplies	Paper	559	493,232	75,040	15.2	7.2
Furniture	Furnishings	442	1,429,307	78,632	5.5	7.4
Furniture	Chairs	493	8,091,474	342,043	4.2	8.3
Technology	Accessories	445	3,178,930	369,554	11.6	7.5
Office Supplies	Storage	507	2,659,340	397,228	14.9	7.6
Office Supplies	Appliances	492	6,333,529	944,927	14.9	7.5
Technology	Phones	488	31,919,223	3,663,117	11.5	7.1
Technology	Machines	491	42,590,335	5,186,711	12.2	6.7

13 rows × 7 columns (first 12 shown)

Finding 2. Tables and Bookcases lose money, and they also carry the heaviest average discounts. Is discounting the cause? Compare margin by discount level and by whether the line is Furniture:

SQL

```
SELECT CASE WHEN p.category = 'Furniture' THEN 'Furniture' ELSE 'Other' END AS group_name,
       CASE WHEN oi.discount = 0 THEN '0%' WHEN oi.discount ≤ 0.2 THEN '1-20%' ELSE '>20%'
END AS discount_band,
       COUNT(*) AS lines,
       ROUND(SUM(oi.profit), 0) AS profit,
       ROUND(100.0 * SUM(oi.profit) / SUM(oi.sales), 1) AS margin_pct
FROM order_items oi JOIN products p ON p.product_id = oi.product_id
GROUP BY CASE WHEN p.category = 'Furniture' THEN 'Furniture' ELSE 'Other' END,
         CASE WHEN oi.discount = 0 THEN '0%' WHEN oi.discount ≤ 0.2 THEN '1-20%' ELSE
'>20%' END
ORDER BY group_name, discount_band;
```

RESULT

group_name	discount_band	lines	profit	margin_pct
Furniture	0%	850	1,861,761	11.7
Furniture	1-20%	509	-317,393	-3.9
Furniture	>20%	558	-2,362,521	-21.9
Other	0%	2630	19,857,203	18.4
Other	1-20%	1434	1,014,534	2.0
Other	>20%	342	-1,701,419	-18.9

6 rows × 5 columns

Finding 3. In every group, discounts above 20% destroy margin. Furniture is hit hardest because it is discounted more often. What would profit be if lines discounted above 20% had been sold at a 20% discount instead? A rough "what-if":

SQL

```
WITH x AS (
    SELECT oi.sales, oi.profit, oi.discount,
           oi.sales / (1 - oi.discount) AS list_value,
           (oi.sales / (1 - oi.discount)) * 0.8 AS sales_at_20,
           oi.sales - oi.profit AS cost
    FROM order_items oi WHERE oi.discount > 0.2
)
SELECT COUNT(*) AS lines_over_20pct,
       ROUND(SUM(profit), 0) AS actual_profit,
       ROUND(SUM(sales_at_20 - cost), 0) AS profit_if_capped_at_20pct,
       ROUND(SUM(sales_at_20 - cost) - SUM(profit), 0) AS potential_gain
FROM x;
```

RESULT

lines_over_20pct	actual_profit	profit_if_capped_at_20pct	potential_gain
900	-4,063,940	-422,474	3,641,466

1 rows × 4 columns

(This ignores that a lower discount might lose some of those sales. It is an upper bound to size the opportunity, and it must be labelled as such for management.)

Step 5: customers

Top customers and concentration

SQL

```
WITH c AS (  
    SELECT o.customer_id, SUM(oi.sales) AS sales, SUM(oi.profit) AS profit, COUNT(DISTINCT  
o.order_id) AS orders  
    FROM orders o JOIN order_items oi ON oi.order_id = o.order_id  
    GROUP BY o.customer_id  
) , r AS (  
    SELECT *, NTILE(10) OVER (ORDER BY sales DESC) AS decile FROM c  
)  
SELECT decile,  
    COUNT(*) AS customers,  
    ROUND(100.0 * SUM(sales) / SUM(SUM(sales)) OVER (), 1) AS pct_of_sales,  
    ROUND(100.0 * SUM(profit) / SUM(SUM(profit)) OVER (), 1) AS pct_of_profit  
FROM r  
GROUP BY decile  
ORDER BY decile;
```

RESULT

decile	customers	pct_of_sales	pct_of_profit
1	79	28.8	34.1
2	79	18.7	21.0
3	79	14.6	14.3
4	79	11.3	10.4
5	79	9.0	8.8
6	78	6.7	5.3
7	78	5.2	2.5
8	78	3.4	2.2
9	78	1.9	1.1
10	78	0.5	0.3

10 rows × 4 columns

RFM segmentation

RFM scores each customer by **R**ecency (days since last order), **F**requency (number of orders) and **M**onetary value (total sales). The reference date is the day after the last order in the data.

SQL

```
WITH base AS (  
    SELECT o.customer_id,  
           DATEDIFF(day, MAX(o.order_date), '2026-01-01') AS recency_days,  
           COUNT(DISTINCT o.order_id) AS frequency,  
           SUM(oi.sales) AS monetary  
    FROM orders o JOIN order_items oi ON oi.order_id = o.order_id  
    GROUP BY o.customer_id  
) , scored AS (  
    SELECT *,  
           NTILE(4) OVER (ORDER BY recency_days DESC) AS r,           -- 4 = most recent  
           NTILE(4) OVER (ORDER BY frequency) AS f,  
           NTILE(4) OVER (ORDER BY monetary) AS m  
    FROM base  
)  
SELECT customer_id, recency_days, frequency, ROUND(monetary, 0) AS monetary, r, f, m,  
       CASE WHEN r ≥ 3 AND f ≥ 3 AND m ≥ 3 THEN 'Champions'  
            WHEN r ≥ 3 AND f ≥ 2 THEN 'Loyal'  
            WHEN r ≤ 2 AND f ≥ 3 AND m ≥ 3 THEN 'At risk (high value)'  
            WHEN r = 1 THEN 'Lost / hibernating'  
            ELSE 'Potential' END AS segment  
FROM scored  
ORDER BY monetary DESC;
```

RESULT

customer_id	recency_days	frequency	monetary	r	f	m	segment
C0741	441	6	1,204,439	1	4	4	At risk (high value)
C0092	11	8	1,140,540	4	4	4	Champions
C0306	278	7	1,126,867	2	4	4	At risk (high value)
C0781	130	5	1,123,527	2	3	4	At risk (high value)
C0443	92	7	1,113,058	3	4	4	Champions
C0720	12	6	1,100,213	4	4	4	Champions
C0493	514	6	1,004,382	1	4	4	At risk (high value)
C0028	266	5	1,000,189	2	3	4	At risk (high value)
C0630	331	5	1,000,175	1	3	4	At risk (high value)
C0262	70	6	952,738	3	4	4	Champions

785 rows × 8 columns (first 10 shown)

SQL

```
WITH base AS (  
    SELECT o.customer_id, DATEDIFF(day, MAX(o.order_date), '2026-01-01') AS recency_days,  
           COUNT(DISTINCT o.order_id) AS frequency, SUM(oi.sales) AS monetary  
    FROM orders o JOIN order_items oi ON oi.order_id = o.order_id GROUP BY o.customer_id  
) , scored AS (  
    SELECT *, NTILE(4) OVER (ORDER BY recency_days DESC) AS r, NTILE(4) OVER (ORDER BY  
frequency) AS f, NTILE(4) OVER (ORDER BY monetary) AS m FROM base  
) , seg AS (  
    SELECT *, CASE WHEN r ≥ 3 AND f ≥ 3 AND m ≥ 3 THEN 'Champions' WHEN r ≥ 3 AND f ≥  
2 THEN 'Loyal'  
                WHEN r ≤ 2 AND f ≥ 3 AND m ≥ 3 THEN 'At risk (high value)' WHEN r = 1  
THEN 'Lost / hibernating' ELSE 'Potential' END AS segment FROM scored  
)  
SELECT segment, COUNT(*) AS customers, ROUND(SUM(monetary), 0) AS sales,  
       ROUND(100.0 * SUM(monetary) / SUM(SUM(monetary)) OVER (), 1) AS pct_of_sales  
FROM seg GROUP BY segment ORDER BY sales DESC;
```

RESULT

segment	customers	sales	pct_of_sales
Champions	178	80,496,072	40.0
At risk (high value)	92	42,616,303	21.2
Potential	198	29,644,898	14.7
Lost / hibernating	169	24,817,833	12.3
Loyal	148	23,838,464	11.8

5 rows × 4 columns

The "At risk (high value)" group is the first win: high-spending customers who have gone quiet deserve a personal outreach before they churn. The same RFM idea is implemented with pandas in the ML track.

Retention and cohorts

A **cohort** groups customers by the year of their first order. **Retention** shows how many come back in later years.

SQL

```
WITH first_order AS (  
    SELECT customer_id, YEAR(MIN(order_date)) AS cohort_year FROM orders GROUP BY  
    customer_id  
) , activity AS (  
    SELECT DISTINCT customer_id, YEAR(order_date) AS active_year FROM orders  
)  
SELECT f.cohort_year,  
       COUNT(DISTINCT f.customer_id) AS  
cohort_size,  
       COUNT(DISTINCT CASE WHEN a.active_year = f.cohort_year THEN a.customer_id END)  
AS y0,  
       COUNT(DISTINCT CASE WHEN a.active_year = f.cohort_year + 1 THEN a.customer_id END)  
AS y1,  
       COUNT(DISTINCT CASE WHEN a.active_year = f.cohort_year + 2 THEN a.customer_id END)  
AS y2  
FROM first_order f  
JOIN activity a ON a.customer_id = f.customer_id  
GROUP BY f.cohort_year  
ORDER BY f.cohort_year;
```

RESULT

cohort_year	cohort_size	y0	y1	y2
2023	540	540	397	415
2024	193	193	156	0
2025	52	52	0	0

3 rows × 5 columns

Divide y1 by cohort_size to get year-1 retention. About 74% of the 2023 cohort (397 of 540) ordered again in 2024, and some who skipped a year returned in 2025: this is a business with sticky customers, so reducing discounts is likely safer than the CFO fears.

Step 6: recommendations

KEY TAKEAWAYS**Findings and actions**

1. **Sales and profit are growing, but the margin is only 9.6%**, far below the 17–18% earned on undiscounted lines.
2. **Technology produces nearly all the profit; Furniture is net loss-making** because Tables and Bookcases are sold with heavy discounts.
3. **Discounts above 20% lose money in every category.** Cap discounts at 20% (the what-if sizes the upper-bound upside at about ₹36 lakh over three years; the real gain will be lower if some customers buy less), test in two regions with an A/B design.
4. **Customer base is sticky** (strong year-on-year retention), so price discipline is lower-risk than assumed.
5. **Re-engage "At risk (high value)" customers** with a targeted offer instead of blanket discounts.
6. Build a **monthly SQL view/dashboard** with these KPIs, with data-quality checks as a first step.

SQL interview practice: 25 questions with answers

Run these on the retail database. Each answer is a runnable query.

INTERVIEW CORNER**Q1. Second-highest sales line.****SQL**

```
SELECT MAX(sales) AS second_highest FROM order_items WHERE sales < (SELECT MAX(sales) FROM order_items);
```

RESULT

second_highest

551,530

1 rows × 1 columns

INTERVIEW CORNER

Q2. Customers with at least 8 orders.

SQL

```
SELECT customer_id, COUNT(*) AS orders FROM orders GROUP BY customer_id HAVING  
COUNT(*) ≥ 8 ORDER BY orders DESC;
```

RESULT

customer_id	orders
C0219	10
C0396	10
C0291	9
C0664	9
C0110	9
C0384	9
C0596	9
C0419	9
C0271	9
C0527	9
C0719	9
C0069	9

41 rows × 2 columns (first 12 shown)

INTERVIEW CORNER

Q3. Duplicate customer names (possible duplicate records).

SQL

```
SELECT customer_name, COUNT(*) AS n FROM customers GROUP BY customer_name HAVING  
COUNT(*) > 1 ORDER BY n DESC;
```

RESULT

customer_name	n
Sneha Joshi	6
Ananya Gupta	5
Rahul Shah	5
Asha Rao	5
Ananya Iyer	5
Saanvi Khan	5
Rohan Gupta	5
Reyansh Shah	4
Simran Khan	4
Vikram Kulkarni	4
Arjun Sharma	4
Nikhil Das	4

238 rows × 2 columns (first 12 shown)

INTERVIEW CORNER

Q4. Highest-selling product in each category.

SQL

```
WITH t AS (SELECT p.category, p.product_name, SUM(oi.sales) AS sales FROM order_items
oi JOIN products p ON p.product_id = oi.product_id GROUP BY p.category,
p.product_name),
r AS (SELECT *, ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS
rn FROM t)
SELECT category, product_name, ROUND(sales, 0) AS sales FROM r WHERE rn = 1;
```

RESULT

category	product_name	sales
Furniture	Crest Table 572	3,122,680
Technology	Orbit Copier 746	16,026,341
Office Supplies	Crest Appliance 402	1,909,628

3 rows × 3 columns

INTERVIEW CORNER

Q5. Month-over-month growth for 2025.

SQL

```
WITH m AS (SELECT MONTH(order_date) AS mth, SUM(sales) AS sales FROM superstore WHERE  
YEAR(order_date) = 2025 GROUP BY MONTH(order_date))  
SELECT mth, ROUND(sales, 0) AS sales, ROUND(100.0 * (sales - LAG(sales) OVER (ORDER BY  
mth)) / LAG(sales) OVER (ORDER BY mth), 1) AS mom_pct FROM m ORDER BY mth;
```

RESULT

mth	sales	mom_pct
1	5,357,761	NULL
2	3,879,338	-27.6
3	6,512,645	67.9
4	6,699,805	2.9
5	6,421,274	-4.2
6	3,968,572	-38.2
7	7,190,087	81.2
8	5,802,184	-19.3
9	8,468,060	45.9
10	6,140,290	-27.5
11	10,284,516	67.5
12	7,652,267	-25.6

12 rows × 3 columns

INTERVIEW CORNER

Q6. Running total of sales by month in 2025.

SQL

```
WITH m AS (SELECT MONTH(order_date) AS mth, SUM(sales) AS sales FROM superstore WHERE  
YEAR(order_date) = 2025 GROUP BY MONTH(order_date))  
SELECT mth, ROUND(sales, 0) AS sales, ROUND(SUM(sales) OVER (ORDER BY mth ROWS BETWEEN  
UNBOUNDED PRECEDING AND CURRENT ROW), 0) AS running_total FROM m ORDER BY mth;
```

RESULT

mth	sales	running_total
1	5,357,761	5,357,761
2	3,879,338	9,237,099
3	6,512,645	15,749,743
4	6,699,805	22,449,548
5	6,421,274	28,870,822
6	3,968,572	32,839,394
7	7,190,087	40,029,481
8	5,802,184	45,831,665
9	8,468,060	54,299,725
10	6,140,290	60,440,015
11	10,284,516	70,724,530
12	7,652,267	78,376,797

12 rows × 3 columns

INTERVIEW CORNER**Q7. Customers who ordered in every year (2023, 2024 and 2025).****SQL**

```
SELECT COUNT(*) AS customers_all_three_years
FROM (SELECT customer_id FROM orders GROUP BY customer_id HAVING COUNT(DISTINCT
YEAR(order_date)) = 3) AS t;
```

RESULT**customers_all_three_years**

311

1 rows × 1 columns

INTERVIEW CORNER**Q8. Employees earning more than their department's average (HR data).****SQL**

```
SELECT COUNT(*) AS above_dept_avg
FROM (SELECT emp_id, monthly_income, AVG(monthly_income) OVER (PARTITION BY
department) AS avg_income FROM employees) AS t
WHERE monthly_income > avg_income;
```

RESULT**above_dept_avg**

638

1 rows × 1 columns

INTERVIEW CORNER**Q9. Delete duplicates keeping the lowest id (pattern).****SQL**

```
WITH d AS (SELECT *, ROW_NUMBER() OVER (PARTITION BY customer_name, city ORDER BY  
customer_id) AS rn FROM dbo.customers)  
DELETE FROM d WHERE rn > 1;
```

INTERVIEW CORNER**Q10. Customers with no orders in the last 6 months of 2025 but orders before that (churn candidates).****SQL**

```
SELECT COUNT(*) AS churn_candidates  
FROM (SELECT customer_id FROM orders WHERE order_date < '2025-07-01' GROUP BY  
customer_id  
EXCEPT  
SELECT customer_id FROM orders WHERE order_date ≥ '2025-07-01') AS t;
```

RESULT**churn_candidates**

310

1 rows × 1 columns

INTERVIEW CORNER**Q11. Percentage of orders shipped late (more than 5 days).****SQL**

```
SELECT ROUND(100.0 * SUM(CASE WHEN DATEDIFF(day, order_date, ship_date) > 5 THEN 1  
ELSE 0 END) / COUNT(*), 1) AS pct_late FROM orders;
```

RESULT

pct_late
31.3

1 rows × 1 columns

INTERVIEW CORNER**Q12. Median order-line sales.****SQL**

```
SELECT DISTINCT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY sales) OVER () AS  
median_sales FROM order_items;
```

RESULT

median_sales
8,430

1 rows × 1 columns

INTERVIEW CORNER

Q13. Categories where more than 30% of lines are loss-making.

SQL

```
SELECT p.category, ROUND(100.0 * AVG(CASE WHEN oi.profit < 0 THEN 1.0 ELSE 0 END), 1)
AS pct_loss_lines
FROM order_items oi JOIN products p ON p.product_id = oi.product_id
GROUP BY p.category HAVING AVG(CASE WHEN oi.profit < 0 THEN 1.0 ELSE 0 END) > 0.3;
```

RESULT

category	pct_loss_lines
Furniture	48.9

1 rows × 2 columns

INTERVIEW CORNER

Q14. Cumulative percentage of sales by customer (Pareto): how many customers make 50% of sales?

SQL

```
WITH c AS (SELECT customer_id, SUM(sales) AS s FROM order_items oi JOIN orders o ON
o.order_id = oi.order_id GROUP BY customer_id),
p AS (SELECT s, SUM(s) OVER (ORDER BY s DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) * 1.0 / SUM(s) OVER () AS cum FROM c)
SELECT COUNT(*) AS customers_for_half_of_sales, (SELECT COUNT(*) FROM c) AS
total_customers FROM p WHERE cum ≤ 0.5;
```

RESULT

customers_for_half_of_sales	total_customers
170	785

1 rows × 2 columns

INTERVIEW CORNER**Q15. Swap two values in a column (pattern).****SQL**

```
UPDATE dbo.staff SET gender = CASE gender WHEN 'M' THEN 'F' WHEN 'F' THEN 'M' ELSE
gender END;
```

INTERVIEW CORNER**Q16. Find consecutive order days (streaks) for a customer: see Lesson 12 (gaps and islands).****INTERVIEW CORNER****Q17. Average order value per segment.****SQL**

```
SELECT segment, ROUND(AVG(t.total), 0) AS aov
FROM (SELECT c.segment, o.order_id, SUM(oi.sales) AS total FROM orders o JOIN
customers c ON c.customer_id = o.customer_id JOIN order_items oi ON oi.order_id =
o.order_id GROUP BY c.segment, o.order_id) AS t
GROUP BY segment;
```

RESULT

segment	aov
Consumer	63,451
Corporate	61,899
Home Office	63,237

3 rows × 2 columns

INTERVIEW CORNER**Q18. Products priced above the average of their sub-category.****SQL**

```
SELECT COUNT(*) AS products_above_subcat_avg
FROM (SELECT product_id, list_price, AVG(list_price) OVER (PARTITION BY sub_category)
      AS avg_price FROM products) AS t WHERE list_price > avg_price;
```

RESULT

products_above_subcat_avg

36

1 rows × 1 columns

INTERVIEW CORNER**Q19. Rows in A not in B without NOT IN trap: use NOT EXISTS (Lesson 10).**

INTERVIEW CORNER

Q20. Pivot category sales into columns by year.

SQL

```
SELECT category,
       ROUND(SUM(CASE WHEN YEAR(order_date) = 2023 THEN sales ELSE 0 END), 0) AS
y2023,
       ROUND(SUM(CASE WHEN YEAR(order_date) = 2024 THEN sales ELSE 0 END), 0) AS
y2024,
       ROUND(SUM(CASE WHEN YEAR(order_date) = 2025 THEN sales ELSE 0 END), 0) AS y2025
FROM superstore GROUP BY category ORDER BY category;
```

RESULT

category	y2023	y2024	y2025
Furniture	9,952,975	12,128,633	12,760,246
Office Supplies	2,611,874	3,550,109	3,971,321
Technology	45,075,369	49,717,814	61,645,230

3 rows × 4 columns

INTERVIEW CORNER

Q21. First and last order date and the number of days between for each customer (top 3 longest tenures).

SQL

```
SELECT TOP 3 customer_id, MIN(order_date) AS first_order, MAX(order_date) AS
last_order, DATEDIFF(day, MIN(order_date), MAX(order_date)) AS days_active
FROM orders GROUP BY customer_id ORDER BY days_active DESC;
```

RESULT

customer_id	first_order	last_order	days_active
C0079	2023-01-08	2025-12-27	1084
C0192	2023-01-24	2025-12-31	1072
C0070	2023-01-09	2025-12-13	1069

3 rows × 4 columns

INTERVIEW CORNER**Q22. Which month has the highest sales in each year?****SQL**

```
WITH m AS (SELECT YEAR(order_date) AS yr, MONTH(order_date) AS mth, SUM(sales) AS sales FROM superstore GROUP BY YEAR(order_date), MONTH(order_date)),  
      r AS (SELECT *, RANK() OVER (PARTITION BY yr ORDER BY sales DESC) AS rnk FROM m)  
SELECT yr, mth, ROUND(sales, 0) AS sales FROM r WHERE rnk = 1 ORDER BY yr;
```

RESULT

yr	mth	sales
2023	12	7,152,348
2024	11	7,284,664
2025	11	10,284,516

3 rows × 3 columns

INTERVIEW CORNER**Q23. How do you find the number of rows in each table of a database?****SQL**

```
SELECT t.name AS table_name, SUM(p.rows) AS row_count  
FROM sys.tables t JOIN sys.partitions p ON p.object_id = t.object_id AND p.index_id IN  
(0, 1)  
GROUP BY t.name ORDER BY row_count DESC;
```

INTERVIEW CORNER**Q24. Explain the difference between RANK() and ROW_NUMBER() and when each is right: Lesson 11.**

INTERVIEW CORNER

Q25. How would you explain the result of a query to a non-technical manager?

State the answer first, in one sentence with numbers ("Furniture loses ₹8.2 lakh a year, almost entirely on heavily discounted tables and bookcases"), then give the 1–2 facts that support it, the assumption or limitation, and the action you recommend.

Summary

KEY TAKEAWAYS

- A good analysis follows a workflow: understand the grain, check data quality, calculate KPIs, find drivers, study customers, recommend actions.
- Always check data quality first; label what-if estimates as estimates.
- Combine CTEs, joins and window functions to build RFM segments, concentration (Pareto), cohorts and retention.
- In interviews, state your assumptions, clarify the grain, write clean, formatted SQL, and explain the result in business language.

Quick quiz

6 questions · tap an answer to check it

1. Why must "number of orders" use `COUNT(DISTINCT order_id)` when joining orders to order_items?

- A DISTINCT is faster
- B Each order appears once per item, so `COUNT(*)` would count lines**
- C order_id is a text column
- D It avoids NULLs

Answer: B. The join repeats the order for each of its items.

2. In RFM, a customer with recency 400 days, frequency 1 and low monetary value is probably:

- A** A champion
- B** Lost or hibernating
- C** A loyal customer
- D** A new high-value customer

Answer: B. Long time since the last order, few orders and low spend.

3. A cohort table shows 2023 cohort size 500, year-1 active 350. Year-1 retention is:

- A** 35%
- B** 70%
- C** 50%
- D** 150%

Answer: B. $350 \div 500 = 70\%$.

4. Which should you do before reporting KPIs?

- A** Build a dashboard
- B** Check data quality: nulls, duplicates, orphans, ranges
- C** Email the CFO
- D** Add indexes

Answer: B. Untrusted data produces untrusted numbers.

5. The what-if "profit if discounts were capped at 20%" is best described as:

A A forecast

B A guarantee

C An upper-bound estimate that ignores possible lost sales

D A data error

Answer: C. It assumes demand would not change, so it overstates the benefit.

6. NTILE(10) over customers ordered by sales gives:

A Ten customers

B Ten equal-sized groups (deciles) of customers

C The top 10% only

D The median

Answer: B. NTILE splits rows into n buckets.

EXERCISE**Capstone: your own analysis**

Using only SQL on the retail database, prepare a one-page analysis answering: (1) Which three cities have the highest profit and which three the lowest? (2) What is the average days to ship by ship mode, and does a slower mode save money (profit per line)? (3) Which product's sales fell the most from 2024 to 2025? (4) Propose two actions with the numbers that support them.

Show solution (worked answer)

Hints: (1) GROUP BY city with ORDER BY profit ascending and descending using TOP 3 in two queries or RANK() in a CTE. (2) Join orders to order_items, AVG(DATEDIFF(day, order_date, ship_date)) and SUM(profit)/COUNT(*) by ship_mode. (3) Pivot product sales by year with conditional aggregation and compute the difference. (4) Example actions: cap Furniture discounts at 20% in two pilot regions; re-engage at-risk high-value customers. Present each action with a baseline number, an expected impact (labelled as an estimate) and a way to measure success.