

Note 1: Introduction to SQL & Databases

Overview

This note covers the fundamentals of SQL, databases, and the foundational concepts you need to understand before writing SQL queries.

Theory

What is SQL?

SQL (Structured Query Language) is a **declarative programming language** designed specifically for managing and manipulating relational databases. Unlike procedural languages where you specify *how* to do something, SQL lets you specify *what* you want, and the database engine figures out *how* to get it.

Key Characteristics of SQL:

- **Declarative:** You describe the result you want, not the steps to get it
- **Set-based:** Operates on sets of rows, not one row at a time
- **Standardized:** ANSI/ISO standard (though vendors add extensions)
- **Non-procedural:** No loops or conditional branching in basic SQL

 **Pronunciation:** "SQL" (S-Q-L) is the ANSI standard pronunciation. "Sequel" comes from IBM's original SEQUEL language. Both are widely accepted!

The Evolution of SQL

1970: E.F. Codd publishes relational model theory at IBM
 1974: IBM develops SEQUEL (Structured English Query Language)
 1979: Oracle releases first commercial SQL database
 1986: SQL becomes ANSI standard
 1987: SQL becomes ISO standard
 Today: SQL is the universal language for relational databases

Why Data is Everywhere (The Data Explosion)

Every digital interaction generates data:

- **Personal data:** Names, phone numbers, emails, preferences
- **Mobile devices:** Apps generate 2.5 quintillion bytes of data daily
- **IoT devices:** Smart cars, wearables, home automation
- **Financial systems:** Real-time transactions, fraud detection
- **E-commerce:** User behavior, purchases, recommendations
- **Healthcare:** Patient records, diagnostics, research data

 **Fact:** 90% of the world's data was created in the last 2 years. This is why database skills are essential!

Why Use Databases Instead of Files?

Aspect	Files (Excel, CSV, Text)	Databases
Data Size	Slows/crashes with large data	Handles billions of records
Concurrent Access	File locking issues	Multiple users simultaneously
Data Integrity	No validation rules	Constraints ensure valid data
Security	Basic file permissions	Row-level, column-level security
Relationships	Manual VLOOKUP	Built-in foreign keys

Aspect	Files (Excel, CSV, Text)	Databases
Query Speed	Sequential scanning	Indexed searching (milliseconds)
ACID Compliance	None	Full transaction support
Backup/Recovery	Manual	Automated, point-in-time recovery

What is ACID?

ACID properties ensure reliable database transactions:

- **Atomicity:** Transaction is all-or-nothing
- **Consistency:** Data remains valid after transaction
- **Isolation:** Concurrent transactions don't interfere
- **Durability:** Committed data survives system failures

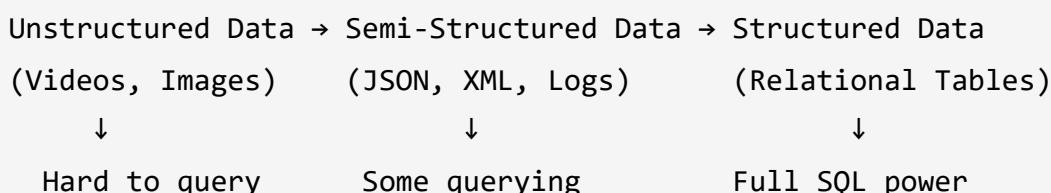
Database Concepts

What is a Database?

A **database** is a structured, organized collection of data stored electronically. Think of it as a highly organized digital filing cabinet where:

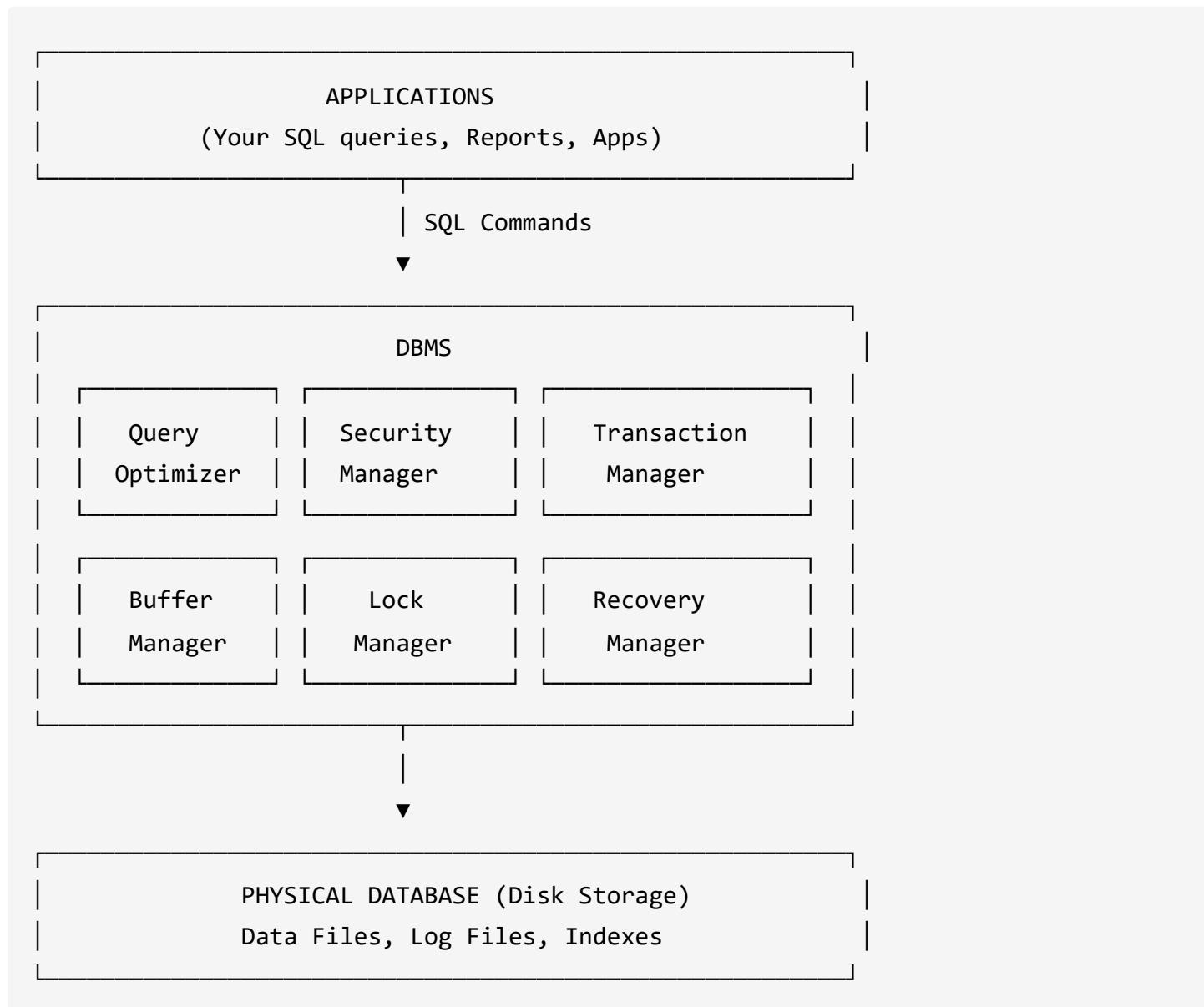
- Data is stored in a structured format
- Relationships between data are maintained
- Data can be quickly accessed, managed, and updated

Types of Data Organization:



Database Management System (DBMS)

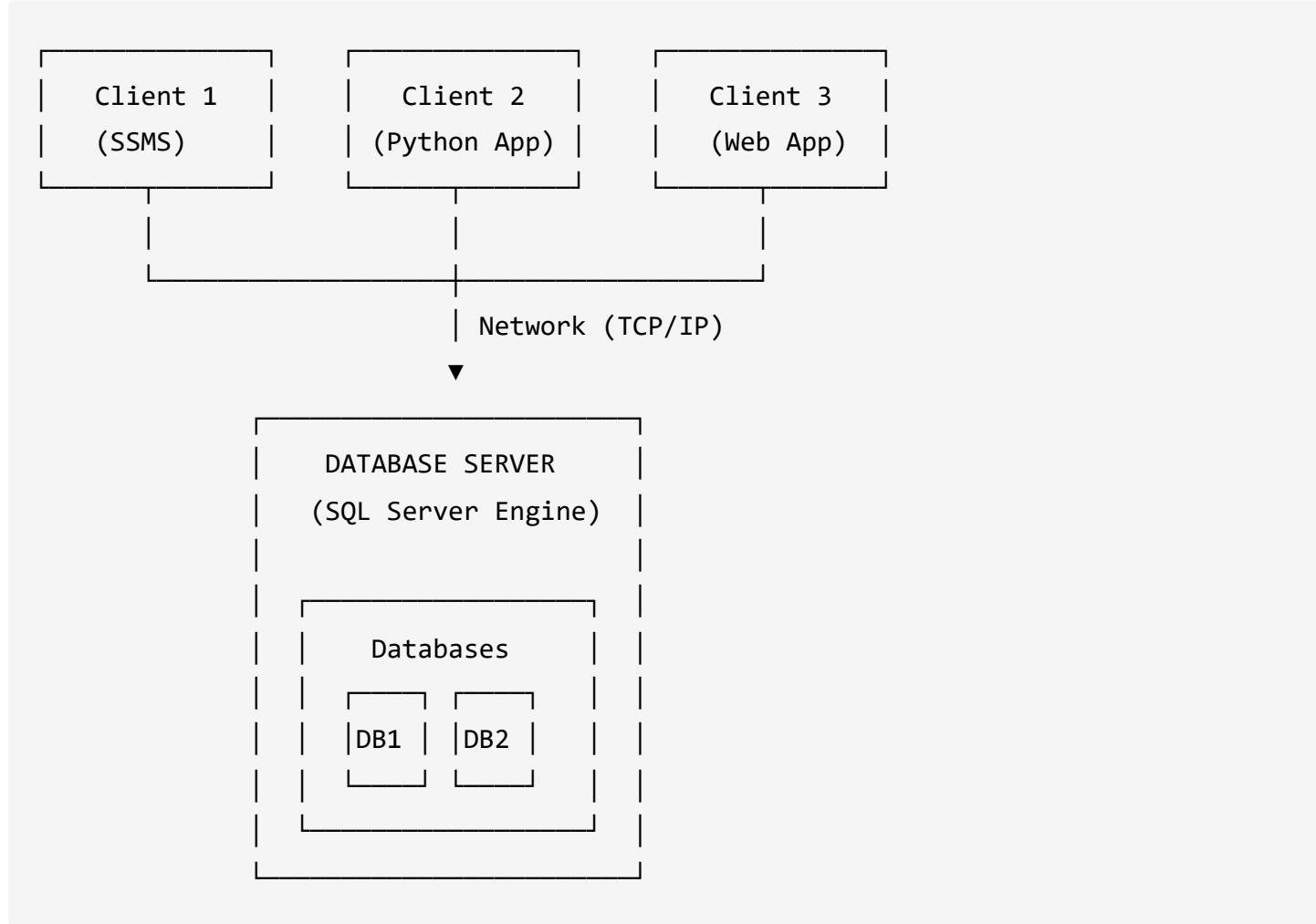
The **DBMS** is the software layer between users and the physical database:



Popular DBMS Options:

DBMS	Type	Best For
SQL Server	Commercial	Enterprise, Windows environments
PostgreSQL	Open Source	Complex queries, GIS data
MySQL	Open Source	Web applications, startups
Oracle	Commercial	Large enterprises, mission-critical
SQLite	Embedded	Mobile apps, small applications

Client-Server Architecture



Types of Databases

1. Relational Database (RDBMS)

The most common type for structured business data:

- Organizes data in **tables** (rows and columns)
- Tables have **relationships** via foreign keys
- Uses **SQL** as query language
- Enforces **ACID** properties

When to Use RDBMS:

- Structured data with clear relationships
- Need for complex queries and joins

- Transaction integrity is critical
- Data consistency is paramount

2. NoSQL Databases

Designed for specific use cases where RDBMS falls short:

Type	How It Works	Use Case	Example
Key-Value	Simple key→value pairs	Caching, sessions	Redis, DynamoDB
Document	JSON-like documents	Content management, catalogs	MongoDB, CouchDB
Column-Family	Columns grouped together	Analytics, time-series	Cassandra, HBase
Graph	Nodes + edges (relationships)	Social networks, recommendations	Neo4j, Amazon Neptune

SQL vs NoSQL Decision Matrix:

Choose SQL when:

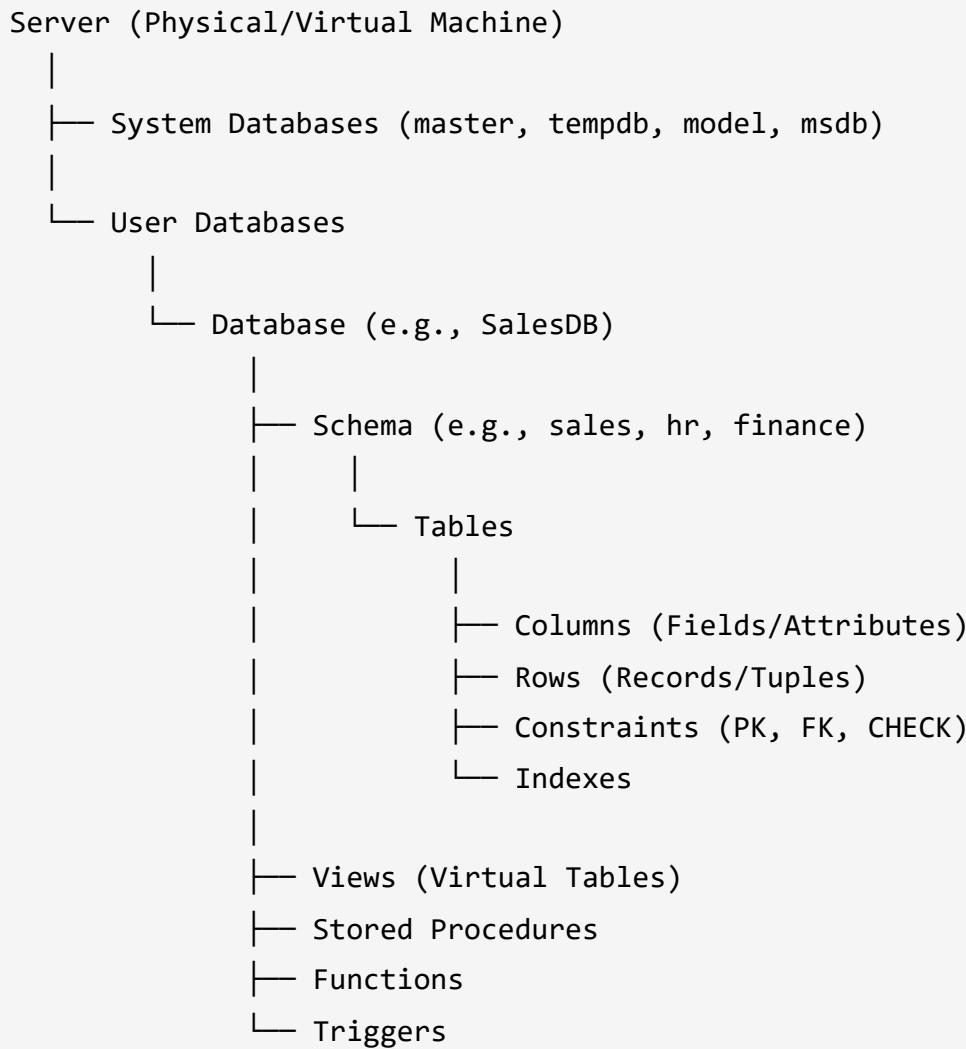
- ✓ Data is structured
- ✓ Complex queries needed
- ✓ ACID compliance required
- ✓ Data relationships exist
- ✓ Reporting/analytics focus

Choose NoSQL when:

- ✓ Data is unstructured/semi-structured
- ✓ Simple key-based lookups
- ✓ Massive scale needed
- ✓ Schema changes frequently
- ✓ High write throughput needed



Database Hierarchy



Understanding Table Structure

Table: Customers

CustomerID	FirstName	LastName	Email	Score	← Columns (Attributes)
(PK, INT)	(VARCHAR)	(VARCHAR)	(VARCHAR)	(INT)	
1	John	Smith	john@email.com	850	← Row (Record/Tuple)
2	Sarah	Jones	sarah@email.com	720	
3	Mike	Brown	mike@email.com	NULL	

↑ ↑

Primary Key
(Unique identifier)

Cell
(Single value)

Key Database Objects

Object	Purpose	Example
Table	Store actual data	Customers , Orders
View	Virtual table from query	v_ActiveCustomers
Index	Speed up data retrieval	idx_CustomerEmail
Stored Procedure	Reusable SQL program	sp_GetCustomerOrders
Function	Return calculated values	fn_CalculateDiscount
Trigger	Auto-execute on events	tr_AuditChanges
Constraint	Enforce data rules	PK_Customers , FK_Orders_Customer



Data Types

Numeric Data Types

Data Type	Description	Range/Size	Example
TINYINT	Very small integers	0 to 255	Age: 25
SMALLINT	Small integers	-32,768 to 32,767	Year: 2024
INT	Standard integers	±2.1 billion	OrderID: 1000000
BIGINT	Large integers	±9.2 quintillion	TransactionID
DECIMAL(p,s)	Exact numeric	User-defined precision	Price: 99.99
FLOAT	Approximate numeric	15 digits precision	Scientific data

Character Data Types

Data Type	Description	Storage	Use Case
CHAR(n)	Fixed-length	Always n bytes	Country codes: 'US'
VARCHAR(n)	Variable-length	Actual length + 2 bytes	Names, descriptions
NVARCHAR(n)	Unicode variable	Actual length × 2 + 2	International text
TEXT	Large text	Up to 2GB	Articles, documents

Date/Time Data Types

Data Type	Description	Format	Example
DATE	Date only	YYYY-MM-DD	'2024-01-15'
TIME	Time only	HH:MM:SS	'14:30:00'

Data Type	Description	Format	Example
DATETIME	Date and time	YYYY-MM-DD HH:MM:SS	'2024-01-15 14:30:00'
DATETIME2	High precision	100 nanoseconds	Audit timestamps

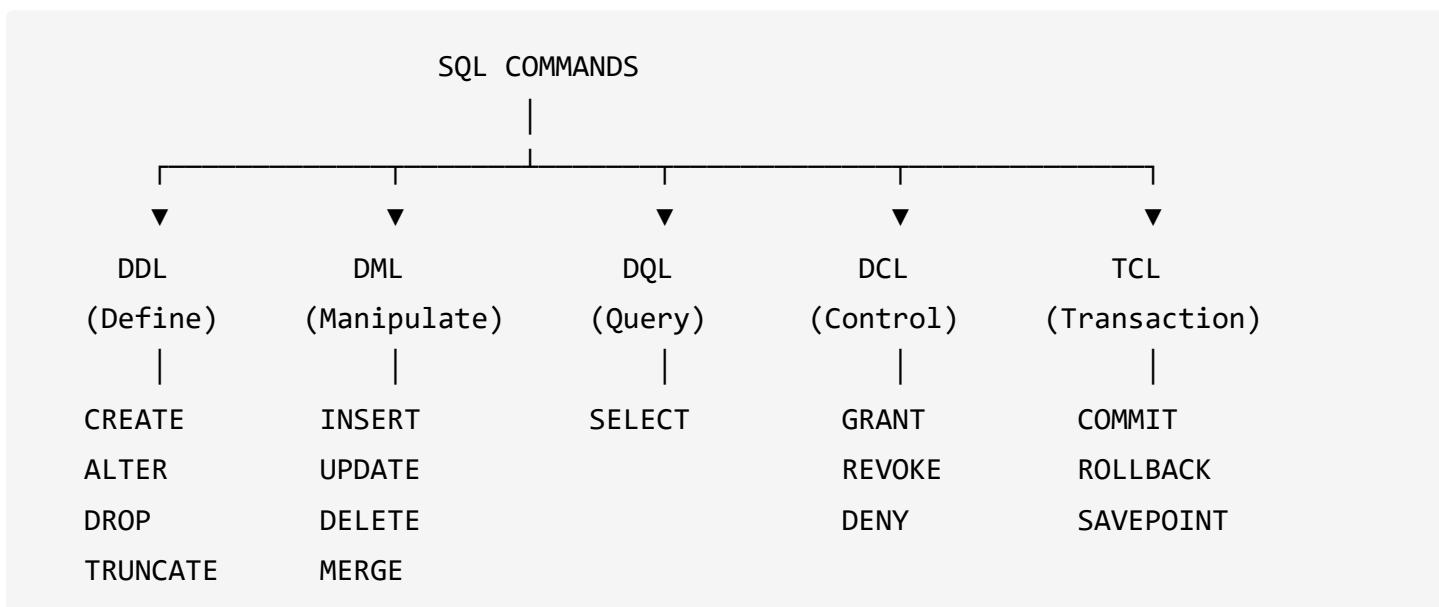
Other Important Types

Data Type	Description	Use Case
BIT	Boolean (0/1)	Flags: IsActive
UNIQUEIDENTIFIER	GUID	Distributed systems
BINARY/VARBINARY	Binary data	Files, images
XML	XML documents	Configuration data



SQL Command Categories

Complete SQL Command Classification



1. DDL (Data Definition Language)

Purpose: Define and modify database structure

Command	Description	Example
CREATE	Create new objects	CREATE TABLE Customers (...)
ALTER	Modify existing objects	ALTER TABLE Customers ADD Email VARCHAR(100)
DROP	Delete objects permanently	DROP TABLE OldCustomers
TRUNCATE	Remove all rows (fast)	TRUNCATE TABLE TempData

2. DML (Data Manipulation Language)

Purpose: Manipulate data within tables

Command	Description	Example
INSERT	Add new rows	INSERT INTO Customers VALUES (...)
UPDATE	Modify existing rows	UPDATE Customers SET Score = 100 WHERE ...
DELETE	Remove specific rows	DELETE FROM Customers WHERE CustomerID = 5
MERGE	Insert or update (upsert)	MERGE INTO Target USING Source ...

3. DQL (Data Query Language)

Purpose: Retrieve data from database

Command	Description	Example
SELECT	Query and retrieve data	SELECT * FROM Customers WHERE Country = 'USA'

4. DCL (Data Control Language)

Purpose: Control access and permissions

Command	Description	Example
GRANT	Give permissions	GRANT SELECT ON Customers TO AnalystRole
REVOKE	Remove permissions	REVOKE INSERT ON Customers FROM UserX
DENY	Explicitly deny access	DENY DELETE ON Customers TO PublicRole

5. TCL (Transaction Control Language)

Purpose: Manage transactions

Command	Description	Example
COMMIT	Save transaction permanently	COMMIT TRANSACTION
ROLLBACK	Undo transaction	ROLLBACK TRANSACTION
SAVEPOINT	Create checkpoint	SAVE TRANSACTION SavePoint1

🎯 Interview Questions

Q1: What is SQL and why is it important?

Answer: SQL (Structured Query Language) is a declarative programming language used to manage and manipulate relational databases. It's important because:

- It's the **universal standard** for relational databases
- It allows **efficient data retrieval** from large datasets
- It provides **data integrity** through constraints and transactions
- It's **vendor-neutral** (works across SQL Server, MySQL, PostgreSQL, etc.)

Q2: What is the difference between SQL and MySQL?

Answer:

- **SQL** is a *language* (Structured Query Language) - the standard for querying databases
- **MySQL** is a *DBMS* (Database Management System) - software that implements SQL

Think of it like: SQL is like English (the language), MySQL is like a book publisher (the platform that uses the language).

Q3: Explain the difference between CHAR and VARCHAR.

Answer:

Aspect	CHAR(10)	VARCHAR(10)
Storage	Always 10 bytes	Actual length + 2 bytes
Padding	Pads with spaces	No padding
Performance	Slightly faster (fixed)	Slightly slower (variable)
Use Case	Fixed-length codes	Variable-length text

Example: CHAR(10) storing 'ABC' = 'ABC ' (7 spaces added)

Q4: What is ACID in databases?

Answer: ACID is a set of properties ensuring reliable database transactions:

- **Atomicity:** Transaction is "all or nothing" - if any part fails, entire transaction rolls back
- **Consistency:** Database moves from one valid state to another valid state
- **Isolation:** Concurrent transactions don't interfere with each other
- **Durability:** Once committed, data survives system failures

Example: Bank transfer - debit from Account A and credit to Account B must both succeed or both fail.

Q5: What is the difference between DELETE, TRUNCATE, and DROP?

Answer:

Aspect	DELETE	TRUNCATE	DROP
Type	DML	DDL	DDL
What it removes	Specific rows	All rows	Entire table
WHERE clause	Yes	No	No
Rollback	Yes	No (usually)	No
Triggers	Fires triggers	No triggers	No triggers
Speed	Slower	Faster	Fastest
Identity reset	No	Yes	N/A

Q6: What is a Primary Key? What are its properties?

Answer: A Primary Key is a column (or combination of columns) that uniquely identifies each row in a table.

Properties:

1. **Unique:** No duplicate values allowed
2. **NOT NULL:** Cannot contain NULL values
3. **Immutable:** Should not change once assigned
4. **Single per table:** Only one primary key per table
5. **Can be composite:** Multiple columns can form a primary key

Q7: What is the difference between Primary Key and Unique Key?

Answer:

Aspect	Primary Key	Unique Key
NULL values	Not allowed	Allows one NULL
Per table	Only one	Multiple allowed
Purpose	Row identifier	Prevent duplicates
Clustered index	Created by default	Non-clustered by default

Q8: What is a Foreign Key?

Answer: A Foreign Key is a column that creates a relationship between two tables by referencing the Primary Key of another table.

```
-- Orders table has Foreign Key referencing Customers
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID),
    OrderDate DATE
);
```

Purpose:

- Enforces **referential integrity**
- Prevents orphan records
- Defines relationships between tables

🔑 Key Takeaways

1. **SQL** is a declarative language for relational databases
2. **DBMS** manages databases (SQL Server, MySQL, PostgreSQL)
3. **ACID** properties ensure transaction reliability
4. **5 command types:** DDL, DML, DQL, DCL, TCL
5. **Primary Keys** uniquely identify rows (no duplicates, no NULLs)
6. **Foreign Keys** create relationships between tables
7. Choose correct **data types** for storage efficiency

Best Practices

- Always understand your database structure before querying
- Use meaningful names for tables and columns
- Each table should have a primary key
- Choose appropriate data types for your data
- Document your database schema

Note 2: Environment Setup & SQL Server Installation

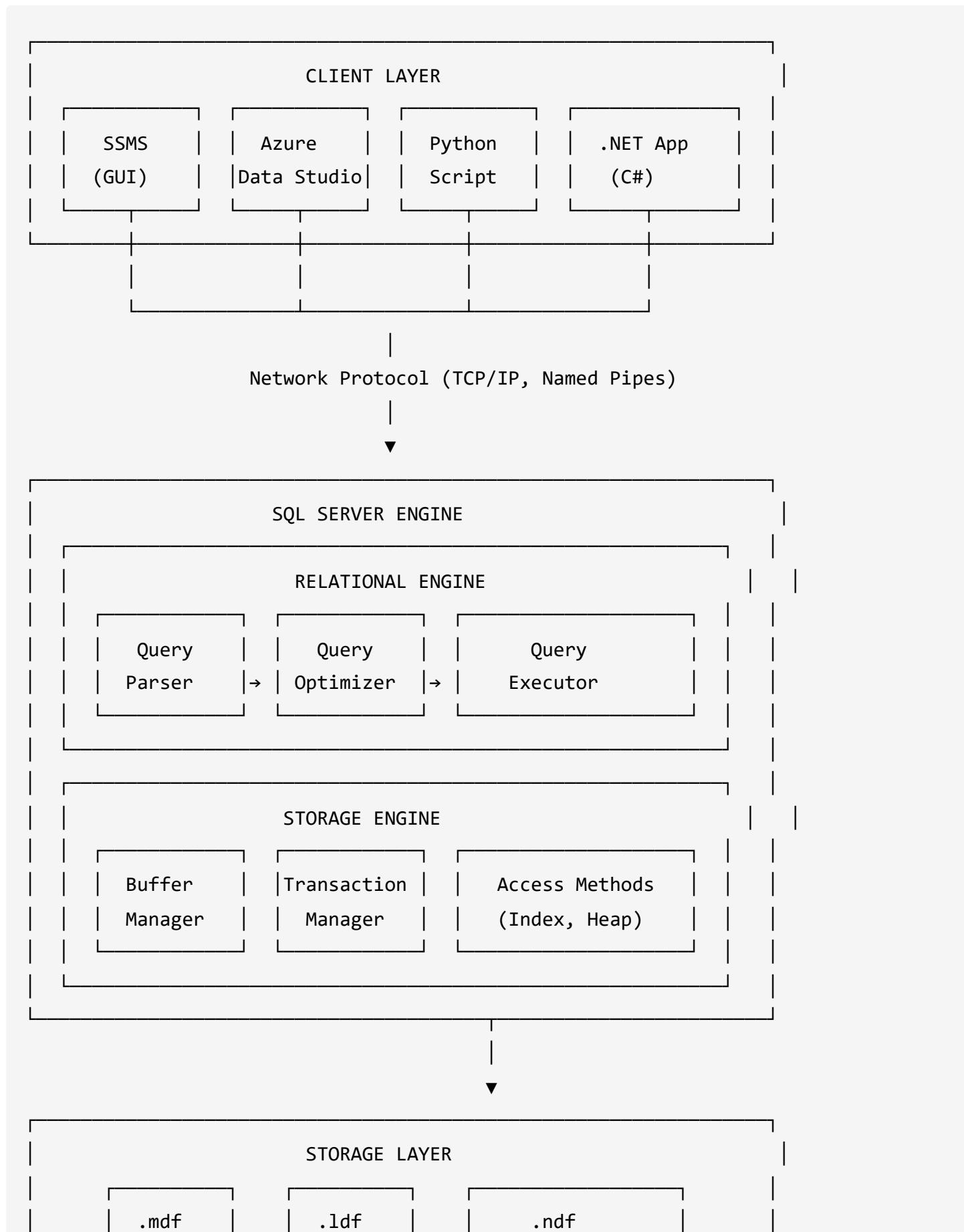
Overview

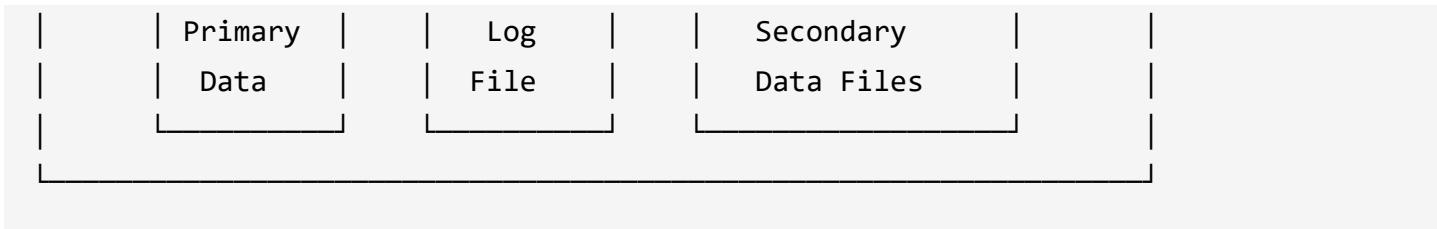
This note covers setting up your SQL development environment including SQL Server Express installation, SQL Server Management Studio (SSMS), and importing sample databases.



Theory

Understanding the SQL Server Architecture





SQL Server Editions

Edition	Cost	Use Case	Limitations
Express	Free	Learning, Small apps	10GB database, 1GB RAM
Developer	Free	Development only	Full features, not for production
Standard	Paid	SMB, Departmental	Limited features
Enterprise	Premium	Large enterprise	Full features, unlimited
Azure SQL	Pay-as-you-go	Cloud workloads	Managed service

SQL Server Components

Component	Purpose	Description
SQL Server Database Engine	Core service	Stores, processes, and secures data
SSMS	Client tool	GUI to manage and query databases
SQL Server Agent	Automation	Schedules jobs and alerts
SQL Server Browser	Discovery	Helps clients find SQL instances
Integration Services (SSIS)	ETL	Data transformation and loading
Reporting Services (SSRS)	Reports	Business intelligence reports
Analysis Services (SSAS)	Analytics	OLAP and data mining

Database Files Explained

File Type	Extension	Purpose
Primary Data File	.mdf	Main data storage (one per database)
Secondary Data File	.ndf	Additional data storage (optional)
Log File	.ldf	Transaction logs for recovery
Backup File	.bak	Database backup

System Databases (Don't Modify!)

Database	Purpose
master	System configuration, login info, linked servers
model	Template for new databases
msdb	SQL Agent jobs, backups, alerts
tempdb	Temporary objects, query sorting, intermediate results

Installation Steps

Step 1: Download SQL Server Express

1. Go to Microsoft's SQL Server download page
2. Choose **Express Edition** (free, quick installation)
3. Click "Download Now"
4. Run the installer
5. Select "**Basic**" installation type
6. Accept license terms
7. Click **Install**

Step 2: Install SSMS (SQL Server Management Studio)

1. Download SSMS from Microsoft
2. Run the installer
3. Choose installation location (default is fine)
4. Click **Install**
5. Wait for completion

Step 3: Connect to SQL Server

1. Open SSMS from Start Menu
2. Connection settings:
 - **Server type:** Database Engine
 - **Server name:** Your-PC-Name\SQLEXPRESS
 - **Authentication:** Windows Authentication
3. Click **Connect**

 **Tip:** If you don't know your PC name, open Command Prompt and type `whoami`

Creating Databases

Method 1: Using SQL Script

```
-- Execute the SQL script provided with course materials  
-- This creates the database with tables and data
```

Steps:

1. Open SSMS
2. Click **New Query**
3. Paste the SQL script
4. Click **Execute** (or press F5)
5. Refresh the databases folder to see new database

Method 2: Restoring from Backup (.bak file)

1. Place .bak file in SQL Server backup folder:

```
C:\Program Files\Microsoft SQL Server\MSSQL\MSSQL\Backup\
```

2. In SSMS: Right-click **Databases** → **Restore Database**
 3. Select **Device** option
 4. Browse and select the .bak file
 5. Click **OK** to restore
-



SSMS Interface

Main Components:

Area	Description
Object Explorer (Left)	Browse servers, databases, tables, etc.
Query Editor (Center)	Write and execute SQL code
Results Panel (Bottom)	View query results and messages
Toolbar (Top)	Common actions and database selector

Important Toolbar Elements:

- **Database dropdown:** Select which database to use
 - **Execute button:** Run your SQL query
 - **New Query:** Open new query window
-



Sample Databases

MyDatabase (Simple)

- **customers** table
- **orders** table

SalesDB (Intermediate)

Contains 5 tables:

- customers
- employees
- orders
- products
- categories

AdventureWorks (Advanced)

- Microsoft's sample database
- Many tables for complex queries
- Good for advanced practice



Basic SSMS Operations

Viewing Table Data

```
-- Right-click table → Select Top 1000 Rows  
-- Or write:  
SELECT * FROM TableName;
```

Switching Databases

```
USE DatabaseName;  
-- Example:  
USE MyDatabase;
```

Checking Database Connection

- Look at the toolbar dropdown
 - Or use: `SELECT DB_NAME();`
-



Writing Your First Query

```
-- Ensure you're connected to the right database  
USE MyDatabase;  
  
-- View all customers  
SELECT * FROM customers;  
  
-- View all orders  
SELECT * FROM orders;
```



SQL Comments

Single-Line Comment

```
-- This is a single line comment  
SELECT * FROM customers; -- This is also a comment
```

Multi-Line Comment

```
/*
This is a
multi-line comment
*/
SELECT * FROM customers;
```

? Practice Questions

Q1: What is the difference between SQL Server and SSMS?

Answer:

- **SQL Server** is the database engine that stores and processes data
- **SSMS** is a graphical tool used to connect to SQL Server and write/execute queries

Q2: What are the two methods to create a database?

Answer:

1. **SQL Script**: Execute a CREATE DATABASE script with table definitions
2. **Restore**: Use a backup file (.bak) to restore an existing database

Q3: How do you switch between databases in SQL?

Answer: Use the `USE` command:

```
USE DatabaseName;
```

Q4: What is the purpose of the Object Explorer in SSMS?

Answer: Object Explorer provides a tree-view navigation of all database objects including servers, databases, tables, views, stored procedures, etc.

Q5: How do you execute a SQL query in SSMS?

Answer:

1. Write the query in the Query Editor
 2. Click the **Execute** button in the toolbar, OR
 3. Press **F5** key
-

Interview Questions

Q1: What are the different SQL Server editions and their use cases?

Answer:

Edition	Use Case
Express	Learning, small applications (free, 10GB limit)
Developer	Development and testing (free, full features)
Standard	Small to medium business workloads
Enterprise	Mission-critical applications, large scale
Azure SQL	Cloud-native applications

Q2: Explain the SQL Server database files.

Answer:

- **.mdf (Primary Data File):** Stores the actual data, tables, indexes. One per database.

- **.ndf (Secondary Data File)**: Additional data files for large databases. Optional.
- **.ldf (Log File)**: Stores transaction logs for recovery. Required for ACID compliance.

Q3: What is tempdb and why is it important?

Answer: `tempdb` is a system database that stores:

- **Temporary tables** (#temp, ##temp)
- **Table variables**
- **Intermediate query results** (sorting, grouping)
- **Version store** for snapshot isolation

Important because:

- Shared by all databases on the server
- Recreated fresh on every SQL Server restart
- Performance bottleneck if poorly configured

Q4: What authentication modes does SQL Server support?

Answer:

1. **Windows Authentication**: Uses Windows/AD credentials. More secure, recommended.
2. **SQL Server Authentication**: Uses SQL Server logins (username/password).
3. **Mixed Mode**: Allows both authentication types.

Q5: How would you troubleshoot "Cannot connect to SQL Server"?

Answer:

1. **Verify server name**: Should include instance name (PC-NAME\SQLEXPRESS)
2. **Check SQL Server service**: Ensure it's running (services.msc)
3. **Check SQL Browser service**: Needed for named instances
4. **Firewall**: Port 1433 (default) may be blocked
5. **Authentication mode**: Try Windows Authentication first

6. Network protocol: Ensure TCP/IP is enabled in SQL Configuration Manager

Q6: What is the difference between a database and a schema?

Answer:

- **Database:** Physical container for data, files, and objects
- **Schema:** Logical namespace within a database to organize objects

```

Database (SalesDB)
├── Schema: dbo (default)
│   └── Tables: Customers, Orders
├── Schema: hr
│   └── Tables: Employees, Departments
└── Schema: finance
    └── Tables: Invoices, Payments

```

Schemas help with:

- Security (grant permissions at schema level)
- Organization (group related objects)
- Avoiding naming conflicts

Q7: What system databases exist in SQL Server?

Answer:

Database	Purpose
master	System configuration, all logins, linked servers
model	Template for creating new databases
msdb	SQL Agent jobs, backup history, alerts
tempdb	Temporary objects, intermediate results
resource	Read-only system objects (hidden)



Key Takeaways

1. **SQL Server Express** is free with 10GB database limit
 2. **SSMS** is the primary GUI tool for SQL Server management
 3. Database files: **.mdf** (data), **.ldf** (logs), **.ndf** (secondary)
 4. **System databases** (master, model, msdb, tempdb) - don't modify!
 5. Use `USE DatabaseName` to switch databases
 6. Comments: `--` single line, `/* */` multi-line
 7. **Windows Authentication** is preferred for security
-



Common Issues & Solutions

Issue	Solution
Can't connect to server	Check server name includes \SQLEXPRESS
Database not visible	Right-click and Refresh
Query runs on wrong database	Check toolbar dropdown or use USE command
Permission denied	Use Windows Authentication
Backup restore fails	Check .bak file is in correct folder



Best Practices

1. Always back up your databases regularly
2. Use Windows Authentication for local development
3. Keep your SSMS updated to the latest version
4. Create separate databases for different projects
5. Always verify your connected database before executing queries

Note 3: SELECT and FROM - Basic Queries

Overview

This note covers the fundamental building blocks of SQL queries - the SELECT and FROM clauses. These are the essential components of every SQL query.

Theory

What is a SQL Query?

A **SQL Query** is a declarative statement that instructs the database to retrieve, filter, transform, and present data. Understanding queries is fundamental because:

- **Declarative Nature:** You specify *what* you want, not *how* to get it
- **Set-Based:** Operates on entire sets of data, not row-by-row
- **Non-Destructive:** SELECT queries don't modify source data
- **Reproducible:** Same query always returns same results (given same data)

The Anatomy of a SQL Query

SQL QUERY STRUCTURE

SELECT	column1, column2, expression	← Projection
FROM	table_name	← Data Source
WHERE	condition	← Row Filter
GROUP BY	column	← Aggregation
HAVING	aggregate_condition	← Group Filter
ORDER BY	column	← Sorting

Understanding Projection vs Selection

In relational algebra (the foundation of SQL):

- **Projection:** Choosing which **columns** to include (SELECT clause)
- **Selection:** Choosing which **rows** to include (WHERE clause)

Original Table			After Projection (SELECT name)			After Selection (WHERE score > 80)		
ID	Name	Score		Name				
1	Alice	85	→	Alice				
2	Bob	72		Bob				
3	Carol	91		Carol				

Query Components (Clauses)

Every SQL query is made up of different sections called **clauses**:

Clause	Purpose	Required?
SELECT	Columns to retrieve (projection)	Yes
FROM	Data source (table/view)	Yes*
WHERE	Filter rows	No
GROUP BY	Group rows for aggregation	No
HAVING	Filter groups	No
ORDER BY	Sort results	No

*FROM is optional only for constant expressions



The SELECT Statement

Basic Syntax

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

SELECT All Columns

```
-- The asterisk (*) means "all columns"
SELECT *
FROM customers;
```

SELECT Specific Columns

```
-- List only the columns you need
SELECT first_name, country, score
FROM customers;
```



Execution Order

Important: SQL executes clauses in a specific order (not the order you write them!)

Step	Clause	Action
1	FROM	Find and retrieve data from table
2	SELECT	Choose which columns to display

Visual Example:

```
Step 1: FROM customers → Gets ALL data from customers table
Step 2: SELECT first_name, country → Keeps only these columns
```



Practical Examples

Example 1: Select Everything

```
-- Task: Retrieve all customer data
SELECT *
FROM customers;

-- Result: All columns and all rows from customers table
```

Example 2: Select Specific Columns

```
-- Task: Retrieve customer names, countries, and scores
SELECT first_name, country, score
FROM customers;

-- Result: Only these 3 columns are displayed
```

Example 3: Querying Different Tables

```
-- Get all order data
SELECT *
FROM orders;
```

```
-- Get all product data
SELECT *
FROM products;
```



Column Selection Rules

Column Order Matters

The order you list columns in SELECT determines the output order:

```
-- Country appears first
SELECT country, first_name, score
FROM customers;
```

```
-- First_name appears first
SELECT first_name, country, score
FROM customers;
```

Comma Placement

- Separate columns with commas
- NO comma after the last column

-- Correct

```
SELECT first_name, country, score  
FROM customers;
```

-- Wrong - comma after last column

```
SELECT first_name, country, score,  
FROM customers; -- ERROR!
```

Multiple Queries

Running Multiple Queries

You can have multiple queries in one script:

```
-- Query 1  
SELECT * FROM customers;  
  
-- Query 2  
SELECT * FROM orders;
```

Using Semicolons

Semicolons separate queries (required in some databases):

```
SELECT * FROM customers;  
SELECT * FROM orders;  
-- Results in 2 separate result sets
```



Static Values

Selecting Without a Table

```
-- Select a static number
```

```
SELECT 123;
```

```
-- Select a static string
```

```
SELECT 'Hello World';
```

Combining Static and Table Data

```
-- Add a static column to table data
```

```
SELECT
    id,
    first_name,
    'New Customer' AS customer_type
FROM customers;
```

Naming Columns (Aliases)

```
-- Use AS to give columns custom names
```

```
SELECT
    123 AS static_number,
    'Hello' AS greeting;
```



Executing Partial Queries

Highlight and Execute

In SSMS, you can:

1. Highlight specific portion of your code

2. Press Execute (F5)
3. Only highlighted code runs

This is useful for:

- Testing parts of complex queries
 - Quick data exploration
 - Debugging
-

❓ Practice Questions

Q1: What does `SELECT *` mean?

Answer: The asterisk (*) is a wildcard that means "select all columns" from the specified table. It retrieves every column in the table.

Q2: Write a query to get `first_name` and `score` from `customers` table.

Answer:

```
SELECT first_name, score  
FROM customers;
```

Q3: What is the execution order of `SELECT` and `FROM`?

Answer:

1. **FROM** executes first - retrieves all data from the table
2. **SELECT** executes second - filters to only specified columns

Q4: What happens if you put a comma after the last column?

Answer: You will get a syntax error. SQL expects another column name after a comma, but finds FROM instead.

Q5: How do you run only part of a query in SSMS?

Answer: Highlight the portion of the query you want to execute, then press F5 or click Execute. Only the highlighted code will run.

Q6: Write a query that selects all columns from the orders table.

Answer:

```
SELECT *
FROM orders;
```

Interview Questions

Q1: What is the difference between SELECT * and SELECT column_names?

Answer:

Aspect	SELECT *	SELECT columns
Readability	Unclear what's returned	Clear, explicit
Performance	Returns all data (slower)	Returns only needed data (faster)
Network	More data transferred	Less data transferred

Aspect	SELECT *	SELECT columns
Maintenance	Breaks if columns added/removed	More stable
Schema Changes	May return unexpected columns	Predictable results

Best Practice: Always list specific columns in production code.

Q2: Can you use SELECT without FROM?

Answer: Yes! You can select constant values, expressions, or system functions:

```
SELECT 1 + 1;           -- Returns: 2
SELECT 'Hello World';   -- Returns: Hello World
SELECT GETDATE();        -- Returns: Current date/time
SELECT @@VERSION;       -- Returns: SQL Server version
```

Q3: What is a Column Alias and why use it?

Answer: An alias provides a temporary name for a column in the result set.

Syntax:

```
SELECT column_name AS alias_name
-- or without AS
SELECT column_name alias_name
```

Use Cases:

1. Make column names more readable
2. Required when selecting expressions
3. Necessary for calculated columns
4. Essential when same column appears twice

```

SELECT
    first_name AS [Customer Name], -- Readable name
    score * 10 AS score_percentage, -- Calculated column
    UPPER(country) AS country_upper -- Function result
FROM customers;

```

Q4: Explain the SQL query execution order.

Answer: SQL has a **logical processing order** different from written order:

Written Order:

```
SELECT → FROM → WHERE → GROUP BY → HAVING → ORDER BY
```

Execution Order:

```
FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY
```

1. **FROM:** Identify data source
2. **WHERE:** Filter rows
3. **GROUP BY:** Group rows
4. **HAVING:** Filter groups
5. **SELECT:** Choose columns, calculate expressions
6. **ORDER BY:** Sort final results

Why This Matters: You can't use a SELECT alias in WHERE (because SELECT runs after WHERE).

Q5: What happens if you SELECT a column that doesn't exist?

Answer: You get an error: "Invalid column name 'column_name'"

```

SELECT nonexistent_column FROM customers;
-- Error: Invalid column name 'nonexistent_column'

```

Q6: What is the difference between a column and an expression in SELECT?

Answer:

- **Column:** References actual data stored in a table
- **Expression:** Calculates a value using operators, functions, or literals

```
SELECT
```

```
    first_name,          -- Column
    score,              -- Column
    score * 2,          -- Expression (calculation)
    UPPER(first_name),  -- Expression (function)
    'Active' AS status, -- Expression (literal)
    score + 100 AS adjusted_score -- Expression with alias
FROM customers;
```

Q7: How do you handle column names with spaces or special characters?

Answer: Use square brackets `[]` or double quotes `""`:

```
SELECT
```

```
    [First Name],      -- Column with space
    [Order-ID],        -- Column with hyphen
    "Customer Name"   -- ANSI standard (if enabled)
FROM customers;
```

Best Practice: Avoid spaces in column names. Use underscores: `first_name`, `order_id`

Q8: What is the purpose of the FROM clause?

Answer: FROM specifies the data source(s) for the query:

Can reference:

- Tables: `FROM customers`
- Views: `FROM v_active_customers`

- Subqueries: `FROM (SELECT ...) AS subquery`
 - Table-valued functions: `FROM dbo.GetOrders(2024)`
 - Multiple tables: `FROM customers, orders` (cross join)
-

🔑 Key Takeaways

1. `SELECT` specifies columns (projection) - what to show
 2. `FROM` specifies data source - where to get data
 3. `SELECT *` retrieves all columns (avoid in production)
 4. Execution order: `FROM → WHERE → SELECT → ORDER BY`
 5. Use aliases (AS) for readable column names
 6. No comma after the last column
 7. Semicolons separate multiple queries
-

📌 Best Practices

Practice	Reason
List specific columns	Better performance than <code>SELECT *</code>
Use meaningful aliases	Makes results clearer
Format queries on multiple lines	Easier to read and maintain
Avoid <code>SELECT *</code> in production	Returns unnecessary data
Always specify <code>FROM</code>	Required for table queries

Formatting Recommendations:

```
-- Good formatting  
SELECT  
    first_name,  
    country,  
    score  
FROM customers;  
  
-- Less readable  
SELECT first_name, country, score FROM customers;
```



Exercises

Exercise 1:

Write a query to display only the ID and first_name from customers.

```
-- Your solution:  
SELECT id, first_name  
FROM customers;
```

Exercise 2:

Write a query to show order_id and sales from orders table.

```
-- Your solution:  
SELECT order_id, sales  
FROM orders;
```

Exercise 3:

Write two separate queries in one script - one for customers, one for orders.

```
-- Your solution:
```

```
SELECT * FROM customers;  
SELECT * FROM orders;
```

Note 4: WHERE Clause - Filtering Data

Overview

This note covers the WHERE clause, which allows you to filter data based on conditions. You'll learn how to retrieve only the rows that meet specific criteria.

Theory

What is the WHERE Clause?

The **WHERE clause** performs **row selection** (filtering) in SQL queries. It evaluates a Boolean expression for each row and includes only rows where the expression evaluates to TRUE.

Key Concepts:

- **Predicate:** The condition that evaluates to TRUE, FALSE, or UNKNOWN
- **Filtering:** Process of excluding rows that don't match
- **Short-circuit evaluation:** SQL may stop evaluating if outcome is determined

How WHERE Works Internally

TABLE: Customers

ID	Name	Country	Score
1	Alice	USA	850
2	Bob	Germany	420
3	Charlie	USA	910
4	Diana	UK	380
5	Eve	Germany	750

All rows

WHERE score > 500

- Row 1: $850 > 500 = \text{TRUE}$ ✓ Include
- Row 2: $420 > 500 = \text{FALSE}$ ✗ Exclude
- Row 3: $910 > 500 = \text{TRUE}$ ✓ Include
- Row 4: $380 > 500 = \text{FALSE}$ ✗ Exclude
- Row 5: $750 > 500 = \text{TRUE}$ ✓ Include

FILTERED RESULT

ID	Name	Country	Score
1	Alice	USA	850
3	Charlie	USA	910
5	Eve	Germany	750

Three-Valued Logic (TRUE, FALSE, UNKNOWN)

SQL uses **three-valued logic** because of NULL values:

Condition	NULL Involvement	Result
5 > 3	No NULLs	TRUE
3 > 5	No NULLs	FALSE
NULL > 5	Has NULL	UNKNOWN
NULL = NULL	Has NULL	UNKNOWN

Important: WHERE only returns rows where condition is TRUE (not FALSE, not UNKNOWN)

Basic Syntax

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



Execution Order

Step	Clause	Action
1	FROM	Get all data from table
2	WHERE	Filter rows based on condition
3	SELECT	Choose columns to display



Building Conditions

Condition Structure

A condition compares values:

```
column_name operator value
```

Examples of Conditions:

```
-- Numeric comparison  
score > 500  
  
-- Text comparison (use single quotes)  
country = 'Germany'  
  
-- Not equal  
country <> 'USA'
```



Practical Examples

Example 1: Filter by Number

```
-- Get customers with score greater than 500  
SELECT *  
FROM customers  
WHERE score > 500;
```

Example 2: Filter by Text

```
-- Get customers from Germany
SELECT *
FROM customers
WHERE country = 'Germany';
```

Example 3: Not Equal

```
-- Get customers NOT with zero score
SELECT *
FROM customers
WHERE score <> 0;
```

Example 4: Combining Column Selection and Filtering

```
-- Get name and country for German customers
SELECT first_name, country
FROM customers
WHERE country = 'Germany';
```

⚠️ Important Rules

String Values Require Single Quotes

```
-- ✓ Correct - text in single quotes
WHERE country = 'Germany'

-- ✗ Wrong - no quotes around text
WHERE country = Germany -- ERROR!
```

Numbers Don't Need Quotes

-- Correct

```
WHERE score > 500
```

-- Also works (but unnecessary)

```
WHERE score > '500'
```

Case Sensitivity

- SQL keywords are NOT case-sensitive
- String values MAY be case-sensitive (depends on database settings)

-- These may give different results:

```
WHERE country = 'germany'
```

```
WHERE country = 'Germany'
```

🔍 How Filtering Works

Step-by-Step Process:

1. SQL reads each row from the table
2. Evaluates the condition for that row
3. If TRUE → row is included in results
4. If FALSE → row is excluded

Visual Example:

```
Data: Maria - Germany - 350  
Condition: score > 500  
350 > 500? FALSE → Row excluded
```

```
Data: John - USA - 900  
Condition: score > 500  
900 > 500? TRUE → Row included
```



Filtering + Column Selection

You can combine WHERE with specific column selection:

```
-- Filter rows AND select specific columns  
SELECT first_name, country  
FROM customers  
WHERE country = 'Germany';  
  
-- Result: Only 2 columns, only German customers
```

Order of Clauses (Fixed):

```
SELECT columns      -- 1st in code  
FROM table          -- 2nd in code  
WHERE condition;    -- 3rd in code
```

❓ Practice Questions

Q1: What does the WHERE clause do?

Answer: The WHERE clause filters rows from the table, returning only those that meet the specified condition.

Q2: Write a query to get all customers from the USA.

Answer:

```
SELECT *
FROM customers
WHERE country = 'USA';
```

Q3: Why do we use single quotes around 'Germany' but not around 500?

Answer:

- Single quotes are required for text/string values
- Numbers don't need quotes

Q4: Write a query to get customers with score NOT equal to zero.

Answer:

```
SELECT *
FROM customers
WHERE score <> 0;
-- OR
WHERE score != 0;
```

Q5: What is the execution order: SELECT, FROM, WHERE?

Answer:

1. FROM (get data)
2. WHERE (filter rows)
3. SELECT (choose columns)

Q6: Write a query to get first_name and score for customers with score > 400.

Answer:

```
SELECT first_name, score
FROM customers
WHERE score > 400;
```

🎯 Interview Questions

Q1: What is the difference between WHERE and HAVING?

Answer:

Aspect	WHERE	HAVING
Filters	Individual rows	Groups (after GROUP BY)
Execution	Before GROUP BY	After GROUP BY
Aggregates	Cannot use (e.g., SUM, COUNT)	Can use aggregates
Index Usage	Can use indexes	Usually cannot use indexes

```
-- WHERE: Filter rows before grouping
SELECT country, COUNT(*)
FROM customers
WHERE score > 500          -- Filter individual rows
GROUP BY country;

-- HAVING: Filter groups after grouping
SELECT country, COUNT(*)
FROM customers
GROUP BY country
HAVING COUNT(*) > 5;       -- Filter aggregated groups
```

Q2: Why doesn't WHERE work with column aliases?

Answer: Because of SQL's logical execution order:

1. FROM → 2. WHERE → 3. GROUP BY → 4. HAVING → 5. SELECT → 6. ORDER BY

Aliases are defined in SELECT, which runs AFTER WHERE. So WHERE doesn't "know" the alias yet.

```
-- ❌ Wrong - alias doesn't exist when WHERE runs
SELECT score * 2 AS double_score
FROM customers
WHERE double_score > 100;  -- Error!

-- ✅ Correct - use the original expression
SELECT score * 2 AS double_score
FROM customers
WHERE score * 2 > 100;      -- Works!
```

Q3: How does NULL behave in WHERE conditions?

Answer: NULL represents "unknown" and behaves specially:

```
-- This returns NO rows where score is NULL
SELECT * FROM customers WHERE score = NULL;      -- Wrong!
SELECT * FROM customers WHERE score <> NULL;      -- Wrong!
```

```
-- Correct way to check for NULL
SELECT * FROM customers WHERE score IS NULL;      -- Right!
SELECT * FROM customers WHERE score IS NOT NULL;   -- Right!
```

Why? Because NULL = NULL returns UNKNOWN (not TRUE), and WHERE only includes TRUE rows.

Q4: What is the difference between = and LIKE?

Answer:

Operator	Purpose	Wildcards	Performance
=	Exact match	None	Fast (uses index)
LIKE	Pattern match	% and _	Slower (may not use index)

```
-- Exact match
WHERE name = 'John'          -- Only 'John'

-- Pattern match
WHERE name LIKE 'John%'     -- John, Johnson, Johnny...
WHERE name LIKE 'J_hn'        -- John, Jahn, Jhn...
WHERE name LIKE '%son'       -- Johnson, Wilson, Anderson...
```

Q5: What operators can be used in WHERE clause?

Answer:

Comparison Operators:

Operator	Meaning	Example
=	Equal	score = 100

Operator	Meaning	Example
<> or !=	Not equal	status <> 'Inactive'
> , <	Greater/Less than	score > 500
>= , <=	Greater/Less or equal	date >= '2024-01-01'

Logical Operators:

Operator	Purpose	Example
AND	Both conditions true	score > 500 AND country = 'USA'
OR	Either condition true	country = 'USA' OR country = 'UK'
NOT	Negates condition	NOT country = 'USA'

Special Operators:

Operator	Purpose	Example
BETWEEN	Range (inclusive)	score BETWEEN 100 AND 500
IN	List of values	country IN ('USA', 'UK', 'Germany')
LIKE	Pattern matching	name LIKE 'A%'
IS NULL	Check for NULL	score IS NULL
IS NOT NULL	Check for non-NUL	score IS NOT NULL

Q6: How does BETWEEN work and is it inclusive?

Answer: BETWEEN is **inclusive** on both ends:

```
WHERE score BETWEEN 100 AND 500
-- Equivalent to:
WHERE score >= 100 AND score <= 500
-- Includes both 100 and 500!
```

Q7: What is short-circuit evaluation in SQL?

Answer: Short-circuit evaluation means SQL stops evaluating a condition once the outcome is determined:

```
-- If first condition is FALSE, AND doesn't evaluate second
WHERE 1 = 0 AND expensive_function() -- expensive_function() may not run

-- If first condition is TRUE, OR doesn't evaluate second
WHERE 1 = 1 OR expensive_function() -- expensive_function() may not run
```

Note: SQL Server doesn't guarantee short-circuit evaluation due to query optimization. Don't rely on it!

Q8: How do you filter by date in SQL?

Answer:

```
-- Exact date
WHERE order_date = '2024-01-15'

-- Date range
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31'

-- Year extraction
WHERE YEAR(order_date) = 2024

-- Recent records
WHERE order_date >= DATEADD(day, -30, GETDATE())
```

Best Practice: Avoid functions on columns (like YEAR()) as they prevent index usage.

🔑 Key Takeaways

1. **WHERE** filters rows where condition is TRUE
2. Uses **three-valued logic**: TRUE, FALSE, UNKNOWN

3. Text values need **single quotes**, numbers don't
 4. **NULL comparisons** require IS NULL / IS NOT NULL
 5. WHERE runs **before SELECT** (can't use aliases)
 6. Use **AND/OR** for multiple conditions
 7. **BETWEEN is inclusive** on both ends
-

Common Operators for WHERE

Operator	Meaning	Example
=	Equal to	country = 'USA'
<> or !=	Not equal to	score <> 0
>	Greater than	score > 500
<	Less than	score < 100
>=	Greater than or equal	score >= 500
<=	Less than or equal	score <= 100



Exercises

Exercise 1:

Get all orders with sales greater than 100.

```
SELECT *
FROM orders
WHERE sales > 100;
```

Exercise 2:

Get customer ID and name for customers from the UK.

```
SELECT id, first_name  
FROM customers  
WHERE country = 'UK';
```

Exercise 3:

Get all customers who don't have zero score.

```
SELECT *  
FROM customers  
WHERE score <> 0;
```



Tips

- Always test your condition logic before running complex queries
- Start with `SELECT *` to see all data, then refine
- Check string values match exactly (including case)
- WHERE clause only affects rows, not columns

Note 5: ORDER BY - Sorting Data

Overview

This note covers the ORDER BY clause, which allows you to sort query results in ascending or descending order based on one or more columns.



Theory

What is ORDER BY?

ORDER BY is the clause that controls the sequence of rows in your result set. Without ORDER BY, SQL makes **no guarantee about row order** - you might get different orderings each time you run the same query.

Why Row Order is Not Guaranteed Without ORDER BY

WITHOUT ORDER BY - Unpredictable!

Query: `SELECT * FROM customers`

Run 1: Run 2: Run 3:

Alice
Bob
Carol

Bob
Carol
Alice

Carol
Alice
Bob

Order depends on: indexes, parallel execution, data pages

Key Concepts

Deterministic Ordering: To get consistent, reproducible results, **ALWAYS** use ORDER BY when order matters.

Stable vs Unstable Sort:

- If two rows have the same value in the ORDER BY column, their relative order is unpredictable
- Add a tie-breaker column (like primary key) for fully deterministic ordering

Sorting Mechanisms

Keyword	Meaning	Order	NULLs
ASC	Ascending	Lowest → Highest (A→Z, 0→9)	First (SQL Server)
DESC	Descending	Highest → Lowest (Z→A, 9→0)	Last (SQL Server)

 **Default:** If not specified, ASC is used automatically

How Sorting Works for Different Data Types

Data Type	ASC Order	DESC Order
Numbers	1, 2, 3...	100, 99, 98...
Text	A, B, C...	Z, Y, X...
Dates	Oldest → Newest	Newest → Oldest
NULL	First (varies by DB)	Last (varies by DB)

NULL Handling in ORDER BY

```
-- In SQL Server:
-- ASC: NULLs appear FIRST
-- DESC: NULLs appear LAST

-- To control NULL position explicitly (PostgreSQL/Oracle):
ORDER BY column NULLS FIRST
ORDER BY column NULLS LAST
```



Basic Syntax

```
SELECT columns  
FROM table_name  
ORDER BY column_name [ASC|DESC];
```

Syntax Rules:

- ORDER BY comes after WHERE (if present)
 - Specify column(s) to sort by
 - Specify mechanism (ASC or DESC)
 - Can use column names or column positions
-



Practical Examples

Example 1: Sort Ascending (Lowest First)

```
-- Sort customers by score, lowest first  
SELECT *  
FROM customers  
ORDER BY score ASC;  
  
-- Result: 0, 350, 500, 750, 900
```

Example 2: Sort Descending (Highest First)

```
-- Sort customers by score, highest first  
SELECT *  
FROM customers  
ORDER BY score DESC;  
  
-- Result: 900, 750, 500, 350, 0
```

Example 3: Sort Alphabetically

```
-- Sort by country A-Z
SELECT *
FROM customers
ORDER BY country ASC;

-- Result: Germany, Germany, UK, USA, USA
```

Example 4: Sort with WHERE

```
-- Filter AND sort
SELECT *
FROM customers
WHERE score > 0
ORDER BY score DESC;
```



Execution Order

Step	Clause	Action
1	FROM	Get data from table
2	WHERE	Filter rows
3	SELECT	Choose columns
4	ORDER BY	Sort results

ORDER BY is the LAST clause to execute!



Nested Sorting (Multiple Columns)

When to Use Nested Sorting

When you have duplicate values in the first sort column, use a second column to refine the order.

Syntax:

```
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC]
```

Example: Sort by Country, then by Score

```
SELECT *
FROM customers
ORDER BY country ASC, score DESC;
```

How it works:

1. First sorts by country (alphabetically)
2. Within each country, sorts by score (highest first)

Priority Rules:

- First column has highest priority
- Second column only affects ties in first column
- Each column can have different mechanisms (ASC/DESC)



Practical Nested Sorting Examples

Task: Sort by country, then by highest score

```
SELECT *
FROM customers
ORDER BY country ASC, score DESC;
```

```
/* Result:
Germany - 500
Germany - 350
UK - 750
USA - 900
USA - 0
*/
```

Task: Get top customers by country

```
SELECT country, first_name, score
FROM customers
ORDER BY country, score DESC;
```

🎯 Common Use Cases

Top N Analysis

```
-- Highest scoring customers
SELECT *
FROM customers
ORDER BY score DESC;
```

```
-- Most recent orders
SELECT *
FROM orders
ORDER BY order_date DESC;
```

Alphabetical Lists

```
-- Customer directory A-Z
SELECT first_name, country
FROM customers
ORDER BY first_name ASC;
```

Chronological Order

```
-- Orders from oldest to newest
SELECT *
FROM orders
ORDER BY order_date ASC;
```

```
-- Orders from newest to oldest
SELECT *
FROM orders
ORDER BY order_date DESC;
```

?

Practice Questions

Q1: What is the default sort order if you don't specify ASC or DESC?

Answer: ASC (Ascending) is the default. Lowest values appear first.

Q2: Write a query to sort customers by score from highest to lowest.

Answer:

```
SELECT *
FROM customers
ORDER BY score DESC;
```

Q3: Can you sort by multiple columns? How?

Answer: Yes, separate columns with commas. Each column can have its own ASC/DESC:

```
ORDER BY country ASC, score DESC
```

Q4: When does ORDER BY execute in the query?

Answer: ORDER BY executes LAST, after FROM, WHERE, and SELECT.

Q5: Write a query to get the 2 most recent orders.

Answer:

```
SELECT TOP 2 *
FROM orders
ORDER BY order_date DESC;
```

Q6: Sort customers by country (A-Z), then by first_name (A-Z).

Answer:

```
SELECT *
FROM customers
ORDER BY country ASC, first_name ASC;
```

Interview Questions

Q1: Can you ORDER BY a column not in the SELECT list?

Answer: Yes! ORDER BY can reference any column in the source table(s), not just those in SELECT.

```
-- Works: ordering by score but not selecting it
SELECT first_name, country
FROM customers
ORDER BY score DESC;
```

Exception: When using SELECT DISTINCT, you can only ORDER BY columns in the SELECT list.

Q2: Can you ORDER BY column alias or column position?

Answer: Yes, both work:

```
-- ORDER BY alias (works because ORDER BY runs after SELECT)
SELECT first_name, score * 2 AS double_score
FROM customers
ORDER BY double_score DESC;

-- ORDER BY position (1 = first column, 2 = second column)
SELECT first_name, score
FROM customers
ORDER BY 2 DESC; -- Sorts by score (2nd column)
```

Best Practice: Avoid positional ordering - it breaks if columns change.

Q3: How does ORDER BY handle NULL values?

Answer: Behavior varies by database:

Database	ASC	DESC
SQL Server	NULLs first	NULLs last
PostgreSQL	NULLs last	NULLs first
Oracle	NULLs last	NULLs first
MySQL	NULLs first	NULLs last

Control with NULLS FIRST/LAST (not SQL Server):

```
ORDER BY score DESC NULLS LAST
```

SQL Server workaround:

```
ORDER BY CASE WHEN score IS NULL THEN 1 ELSE 0 END, score DESC
```

Q4: What is the difference between ORDER BY in a subquery vs main query?

Answer: ORDER BY in a subquery is usually meaningless and may be ignored!

```
-- Subquery ORDER BY is ignored (no guarantee of order)
SELECT * FROM (
    SELECT * FROM customers ORDER BY score DESC
) AS sub;

-- Only main query ORDER BY matters
SELECT * FROM (
    SELECT * FROM customers
) AS sub
ORDER BY score DESC;
```

Q5: How does ORDER BY affect performance?

Answer:

Without Index:

- SQL must sort all rows (expensive for large tables)
- Uses tempdb for large sorts
- O(n log n) operation

With Index on ORDER BY column:

- Data already sorted in index
- No additional sort needed
- Much faster

```
-- If index exists on score:
SELECT * FROM customers ORDER BY score; -- Uses index (fast)

-- If no index on score:
SELECT * FROM customers ORDER BY score; -- Full sort (slow)
```

Q6: Can you use expressions in ORDER BY?

Answer: Yes! You can sort by calculated values:

```
-- Sort by calculated expression
SELECT first_name, score
FROM customers
ORDER BY score * 2 DESC;

-- Sort by function result
SELECT first_name, last_name
FROM customers
ORDER BY LEN(first_name) DESC; -- By name length

-- Sort by CASE expression
SELECT first_name, country
FROM customers
ORDER BY CASE country
    WHEN 'USA' THEN 1
    WHEN 'UK' THEN 2
    ELSE 3
END;
```

Q7: What happens if you ORDER BY a non-unique column?

Answer: Rows with the same value have **undefined relative order** (unstable sort).

```
-- Multiple customers in 'USA' - their order is unpredictable
SELECT * FROM customers ORDER BY country;

-- Solution: Add tie-breaker for deterministic ordering
SELECT * FROM customers ORDER BY country, customer_id;
```

Q8: Can you combine ORDER BY with UNION?

Answer: Yes, but ORDER BY applies to the entire result, placed at the very end:

```
SELECT first_name, 'Customer' AS type FROM customers
UNION
SELECT first_name, 'Employee' AS type FROM employees
ORDER BY first_name; -- Sorts combined result
```

🔑 Key Takeaways

1. **ORDER BY** is the ONLY way to guarantee row order
2. **ASC** = Ascending (default), **DESC** = Descending
3. ORDER BY executes **last** in the query
4. Can ORDER BY **columns not in SELECT** (except with DISTINCT)
5. Can use **aliases** and **positions** (prefer aliases)
6. **NULL handling** varies by database
7. Use **indexes** on frequently sorted columns for performance
8. Add **tie-breaker** for fully deterministic ordering

📌 Best Practices

Practice	Reason
Always specify ASC or DESC	Makes intent clear
Use ORDER BY with TOP/LIMIT	For meaningful "top N" results
Consider indexes on sort columns	Improves performance
Test nested sorting	Verify expected order

🧪 Exercises

Exercise 1:

Sort all customers by `first_name` alphabetically.

```
SELECT *
FROM customers
ORDER BY first_name ASC;
```

Exercise 2:

Get orders sorted by sales (highest first).

```
SELECT *
FROM orders
ORDER BY sales DESC;
```

Exercise 3:

Sort customers by country (A-Z), then score (highest first).

```
SELECT *
FROM customers
ORDER BY country ASC, score DESC;
```

Exercise 4:

Get the lowest scoring customer.

```
SELECT TOP 1 *
FROM customers
ORDER BY score ASC;
```

⚠ Common Mistakes

Mistake	Correction
ORDER BY before WHERE	ORDER BY must come after WHERE

Mistake	Correction
Forgetting sort direction	Always specify ASC or DESC
Wrong column order in nested sort	First column has highest priority
Expecting ORDER BY to affect original table	It only affects the result set

Note 6: GROUP BY - Aggregating Data

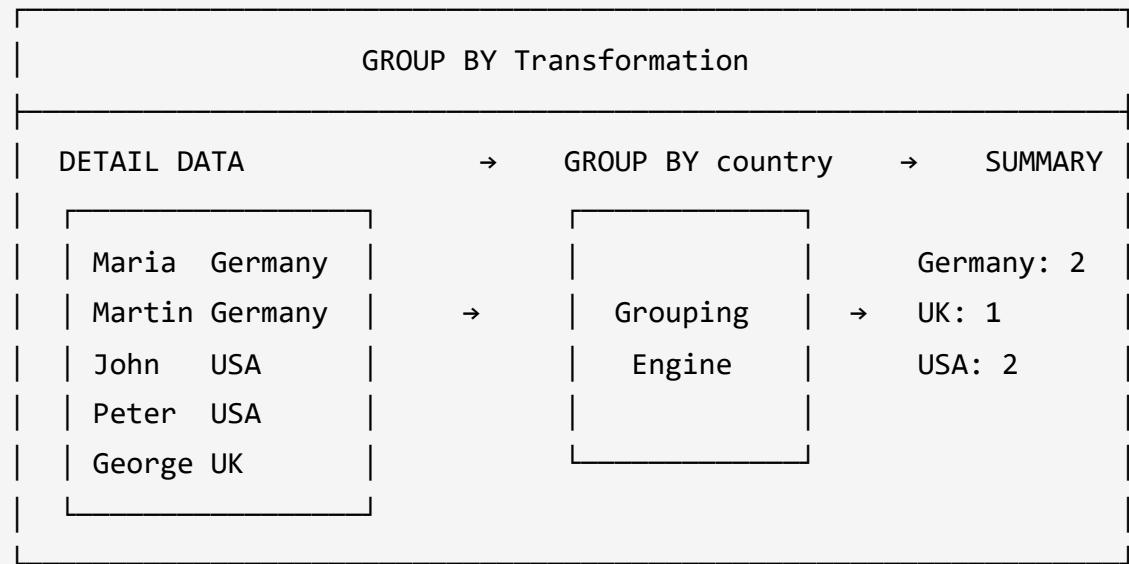
Overview

This note covers the GROUP BY clause, which allows you to combine rows with the same values and perform aggregate calculations like SUM, COUNT, AVG, etc.

Theory

What is GROUP BY?

GROUP BY transforms detail-level data into summary-level data by combining rows with identical values:



The Aggregation Concept

Aggregation = Combining multiple values into a single summary value

Level	Example	Description
Row Level	<code>SELECT score FROM customers</code>	Each row is separate
Group Level	<code>SELECT country, SUM(score) GROUP BY country</code>	Rows combined by country
Table Level	<code>SELECT SUM(score) FROM customers</code>	All rows into one value

The GROUP BY Rule (Most Important!)

Any column in SELECT must EITHER be in GROUP BY OR wrapped in an aggregate function

-- 🧐 Think of it this way:
 -- GROUP BY creates ONE row per group
 -- So which "first_name" should it show if there are 5 Johns?
 -- SQL cannot decide, so it throws an error!

```
SELECT country, first_name, SUM(score) -- ✗ ERROR
FROM customers
GROUP BY country; -- first_name not grouped or aggregated
```

-- Solution 1: Add to GROUP BY (creates more groups)

```
SELECT country, first_name, SUM(score) -- ✓
FROM customers
GROUP BY country, first_name;
```

-- Solution 2: Aggregate it (pick one representative value)

```
SELECT country, MAX(first_name), SUM(score) -- ✓
FROM customers
GROUP BY country;
```

Aggregate Functions - Complete Reference

Function	NULLs	Duplicates	Use Case
COUNT(*)	Counts ALL rows	Counts all	Total row count
COUNT(col)	Ignores NULLs	Counts all	Non-null value count
COUNT(DISTINCT col)	Ignores NULLs	Ignores dupes	Unique value count
SUM(col)	Ignores NULLs	Sums all	Total of values
AVG(col)	Ignores NULLs	Averages all	Mean value
MAX(col)	Ignores NULLs	Returns highest	Largest value
MIN(col)	Ignores NULLs	Returns lowest	Smallest value

When to Use GROUP BY

- **"How many X per Y?"** → COUNT(*) GROUP BY
- **"Total X for each Y?"** → SUM(x) GROUP BY y

- "Average X by Y?" → `AVG(x) GROUP BY y`
 - "Maximum X in each Y?" → `MAX(x) GROUP BY y`
-



Basic Syntax

```
SELECT column, AGGREGATE_FUNCTION(column)
FROM table_name
GROUP BY column;
```

Aggregate Functions

Function	Description	Example
<code>SUM()</code>	Total of values	<code>SUM(score)</code>
<code>COUNT()</code>	Number of rows	<code>COUNT(id)</code>
<code>AVG()</code>	Average value	<code>AVG(score)</code>
<code>MAX()</code>	Maximum value	<code>MAX(score)</code>
<code>MIN()</code>	Minimum value	<code>MIN(score)</code>



Practical Examples

Example 1: Total Score by Country

```
SELECT country, SUM(score) AS total_score  
FROM customers  
GROUP BY country;
```

```
/* Result:  
Germany    850  
UK         750  
USA        900  
*/
```

Example 2: Count Customers by Country

```
SELECT country, COUNT(id) AS total_customers  
FROM customers  
GROUP BY country;
```

```
/* Result:  
Germany    2  
UK         1  
USA        2  
*/
```

Example 3: Multiple Aggregations

```
SELECT  
    country,  
    COUNT(id) AS total_customers,  
    SUM(score) AS total_score,  
    AVG(score) AS avg_score  
FROM customers  
GROUP BY country;
```



Execution Order

Step	Clause	Action
1	FROM	Get data from table
2	WHERE	Filter rows (before grouping)
3	GROUP BY	Combine rows into groups
4	SELECT	Choose columns and calculate aggregates
5	ORDER BY	Sort results



How GROUP BY Works

Visual Process:

Original Data:

Maria - Germany - 350
 John - USA - 900
 George - UK - 750
 Martin - Germany - 500
 Peter - USA - 0

After GROUP BY country:

Germany → Maria(350) + Martin(500) = 850
 UK → George(750) = 750
 USA → John(900) + Peter(0) = 900

Key Points:

1. Rows with same value are combined
2. Non-aggregated columns must be in GROUP BY
3. Output has one row per unique group

⚠️ Important Rules

Rule 1: Non-aggregated columns must be in GROUP BY

-- ❌ Wrong - first_name not in GROUP BY or aggregate

```
SELECT country, first_name, SUM(score)  
FROM customers  
GROUP BY country; -- ERROR!
```

-- ✅ Correct - add first_name to GROUP BY

```
SELECT country, first_name, SUM(score)  
FROM customers  
GROUP BY country, first_name;
```

-- ✅ Correct - aggregate first_name

```
SELECT country, COUNT(first_name), SUM(score)  
FROM customers  
GROUP BY country;
```

Rule 2: Aliases for Aggregated Columns

-- Without alias - column has no name

```
SELECT country, SUM(score)  
FROM customers  
GROUP BY country; -- Column shows "No column name"
```

-- With alias - clear naming

```
SELECT country, SUM(score) AS total_score  
FROM customers  
GROUP BY country; -- Column shows "total_score"
```

🎯 Multiple Column Grouping

Grouping by Two Columns

```
SELECT country, first_name, SUM(score) AS total_score  
FROM customers  
GROUP BY country, first_name;
```

Effect: Creates groups based on unique combinations of BOTH columns.

Example:

Groups created:

(Germany, Maria) → One group
(Germany, Martin) → One group
(USA, John) → One group
(USA, Peter) → One group
(UK, George) → One group

🔧 GROUP BY with WHERE

Filter BEFORE Grouping

```
-- Only count customers with score > 0  
SELECT country, COUNT(id) AS customer_count  
FROM customers  
WHERE score > 0  
GROUP BY country;
```

Order: WHERE filters rows FIRST, then GROUP BY aggregates.

❓ Practice Questions

Q1: What does GROUP BY do?

Answer: GROUP BY combines rows with the same values in specified columns into single summary rows, typically used with aggregate functions.

Q2: Write a query to find total score for each country.

Answer:

```
SELECT country, SUM(score) AS total_score  
FROM customers  
GROUP BY country;
```

Q3: Can you select a column that is not in GROUP BY?

Answer: Only if it's wrapped in an aggregate function. Otherwise, you'll get an error.

Q4: What's the difference between COUNT(*) and COUNT(column)?

Answer:

- COUNT(*) counts all rows including NULLs
- COUNT(column) counts only non-NULL values in that column

Q5: Write a query to find average score by country.

Answer:

```
SELECT country, AVG(score) AS avg_score  
FROM customers  
GROUP BY country;
```

Q6: When does GROUP BY execute in the query order?

Answer: After WHERE but before SELECT and ORDER BY.

🎯 Interview Questions

Q1: What is the difference between WHERE and HAVING?

Answer:

Aspect	WHERE	HAVING
Filters	Individual rows	Groups/aggregates
Executes	Before GROUP BY	After GROUP BY
Can use aggregates	✗ No	✓ Yes

```
-- WHERE filters rows BEFORE grouping
SELECT country, SUM(score)
FROM customers
WHERE score > 100          -- Filter individual scores
GROUP BY country;
```

```
-- HAVING filters AFTER grouping
SELECT country, SUM(score)
FROM customers
GROUP BY country
HAVING SUM(score) > 500; -- Filter group totals
```

Q2: What is the difference between COUNT(*), COUNT(column), and COUNT(DISTINCT column)?

Answer:

```
-- Sample data:
-- id | country
-- 1  | USA
-- 2  | USA
-- 3  | NULL
-- 4  | UK
```

SELECT

```
COUNT(*) AS all_rows,          -- 4 (counts NULLs)
COUNT(country) AS non_null,    -- 3 (ignores NULLs)
COUNT(DISTINCT country) AS unique_countries; -- 2 (USA, UK)
```

Q3: Can you GROUP BY a column not in SELECT?

Answer: Yes! Unlike the reverse (SELECT without GROUP BY), this is allowed:

```
-- Group by country but don't show it
SELECT SUM(score) AS total
FROM customers
GROUP BY country; --  Valid
```

Q4: What happens if you use GROUP BY without aggregate functions?

Answer: It works like DISTINCT - returns unique values:

```
-- These produce the same result:
SELECT DISTINCT country FROM customers;
SELECT country FROM customers GROUP BY country;
```

Q5: Can you GROUP BY an alias or expression?

Answer:

Database	GROUP BY Alias	GROUP BY Expression
SQL Server	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes

Database	GROUP BY Alias	GROUP BY Expression
MySQL	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
PostgreSQL	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes

-- SQL Server: Must repeat expression

```
SELECT YEAR(order_date) AS order_year, COUNT(*)
FROM orders
GROUP BY YEAR(order_date); -- Repeat, not alias
```

-- MySQL: Can use alias

```
SELECT YEAR(order_date) AS order_year, COUNT(*)
FROM orders
GROUP BY order_year; -- Alias works
```

Q6: How do aggregate functions handle NULL values?

Answer: All aggregate functions **ignore NULL** except COUNT(*):

-- Data: scores = [100, 200, NULL, 300]

```
SELECT
    COUNT(*) AS total_rows,          -- 4 (includes NULL row)
    COUNT(score) AS non_null,        -- 3 (excludes NULL)
    SUM(score) AS total,            -- 600 (ignores NULL)
    AVG(score) AS average;         -- 200 (600/3, not 600/4!)
```

⚠️ **AVG Warning:** AVG ignores NULLs, so average of [100, NULL, 200] = 150, not 100!

Q7: What is the execution order with GROUP BY?

Answer:

1. FROM → Get table data
2. WHERE → Filter rows (no aggregates allowed)
3. GROUP BY → Create groups
4. HAVING → Filter groups (aggregates allowed)
5. SELECT → Calculate aggregates, pick columns
6. ORDER BY → Sort results

Q8: Can you ORDER BY an aggregate not in SELECT?

Answer: Yes in most databases:

```
SELECT country
FROM customers
GROUP BY country
ORDER BY SUM(score) DESC; -- ✓ Valid
```

Q9: What is GROUP BY ROLLUP / CUBE?

Answer: They create subtotals and grand totals:

```
-- ROLLUP: Hierarchical subtotals
SELECT country, city, SUM(score)
FROM customers
GROUP BY ROLLUP(country, city);
-- Returns: Each city, Each country total, Grand total

-- CUBE: All combinations of subtotals
SELECT country, city, SUM(score)
FROM customers
GROUP BY CUBE(country, city);
-- Returns: All possible subtotal combinations
```

🔑 Key Takeaways

1. **GROUP BY** combines rows with same values

2. Must use **aggregate functions** with GROUP BY
 3. Non-aggregated columns **must** be in GROUP BY
 4. Use **aliases** to name calculated columns
 5. **WHERE** filters before grouping
 6. Multiple columns create combination groups
-

Best Practices

Practice	Reason
Always use aliases for aggregates	Makes results readable
Group only by needed columns	More columns = more groups
Use WHERE to pre-filter	Better performance
Test with small data first	Verify grouping logic

Exercises

Exercise 1:

Count the number of orders per customer_id.

```
SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id;
```

Exercise 2:

Find total sales by product_id.

```
SELECT product_id, SUM(sales) AS total_sales  
FROM orders  
GROUP BY product_id;
```

Exercise 3:

Find the highest score in each country.

```
SELECT country, MAX(score) AS highest_score  
FROM customers  
GROUP BY country;
```

Exercise 4:

Count customers by country, only for customers with score > 100.

```
SELECT country, COUNT(*) AS customer_count  
FROM customers  
WHERE score > 100  
GROUP BY country;
```



Common Aggregate Function Examples

-- Count all rows

```
SELECT COUNT(*) FROM customers;
```

-- Count non-null values

```
SELECT COUNT(score) FROM customers;
```

-- Sum of values

```
SELECT SUM(score) FROM customers;
```

-- Average

```
SELECT AVG(score) FROM customers;
```

-- Maximum

```
SELECT MAX(score) FROM customers;
```

-- Minimum

```
SELECT MIN(score) FROM customers;
```

Note 7: HAVING Clause - Filtering Aggregated Data



Overview

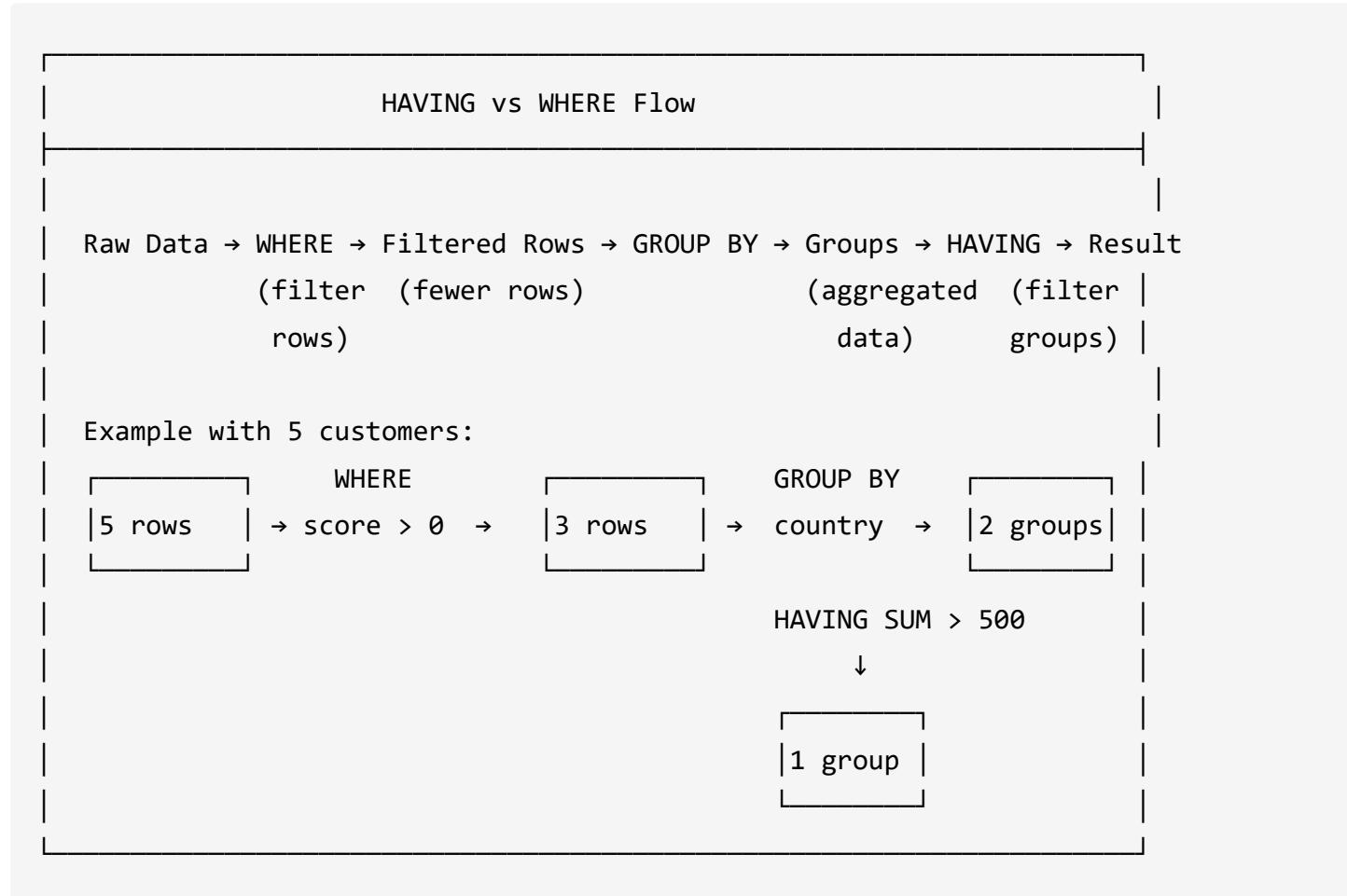
This note covers the HAVING clause, which allows you to filter data AFTER aggregation. While WHERE filters before grouping, HAVING filters the grouped results.



Theory

What is HAVING?

HAVING is the WHERE clause for groups. It filters aggregated results:



The Filter Timing Difference

Aspect	WHERE	HAVING
When	Before GROUP BY	After GROUP BY
Filters	Individual rows	Aggregated groups
Can use	Column values only	Aggregate functions
Performance	Faster (fewer rows to aggregate)	Runs after expensive aggregation
Requires	Nothing	GROUP BY clause

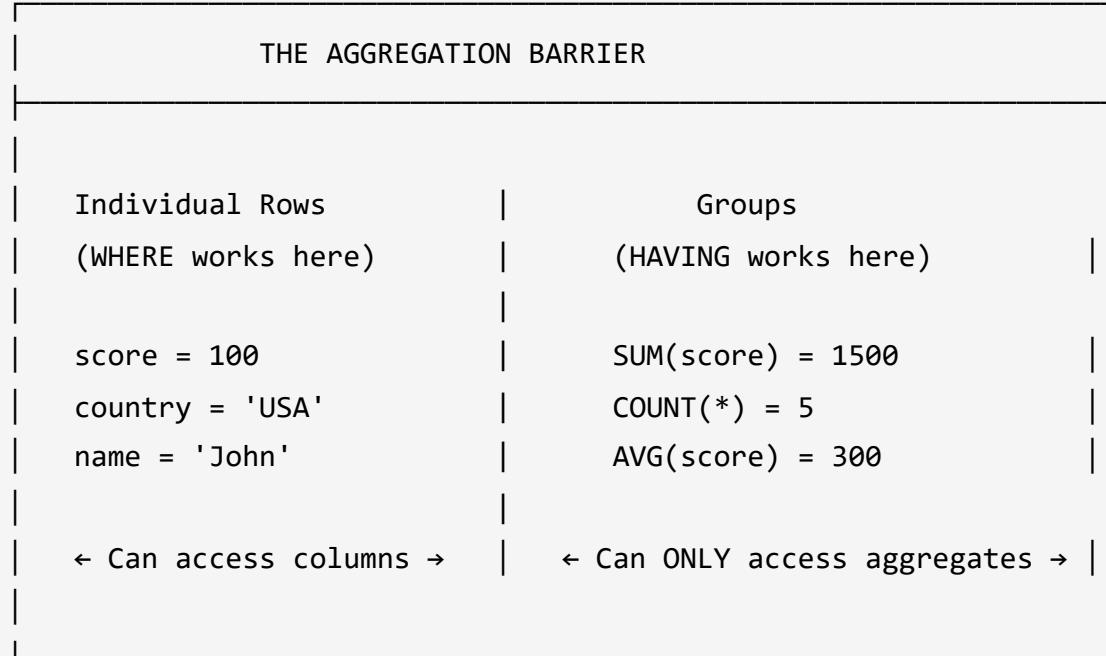
Why Two Filter Clauses?

Think of it like making a summary report:

1. **WHERE** = "Don't include these records in my report" (pre-filter)
2. **GROUP BY** = "Now summarize by category"
3. **HAVING** = "Only show categories that meet this criteria" (post-filter)

```
-- Real example: Sales report
SELECT region, SUM(amount) AS total_sales
FROM orders
WHERE status = 'completed'      -- Only count completed orders (row filter)
GROUP BY region
HAVING SUM(amount) > 10000;    -- Only show high-performing regions (group filter)
```

The Aggregation Barrier





Basic Syntax

```
SELECT column, AGGREGATE_FUNCTION(column)
FROM table_name
GROUP BY column
HAVING condition_on_aggregate;
```



Practical Examples

Example 1: Filter by Total Score

```
-- Show countries with total score > 800
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING SUM(score) > 800;

/* Result:
Germany      850
USA          900
*/
```

Example 2: Filter by Count

```
-- Show countries with more than 1 customer
SELECT country, COUNT(*) AS customer_count
FROM customers
GROUP BY country
HAVING COUNT(*) > 1;
```

Example 3: Filter by Average

```
-- Show countries with average score > 400
SELECT country, AVG(score) AS avg_score
FROM customers
GROUP BY country
HAVING AVG(score) > 400;
```



Execution Order

Step	Clause	Description
1	FROM	Get data
2	WHERE	Filter rows (before grouping)
3	GROUP BY	Create groups
4	HAVING	Filter groups (after aggregation)
5	SELECT	Select columns
6	ORDER BY	Sort results



Visual Comparison: WHERE vs HAVING

Original Data:

Maria	-	Germany	-	350
John	-	USA	-	900
George	-	UK	-	750
Martin	-	Germany	-	500
Peter	-	USA	-	0

Using WHERE (Before Grouping):

```
SELECT country, SUM(score) AS total
FROM customers
WHERE score > 400 -- Filters BEFORE grouping
GROUP BY country;

-- Only rows with score > 400 are grouped
-- Maria (350) excluded, Peter (0) excluded
```

Using HAVING (After Grouping):

```
SELECT country, SUM(score) AS total
FROM customers
GROUP BY country
HAVING SUM(score) > 800; -- Filters AFTER grouping

-- All rows are grouped first
-- Then groups with total <= 800 are removed
```



Combining WHERE and HAVING

You can use BOTH in the same query:

```
SELECT country, AVG(score) AS avg_score
FROM customers
WHERE score >> 0 -- Filter 1: Exclude zero scores
GROUP BY country
HAVING AVG(score) > 430; -- Filter 2: Only high averages
```

Execution Flow:

1. **FROM**: Get all customers
2. **WHERE**: Remove customers with score = 0
3. **GROUP BY**: Group remaining by country

4. **HAVING**: Keep only groups with avg > 430

5. **SELECT**: Show country and average



When to Use Which

Use WHERE When:

- Filtering on actual column values
- Condition is on non-aggregated data
- Want to exclude rows before aggregation

Use HAVING When:

- Filtering on aggregate results (SUM, COUNT, AVG, etc.)
 - Need to filter AFTER grouping
 - Condition involves aggregate function
-

Practice Questions

Q1: What is the main difference between WHERE and HAVING?

Answer:

- WHERE filters rows BEFORE aggregation
- HAVING filters groups AFTER aggregation

Q2: Write a query to show countries with total score > 500.

Answer:

```
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING SUM(score) > 500;
```

Q3: Can you use aggregate functions in WHERE clause?

Answer: No, aggregate functions can only be used in HAVING clause because WHERE executes before aggregation.

Q4: Write a query to find countries with more than 2 customers.

Answer:

```
SELECT country, COUNT(*) AS customer_count
FROM customers
GROUP BY country
HAVING COUNT(*) > 2;
```

Q5: Can you use both WHERE and HAVING in one query?

Answer: Yes! WHERE filters rows first, then GROUP BY aggregates, then HAVING filters groups.

Q6: Where does HAVING go in the query order?

Answer: HAVING comes after GROUP BY and before ORDER BY.



Interview Questions

Q1: Why can't you use aggregate functions in the WHERE clause?

Answer: Because of **execution order**. WHERE executes before GROUP BY, so:

- At WHERE stage, data is still individual rows
- Aggregates like SUM() don't exist yet
- You can't filter by something that hasn't been calculated

```
-- ✗ ERROR: Aggregate functions in WHERE
SELECT country, SUM(score)
FROM customers
WHERE SUM(score) > 500 -- SUM doesn't exist yet!
GROUP BY country;
```

```
-- ✓ CORRECT: Use HAVING
SELECT country, SUM(score)
FROM customers
GROUP BY country
HAVING SUM(score) > 500; -- SUM has been calculated
```

Q2: Can you use HAVING without GROUP BY?

Answer: Technically yes, but rarely useful. Without GROUP BY, the entire table is one group:

```
-- Without GROUP BY: entire table = one group
SELECT COUNT(*) AS total
FROM customers
HAVING COUNT(*) > 5;

-- Returns result if more than 5 customers, nothing otherwise
```

Q3: Can you use column aliases in HAVING?

Answer: It depends on the database:

Database	Alias in HAVING?
SQL Server	✗ No
MySQL	✓ Yes
PostgreSQL	✗ No
Oracle	✗ No

```
-- SQL Server: Must repeat aggregate
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING SUM(score) > 500; -- Must use SUM(), not alias
```

```
-- MySQL: Can use alias
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING total_score > 500; -- Alias works
```

Q4: Can HAVING filter on non-aggregated columns?

Answer: Only if the column is in GROUP BY:

```
--  Works: country is in GROUP BY
SELECT country, SUM(score)
FROM customers
GROUP BY country
HAVING country = 'USA'; -- Valid but WHERE is better

--  Error: first_name not in GROUP BY
SELECT country, SUM(score)
FROM customers
GROUP BY country
HAVING first_name = 'John'; -- Invalid!
```

Best Practice: Use WHERE for non-aggregate filters (better performance).

Q5: What's the performance difference between WHERE and HAVING?

Answer:

Filter	Performance	Why
WHERE	<input checked="" type="checkbox"/> Better	Filters BEFORE aggregation = fewer rows to process
HAVING	<input type="checkbox"/> Slower	Aggregates ALL rows first, then filters

```
--  Slow: Aggregates all rows, then filters
SELECT country, SUM(score)
FROM customers
GROUP BY country
HAVING country = 'USA';

--  Fast: Filters first, aggregates only matching rows
SELECT country, SUM(score)
FROM customers
WHERE country = 'USA'
GROUP BY country;
```

Q6: Can HAVING use multiple conditions?

Answer: Yes, with AND/OR:

```
SELECT country, COUNT(*) AS cnt, SUM(score) AS total
FROM customers
GROUP BY country
HAVING COUNT(*) > 2 AND SUM(score) > 1000
OR AVG(score) > 500;
```

Q7: Can you ORDER BY an aggregate that's filtered by HAVING?

Answer: Yes, they work together:

```
SELECT country, SUM(score) AS total
FROM customers
GROUP BY country
HAVING SUM(score) > 500
ORDER BY SUM(score) DESC; -- Sort surviving groups
```

Q8: In what scenarios would you use both WHERE and HAVING together?

Answer: When you need to:

1. Exclude certain rows from calculation (WHERE)
2. Filter the aggregated results (HAVING)

```
-- Example: Active customers only, show countries with >1000 sales
SELECT country, SUM(order_amount) AS sales
FROM customers
WHERE status = 'active'          -- Only count active customers
GROUP BY country
HAVING SUM(order_amount) > 1000; -- Only show high-volume countries
```

🔑 Key Takeaways

1. **HAVING** filters **after** aggregation
 2. **WHERE** filters **before** aggregation
 3. HAVING requires **GROUP BY**
 4. HAVING uses **aggregate functions**
 5. Can combine **WHERE** and **HAVING** in same query
 6. Think: WHERE → rows, HAVING → groups
-

📌 Decision Guide

Need to filter?

- └─ On original row data?
 - | → Use WHERE
 - └─ On aggregated results?
 - Use HAVING
-

🧪 Exercises

Exercise 1:

Show products with total sales > 1000.

```
SELECT product_id, SUM(sales) AS total_sales
FROM orders
GROUP BY product_id
HAVING SUM(sales) > 1000;
```

Exercise 2:

Find customers who placed more than 3 orders.

```
SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 3;
```

Exercise 3:

Show countries with average score > 300, excluding zero scores.

```
SELECT country, AVG(score) AS avg_score
FROM customers
WHERE score <> 0
GROUP BY country
HAVING AVG(score) > 300;
```

Exercise 4:

Find categories with at least 5 products.

```
SELECT category, COUNT(*) AS product_count
FROM products
GROUP BY category
HAVING COUNT(*) >= 5;
```

⚠ Common Mistakes

Mistake	Correction
Using aggregate in WHERE	Move to HAVING
HAVING without GROUP BY	HAVING requires GROUP BY
Using column alias in HAVING	Use the full aggregate function
Confusing execution order	Remember: WHERE → GROUP BY → HAVING

Alias Issue:

```
-- ❌ Wrong - can't use alias in HAVING
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING total_score > 800; -- ERROR!

-- ✅ Correct - use the aggregate function
SELECT country, SUM(score) AS total_score
FROM customers
GROUP BY country
HAVING SUM(score) > 800; -- Works!
```



Pro Tips

1. Use **WHERE** to reduce data BEFORE expensive aggregations
2. HAVING on large datasets can be slow - filter early with WHERE
3. Remember you can't use SELECT aliases in HAVING
4. Always test your conditions step by step

Note 8: DISTINCT - Removing Duplicates

Overview

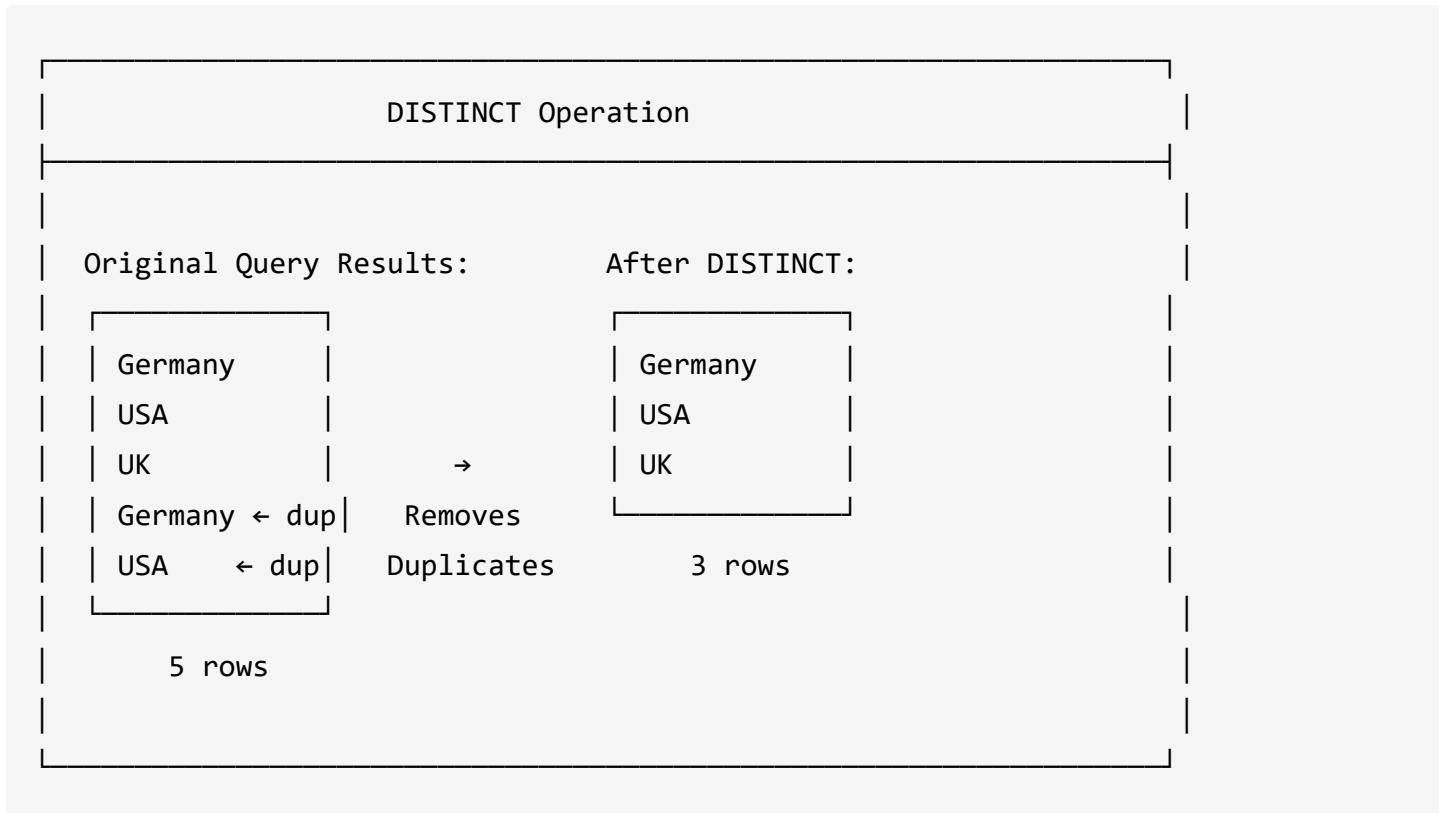
This note covers the DISTINCT keyword, which removes duplicate values from your query results, returning only unique values.



Theory

What is DISTINCT?

DISTINCT is a de-duplication operator that removes duplicate rows from your result set:



The DISTINCT Scope

DISTINCT applies to **ALL columns in SELECT** combined, not individual columns:

```
-- This finds unique (country, city) COMBINATIONS
SELECT DISTINCT country, city FROM customers;

-- NOT the same as this (which isn't valid SQL):
-- SELECT DISTINCT(country), DISTINCT(city) -- ✗ Invalid
```

DISTINCT vs ALL

```
SELECT ALL country FROM customers;      -- Default: shows all rows
SELECT DISTINCT country FROM customers; -- Shows unique values only

-- ALL is the default (rarely written explicitly)
```

How DISTINCT Works Internally

1. **Sort/Hash** the result set
2. **Compare** consecutive rows (or hash buckets)
3. **Keep first occurrence**, eliminate duplicates
4. Return unique rows

Method	When Used	Speed
Hash Distinct	Unsorted data	Fast for small sets
Sort Distinct	Large data or with ORDER BY	Uses tempdb
Stream Distinct	Already sorted data	Very fast

When to Use DISTINCT

- Get unique values from a column with duplicates
- Find distinct combinations of values
- Create dropdown/filter lists
- Don't use on primary keys (already unique)
- Don't use to "fix" a bad JOIN (investigate instead)



Basic Syntax

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

Position in Query:

```
SELECT DISTINCT columns -- DISTINCT comes right after SELECT
FROM table_name
WHERE condition
ORDER BY column;
```



Practical Examples

Example 1: Unique Countries

```
-- Get list of unique countries
SELECT DISTINCT country
FROM customers;

/* Result:
Germany
UK
USA
*/
```

Example 2: Without DISTINCT (Shows Duplicates)

```
SELECT country
FROM customers;

/* Result:
Germany
USA
UK
Germany
USA
*/
```

Example 3: Multiple Columns

```
-- Unique combinations of country and score  
SELECT DISTINCT country, score  
FROM customers;  
  
/* Each unique combination appears once */
```



How DISTINCT Works

Visual Process:

Original Data:	After DISTINCT:
Germany	Germany
USA	UK
UK	→ USA
Germany	(removed)
USA	(removed)

Step by Step:

1. SQL reads all rows
2. Checks if value already exists in results
3. If exists → Skip (don't add)
4. If new → Add to results



DISTINCT with Multiple Columns

When using DISTINCT with multiple columns, SQL looks for unique **combinations**:

```
SELECT DISTINCT country, first_name  
FROM customers;
```

country	first_name	Included?
Germany	Maria	<input checked="" type="checkbox"/> First time
USA	John	<input checked="" type="checkbox"/> First time
UK	George	<input checked="" type="checkbox"/> First time
Germany	Martin	<input checked="" type="checkbox"/> Different combo
USA	Peter	<input checked="" type="checkbox"/> Different combo

All 5 rows kept because each country + name combination is unique!

⚠️ Important Considerations

Performance Impact

```
-- DISTINCT requires additional processing
-- Database must check for duplicates
-- Can be slow on large datasets
```

Don't Use DISTINCT Unnecessarily

```
-- ❌ Unnecessary - IDs are already unique
SELECT DISTINCT id FROM customers;

-- ✅ Better - no DISTINCT needed
SELECT id FROM customers;
```

When DISTINCT is Useful

-- Good use - column has duplicates

```
SELECT DISTINCT country FROM customers;
```

-- Good use - finding unique categories

```
SELECT DISTINCT category FROM products;
```



DISTINCT vs GROUP BY

Both can return unique values, but they serve different purposes:

-- Using DISTINCT

```
SELECT DISTINCT country  
FROM customers;
```

-- Using GROUP BY (same result for unique values)

```
SELECT country  
FROM customers  
GROUP BY country;
```

DISTINCT	GROUP BY
Just removes duplicates	Groups for aggregation
Simpler syntax	Allows aggregate functions
Cannot add aggregates	Can include SUM, COUNT, etc.

❓ Practice Questions

Q1: What does DISTINCT do?

Answer: DISTINCT removes duplicate rows from the result set, returning only unique values.

Q2: Where is DISTINCT placed in a query?

Answer: Directly after SELECT: `SELECT DISTINCT column FROM table;`

Q3: Write a query to get unique countries from customers.

Answer:

```
SELECT DISTINCT country  
FROM customers;
```

Q4: Does DISTINCT work on multiple columns?

Answer: Yes, it returns unique combinations of all specified columns.

Q5: Why shouldn't you always use DISTINCT?

Answer:

- It has performance overhead
- If data is already unique (like IDs), it's unnecessary
- It can mask data quality issues

Q6: What's the difference between DISTINCT and GROUP BY?

Answer:

- DISTINCT: Simply removes duplicates

- GROUP BY: Groups data for aggregation (SUM, COUNT, etc.)

Interview Questions

Q1: What is the difference between DISTINCT and GROUP BY for getting unique values?

Answer:

Aspect	DISTINCT	GROUP BY
Purpose	Remove duplicates	Group for aggregation
Syntax	Simpler	More verbose
Aggregates	✗ Cannot use	✓ Can use SUM, COUNT, etc.
Performance	Generally similar	Same optimization

-- Same result:

```
SELECT DISTINCT country FROM customers;
SELECT country FROM customers GROUP BY country;
```

-- Only GROUP BY can do this:

```
SELECT country, SUM(score) FROM customers GROUP BY country;
```

Q2: How does DISTINCT handle NULL values?

Answer: DISTINCT treats all NULLs as **equal** (one NULL value in result):

```
-- Data: countries = [USA, NULL, UK, NULL, USA]
SELECT DISTINCT country FROM customers;
-- Result: USA, UK, NULL (only one NULL)
```

Q3: Does DISTINCT affect performance?

Answer: Yes, it adds overhead:

Operation	Impact
Sorting/Hashing	CPU and memory usage
Comparison	Extra processing
Large datasets	May spill to disk (tempdb)

```
-- Check execution plan difference:
```

```
SELECT country FROM customers;           -- No distinct = faster
SELECT DISTINCT country FROM customers;  -- Adds Sort/Hash operation
```

Best Practice: Only use DISTINCT when duplicates actually exist.

Q4: Can you use DISTINCT with aggregate functions?

Answer: Yes! Two ways:

```
-- 1. COUNT(DISTINCT column) - count unique values
```

```
SELECT COUNT(DISTINCT country) AS unique_countries
FROM customers;
```

```
-- 2. DISTINCT on aggregated results (less common)
```

```
SELECT DISTINCT COUNT(*) -- Unique counts
FROM orders
GROUP BY customer_id;
```

Q5: Why does adding DISTINCT sometimes hide bugs?

Answer: It can mask problems like:

- **Bad JOINs** creating duplicate rows
- **Missing GROUP BY** causing unintended repetition
- **Data quality issues** with actual duplicates

```
-- ❌ Wrong: JOIN creates duplicates, DISTINCT hides the problem
SELECT DISTINCT c.name
FROM customers c
JOIN orders o ON c.id = o.customer_id; -- One customer, many orders

-- ✅ Better: Investigate why duplicates exist
SELECT c.name, COUNT(*) AS times_repeated
FROM customers c
JOIN orders o ON c.id = o.customer_id
GROUP BY c.name;
```

Q6: Can you ORDER BY a column not in SELECT DISTINCT?

Answer: No! With DISTINCT, ORDER BY can only use columns in SELECT:

```
-- ❌ Error: score not in SELECT
SELECT DISTINCT country
FROM customers
ORDER BY score;

-- ✅ Works: country is in SELECT
SELECT DISTINCT country
FROM customers
ORDER BY country;

-- ✅ Works: include score in SELECT
SELECT DISTINCT country, score
FROM customers
ORDER BY score;
```

Q7: What is DISTINCT ON (PostgreSQL specific)?

Answer: PostgreSQL allows selecting first row per group:

```
-- PostgreSQL only: Get first customer per country
SELECT DISTINCT ON (country) country, first_name, score
FROM customers
ORDER BY country, score DESC;

-- SQL Server equivalent uses ROW_NUMBER():
SELECT country, first_name, score
FROM (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY country ORDER BY score DESC) AS rn
    FROM customers
) t
WHERE rn = 1;
```

Q8: Does DISTINCT consider all columns for uniqueness?

Answer: Yes, DISTINCT evaluates the **entire row** across all selected columns:

```
SELECT DISTINCT country, city FROM customers;
-- (USA, New York) and (USA, Los Angeles) are both kept
-- because the COMBINATION is different
```

Q9: When should you use EXISTS instead of DISTINCT?

Answer: When checking for existence in JOINs, EXISTS is often better:

```
-- ❌ Slow: Gets all matches, then deduplicates
SELECT DISTINCT c.name
FROM customers c
JOIN orders o ON c.id = o.customer_id;

-- ✅ Faster: Stops at first match
SELECT c.name
FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id);
```



Key Takeaways

1. **DISTINCT** removes duplicate values
 2. Place **after SELECT**, before columns
 3. Works on **all specified columns** combined
 4. Has **performance cost** - use when needed
 5. Don't use on **already unique** columns (like IDs)
 6. For aggregation, prefer **GROUP BY**
-



Best Practices

Practice	Reason
Use only when needed	Performance impact
Check if data is naturally unique	Avoid unnecessary DISTINCT
Don't use to hide data issues	Investigate duplicates
Consider GROUP BY for aggregation	More appropriate tool



Exercises

Exercise 1:

Get a unique list of product categories.

```
SELECT DISTINCT category
FROM products;
```

Exercise 2:

Find all unique country and city combinations.

```
SELECT DISTINCT country, city  
FROM customers;
```

Exercise 3:

Count how many unique countries we have.

```
SELECT COUNT(DISTINCT country) AS unique_countries  
FROM customers;
```

Exercise 4:

Get unique order dates.

```
SELECT DISTINCT order_date  
FROM orders  
ORDER BY order_date;
```



COUNT with DISTINCT

You can combine COUNT with DISTINCT:

```
-- Count unique countries  
SELECT COUNT(DISTINCT country) AS unique_countries  
FROM customers;  
  
-- Compare with total count  
SELECT  
    COUNT(*) AS total_rows,  
    COUNT(DISTINCT country) AS unique_countries  
FROM customers;
```



Common Mistakes

Mistake	Correction
Using DISTINCT on IDs	IDs are unique, no need
DISTINCT after column name	Must be right after SELECT
Expecting DISTINCT per column	Works on all columns combined
Overusing DISTINCT	Only use when duplicates exist

Wrong Placement:

-- Wrong

```
SELECT country DISTINCT FROM customers;
```

-- Correct

```
SELECT DISTINCT country FROM customers;
```

Note 9: TOP/LIMIT - Limiting Results



Overview

This note covers the TOP clause (SQL Server) or LIMIT (MySQL, PostgreSQL), which restricts the number of rows returned by a query.

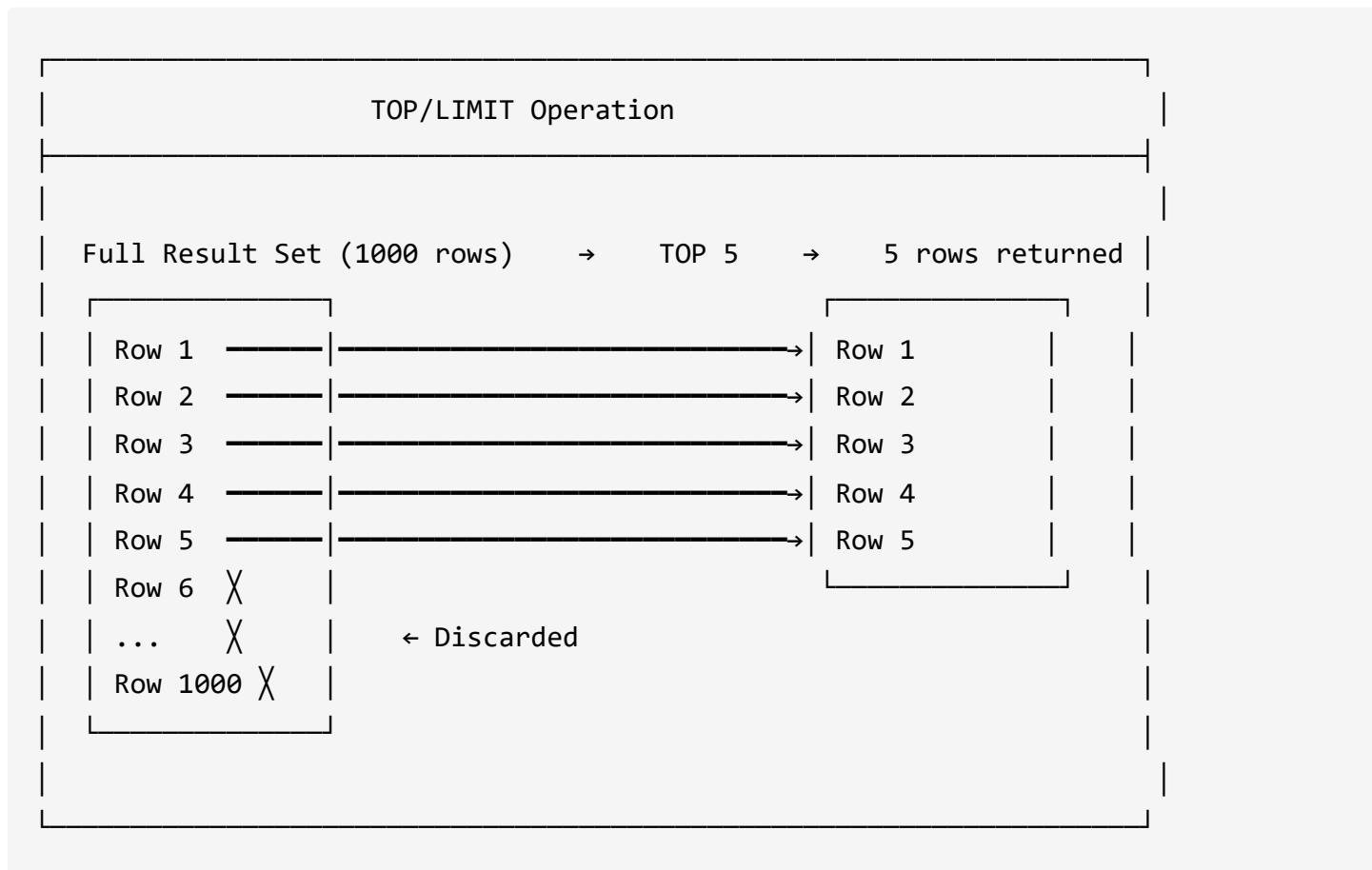


Theory

What is TOP/LIMIT?

TOP/LIMIT restricts the number of rows returned by a query. It's essential for:

- Pagination (showing page 1 of results)
- Top-N analysis (best, worst, most recent)
- Performance (previewing large tables)
- Sampling data for testing



⚠ The ORDER BY Requirement

TOP/LIMIT without ORDER BY = Unpredictable Results!

-- **✗** Arbitrary rows (SQL picks any 3 rows)

```
SELECT TOP 3 * FROM customers;
```

-- **✓** Deterministic (highest 3 scores)

```
SELECT TOP 3 * FROM customers ORDER BY score DESC;
```

Why? Without ORDER BY, SQL returns rows in whatever order it finds them (physical order, index order, etc.) - this can change between executions!

Database Syntax Comparison

Database	Syntax	Position	Notes
SQL Server	TOP n	After SELECT	SELECT TOP 5 *
MySQL	LIMIT n	End of query	SELECT * ... LIMIT 5
PostgreSQL	LIMIT n	End of query	SELECT * ... LIMIT 5
Oracle	FETCH FIRST n ROWS	After ORDER BY	FETCH FIRST 5 ROWS ONLY
ANSI SQL	FETCH FIRST n ROWS	After ORDER BY	Standard syntax

TOP Variants in SQL Server

```
-- Fixed number of rows
SELECT TOP 5 * FROM customers;

-- Percentage of rows
SELECT TOP 10 PERCENT * FROM customers;

-- With ties (includes all rows with same value as last row)
SELECT TOP 3 WITH TIES * FROM customers ORDER BY score DESC;
```

OFFSET-FETCH for Pagination (ANSI SQL)

```
-- Skip first 10 rows, get next 5 (rows 11-15)
SELECT * FROM customers
ORDER BY score DESC
OFFSET 10 ROWS
FETCH NEXT 5 ROWS ONLY;

-- MySQL equivalent
SELECT * FROM customers
ORDER BY score DESC
LIMIT 5 OFFSET 10;
```



Basic Syntax

SQL Server (TOP)

```
SELECT TOP n column1, column2  
FROM table_name;
```

MySQL/PostgreSQL (LIMIT)

```
SELECT column1, column2  
FROM table_name  
LIMIT n;
```



Practical Examples

Example 1: Get First 3 Rows

```
-- SQL Server  
SELECT TOP 3 *  
FROM customers;  
  
-- MySQL/PostgreSQL  
SELECT *  
FROM customers  
LIMIT 3;
```

Example 2: TOP with ORDER BY (Most Common)

```
-- Get top 3 highest scores
SELECT TOP 3 *
FROM customers
ORDER BY score DESC;
```

Example 3: Bottom N Records

```
-- Get 2 lowest scoring customers
SELECT TOP 2 *
FROM customers
ORDER BY score ASC;
```

Example 4: Most Recent Orders

```
-- Get 5 most recent orders
SELECT TOP 5 *
FROM orders
ORDER BY order_date DESC;
```



Execution Order

Step	Clause	Action
1	FROM	Get data
2	WHERE	Filter rows
3	GROUP BY	Aggregate
4	HAVING	Filter groups
5	SELECT	Choose columns
6	ORDER BY	Sort results

Step	Clause	Action
7	TOP/LIMIT	Limit rows (LAST!)

 **Important:** TOP executes LAST, after sorting!

TOP + ORDER BY Combinations

Top N Analysis

```
-- Top 5 customers by score
SELECT TOP 5 first_name, score
FROM customers
ORDER BY score DESC;
```

Bottom N Analysis

```
-- Bottom 5 customers by score
SELECT TOP 5 first_name, score
FROM customers
ORDER BY score ASC;
```

Most Recent Records

```
-- Latest 10 orders
SELECT TOP 10 *
FROM orders
ORDER BY order_date DESC;
```

Oldest Records

```
-- First 10 orders ever
SELECT TOP 10 *
FROM orders
ORDER BY order_date ASC;
```

Using TOP PERCENT

Percentage of Rows

```
-- Get top 20% of customers by score
SELECT TOP 20 PERCENT *
FROM customers
ORDER BY score DESC;
```

Practical Use Cases

1. Quick Data Preview

```
-- Just see a few rows quickly
SELECT TOP 5 * FROM large_table;
```

2. Performance Optimization

```
-- Testing queries on large tables
SELECT TOP 100 * FROM millions_of_rows;
```

3. Top Performers

```
-- Best selling products
SELECT TOP 10 product_name, SUM(sales) AS total_sales
FROM orders
GROUP BY product_name
ORDER BY total_sales DESC;
```

4. Recent Activity

```
-- Latest customer orders
SELECT TOP 5 customer_id, order_date, sales
FROM orders
ORDER BY order_date DESC;
```

❓ Practice Questions

Q1: What does TOP do?

Answer: TOP limits the number of rows returned in the query results.

Q2: Write a query to get the top 3 customers by score.

Answer:

```
SELECT TOP 3 *
FROM customers
ORDER BY score DESC;
```

Q3: Why is ORDER BY important with TOP?

Answer: Without ORDER BY, the rows returned are arbitrary. ORDER BY ensures you get the "top" based on specific criteria.

Q4: Get the 2 most recent orders.

Answer:

```
SELECT TOP 2 *
FROM orders
ORDER BY order_date DESC;
```

Q5: When does TOP execute in the query order?

Answer: TOP executes LAST, after all other clauses including ORDER BY.

Q6: Write a query for the 5 lowest scoring customers.

Answer:

```
SELECT TOP 5 *
FROM customers
ORDER BY score ASC;
```

Interview Questions

Q1: What happens if you use TOP without ORDER BY?

Answer: The results are **non-deterministic** - SQL returns arbitrary rows based on internal factors like:

- Physical row order on disk
- Index access order
- Parallel processing
- Query plan changes

-- X Unpredictable: might return different rows each time

```
SELECT TOP 5 * FROM customers;
```

-- ✓ Predictable: always returns highest 5 scores

```
SELECT TOP 5 * FROM customers ORDER BY score DESC;
```

Q2: What is the difference between TOP and TOP WITH TIES?

Answer:

TOP	TOP WITH TIES
Returns exactly N rows	Returns N rows + ties at Nth position
Arbitrary cutoff on ties	Fair handling of equal values

-- Data: scores = [100, 100, 90, 90, 80]

```
SELECT TOP 3 score FROM customers ORDER BY score DESC;
```

-- Returns: 100, 100, 90 (exactly 3)

```
SELECT TOP 3 WITH TIES score FROM customers ORDER BY score DESC;
```

-- Returns: 100, 100, 90, 90 (4 rows - includes tie at 3rd)

Q3: How do you implement pagination with TOP/OFFSET?

Answer:

```
-- SQL Server (OFFSET-FETCH)
-- Page 1 (rows 1-10)
SELECT * FROM customers ORDER BY id OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;

-- Page 2 (rows 11-20)
SELECT * FROM customers ORDER BY id OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;

-- Page 3 (rows 21-30)
SELECT * FROM customers ORDER BY id OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY;

-- MySQL pagination
SELECT * FROM customers ORDER BY id LIMIT 10 OFFSET 20; -- Page 3
```

Q4: Is TOP faster than fetching all rows?

Answer: It depends on whether ORDER BY is present:

```
--  Fast: No ORDER BY, stops after 5 rows found
SELECT TOP 5 * FROM customers;

--  May be slow: Must sort ALL rows first, then take 5
SELECT TOP 5 * FROM customers ORDER BY score DESC;

--  Fast with ORDER BY: If index on score exists
SELECT TOP 5 * FROM customers ORDER BY score DESC; -- Index seek
```

Key Insight: With ORDER BY, SQL must determine the order of ALL rows before knowing which 5 are "top".

Q5: How do you get rows 11-20 (skip first 10)?

Answer:

```
-- SQL Server
SELECT * FROM customers
ORDER BY id
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;

-- MySQL
SELECT * FROM customers ORDER BY id LIMIT 10 OFFSET 10;
-- OR
SELECT * FROM customers ORDER BY id LIMIT 10, 10; -- LIMIT offset, count

-- Old SQL Server (pre-2012)
SELECT * FROM (
    SELECT *, ROW_NUMBER() OVER (ORDER BY id) AS rn
    FROM customers
) t
WHERE rn BETWEEN 11 AND 20;
```

Q6: Can you use TOP with GROUP BY?

Answer: Yes! TOP applies to the aggregated results:

```
-- Top 3 countries by total score
SELECT TOP 3 country, SUM(score) AS total_score
FROM customers
GROUP BY country
ORDER BY total_score DESC;
```

Q7: What is TOP PERCENT?

Answer: Returns a percentage of total rows:

```
-- Top 20% of customers by score
SELECT TOP 20 PERCENT *
FROM customers
ORDER BY score DESC;

-- If 100 customers exist, returns 20 rows
-- If 50 customers exist, returns 10 rows
```

Note: Result is always rounded UP (11 rows from 55 at 20%).

Q8: How does TOP work with UNION?

Answer: TOP can be applied at different levels:

```
-- TOP on each individual query
SELECT TOP 2 name FROM customers
UNION ALL
SELECT TOP 2 name FROM suppliers;
-- Returns: up to 4 rows (2 + 2)

-- TOP on combined results
SELECT TOP 2 name FROM (
    SELECT name FROM customers
    UNION ALL
    SELECT name FROM suppliers
) combined;
-- Returns: exactly 2 rows from combined set
```

Q9: What is the ANSI SQL standard syntax for TOP?

Answer: `FETCH FIRST n ROWS ONLY`:

```
-- ANSI Standard (works in SQL Server 2012+, Oracle, PostgreSQL)
SELECT * FROM customers
ORDER BY score DESC
FETCH FIRST 5 ROWS ONLY;

-- With OFFSET (skip + limit)
SELECT * FROM customers
ORDER BY score DESC
OFFSET 10 ROWS
FETCH NEXT 5 ROWS ONLY;
```

🔑 Key Takeaways

1. **TOP** limits number of returned rows
 2. Place after **SELECT** keyword
 3. **Always use with ORDER BY** for meaningful results
 4. Executes **LAST** in query order
 5. Use **PERCENT** for percentage-based limits
 6. Great for **performance** when testing
-

📌 Best Practices

Practice	Reason
Always include ORDER BY	Without it, results are random
Use for testing queries	Faster on large tables
Combine with aggregation	For "top N" analysis
Avoid TOP without ORDER BY	Results unpredictable

💡 Exercises

Exercise 1:

Get the first 5 customers.

```
SELECT TOP 5 *
FROM customers;
```

Exercise 2:

Get the top 3 products by price.

```
SELECT TOP 3 *
FROM products
ORDER BY price DESC;
```

Exercise 3:

Get the 2 most recent orders for a specific customer.

```
SELECT TOP 2 *
FROM orders
WHERE customer_id = 1
ORDER BY order_date DESC;
```

Exercise 4:

Get top 10% of orders by sales amount.

```
SELECT TOP 10 PERCENT *
FROM orders
ORDER BY sales DESC;
```



TOP with Ties

WITH TIES Option

If multiple rows have the same value, you might want all of them:

```
-- Include all ties for the last position
SELECT TOP 3 WITH TIES *
FROM customers
ORDER BY score DESC;

-- If 3rd and 4th have same score, both included
```

⚠ Common Mistakes

Mistake	Correction
TOP without ORDER BY	Results are meaningless
Forgetting TOP position	Must be after SELECT
Using in wrong database	LIMIT for MySQL, TOP for SQL Server
Expecting TOP before ORDER BY	TOP applies after sorting

Wrong Order:

-- Wrong

```
SELECT * TOP 5 FROM customers;
```

-- Correct

```
SELECT TOP 5 * FROM customers;
```

🔧 Database-Specific Syntax

SQL Server

```
SELECT TOP 10 * FROM table_name ORDER BY column DESC;
```

MySQL

```
SELECT * FROM table_name ORDER BY column DESC LIMIT 10;
```

PostgreSQL

```
SELECT * FROM table_name ORDER BY column DESC LIMIT 10;
```

Oracle

```
SELECT * FROM table_name ORDER BY column DESC FETCH FIRST 10 ROWS ONLY;
```

Note 10: Query Execution Order

Overview

This note explains the difference between SQL writing order and execution order - one of the most important concepts for understanding how SQL queries actually work.

Theory

Two Orders to Understand

1. **Coding Order** (Syntax Order) - How you write the query
2. **Execution Order** (Logical Order) - How SQL Server processes it

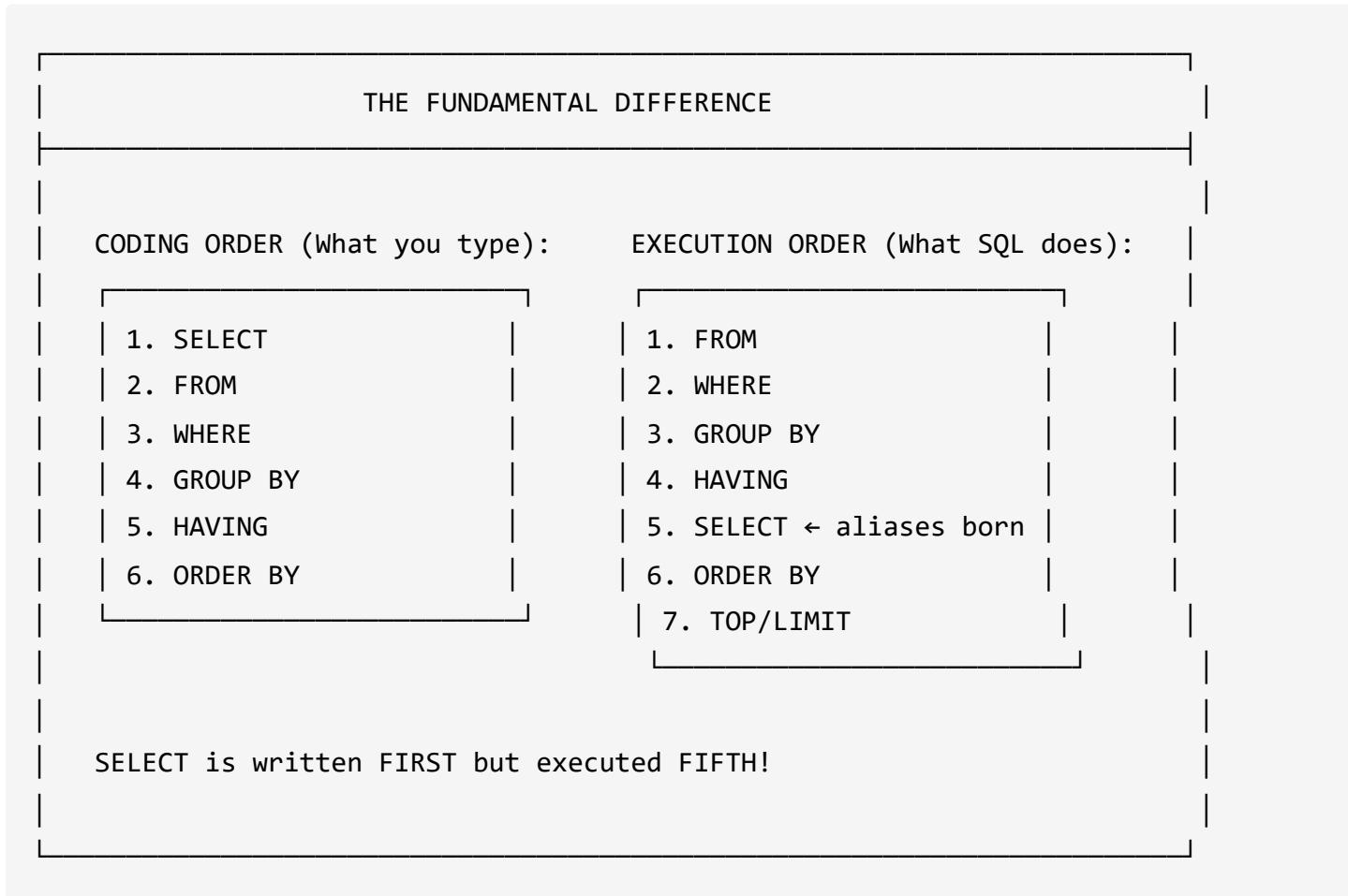
 **Key Insight:** SQL is a **declarative** language - you tell it WHAT you want, not HOW to get it. The engine decides the execution path!

Why This Matters

Understanding execution order explains:

- **Why** certain errors occur (alias not found, aggregate not allowed)
- **What** you can reference at each stage
- **How** to optimize queries

- **When** filters and transformations happen



Logical vs Physical Execution

Aspect	Logical Order	Physical Order
What it is	Conceptual processing sequence	Actual operations by query engine
Determined by	SQL standard	Query optimizer
Visible to you	Yes (affects what you can reference)	No (internal optimization)
Changes	Never (fixed rules)	Every query (optimizer chooses best)

The optimizer may reorder operations, use parallel processing, or skip steps entirely - but it **MUST** produce results as if it followed the logical order!

The Scope Visibility Rule

At each execution step, you can only reference:

1. Items from **previous** steps
2. Items from the **current** step
3. **NOT** items from **future** steps

```
Step 1 (FROM)      → Can see: Table columns
Step 2 (WHERE)     → Can see: Table columns (NOT aliases)
Step 3 (GROUP BY)  → Can see: Table columns (NOT aliases)
Step 4 (HAVING)    → Can see: Table columns + aggregates
Step 5 (SELECT)    → Creates aliases HERE
Step 6 (ORDER BY)  → Can see: Aliases + table columns
Step 7 (TOP)       → Works on final sorted set
```



Coding Order vs Execution Order

How We Write (Coding Order)

```
SELECT column1, column2      -- 1st written
      FROM table_name        -- 2nd written
      WHERE condition         -- 3rd written
      GROUP BY column         -- 4th written
      HAVING aggregate_condition -- 5th written
      ORDER BY column         -- 6th written
```

How SQL Executes (Execution Order)

```

FROM table_name           -- 1st executed
WHERE condition          -- 2nd executed
GROUP BY column          -- 3rd executed
HAVING aggregate_condition -- 4th executed
SELECT column1, column2    -- 5th executed
ORDER BY column          -- 6th executed
TOP/LIMIT                -- 7th executed (LAST!)

```



Detailed Execution Order

Step	Clause	What It Does
1	FROM	Identifies the table(s)
2	WHERE	Filters individual rows
3	GROUP BY	Groups rows together
4	HAVING	Filters groups
5	SELECT	Chooses columns
6	ORDER BY	Sorts the result
7	TOP/LIMIT	Limits rows returned



Visual Example

Query:

```
SELECT department, SUM(salary) AS total_salary
FROM employees
WHERE status = 'Active'
GROUP BY department
HAVING SUM(salary) > 50000
ORDER BY total_salary DESC
```

Execution Steps:

Step 1 - FROM employees
→ Gets all rows from employees table

Step 2 - WHERE status = 'Active'
→ Keeps only active employees

Step 3 - GROUP BY department
→ Groups rows by department

Step 4 - HAVING SUM(salary) > 50000
→ Keeps groups with total > 50000

Step 5 - SELECT department, SUM(salary)
→ Shows only requested columns

Step 6 - ORDER BY total_salary DESC
→ Sorts by total salary descending

🎯 Why Does Order Matter?

1. Alias Availability

```
-- ❌ This FAILS (alias not available in WHERE)
SELECT first_name, salary * 12 AS annual_salary
FROM employees
WHERE annual_salary > 50000; -- Error!
```

```
-- ✅ This WORKS (alias available in ORDER BY)
SELECT first_name, salary * 12 AS annual_salary
FROM employees
ORDER BY annual_salary DESC; -- Works!
```

Why?

- WHERE executes BEFORE SELECT (alias doesn't exist yet)
- ORDER BY executes AFTER SELECT (alias exists)

2. Aggregate in WHERE

```
-- ❌ This FAILS
SELECT department, COUNT(*) AS emp_count
FROM employees
WHERE COUNT(*) > 5 -- Error! Can't use aggregate
GROUP BY department;
```

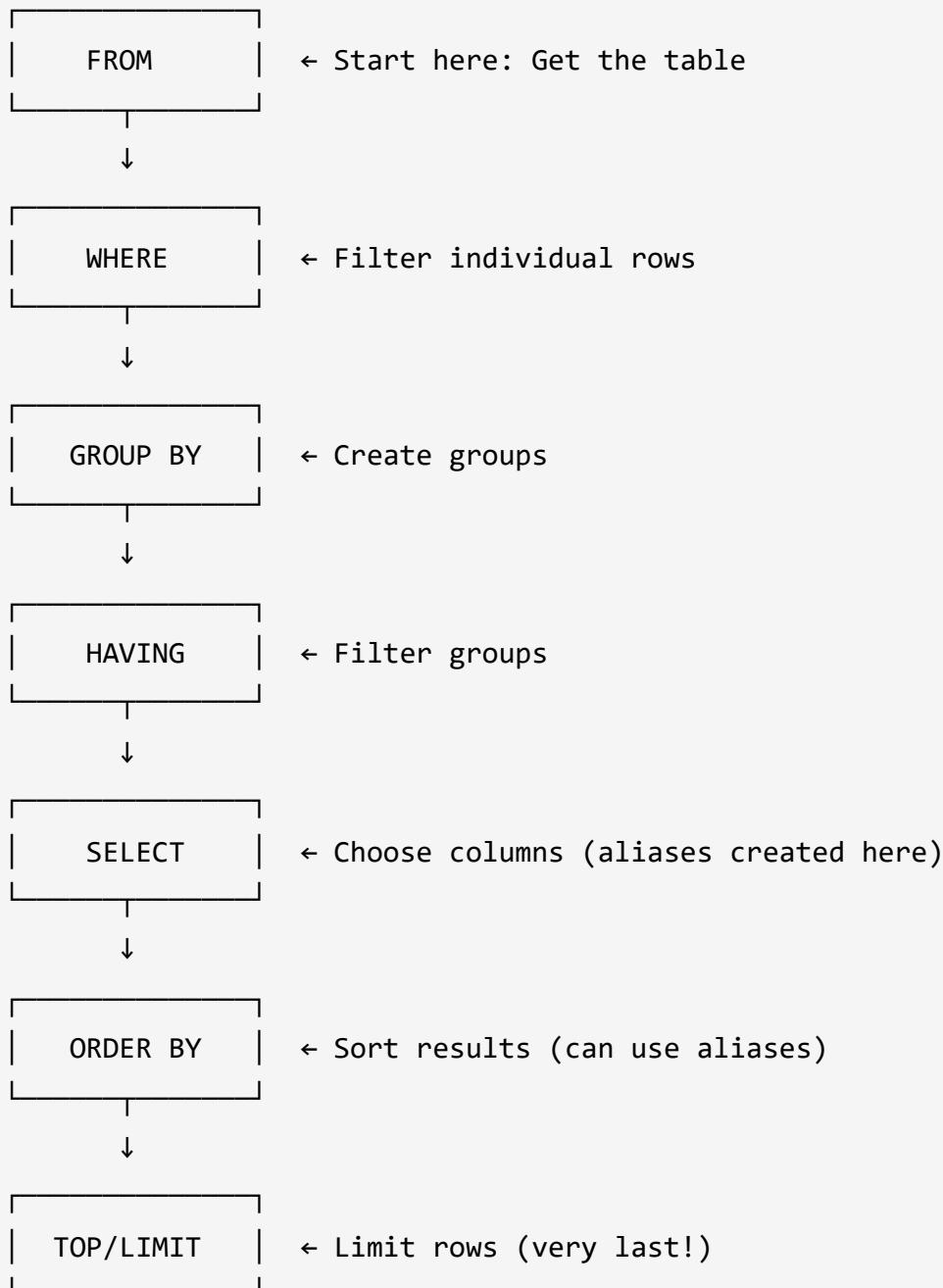
```
-- ✅ This WORKS
SELECT department, COUNT(*) AS emp_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 5; -- Correct! Use HAVING
```

Why?

- WHERE filters BEFORE grouping (no aggregate exists)
- HAVING filters AFTER grouping (aggregate is calculated)



Execution Order Diagram





Practical Implications

Where Aliases Work

Clause	Can Use Alias?	Reason
FROM	<input checked="" type="checkbox"/> No	Executes before SELECT
WHERE	<input checked="" type="checkbox"/> No	Executes before SELECT
GROUP BY	<input checked="" type="checkbox"/> No	Executes before SELECT
HAVING	<input checked="" type="checkbox"/> No	Executes before SELECT
SELECT	<input checked="" type="checkbox"/> Creating	This is where aliases are defined
ORDER BY	<input checked="" type="checkbox"/> Yes	Executes after SELECT

Where Aggregates Work

Clause	Can Use Aggregate?
WHERE	<input checked="" type="checkbox"/> No (use HAVING)
HAVING	<input checked="" type="checkbox"/> Yes
SELECT	<input checked="" type="checkbox"/> Yes
ORDER BY	<input checked="" type="checkbox"/> Yes

? Practice Questions

Q1: What executes first - FROM or SELECT?

Answer: FROM executes first. SQL needs to know which table to work with before it can select columns.

Q2: Why can't you use an alias in WHERE?

Answer: Because WHERE executes before SELECT, and aliases are created in SELECT. The alias doesn't exist yet when WHERE runs.

Q3: Why can you use aliases in ORDER BY?

Answer: Because ORDER BY executes after SELECT, so the alias has already been created.

Q4: Why can't aggregates be used in WHERE?

Answer: WHERE filters individual rows before grouping. Aggregates only exist after GROUP BY creates groups.

Q5: Write the correct execution order.

Answer:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY
7. TOP/LIMIT

Q6: Which executes last - ORDER BY or TOP?

Answer: TOP executes last, after ORDER BY.



Interview Questions

Q1: Why does this query fail?

```
SELECT first_name, salary * 12 AS annual_salary
FROM employees
WHERE annual_salary > 50000;
```

Answer: The alias `annual_salary` is created in SELECT, which executes **after** WHERE. When WHERE runs, the alias doesn't exist yet.

Fix:

```
WHERE salary * 12 > 50000 -- Repeat the expression
-- OR use a subquery/CTE
```

Q2: Can you GROUP BY an alias? Why or why not?

Answer:

Database	GROUP BY Alias?	Reason
SQL Server	✗ No	GROUP BY executes before SELECT
MySQL	✓ Yes	MySQL extends the standard
PostgreSQL	✗ No	Follows standard
Oracle	✗ No	Follows standard

```
-- SQL Server: Must repeat expression
SELECT YEAR(order_date) AS order_year, COUNT(*)
FROM orders
GROUP BY YEAR(order_date); -- Cannot use alias
```

```
-- MySQL: Can use alias
GROUP BY order_year; -- Works
```

Q3: Why can ORDER BY use columns NOT in SELECT?

Answer: Because ORDER BY has access to the entire row context from FROM (after SELECT projection happens conceptually, but before the final result is materialized). However, with DISTINCT, only SELECT columns can be used.

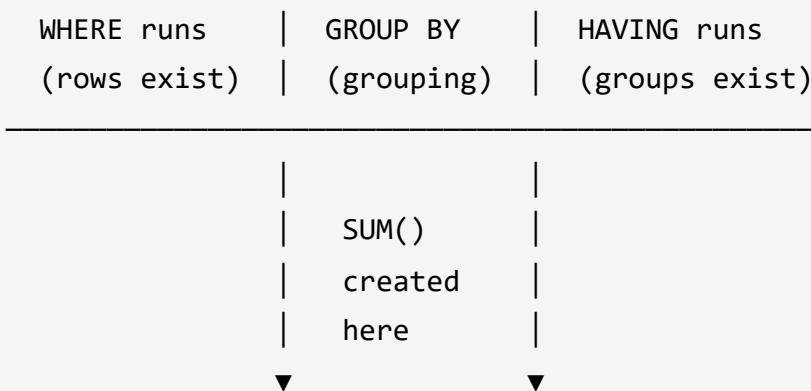
```
-- Works: ORDER BY column not in SELECT
SELECT first_name FROM customers ORDER BY score DESC;

-- Fails with DISTINCT:
SELECT DISTINCT first_name FROM customers ORDER BY score; -- Error!
```

Q4: Explain why HAVING can use aggregates but WHERE cannot?

Answer:

Timeline:



- **WHERE:** Operates on individual rows BEFORE aggregation - SUM() doesn't exist yet
- **HAVING:** Operates on groups AFTER aggregation - SUM() is calculated

Q5: Does the optimizer always follow the logical execution order?

Answer: No! The optimizer may:

- Push filters from HAVING into WHERE when possible
- Reorder JOINS

- Skip unnecessary steps
- Use parallel execution

But the **result** must be identical to following the logical order.

```
-- You write:
SELECT dept, SUM(salary) FROM emp
WHERE status = 'Active'
GROUP BY dept
HAVING dept = 'Sales';

-- Optimizer might:
-- 1. Filter by dept='Sales' early (push down)
-- 2. Then filter by status='Active'
-- 3. Then aggregate (much faster!)
```

Q6: In a query with JOINs, when do JOINs execute?

Answer: JOINs are part of the FROM clause and execute first:

1. FROM table1
JOIN table2 ON ...
JOIN table3 ON ... ← All joins happen here
2. WHERE
3. GROUP BY
- ...

The order of joins in FROM can affect performance (optimizer usually reorders), but logically they all happen at the FROM stage.

Q7: Why does this work?

```
SELECT dept, COUNT(*) AS cnt
FROM employees
GROUP BY dept
ORDER BY COUNT(*); -- Different aggregate expression
```

Answer: ORDER BY can compute its own aggregate on the grouped data. It doesn't have to match the SELECT aggregate exactly (though it's the same calculation here).

Q8: When does TOP/LIMIT execute?

Answer: TOP/LIMIT executes **absolutely last**, after all other operations:

```
SELECT TOP 5 *
FROM customers
WHERE score > 100
ORDER BY score DESC;

-- Order: FROM → WHERE → ORDER BY → TOP
-- TOP takes the first 5 rows from the ALREADY sorted result
```

This is why `TOP 5` without `ORDER BY` gives arbitrary (non-deterministic) results.

Q9: Can you reference a window function in WHERE?

Answer: No! Window functions are calculated during SELECT:

```
-- ✗ Error: Window function not allowed in WHERE
SELECT *, ROW_NUMBER() OVER (ORDER BY score) AS rn
FROM customers
WHERE ROW_NUMBER() OVER (ORDER BY score) <= 5;

-- ✓ Fix: Use subquery or CTE
SELECT * FROM (
    SELECT *, ROW_NUMBER() OVER (ORDER BY score) AS rn
    FROM customers
) t
WHERE rn <= 5;
```

🔑 Key Takeaways

1. **Writing ≠ Execution** - SQL doesn't run top to bottom

2. **FROM is first** - Always starts with the table
 3. **SELECT is late** - Fifth in execution order
 4. **Aliases in SELECT** - Only available in ORDER BY
 5. **WHERE vs HAVING** - Before vs after grouping
 6. **ORDER BY + TOP** - Both execute at the end
-

✖ Common Errors Explained

Error 1: Alias in WHERE

```
-- Error: Invalid column name 'annual_salary'  
SELECT salary * 12 AS annual_salary  
FROM employees  
WHERE annual_salary > 50000;  
  
-- Fix: Repeat the expression  
WHERE salary * 12 > 50000;
```

Error 2: Aggregate in WHERE

```
-- Error: Cannot use aggregate function in WHERE  
SELECT department  
FROM employees  
WHERE COUNT(*) > 5  
GROUP BY department;  
  
-- Fix: Use HAVING  
HAVING COUNT(*) > 5;
```

Error 3: Non-grouped Column in SELECT

```
-- Error: Column must be in GROUP BY or aggregate
SELECT department, first_name, SUM(salary)
FROM employees
GROUP BY department;
```

```
-- Fix: Add to GROUP BY or remove
SELECT department, SUM(salary)
FROM employees
GROUP BY department;
```



Exercises

Exercise 1:

Explain why this query fails:

```
SELECT category, SUM(price) AS total
FROM products
WHERE total > 100
GROUP BY category;
```

Answer: `total` is an alias created in SELECT, which executes after WHERE. Also, `SUM()` can't be in WHERE.

Exercise 2:

Fix the above query.

```
SELECT category, SUM(price) AS total
FROM products
GROUP BY category
HAVING SUM(price) > 100;
```

Exercise 3:

This works - explain why:

```
SELECT category, SUM(price) AS total  
FROM products  
GROUP BY category  
ORDER BY total DESC;
```

Answer: ORDER BY executes after SELECT, so the alias `total` exists and can be used.

Memory Trick

"FROM WHERE GROUP HAVING SELECT ORDER TOP"

Or remember: **"Furry Whales Grow Hairy Shells On Tuesdays"**

- **F**ROM
 - **W**HERE
 - **G**ROUP BY
 - **H**AVING
 - **S**LECT
 - **O**RDERS BY
 - **T**OP
-



Putting It All Together

Complete Query Example:

```
SELECT TOP 5
    department,
    COUNT(*) AS employee_count,
    AVG(salary) AS avg_salary
FROM employees
WHERE hire_date >= '2020-01-01'
GROUP BY department
HAVING COUNT(*) >= 3
ORDER BY avg_salary DESC;
```

Execution Breakdown:

1. **FROM employees** - Get all employees
2. **WHERE hire_date >= '2020-01-01'** - Only recent hires
3. **GROUP BY department** - Group by department
4. **HAVING COUNT(*) >= 3** - Only departments with 3+ employees
5. **SELECT ...** - Get columns and create aliases
6. **ORDER BY avg_salary DESC** - Sort by average salary
7. **TOP 5** - Return only top 5 rows

Note 11: String Functions in SQL



Overview

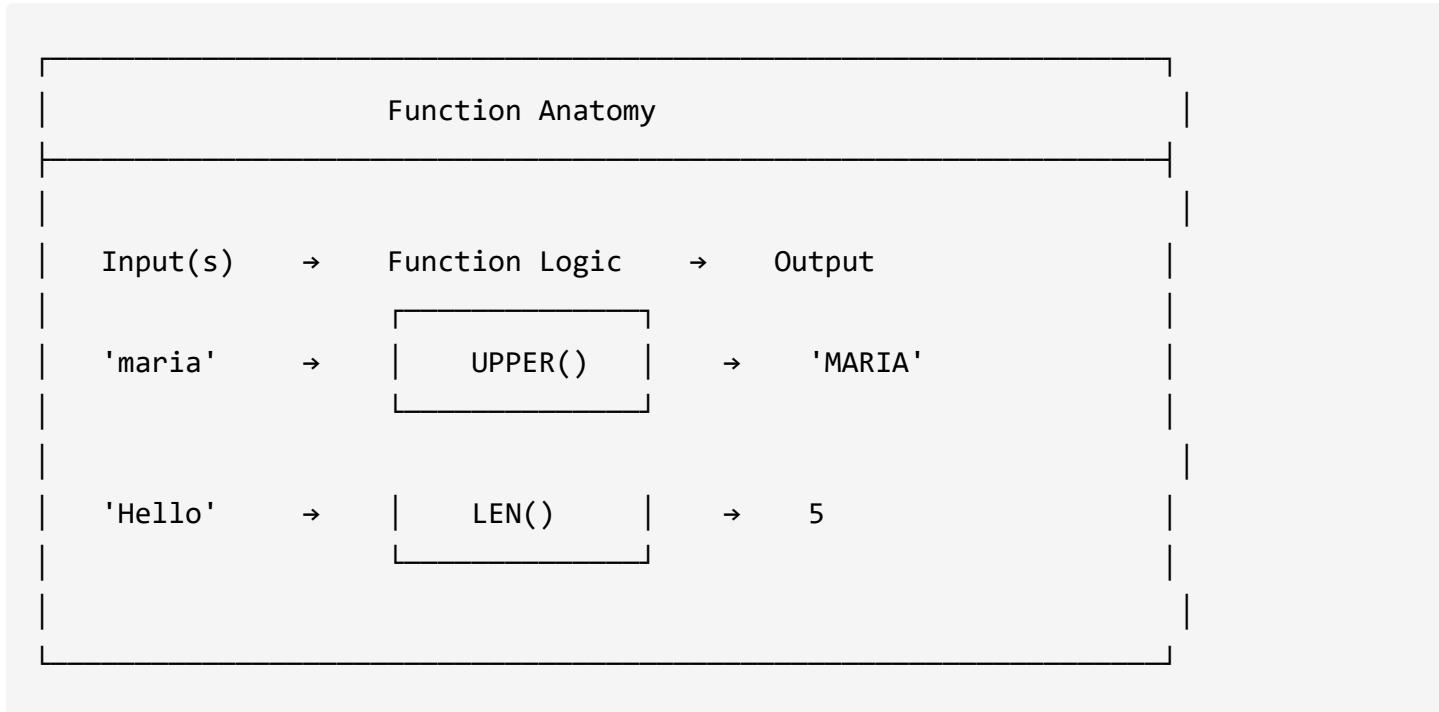
This note covers string functions in SQL - built-in functions that help manipulate, transform, and extract data from text values.



Theory

What are Functions?

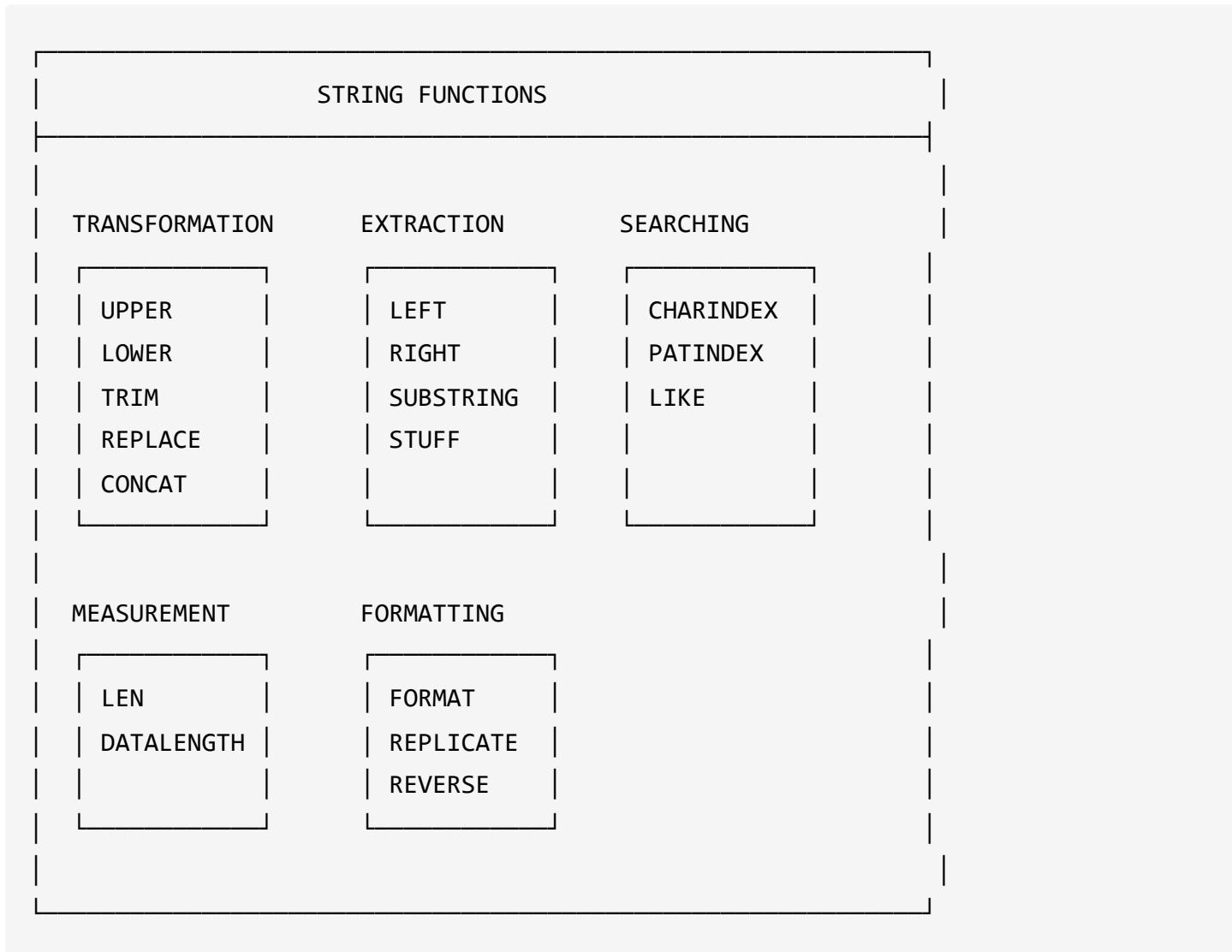
A **function** is a reusable code block that transforms input into output:



Categories of SQL Functions

Category	Also Called	Description	Example
Scalar	Single-row	One value in → one value out	UPPER('hello') → 'HELLO'
Aggregate	Multi-row	Many values in → one value out	SUM(salary) → 50000
Window	Analytic	Many values, preserves rows	ROW_NUMBER()
Table-valued	TVF	Returns a table	STRING_SPLIT()

String Function Categories



NULL Handling in String Functions

Most string functions return NULL if any input is NULL:

```

CONCAT('Hello', NULL, 'World') -- Returns: 'HelloWorld' (CONCAT is special)
UPPER(NULL)                   -- Returns: NULL
LEN(NULL)                     -- Returns: NULL
'Hello' + NULL                -- Returns: NULL (+ operator)
  
```

CONCAT vs + Operator:

- `CONCAT` ignores NULL (treats as empty string)
- `+` operator propagates NULL (result is NULL)



String Manipulation Functions

1. CONCAT - Combining Values

Purpose: Combine multiple string values into one.

```
-- Basic syntax  
CONCAT(value1, value2, value3, ...)  
  
-- Example: Combine first name and country  
SELECT CONCAT(first_name, ' ', country) AS name_country  
FROM customers;  
  
-- Output: "Maria Germany", "John USA"
```

Use Case: Creating full names, combining address fields.

2. UPPER - Convert to Uppercase

Purpose: Convert all characters to uppercase.

```
-- Syntax  
UPPER(string_value)  
  
-- Example  
SELECT UPPER(first_name) AS up_name  
FROM customers;  
  
-- Input: "Maria" → Output: "MARIA"
```

3. LOWER - Convert to Lowercase

Purpose: Convert all characters to lowercase.

```
-- Syntax  
LOWER(string_value)
```

```
-- Example  
SELECT LOWER(first_name) AS low_name  
FROM customers;  
  
-- Input: "MARIA" → Output: "maria"
```

4. TRIM - Remove Spaces

Purpose: Remove leading and trailing spaces from strings.

```
-- Syntax  
TRIM(string_value)  
  
-- Example: Clean up names  
SELECT TRIM(first_name) AS clean_name  
FROM customers;  
  
-- Input: " John " → Output: "John"
```

Finding Values with Spaces:

```
-- Detect customers with spaces in names  
SELECT * FROM customers  
WHERE first_name <> TRIM(first_name);  
  
-- Alternative method using length  
SELECT *  
FROM customers  
WHERE LEN(first_name) <> LEN(TRIM(first_name));
```

5. REPLACE - Replace Characters

Purpose: Replace specific characters with new ones.

```
-- Syntax
REPLACE(value, old_string, new_string)

-- Example 1: Remove dashes from phone number
SELECT REPLACE('123-456-7890', '-', '') AS clean_phone;
-- Output: "1234567890"

-- Example 2: Change file extension
SELECT REPLACE('report.txt', '.txt', '.csv') AS new_filename;
-- Output: "report.csv"
```



String Calculation Functions

LEN - Calculate String Length

Purpose: Count the number of characters in a value.

```
-- Syntax
LEN(string_value)

-- Example
SELECT first_name, LEN(first_name) AS name_length
FROM customers;

-- "Maria" → 5
-- "John"   → 4
```

Works with any data type:

```
LEN('350')      -- Returns: 3
LEN('2026-01-23') -- Returns: 10
```



String Extraction Functions

1. LEFT - Extract from Start

Purpose: Extract specific number of characters from the beginning.

```
-- Syntax  
LEFT(string_value, number_of_characters)  
  
-- Example: Get first 2 characters  
SELECT LEFT(first_name, 2) AS first_two  
FROM customers;  
  
-- "Maria" → "Ma"  
-- "George" → "Ge"
```

2. RIGHT - Extract from End

Purpose: Extract specific number of characters from the end.

```
-- Syntax  
RIGHT(string_value, number_of_characters)  
  
-- Example: Get last 2 characters  
SELECT RIGHT(first_name, 2) AS last_two  
FROM customers;  
  
-- "Maria" → "ia"  
-- "George" → "ge"
```

3. SUBSTRING - Extract from Middle

Purpose: Extract characters from a specific position.

```
-- Syntax  
SUBSTRING(string_value, start_position, length)  
  
-- Example: After 2nd character, get 2 characters  
SELECT SUBSTRING('Maria', 3, 2) AS result;  
-- Output: "ri" (starts at position 3, takes 2 characters)
```

Practical Example:

```
-- Extract middle name from full name  
SELECT SUBSTRING(full_name, 6, 5) AS middle_part  
FROM employees;
```



Nesting Functions

You can combine multiple functions together:

```
-- Step 1: LEFT extracts first 2 characters  
LEFT('Maria', 2) -- Returns: 'Ma'  
  
-- Step 2: LOWER converts to lowercase  
LOWER(LEFT('Maria', 2)) -- Returns: 'ma'  
  
-- Step 3: LEN calculates length  
LEN(LOWER(LEFT('Maria', 2))) -- Returns: 2
```

Execution Order: Inner function → Outer function



Practical Examples

Example 1: Clean and Format Names

```
SELECT  
    UPPER(TRIM(first_name)) AS clean_upper_name  
FROM customers;
```

Example 2: Create Email from Name

```
SELECT  
    LOWER(CONCAT(first_name, '.', last_name, '@company.com')) AS email  
FROM employees;  
-- Output: "john.doe@company.com"
```

Example 3: Extract Area Code

```
SELECT  
    LEFT(phone_number, 3) AS area_code  
FROM contacts;
```

Example 4: Get File Extension

```
SELECT  
    RIGHT(filename, 3) AS extension  
FROM files;  
-- "report.csv" → "csv"
```

❓ Practice Questions

Q1: How do you combine two columns into one?

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name  
FROM customers;
```

Q2: Convert a name to uppercase.

```
SELECT UPPER(first_name) AS name_upper  
FROM customers;
```

Q3: Remove extra spaces from a value.

```
SELECT TRIM(customer_name) AS clean_name  
FROM customers;
```

Q4: Get the first 3 characters of a product code.

```
SELECT LEFT(product_code, 3) AS code_prefix  
FROM products;
```

Q5: Replace hyphens with slashes in a date.

```
SELECT REPLACE('2025-01-15', '-', '/') AS formatted_date;  
-- Output: "2025/01/15"
```



Interview Questions

Q1: What is the difference between CONCAT and the + operator?

Answer:

Aspect	CONCAT()	+ Operator
NULL handling	Ignores NULL (treats as "")	NULL propagates (result = NULL)
Type safety	Auto-converts to string	Requires same types
Multiple values	Any number of parameters	Binary (two at a time)

```
-- CONCAT ignores NULL
SELECT CONCAT('Hello', NULL, 'World'); -- 'HelloWorld'

-- + propagates NULL
SELECT 'Hello' + NULL + 'World'; -- NULL

-- CONCAT auto-converts
SELECT CONCAT('Value: ', 100); -- 'Value: 100'

-- + requires explicit conversion
SELECT 'Value: ' + CAST(100 AS VARCHAR); -- 'Value: 100'
```

Q2: What is the difference between LEN and DATALENGTH?

Answer:

Function	Measures	Unicode Handling
LEN()	Character count	Same for VARCHAR/NVARCHAR
DATALENGTH()	Byte count	NVARCHAR = 2x bytes

```

SELECT LEN('Hello'),          -- 5 characters
       DATALENGTH('Hello');   -- 5 bytes (VARCHAR)

SELECT LEN(N>Hello'),        -- 5 characters
       DATALENGTH(N>Hello'); -- 10 bytes (NVARCHAR = 2 bytes per char)

```

Q3: How do you find the position of a substring?

Answer: Use `CHARINDEX` or `PATINDEX`:

```

-- CHARINDEX: Find exact substring position
SELECT CHARINDEX('@', 'user@email.com'); -- Returns: 5

-- PATINDEX: Find pattern position (supports wildcards)
SELECT PATINDEX('%@%', 'user@email.com'); -- Returns: 5

-- Case-insensitive by default
SELECT CHARINDEX('A', 'banana'); -- Returns: 2 (finds 'a')

```

Q4: How do you split a string into parts?

Answer:

```

-- SQL Server 2016+: STRING_SPLIT
SELECT value FROM STRING_SPLIT('apple,banana,cherry', ',');
-- Returns 3 rows: apple, banana, cherry

-- Extract before/after delimiter
SELECT
    LEFT(email, CHARINDEX('@', email) - 1) AS username,
    RIGHT(email, LEN(email) - CHARINDEX('@', email)) AS domain
FROM users;

```

Q5: What is the difference between TRIM, LTRIM, and RTRIM?

Answer:

Function	Removes
LTRIM()	Leading spaces only (left)
RTRIM()	Trailing spaces only (right)
TRIM()	Both leading and trailing (SQL Server 2017+)

```

SELECT LTRIM(' hello ');
-- 'hello' (right spaces remain)

SELECT RTRIM(' hello ');
-- ' hello' (left spaces remain)

SELECT TRIM(' hello ');
-- 'hello' (both removed)

-- TRIM can also remove specific characters (SQL Server 2017+)
SELECT TRIM('x' FROM 'xxxhelloxx');
-- 'hello'

```

Q6: How do you pad a string to a fixed length?

Answer:

```

-- Right-pad with spaces (fixed width output)
SELECT LEFT(name + REPLICATE(' ', 20), 20) AS padded_name;

-- Left-pad with zeros (common for IDs)
SELECT RIGHT('0000000000' + CAST(id AS VARCHAR), 10) AS padded_id;
-- id=42 becomes '0000000042'

-- Using FORMAT (SQL Server 2012+)
SELECT FORMAT(42, 'd10'); -- '0000000042'

```

Q7: How do you reverse a string?

Answer:

```
SELECT REVERSE('Hello'); -- 'olleH'

-- Check for palindrome
SELECT CASE
    WHEN name = REVERSE(name) THEN 'Palindrome'
    ELSE 'Not palindrome'
END
FROM words;
```

Q8: What is STUFF and when would you use it?

Answer: STUFF deletes and inserts characters at a specific position:

```
-- Syntax: STUFF(string, start, length_to_delete, insert_string)
SELECT STUFF('Hello World', 7, 5, 'SQL'); -- 'Hello SQL'

-- Common use: Mask credit card numbers
SELECT STUFF('1234567890123456', 5, 8, '*****');
-- Result: '1234*****3456'

-- Common use: Remove leading comma from FOR XML PATH
SELECT STUFF(
    (SELECT ', ' + name FROM products FOR XML PATH('')),
    1, 2, ''
); -- Removes first ', '
```

Q9: How do you make case-insensitive comparisons?

Answer:

```
-- Method 1: Convert both to same case
```

```
SELECT * FROM customers WHERE UPPER(name) = UPPER('john');
```

```
-- Method 2: Use COLLATE (more efficient)
```

```
SELECT * FROM customers
WHERE name = 'john' COLLATE SQL_Latin1_General_CI_AS;
```

```
-- Method 3: Database/column default collation
```

```
-- If database is case-insensitive, no conversion needed
```

Q10: How do you count occurrences of a character in a string?

Answer:

```
-- Count 'a' in 'banana'
```

```
SELECT LEN('banana') - LEN(REPLACE('banana', 'a', '')) AS count_of_a;
```

```
-- Returns: 3
```

```
-- Works because:
```

```
-- Original: 'banana' (6 chars)
```

```
-- After replace: 'bnn' (3 chars)
```

```
-- Difference: 6 - 3 = 3 occurrences
```



Key Takeaways

Function	Purpose	Example
CONCAT	Combine strings	CONCAT('A', 'B') → 'AB'
UPPER	Uppercase	UPPER('hi') → 'HI'
LOWER	Lowercase	LOWER('HI') → 'hi'
TRIM	Remove spaces	TRIM(' hi ') → 'hi'
REPLACE	Replace text	REPLACE('a-b', '-', '_') → 'a_b'

Function	Purpose	Example
LEN	Get length	LEN('hello') → 5
LEFT	Extract from start	LEFT('hello', 2) → 'he'
RIGHT	Extract from end	RIGHT('hello', 2) → 'lo'
SUBSTRING	Extract from middle	SUBSTRING('hello', 2, 3) → 'ell'

📌 Best Practices

1. **Always TRIM** user input data
2. **Use UPPER/LOWER** for case-insensitive comparisons
3. **Nest functions** from inner to outer
4. **Use CONCAT** instead of + for safety
5. **Clean data** before analysis

🧪 Exercises

Exercise 1:

Create a full name column from first_name and last_name.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM employees;
```

Exercise 2:

Find all customers whose names contain leading/trailing spaces.

```
SELECT * FROM customers
WHERE first_name <> TRIM(first_name);
```

Exercise 3:

Extract the domain from email addresses.

```
-- For email like "user@domain.com"  
SELECT RIGHT(email, LEN(email) - CHARINDEX('@', email)) AS domain  
FROM users;
```

Exercise 4:

Replace all spaces in product names with underscores.

```
SELECT REPLACE(product_name, ' ', '_') AS url_name  
FROM products;
```

Note 12: Date and Time Functions in SQL

Overview

This note covers date and time functions in SQL - essential tools for extracting parts, formatting, calculating, and validating date/time values.

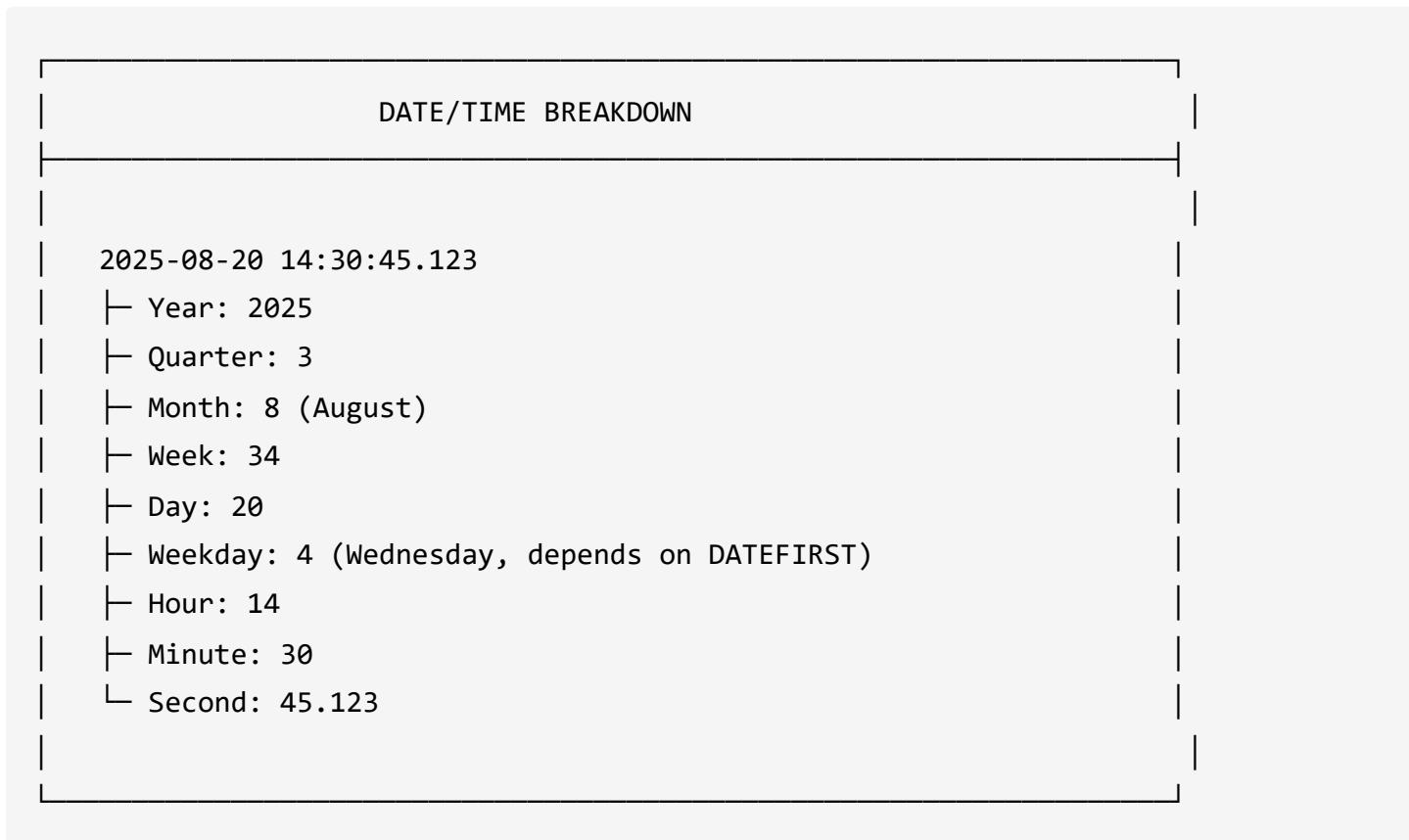
Theory

Date/Time Data Types

Type	Storage	Range	Precision	Example
DATE	3 bytes	0001-9999	Day	'2025-08-20'

Type	Storage	Range	Precision	Example
TIME	3-5 bytes	24 hours	100 nanosec	'14:30:45.1234567'
DATETIME	8 bytes	1753-9999	3.33 ms	'2025-08-20 14:30:45.123'
DATETIME2	6-8 bytes	0001-9999	100 nanosec	'2025-08-20 14:30:45.1234567'
SMALLDATETIME	4 bytes	1900-2079	Minute	'2025-08-20 14:31:00'
DATETIMEOFFSET	10 bytes	0001-9999	100 nanosec	'2025-08-20 14:30:45 +05:30'

Date/Time Hierarchy



Date Function Categories

DATE/TIME FUNCTIONS

EXTRACTION

YEAR
MONTH
DAY
DATEPART
DATENAME

ARITHMETIC

DATEADD
DATEDIFF
EOMONTH

CURRENT VALUES

GETDATE
GETUTCDATE
SYSDATETIME
CURRENT_
TIMESTAMP

CONSTRUCTION

DATEFROMPARTS
DATETRUNC
CAST

FORMATTING

FORMAT
CONVERT

VALIDATION

ISDATE
TRY_CONVERT
TRY_CAST

ISO 8601 Standard Format

YYYY-MM-DD HH:MM:SS.sss

2025-08-20 14:30:45.123

-- Always use ISO format for portability:

WHERE order_date >= '2025-08-01' -- Unambiguous globally



Part Extraction Functions

1. YEAR, MONTH, DAY - Basic Extraction

```
-- Get individual parts
SELECT
    YEAR(order_date) AS order_year,
    MONTH(order_date) AS order_month,
    DAY(order_date) AS order_day
FROM orders;

-- Input: '2025-08-20'
-- Output: 2025, 8, 20
```

2. DATEPART - Extract Any Part

Purpose: Extract specific parts as integers.

```
-- Syntax
DATEPART(part, date)

-- Examples
SELECT
    DATEPART(year, order_date) AS year,
    DATEPART(month, order_date) AS month,
    DATEPART(day, order_date) AS day,
    DATEPART(quarter, order_date) AS quarter,
    DATEPART(week, order_date) AS week,
    DATEPART(weekday, order_date) AS weekday,
    DATEPART(hour, creation_time) AS hour
FROM orders;
```

Available Parts:

Part	Description
year	Year (2025)
quarter	Quarter (1-4)
month	Month (1-12)
week	Week number
day	Day of month
weekday	Day of week (1-7)
hour	Hour (0-23)
minute	Minute (0-59)
second	Second (0-59)

3. DATENAME - Extract Part as Text

Purpose: Get the name of the part (text output).

```
-- Syntax
DATENAME(part, date)

-- Examples
SELECT
    DATENAME(month, order_date) AS month_name,
    DATENAME(weekday, order_date) AS day_name
FROM orders;

-- Output: "August", "Wednesday"
```

Key Difference:

- DATEPART → Returns integer (8)
- DATENAME → Returns string ("August")

4. DATETRUNC - Truncate to Level

Purpose: Reset date to specific precision level.

-- Syntax

```
DATETRUNC(part, date)
```

-- Truncate to month level

```
SELECT DATETRUNC(month, '2025-08-20 14:30:45');
```

-- Output: 2025-08-01 00:00:00

-- Truncate to year level

```
SELECT DATETRUNC(year, '2025-08-20');
```

-- Output: 2025-01-01 00:00:00

-- Truncate to day level

```
SELECT DATETRUNC(day, '2025-08-20 14:30:45');
```

-- Output: 2025-08-20 00:00:00

Use Case - Aggregation by Month:

```
SELECT
    DATETRUNC(month, order_date) AS month,
    COUNT(*) AS orders
FROM orders
GROUP BY DATETRUNC(month, order_date);
```

5. EOMONTH - End of Month

Purpose: Get the last day of a month.

-- Syntax

EOMONTH(**date**)

-- Example

SELECT EOMONTH('2025-08-20');

-- Output: 2025-08-31

SELECT EOMONTH('2025-02-15');

-- Output: 2025-02-28

Get First Day of Month:

-- Use DATETRUNC

```
SELECT CAST(DATETRUNC(month, order_date) AS DATE) AS first_of_month  
FROM orders;
```



Date Formatting Functions

FORMAT - Custom Date Formatting

-- Syntax

FORMAT(date, format_string)

-- Examples

SELECT FORMAT(order_date, 'dd/MM/yyyy') AS formatted;

-- Output: "20/08/2025"

SELECT FORMAT(order_date, 'MMMM dd, yyyy') AS formatted;

-- Output: "August 20, 2025"

SELECT FORMAT(order_date, 'ddd') AS short_day;

-- Output: "Wed"

SELECT FORMAT(order_date, 'dddd') AS full_day;

-- Output: "Wednesday"

Format Specifiers:

Specifier	Output
dd	Day (01-31)
ddd	Short day (Mon)
dddd	Full day (Monday)
MM	Month (01-12)
MMM	Short month (Aug)
MMMM	Full month (August)
yy	Year (25)
yyyy	Year (2025)
HH	Hour 24h (00-23)
hh	Hour 12h (01-12)
mm	Minute (00-59)
ss	Second (00-59)

CAST - Change Data Type

```
-- Convert datetime to date
SELECT CAST(order_datetime AS DATE) AS order_date;

-- Convert string to date
SELECT CAST('2025-08-20' AS DATE) AS converted_date;

-- Convert date to string
SELECT CAST(order_date AS VARCHAR(10)) AS date_string;
```

⊕ Date Calculation Functions

1. GETDATE() - Current Date/Time

```
SELECT GETDATE() AS current_datetime;  
-- Output: 2025-01-15 10:30:45.123
```

2. DATEADD - Add/Subtract Time

Purpose: Add or subtract time from a date.

```
-- Syntax  
DATEADD(part, number, date)  
  
-- Add 5 days  
SELECT DATEADD(day, 5, '2025-08-20');  
-- Output: 2025-08-25  
  
-- Subtract 3 months  
SELECT DATEADD(month, -3, '2025-08-20');  
-- Output: 2025-05-20  
  
-- Add 1 year  
SELECT DATEADD(year, 1, '2025-08-20');  
-- Output: 2026-08-20
```

Practical Example:

```
-- Calculate delivery date (7 days after order)  
SELECT  
    order_date,  
    DATEADD(day, 7, order_date) AS expected_delivery  
FROM orders;
```

3. DATEDIFF - Difference Between Dates

Purpose: Calculate difference between two dates.

```
-- Syntax  
DATEDIFF(part, start_date, end_date)  
  
-- Days between dates  
SELECT DATEDIFF(day, '2025-01-01', '2025-01-15');  
-- Output: 14  
  
-- Months between dates  
SELECT DATEDIFF(month, '2025-01-01', '2025-08-20');  
-- Output: 7  
  
-- Calculate age  
SELECT DATEDIFF(year, birth_date, GETDATE()) AS age  
FROM employees;
```

Practical Examples:

```
-- Shipping duration  
SELECT  
    order_id,  
    DATEDIFF(day, order_date, ship_date) AS days_to_ship  
FROM orders;  
  
-- Average shipping time per month  
SELECT  
    MONTH(order_date) AS month,  
    AVG(DATEDIFF(day, order_date, ship_date)) AS avg_ship_days  
FROM orders  
GROUP BY MONTH(order_date);
```



Date Validation

ISDATE - Check if Value is Valid Date

-- Syntax

```
ISDATE(value)
```

-- Examples

```
SELECT ISDATE('2025-08-20');    -- Returns: 1 (true)
SELECT ISDATE('123');           -- Returns: 0 (false)
SELECT ISDATE('2025');          -- Returns: 1 (true)
SELECT ISDATE('August');         -- Returns: 0 (false)
```

Use Case - Handle Bad Data:

```
SELECT
    order_id,
    CASE
        WHEN ISDATE(order_date) = 1
        THEN CAST(order_date AS DATE)
        ELSE NULL
    END AS valid_date
FROM orders;
```



Practical Examples

Example 1: Orders by Year

```
SELECT
    YEAR(order_date) AS order_year,
    COUNT(*) AS total_orders
FROM orders
GROUP BY YEAR(order_date);
```

Example 2: Orders by Month Name

```
SELECT
    DATENAME(month, order_date) AS month_name,
    COUNT(*) AS total_orders
FROM orders
GROUP BY DATENAME(month, order_date), MONTH(order_date)
ORDER BY MONTH(order_date);
```

Example 3: Filter February Orders

```
SELECT * FROM orders
WHERE MONTH(order_date) = 2;
```

Example 4: Calculate Employee Age

```
SELECT
    employee_id,
    first_name,
    DATEDIFF(year, birth_date, GETDATE()) AS age
FROM employees;
```



Function Output Types

Function	Output Type
YEAR, MONTH, DAY	Integer
DATEPART	Integer
DATENAME	String
DATETRUNC	DateTime
EOMONTH	Date

Function	Output Type
FORMAT	String
GETDATE	DateTime
DATEADD	DateTime
DATEDIFF	Integer
ISDATE	Integer (0/1)

❓ Practice Questions

Q1: How many orders were placed each year?

```
SELECT YEAR(order_date) AS year, COUNT(*) AS orders
FROM orders
GROUP BY YEAR(order_date);
```

Q2: Show orders placed in February.

```
SELECT * FROM orders
WHERE MONTH(order_date) = 2;
```

Q3: Calculate shipping duration in days.

```
SELECT
    order_id,
    DATEDIFF(day, order_date, ship_date) AS shipping_days
FROM orders;
```

Q4: Get the last day of each order's month.

```
SELECT order_id, EOMONTH(order_date) AS month_end
FROM orders;
```

Interview Questions

Q1: What is the difference between DATETIME and DATETIME2?

Answer:

Aspect	DATETIME	DATETIME2
Range	1753-9999	0001-9999
Precision	3.33 ms	100 ns (configurable)
Storage	8 bytes fixed	6-8 bytes (by precision)
Recommended	Legacy	<input checked="" type="checkbox"/> New development

```
-- DATETIME rounds to .000, .003, or .007
SELECT CAST('2025-01-15 12:30:45.999' AS DATETIME);
-- Result: '2025-01-15 12:30:46.000' (rounded!)

-- DATETIME2 preserves precision
SELECT CAST('2025-01-15 12:30:45.9999999' AS DATETIME2);
-- Result: '2025-01-15 12:30:45.9999999'
```

Q2: How do you calculate age accurately?

Answer: DATEDIFF(year, ...) counts year boundaries, not actual years:

-- X Inaccurate: Counts year boundary crossings

```
SELECT DATEDIFF(year, '2000-12-31', '2001-01-01'); -- Returns: 1 (but only 1 day!)
```

-- ✓ Accurate age calculation:

SELECT

```
DATEDIFF(year, birth_date, GETDATE()) -
CASE WHEN DATEADD(year, DATEDIFF(year, birth_date, GETDATE()), birth_date) > GETDATE()
    THEN 1 ELSE 0
END AS accurate_age
FROM employees;
```

-- Explanation: Subtract 1 if birthday hasn't occurred this year yet



Q3: What is the difference between DATEPART and DATENAME?

Answer:

Function	Returns	Example Output
DATEPART(month, date)	Integer	8
DATENAME(month, date)	String	'August'

SELECT

```
DATEPART(month, '2025-08-20'), -- 8
DATENAME(month, '2025-08-20'); -- 'August'
```

-- Use DATEPART for: filtering, calculations, sorting

-- Use DATENAME for: display, reporting, labels

Q4: How do you get the first day of the current month?

Answer:

```
-- Method 1: DATETRUNC (SQL Server 2022+)
```

```
SELECT DATETRUNC(month, GETDATE());
```

```
-- Method 2: DATEFROMPARTS
```

```
SELECT DATEFROMPARTS(YEAR(GETDATE()), MONTH(GETDATE()), 1);
```

```
-- Method 3: EOMONTH of previous month + 1 day
```

```
SELECT DATEADD(day, 1, EOMONTH(GETDATE(), -1));
```

```
-- Method 4: Subtract day-of-month minus 1
```

```
SELECT DATEADD(day, 1 - DAY(GETDATE()), CAST(GETDATE() AS DATE));
```

Q5: How do you filter for "this week" or "last 7 days"?

Answer:

```
-- Last 7 days (including today)
```

```
SELECT * FROM orders  
WHERE order_date >= DATEADD(day, -6, CAST(GETDATE() AS DATE));
```

```
-- This week (Sunday to Saturday)
```

```
SELECT * FROM orders  
WHERE order_date >= DATEADD(day, 1 - DATEPART(weekday, GETDATE()), CAST(GETDATE() AS  
AND order_date < DATEADD(day, 8 - DATEPART(weekday, GETDATE()), CAST(GETDATE() AS
```

```
-- Current month
```

```
SELECT * FROM orders  
WHERE order_date >= DATEFROMPARTS(YEAR(GETDATE()), MONTH(GETDATE()), 1);
```

Q6: Why should you avoid using functions on columns in WHERE?

Answer: Functions prevent index usage (not SARGable):

```
-- ❌ Non-SARGable: Full table scan
SELECT * FROM orders WHERE YEAR(order_date) = 2025;

-- ✅ SARGable: Can use index on order_date
SELECT * FROM orders
WHERE order_date >= '2025-01-01' AND order_date < '2026-01-01';
```

Q7: How do you handle time zones in SQL Server?

Answer:

```
-- Store with offset using DATETIMEOFFSET
CREATE TABLE events (event_time DATETIMEOFFSET);
INSERT INTO events VALUES ('2025-01-15 10:00:00 +05:30');

-- Convert between time zones
SELECT event_time AT TIME ZONE 'Pacific Standard Time';

-- Get current time in different zones
SELECT GETDATE() AT TIME ZONE 'UTC';
SELECT SYSDATETIMEOFFSET(); -- Local time with offset
```

Q8: What is the difference between GETDATE() and SYSDATETIME()?

Answer:

Function	Precision	Returns
GETDATE()	~3.33 ms	DATETIME
SYSDATETIME()	100 ns	DATETIME2
GETUTCDATE()	~3.33 ms	DATETIME (UTC)
SYSUTCDATETIME()	100 ns	DATETIME2 (UTC)

```

SELECT GETDATE();          -- 2025-01-15 10:30:45.123
SELECT SYSDATETIME();      -- 2025-01-15 10:30:45.1234567
SELECT SYSUTCDATETIME();   -- 2025-01-15 05:00:45.1234567 (UTC)

```

Q9: How do you get the number of business days between two dates?

Answer:

```

-- Exclude weekends (Saturday=7, Sunday=1)
SELECT
    order_date,
    ship_date,
    (DATEDIFF(day, order_date, ship_date) + 1) -
    (DATEDIFF(week, order_date, ship_date) * 2) -
    CASE WHEN DATEPART(weekday, order_date) = 1 THEN 1 ELSE 0 END -
    CASE WHEN DATEPART(weekday, ship_date) = 7 THEN 1 ELSE 0 END
    AS business_days
FROM orders;

```

Q10: How do you generate a date series (calendar table)?

Answer:

```

-- Using recursive CTE
WITH DateSeries AS (
    SELECT CAST('2025-01-01' AS DATE) AS dt
    UNION ALL
    SELECT DATEADD(day, 1, dt)
    FROM DateSeries
    WHERE dt < '2025-12-31'
)
SELECT dt FROM DateSeries
OPTION (MAXRECURSION 366);

```



Key Takeaways

1. **YEAR/MONTH/DAY** - Quick extraction
 2. **DATEPART** - Any part as integer
 3. **DATENAME** - Part as text (readable)
 4. **DATETRUNC** - Reset to precision level
 5. **DATEADD** - Add/subtract time units
 6. **DATEDIFF** - Calculate time between dates
 7. **FORMAT** - Custom display formatting
 8. **ISDATE** - Validate date values
-



Best Practices

Practice	Reason
Use DATEPART for filtering	Numbers are faster
Use DATENAME for display	Human-readable
Use DATETRUNC for grouping	Consistent aggregation
Use ISDATE before casting	Avoid errors



Exercises

Exercise 1:

Find orders from the last 30 days.

```
SELECT * FROM orders
WHERE order_date >= DATEADD(day, -30, GETDATE());
```

Exercise 2:

Calculate average order processing time.

```
SELECT AVG(DATEDIFF(day, order_date, ship_date)) AS avg_processing  
FROM orders;
```

Exercise 3:

Group sales by quarter.

```
SELECT  
    DATEPART(quarter, order_date) AS quarter,  
    SUM(sales) AS total_sales  
FROM orders  
GROUP BY DATEPART(quarter, order_date);
```

Note 13: NULL Functions in SQL

Overview

This note covers NULL handling in SQL - understanding what NULLs are, why they matter, and the functions available to handle them effectively.

Theory

What is NULL?

- **NULL** = The absence of a value (unknown, missing, not applicable)
- NULL is NOT zero, NOT an empty string, NOT a space
- NULL represents "I don't know what this value is"

NULL vs Other "Empty" Values

Value	Type	Meaning
NULL	Unknown	"I don't know"
0	Number	"The answer is zero"
''	String	"The answer is an empty string"
' '	String	"The answer is a space"
'N/A'	String	"Not Applicable"

Example: What is John's middle name?

NULL = "We didn't ask / don't know"
 '' = "He has no middle name"

The Three-Valued Logic

SQL uses three-valued logic because of NULL: TRUE, FALSE, UNKNOWN

```
-- Regular comparisons
5 = 5      -- TRUE
5 = 6      -- FALSE
5 = NULL    -- UNKNOWN (not FALSE!)
NULL = NULL -- UNKNOWN (not TRUE!)
```

Expression	Result
TRUE AND UNKNOWN	UNKNOWN
FALSE AND UNKNOWN	FALSE
TRUE OR UNKNOWN	TRUE
FALSE OR UNKNOWN	UNKNOWN
NOT UNKNOWN	UNKNOWN

⚠ Critical: WHERE clause only returns rows where condition is TRUE, not UNKNOWN!

Why NULLs Occur

- Optional form fields left empty
- Missing data during import
- LEFT/RIGHT JOIN with no matches
- Aggregate on empty set
- Division by zero (when using NULLIF)
- NULLIF function explicitly creates NULL

NULL Propagation

NULL "infects" most operations:

```
-- Any arithmetic with NULL = NULL  
100 + NULL = NULL  
100 * NULL = NULL  
  
-- String concatenation (using +) with NULL = NULL  
'Hello' + NULL = NULL -- Result is NULL!  
  
-- Exception: CONCAT ignores NULL  
CONCAT('Hello', NULL) = 'Hello'
```



NULL Handling Functions

1. ISNULL - Replace NULL with Value

Purpose: Replace NULL with a specific value.

-- Syntax

```
ISNULL(value, replacement)
```

-- Example: Replace NULL with 'Unknown'

```
SELECT ISNULL(shipping_address, 'Unknown') AS address
FROM orders;
```

-- Example: Replace NULL with another column

```
SELECT ISNULL(shipping_address, billing_address) AS address
FROM orders;
```

How it works:

ISNULL checks value:

- ├─ If NULL → Return replacement
- └─ If NOT NULL → Return original value

2. COALESCE - Return First Non-NULL

Purpose: Return the first non-NULL value from a list.

-- Syntax

```
COALESCE(value1, value2, value3, ...)
```

-- Example: Check multiple columns

```
SELECT COALESCE(shipping_address, billing_address, 'No Address') AS address
FROM orders;
```

How it works:

COALESCE checks left to right:

1. Check value1 → If not NULL, return it
2. Check value2 → If not NULL, return it
3. Continue until finding non-NULL
4. If all NULL, return last value

Example with 3 values:

```
SELECT COALESCE(phone_mobile, phone_work, phone_home) AS contact_phone
FROM customers;
```

3. ISNULL vs COALESCE

Feature	ISNULL	COALESCE
Values accepted	2 only	Multiple
SQL Standard	SQL Server only	All databases
Performance	Faster	Slightly slower
Oracle equivalent	NVL	COALESCE
MySQL equivalent	IFNULL	COALESCE

Recommendation: Use COALESCE for portability.

4. NULLIF - Convert Value to NULL

Purpose: Return NULL if two values are equal.

```
-- Syntax
NULLIF(value1, value2)

-- Example: Convert -1 to NULL
SELECT NULLIF(price, -1) AS clean_price
FROM products;
-- If price = -1, returns NULL
-- If price = 50, returns 50
```

Common Use - Prevent Divide by Zero:

```
-- Avoid division error  
SELECT sales / NULLIF(quantity, 0) AS price_per_unit  
FROM orders;  
-- If quantity = 0, returns NULL instead of error
```

5. IS NULL / IS NOT NULL - Check for NULLs

Purpose: Test if a value is NULL.

```
-- Find records with NULL  
SELECT * FROM customers  
WHERE score IS NULL;  
  
-- Find records without NULL  
SELECT * FROM customers  
WHERE score IS NOT NULL;
```

 **Important:** Never use = NULL . Always use IS NULL .

--  WRONG
WHERE score = NULL

--  CORRECT
WHERE score IS NULL

Use Cases for NULL Handling

Use Case 1: Before Aggregations

NULLs are ignored in aggregations:

```
-- Without handling NULLs
SELECT AVG(score) FROM customers;
-- Calculates: (80 + 90) / 2 = 85 (ignores NULL)

-- With NULL handling (if NULL means 0)
SELECT AVG(COALESCE(score, 0)) FROM customers;
-- Calculates: (80 + 90 + 0) / 3 = 56.67
```

Use Case 2: Before Mathematical Operations

NULL propagates through calculations:

```
-- Problem: NULL + anything = NULL
SELECT first_name + ' ' + last_name AS full_name
FROM customers;
-- If last_name is NULL, full_name is NULL

-- Solution: Handle NULL first
SELECT first_name + ' ' + COALESCE(last_name, '') AS full_name
FROM customers;
```

Use Case 3: Before Joins

NULLs don't match in joins:

```
-- Problem: NULL keys won't join
SELECT * FROM table1 t1
INNER JOIN table2 t2 ON t1.type = t2.type;
-- Rows with NULL type won't match

-- Solution: Handle NULL in join
SELECT * FROM table1 t1
INNER JOIN table2 t2
ON ISNULL(t1.type, '') = ISNULL(t2.type, '');
```

Use Case 4: Before Sorting

Control where NULLs appear:

```
-- NULLs appear first in ASC, last in DESC

-- Force NULLs to end in ascending sort
SELECT * FROM customers
ORDER BY
CASE WHEN score IS NULL THEN 1 ELSE 0 END,
score ASC;
```

Use Case 5: Anti-Joins (Finding Unmatched Rows)

```
-- Find customers with no orders
SELECT c.*
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.customer_id IS NULL;
```



Practical Examples

Example 1: Default Value for NULL

```
SELECT
customer_id,
COALESCE(score, 0) AS score
FROM customers;
```

Example 2: Safe Division

```
SELECT
    order_id,
    sales / NULLIF(quantity, 0) AS unit_price
FROM orders;
```

Example 3: Find Missing Data

```
SELECT * FROM customers
WHERE email IS NULL OR phone IS NULL;
```

Example 4: Full Name with NULL Handling

```
SELECT
    CONCAT(first_name, ' ', COALESCE(last_name, '')) AS full_name
FROM customers;
```

Example 5: Conditional NULL Replacement

```
SELECT
    customer_id,
    CASE
        WHEN score IS NULL THEN 'No Score'
        WHEN score < 50 THEN 'Low'
        WHEN score < 80 THEN 'Medium'
        ELSE 'High'
    END AS score_category
FROM customers;
```



NULL vs Empty String vs Blank

Aspect	NULL	Empty String ("")	Blank (' ')
Meaning	Unknown	Known, empty	Known, space(s)
Storage	Minimal	Takes space	Takes space
LEN()	NULL	0	1+
Search	IS NULL	= "	= ''
Performance	Best	Good	Worst

Detecting the difference:

```
SELECT
    category,
    DATALENGTH(category) AS length
FROM orders;
-- NULL → NULL
-- '' → 0
-- ' ' → 1
```

🔧 Data Cleanup Policies

Policy 1: Keep NULLs and Empty Strings

```
SELECT TRIM(category) AS clean_category
FROM orders;
-- Removes blank spaces, keeps NULLs and empty strings
```

Policy 2: Convert to NULL Only

```
SELECT NULLIF(TRIM(category), '') AS clean_category  
FROM orders;  
-- Converts blanks and empty strings to NULL
```

Policy 3: Use Default Value

```
SELECT COALESCE(NULLIF(TRIM(category), ''), 'Unknown') AS clean_category  
FROM orders;  
-- Converts everything to 'Unknown' if missing
```

?

Practice Questions

Q1: Replace NULL scores with 0.

```
SELECT customer_id, COALESCE(score, 0) AS score  
FROM customers;
```

Q2: Find all customers with missing phone numbers.

```
SELECT * FROM customers  
WHERE phone IS NULL;
```

Q3: Avoid divide by zero error.

```
SELECT total / NULLIF(count, 0) AS average  
FROM summary;
```

Q4: Use billing address if shipping is NULL.

```
SELECT COALESCE(shipping_address, billing_address) AS address
FROM orders;
```

Q5: Find customers who never placed orders.

```
SELECT c.*
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

Interview Questions

Q1: Why does WHERE column = NULL not work?

Answer: Because NULL represents an unknown value. Comparing anything to unknown gives UNKNOWN (not TRUE or FALSE). The WHERE clause only returns rows where the condition evaluates to TRUE.

--  Never returns any rows (even if NULLs exist)

```
SELECT * FROM customers WHERE score = NULL;
```

--  Also wrong

```
SELECT * FROM customers WHERE score <> NULL;
```

--  Correct

```
SELECT * FROM customers WHERE score IS NULL;
```

```
SELECT * FROM customers WHERE score IS NOT NULL;
```

Q2: What is the difference between ISNULL and COALESCE?

Answer:

Feature	ISNULL	COALESCE
Parameters	Exactly 2	2 or more
Standard	SQL Server only	ANSI SQL (all DBs)
Data type	Takes type of first param	Uses data type precedence
Performance	Slightly faster	More overhead

```
-- ISNULL: Limited to 2 values
```

```
SELECT ISNULL(phone, 'Unknown');
```

```
-- COALESCE: Can check multiple values
```

```
SELECT COALESCE(mobile, work_phone, home_phone, 'Unknown');
```

```
-- Data type difference
```

```
SELECT ISNULL(NULL, 1);      -- Returns INT (type of replacement)
```

```
SELECT COALESCE(NULL, 1);   -- Returns INT (highest precedence)
```

Best Practice: Use COALESCE for portability across databases.

Q3: How do aggregates handle NULL?

Answer: Aggregates **ignore NULL values** (except COUNT(*)):

```
-- Data: scores = [100, NULL, 200, NULL, 300]
```

```
SELECT COUNT(*) AS all_rows;          -- 5 (counts NULLs)
```

```
SELECT COUNT(score) AS non_null;      -- 3 (ignores NULLs)
```

```
SELECT SUM(score) AS total;          -- 600 (ignores NULLs)
```

```
SELECT AVG(score) AS average;        -- 200 (600/3, NOT 600/5!)
```

```
-- If you want NULL treated as 0:
```

```
SELECT AVG(COALESCE(score, 0)) AS avg_with_nulls; -- 120 (600/5)
```

Q4: What does NULLIF do and when would you use it?

Answer: NULLIF returns NULL if two values are equal, otherwise returns the first value.

```
-- Syntax: NULLIF(value1, value2)
NULLIF(5, 5)    -- Returns NULL (values equal)
NULLIF(5, 3)    -- Returns 5 (values different)
NULLIF(NULL, 5) -- Returns NULL (first value is NULL)

-- Common use 1: Prevent divide by zero
SELECT revenue / NULLIF(cost, 0) AS ratio;
-- Returns NULL instead of error when cost = 0

-- Common use 2: Convert placeholder to NULL
SELECT NULLIF(status, 'N/A') AS clean_status;
-- Converts 'N/A' to NULL for consistent handling
```

Q5: Can NULL equal NULL?

Answer: No! NULL = NULL returns UNKNOWN, not TRUE.

```
-- This comparison is UNKNOWN
SELECT CASE WHEN NULL = NULL THEN 'Equal' ELSE 'Not Equal' END;
-- Returns: 'Not Equal' (because UNKNOWN is not TRUE)

-- To check if two nullable values are "same" (including both NULL):
SELECT CASE
    WHEN col1 = col2 THEN 'Same'
    WHEN col1 IS NULL AND col2 IS NULL THEN 'Same'
    ELSE 'Different'
END;
```

Q6: How do NULLs affect JOINs?

Answer: NULLs **never** match in JOIN conditions:

```
-- Table A: id = [1, 2, NULL]
-- Table B: id = [2, NULL, NULL]

SELECT * FROM A INNER JOIN B ON A.id = B.id;
-- Only returns row where id = 2
-- NULL rows don't match (even NULL to NULL!)

-- If you need NULLs to match:
SELECT * FROM A INNER JOIN B
ON A.id = B.id OR (A.id IS NULL AND B.id IS NULL);
```

Q7: How do you handle NULLs in ORDER BY?

Answer: NULL placement varies by database:

Database	ASC	DESC
SQL Server	NULLs first	NULLs last
PostgreSQL	NULLs last	NULLs first
MySQL	NULLs first	NULLs last
Oracle	NULLs last	NULLs first

```
-- SQL Server: Force NULLs to end (ASC)
SELECT * FROM customers
ORDER BY CASE WHEN score IS NULL THEN 1 ELSE 0 END, score ASC;

-- PostgreSQL/Oracle: Use NULLS FIRST/LAST
SELECT * FROM customers ORDER BY score ASC NULLS LAST;
```

Q8: What is the difference between NULL, empty string, and whitespace?

Answer:

Value	LEN()	IS NULL	= ''	Meaning
NULL	NULL	TRUE	UNKNOWN	Unknown/missing
''	0	FALSE	TRUE	Known empty
''	1+	FALSE	FALSE	Known space(s)

```
-- Detecting each:
```

```
SELECT
    col,
    CASE
        WHEN col IS NULL THEN 'NULL'
        WHEN col = '' THEN 'Empty String'
        WHEN LTRIM(col) = '' THEN 'Whitespace Only'
        ELSE 'Has Value'
    END AS type
FROM table;
```

Q9: How do you convert empty strings to NULL?

Answer:

```
-- Convert empty string to NULL
SELECT NULLIF(column, '') AS clean_column;

-- Convert empty string and whitespace to NULL
SELECT NULLIF(LTRIM(RTRIM(column)), '') AS clean_column;

-- Full cleanup pipeline
SELECT COALESCE(NULLIF(LTRIM(RTRIM(column)), ''), 'Default') AS clean_column;
```

Q10: Why is NOT IN dangerous with NULLs?

Answer: If the subquery contains NULL, NOT IN returns no rows:

```
-- Subquery returns: [1, 2, NULL]
SELECT * FROM customers WHERE id NOT IN (SELECT id FROM excluded);
```

-- Logically becomes:
-- WHERE id <> 1 AND id <> 2 AND id <> NULL
-- The last part is UNKNOWN, making entire condition UNKNOWN
-- No rows returned!

-- Safe alternative: Use NOT EXISTS

```
SELECT * FROM customers c
WHERE NOT EXISTS (SELECT 1 FROM excluded e WHERE e.id = c.id);
```

🔑 Key Takeaways

1. **NULL ≠ zero/empty** - NULL means unknown
2. **IS NULL/IS NOT NULL** - Use for checking NULLs
3. **ISNULL** - Replace NULL (2 values only)
4. **COALESCE** - First non-NUL from list
5. **NULLIF** - Convert value to NULL
6. **Handle before operations** - Aggregations, math, joins, sorting
7. **Anti-joins** - LEFT JOIN + IS NULL

📌 Best Practices

Practice	Reason
Use COALESCE over ISNULL	Portability
Handle NULLs before math	Avoid NULL propagation
Handle NULLs before joins	Prevent missing rows
Use NULLIF for divide by zero	Prevent errors

Practice	Reason
Define data policies	Consistency



Exercises

Exercise 1:

Calculate average score treating NULLs as 0.

```
SELECT AVG(COALESCE(score, 0)) AS avg_score  
FROM customers;
```

Exercise 2:

Find products with no category assigned.

```
SELECT * FROM products  
WHERE category IS NULL;
```

Exercise 3:

Create a full address using available parts.

```
SELECT COALESCE(  
    street + ', ' + city + ', ' + state,  
    street + ', ' + city,  
    city,  
    'No Address'  
) AS full_address  
FROM customers;
```

Exercise 4:

Safely calculate price per unit.

```
SELECT
    product_id,
    total_price / NULLIF(quantity, 0) AS unit_price
FROM order_details;
```

Note 14: CASE Statements in SQL

Overview

This note covers CASE statements - SQL's way to implement conditional logic (IF-THEN-ELSE) for data transformations and categorization.

Theory

What is CASE Statement?

A **CASE expression** provides conditional logic in SQL - it's the equivalent of IF-THEN-ELSE in programming languages:

CASE Statement Flow

Input Value	→	Conditions Checked	→	Output Value
75		score >= 90? NO		
	→	score >= 80? NO		
		score >= 70? YES → Return 'C'		
		score >= 60? (not checked)		
		ELSE (not reached)		

⚠ First TRUE condition wins - order matters!

Two CASE Syntax Forms

1. Searched CASE (Most Common):

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ELSE default_result
END
```

- Each WHEN has its own condition
- Conditions can be unrelated
- Most flexible

2. Simple CASE:

```
CASE expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ELSE default_result
END
```

- Compares one expression to multiple values
- Cleaner for single-column comparisons
- Like a switch statement

CASE is an Expression, Not a Statement

CASE returns a **value**, so it can be used anywhere a value is allowed:

- In SELECT (create calculated columns)
- In WHERE (conditional filtering)
- In ORDER BY (custom sorting)
- In GROUP BY (conditional grouping)
- In aggregate functions (conditional aggregation)
- In JOIN conditions
- In UPDATE SET clause

The ELSE Clause

With ELSE	Without ELSE
Returns specified default	Returns NULL
Explicit handling	May hide issues
<input checked="" type="checkbox"/> Recommended	 Use carefully

```
-- Without ELSE: returns NULL when no match
```

```
CASE WHEN score > 90 THEN 'A' END -- NULL for score = 50
```

```
-- With ELSE: explicit default
```

```
CASE WHEN score > 90 THEN 'A' ELSE 'Other' END
```



Basic Syntax

Simple CASE Structure

CASE

```
WHEN condition1 THEN result1  
WHEN condition2 THEN result2  
WHEN condition3 THEN result3  
ELSE default_result  
END AS column_alias
```

Keywords Explained

Keyword	Purpose
CASE	Start the logic
WHEN	Define a condition
THEN	Result if condition is true
ELSE	Default (optional)
END	End the logic



Basic Examples

Example 1: Single Condition

```
SELECT
    order_id,
    sales,
    CASE
        WHEN sales > 50 THEN 'High'
    END AS sales_category
FROM orders;
```

Output:

order_id	sales	sales_category
1	60	High
2	30	NULL

⚠ Without ELSE, non-matching rows return NULL.

Example 2: Multiple Conditions

```
SELECT
    order_id,
    sales,
    CASE
        WHEN sales > 50 THEN 'High'
        WHEN sales > 20 THEN 'Medium'
        ELSE 'Low'
    END AS sales_category
FROM orders;
```

Output:

order_id	sales	sales_category
1	60	High
2	30	Medium
3	15	Low

Example 3: With ELSE (Default Value)

```

SELECT
    customer_id,
    score,
    CASE
        WHEN score >= 90 THEN 'A'
        WHEN score >= 80 THEN 'B'
        WHEN score >= 70 THEN 'C'
        WHEN score >= 60 THEN 'D'
        ELSE 'F'
    END AS grade
FROM students;

```



Execution Flow

CASE evaluates conditions **top to bottom**:

1. Check condition1 → If TRUE, return result1 and STOP
2. Check condition2 → If TRUE, return result2 and STOP
3. Check condition3 → If TRUE, return result3 and STOP
4. If all FALSE → Return ELSE value (or NULL)

Order matters!

--  Wrong order

CASE

```
WHEN sales > 20 THEN 'Medium' -- 60 matches here!
```

```
WHEN sales > 50 THEN 'High' -- Never reached
```

END

--  Correct order

CASE

```
WHEN sales > 50 THEN 'High' -- Check highest first
```

```
WHEN sales > 20 THEN 'Medium'
```

END

Use Cases

Use Case 1: Categorizing Data

SELECT

```
product_id,
```

```
price,
```

CASE

```
WHEN price > 100 THEN 'Premium'
```

```
WHEN price > 50 THEN 'Standard'
```

```
ELSE 'Budget'
```

```
END AS price_tier
```

FROM products;

Use Case 2: Mapping/Decoding Values

```
SELECT
    order_id,
    status,
    CASE status
        WHEN 1 THEN 'Pending'
        WHEN 2 THEN 'Processing'
        WHEN 3 THEN 'Shipped'
        WHEN 4 THEN 'Delivered'
        WHEN 5 THEN 'Cancelled'
        ELSE 'Unknown'
    END AS status_text
FROM orders;
```

Simplified CASE syntax (when comparing one column):

```
CASE column_name
    WHEN value1 THEN result1
    WHEN value2 THEN result2
END
```

Use Case 3: Handling NULLs

```
SELECT
    customer_id,
    CASE
        WHEN score IS NULL THEN 0
        ELSE score
    END AS clean_score
FROM customers;
```

Use Case 4: Conditional Aggregation

```

SELECT
    COUNT(*) AS total_orders,
    SUM(CASE WHEN status = 'Completed' THEN 1 ELSE 0 END) AS completed,
    SUM(CASE WHEN status = 'Pending' THEN 1 ELSE 0 END) AS pending,
    SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END) AS cancelled
FROM orders;

```

Output:

total_orders	completed	pending	cancelled
100	75	15	10

Use Case 5: Pivot-Style Reports

```

SELECT
    product_id,
    SUM(CASE WHEN MONTH(order_date) = 1 THEN sales ELSE 0 END) AS jan_sales,
    SUM(CASE WHEN MONTH(order_date) = 2 THEN sales ELSE 0 END) AS feb_sales,
    SUM(CASE WHEN MONTH(order_date) = 3 THEN sales ELSE 0 END) AS mar_sales
FROM orders
GROUP BY product_id;

```

Use Case 6: Complex Conditions

```

SELECT
    customer_id,
    CASE
        WHEN total_orders > 10 AND total_spent > 1000 THEN 'VIP'
        WHEN total_orders > 5 OR total_spent > 500 THEN 'Regular'
        ELSE 'New'
    END AS customer_tier
FROM customer_summary;

```



CASE in Different Clauses

In SELECT (Create Columns)

```

SELECT
    name,
    CASE WHEN age >= 18 THEN 'Adult' ELSE 'Minor' END AS category
FROM users;

```

In WHERE (Filter Data)

```

SELECT * FROM products
WHERE
    CASE
        WHEN category = 'Electronics' THEN price > 100
        WHEN category = 'Clothing' THEN price > 50
        ELSE price > 20
    END;

```

In ORDER BY (Custom Sorting)

```
SELECT * FROM orders
ORDER BY
CASE status
    WHEN 'Urgent' THEN 1
    WHEN 'High' THEN 2
    WHEN 'Medium' THEN 3
    ELSE 4
END;
```

In GROUP BY

```
SELECT
CASE
    WHEN age < 20 THEN 'Teen'
    WHEN age < 40 THEN 'Adult'
    ELSE 'Senior'
END AS age_group,
COUNT(*) AS count
FROM customers
GROUP BY
CASE
    WHEN age < 20 THEN 'Teen'
    WHEN age < 40 THEN 'Adult'
    ELSE 'Senior'
END;
```

⚠ Rules and Best Practices

Rule: Matching Data Types

All THEN and ELSE results must have compatible data types:

```
-- ✗ Wrong (mixing string and number)
```

CASE

```
WHEN score > 50 THEN 'High'
```

```
ELSE 0
```

END

```
-- ✓ Correct (all strings)
```

CASE

```
WHEN score > 50 THEN 'High'
```

```
ELSE 'Low'
```

END

Best Practices

Practice	Reason
Order conditions correctly	First true wins
Always include ELSE	Avoid NULLs
Check highest first	Prevents overlap
Use aliases	Readable output

❓ Practice Questions

Q1: Categorize products by price.

```
SELECT product_name,
CASE
    WHEN price > 100 THEN 'Expensive'
    WHEN price > 50 THEN 'Moderate'
    ELSE 'Cheap'
END AS price_category
FROM products;
```

Q2: Map status codes to text.

```
SELECT order_id,
CASE status_code
    WHEN 1 THEN 'Open'
    WHEN 2 THEN 'In Progress'
    WHEN 3 THEN 'Closed'
END AS status
FROM orders;
```

Q3: Count orders by category.

```
SELECT
    SUM(CASE WHEN category = 'A' THEN 1 ELSE 0 END) AS cat_a,
    SUM(CASE WHEN category = 'B' THEN 1 ELSE 0 END) AS cat_b
FROM products;
```

Q4: Assign grades based on score.

```
SELECT student_name,
CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >= 80 THEN 'B'
    WHEN score >= 70 THEN 'C'
    WHEN score >= 60 THEN 'D'
    ELSE 'F'
END AS grade
FROM students;
```

Q5: Handle NULL in calculations.

```
SELECT
    AVG(CASE WHEN score IS NULL THEN 0 ELSE score END) AS avg_score
FROM students;
```



Interview Questions

Q1: What is the difference between Simple CASE and Searched CASE?

Answer:

Feature	Simple CASE	Searched CASE
Syntax	CASE expr WHEN val THEN...	CASE WHEN condition THEN...
Compares	One expression to values	Independent conditions
Operators	Equality only (=)	Any operator (>, <, LIKE, etc.)
NULL handling	Cannot check IS NULL	Can check IS NULL

```
-- Simple CASE: equals comparison only
CASE status
    WHEN 1 THEN 'Active'
    WHEN 2 THEN 'Inactive'
END

-- Searched CASE: any condition
CASE
    WHEN status = 1 AND verified = 1 THEN 'Active Verified'
    WHEN status = 1 THEN 'Active Unverified'
    WHEN status IS NULL THEN 'Unknown'
END
```

Q2: What happens if multiple WHEN conditions are true?

Answer: Only the **first true** condition's result is returned. Subsequent conditions are not evaluated.

```
-- score = 95
CASE
    WHEN score >= 70 THEN 'C' -- TRUE - returns 'C', stops here!
    WHEN score >= 80 THEN 'B' -- Never checked
    WHEN score >= 90 THEN 'A' -- Never checked
END
-- Returns 'C' (wrong!) because order is incorrect

-- Correct order:
CASE
    WHEN score >= 90 THEN 'A' -- Check highest first
    WHEN score >= 80 THEN 'B'
    WHEN score >= 70 THEN 'C'
END
```

Q3: Can CASE be used in aggregate functions?

Answer: Yes! This is called **conditional aggregation**:

```
-- Count orders by status in columns (pivot)
SELECT
    COUNT(CASE WHEN status = 'Completed' THEN 1 END) AS completed,
    COUNT(CASE WHEN status = 'Pending' THEN 1 END) AS pending,
    COUNT(CASE WHEN status = 'Cancelled' THEN 1 END) AS cancelled
FROM orders;

-- Sum sales by region
SELECT
    SUM(CASE WHEN region = 'East' THEN amount ELSE 0 END) AS east_sales,
    SUM(CASE WHEN region = 'West' THEN amount ELSE 0 END) AS west_sales
FROM orders;
```

Q4: Can all THEN/ELSE results have different data types?

Answer: No! All results must have **compatible data types**:

```
-- ✗ Error: mixing VARCHAR and INT
CASE WHEN status = 1 THEN 'Active' ELSE 0 END

-- ✓ Correct: all VARCHAR
CASE WHEN status = 1 THEN 'Active' ELSE 'Inactive' END

-- ✓ Correct: all INT
CASE WHEN status = 1 THEN 1 ELSE 0 END
```

SQL uses **data type precedence** to determine the final type.

Q5: How do you use CASE in ORDER BY?

Answer: For custom sorting order:

```
-- Sort by custom priority
SELECT * FROM tasks
ORDER BY
    CASE priority
        WHEN 'Critical' THEN 1
        WHEN 'High' THEN 2
        WHEN 'Medium' THEN 3
        WHEN 'Low' THEN 4
        ELSE 5
    END;

-- Sort NULLs to the end
SELECT * FROM customers
ORDER BY CASE WHEN phone IS NULL THEN 1 ELSE 0 END, phone;
```

Q6: Can you nest CASE statements?

Answer: Yes, CASE can be nested:

```

SELECT
CASE
    WHEN category = 'Electronics'
        THEN CASE
            WHEN price > 500 THEN 'Premium Electronics'
            ELSE 'Basic Electronics'
        END
    WHEN category = 'Clothing'
        THEN CASE
            WHEN price > 100 THEN 'Designer'
            ELSE 'Casual'
        END
    ELSE 'Other'
END AS product_tier
FROM products;

```

However, nested CASE can be hard to read - consider using a lookup table instead.

Q7: How is CASE different from IIF?

Answer: IIF is a shorthand for simple two-outcome CASE:

```

-- IIF: Simple true/false (SQL Server 2012+)
SELECT IIF(score >= 60, 'Pass', 'Fail') AS result;

-- Equivalent CASE:
SELECT CASE WHEN score >= 60 THEN 'Pass' ELSE 'Fail' END AS result;

```

Feature	IIF	CASE
Conditions	2 only	Unlimited
Readability	Simple cases	Complex logic
Portability	SQL Server only	All databases

Q8: How do you handle NULL in CASE?

Answer: Simple CASE cannot check for NULL; use Searched CASE:

```
-- ✗ Simple CASE: Cannot match NULL  
CASE status WHEN NULL THEN 'Missing' END -- Never matches!  
  
-- ✓ Searched CASE: Use IS NULL  
CASE WHEN status IS NULL THEN 'Missing' ELSE status END
```

Q9: Can CASE be used in UPDATE statements?

Answer: Yes, very useful for conditional updates:

```
-- Update prices based on category  
UPDATE products  
SET price = CASE  
    WHEN category = 'Electronics' THEN price * 1.10  
    WHEN category = 'Clothing' THEN price * 1.05  
    ELSE price * 1.02  
END;  
  
-- Update status based on multiple conditions  
UPDATE orders  
SET status = CASE  
    WHEN ship_date IS NOT NULL THEN 'Shipped'  
    WHEN payment_date IS NOT NULL THEN 'Paid'  
    ELSE 'Pending'  
END;
```

Q10: What is CHOOSE and how does it relate to CASE?

Answer: CHOOSE returns a value based on index position:

```
-- CHOOSE: Return nth value from list
SELECT CHOOSE(2, 'First', 'Second', 'Third'); -- Returns 'Second'

-- Useful for day names
SELECT CHOOSE(DATEPART(weekday, GETDATE())),
    'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat') AS day_name;

-- Equivalent CASE:
CASE DATEPART(weekday, GETDATE())
    WHEN 1 THEN 'Sun'
    WHEN 2 THEN 'Mon'
    -- etc.
END
```

🔑 Key Takeaways

1. **CASE = IF-ELSE** in SQL
2. **WHEN-THEN** pairs define conditions
3. **ELSE** is optional but recommended
4. **Order matters** - first true wins
5. **Data types must match** in all results
6. **Use everywhere** - SELECT, WHERE, ORDER BY, GROUP BY

📌 Common Patterns

Pattern 1: Binary Flag

```
CASE WHEN condition THEN 1 ELSE 0 END AS flag
```

Pattern 2: Null Replacement

```
CASE WHEN value IS NULL THEN 'default' ELSE value END
```

Pattern 3: Value Mapping

```
CASE column WHEN 'A' THEN 'Alpha' WHEN 'B' THEN 'Beta' END
```

Pattern 4: Range Categorization

```
CASE
    WHEN value > 100 THEN 'High'
    WHEN value > 50 THEN 'Medium'
    ELSE 'Low'
END
```



Exercises

Exercise 1:

Create a column showing "Weekend" or "Weekday".

```
SELECT order_date,
CASE
    WHEN DATEPART(weekday, order_date) IN (1, 7) THEN 'Weekend'
    ELSE 'Weekday'
END AS day_type
FROM orders;
```

Exercise 2:

Calculate total sales for each quarter.

```
SELECT
```

```
    SUM(CASE WHEN MONTH(order_date) IN (1,2,3) THEN sales ELSE 0 END) AS Q1,  
    SUM(CASE WHEN MONTH(order_date) IN (4,5,6) THEN sales ELSE 0 END) AS Q2,  
    SUM(CASE WHEN MONTH(order_date) IN (7,8,9) THEN sales ELSE 0 END) AS Q3,  
    SUM(CASE WHEN MONTH(order_date) IN (10,11,12) THEN sales ELSE 0 END) AS Q4
```

```
FROM orders;
```

Exercise 3:

Custom sort by priority.

```
SELECT * FROM tasks  
ORDER BY  
CASE priority  
    WHEN 'Critical' THEN 1  
    WHEN 'High' THEN 2  
    WHEN 'Medium' THEN 3  
    ELSE 4  
END;
```

Note 15: Aggregate Functions Overview

Overview

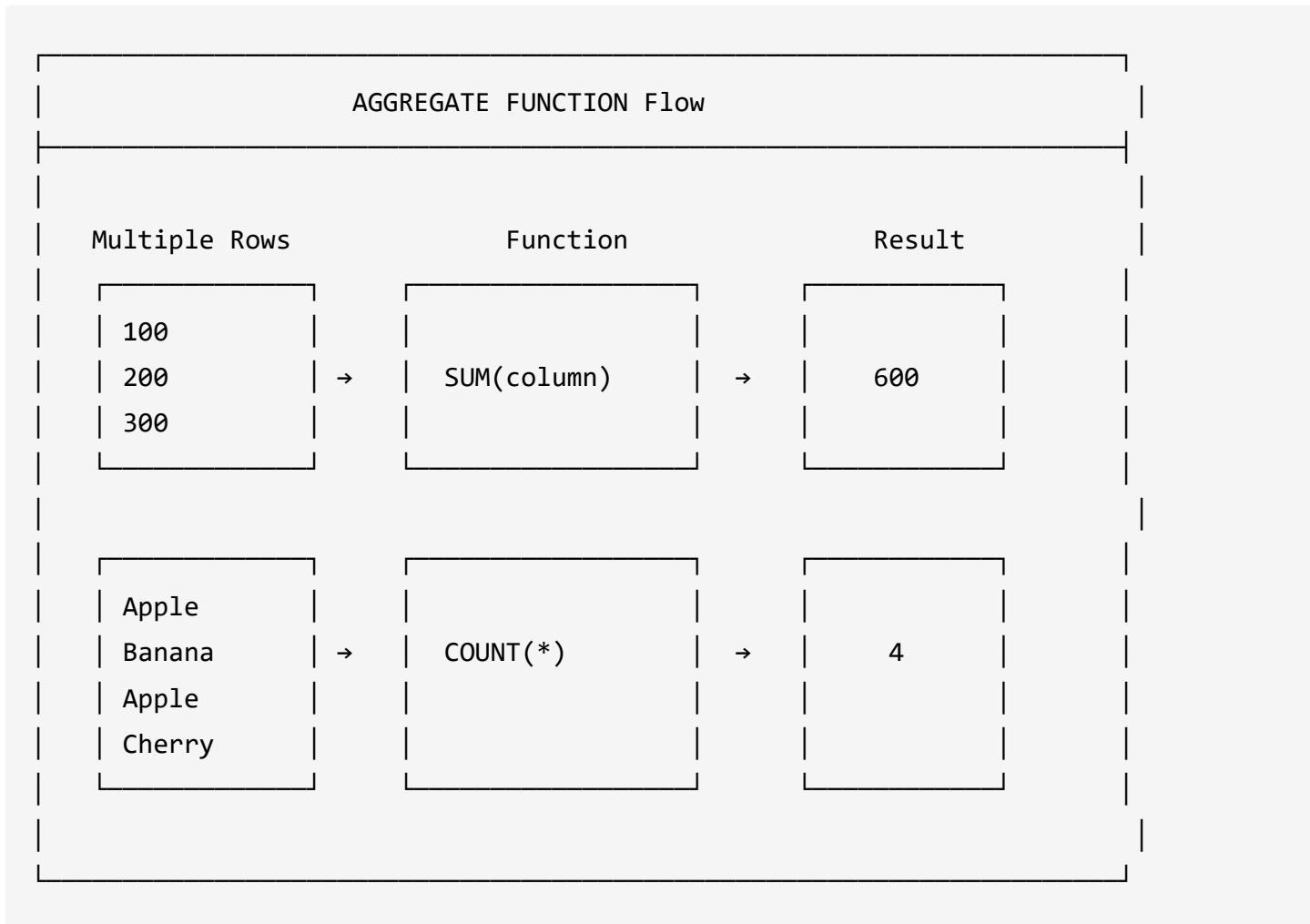
This note covers basic aggregate functions in SQL - essential tools for summarizing data and uncovering insights about your business.



Theory

What are Aggregate Functions?

Aggregate functions collapse multiple rows into a single summary value:



Aggregate vs Scalar Functions

Type	Input	Output	Example
Scalar	1 row	1 value per row	<code>UPPER('hello') → 'HELLO'</code>
Aggregate	Many rows	1 value total	<code>SUM(sales) → 5000</code>
Window	Many rows	1 value per row	<code>SUM(sales) OVER ()</code>

The Aggregation Scope

AGGREGATION LEVELS

1. TABLE-LEVEL (no GROUP BY):

```
SELECT SUM(sales) FROM orders;
→ One value for entire table
```

2. GROUP-LEVEL (with GROUP BY):

```
SELECT region, SUM(sales) FROM orders GROUP BY region;
→ One value per group
```

3. WINDOW-LEVEL (with OVER):

```
SELECT SUM(sales) OVER (PARTITION BY region) FROM orders;
→ Value for each row (within its group context)
```

NULL Behavior - Critical!

All aggregate functions **ignore NULL values** except COUNT(*):

```
-- Data: [100, NULL, 200, NULL, 300]

COUNT(*)          -- 5 (counts rows including NULLs)
COUNT(column)    -- 3 (counts non-NULL values)
SUM(column)       -- 600 (100+200+300, ignores NULLs)
AVG(column)       -- 200 (600÷3, NOT 600÷5!)
MIN(column)       -- 100
MAX(column)       -- 300
```

Complete Aggregate Function Reference

Function	Purpose	NULL Handling	Works On
COUNT(*)	Count rows	Includes NULL rows	Any

Function	Purpose	NULL Handling	Works On
COUNT(col)	Count values	Ignores NULLs	Any
COUNT(DISTINCT)	Count unique	Ignores NULLs	Any
SUM()	Total	Ignores NULLs	Numeric
AVG()	Average	Ignores NULLs	Numeric
MIN()	Minimum	Ignores NULLs	Any sortable
MAX()	Maximum	Ignores NULLs	Any sortable
STRING_AGG()	Concatenate	Ignores NULLs	String
STDEV()	Std deviation	Ignores NULLs	Numeric
VAR()	Variance	Ignores NULLs	Numeric



Core Aggregate Functions

Function	Purpose	Example Output
COUNT(*)	Count rows	100
SUM()	Total of values	5000
AVG()	Average of values	50.5
MIN()	Lowest value	10
MAX()	Highest value	200



Function Details

1. COUNT - Count Rows

```
-- Count all rows
SELECT COUNT(*) AS total_orders
FROM orders;

-- Count specific column (ignores NULLs)
SELECT COUNT(customer_id) AS customers_with_orders
FROM orders;

-- Count distinct values
SELECT COUNT(DISTINCT customer_id) AS unique_customers
FROM orders;
```

COUNT variations:

Syntax	Behavior
COUNT(*)	Counts all rows including NULLs
COUNT(column)	Counts non-NULL values only
COUNT(DISTINCT col)	Counts unique non-NULL values

2. SUM - Total Values

```
-- Total sales
SELECT SUM(sales) AS total_sales
FROM orders;

-- Total by category
SELECT category, SUM(sales) AS category_total
FROM orders
GROUP BY category;
```

⚠ SUM only works with numeric data types.

3. AVG - Average Value

```
-- Average sales
SELECT AVG(sales) AS average_sales
FROM orders;

-- Average score per department
SELECT department, AVG(score) AS avg_score
FROM employees
GROUP BY department;
```

Note: AVG ignores NULL values in calculation.

4. MIN - Minimum Value

```
-- Lowest price
SELECT MIN(price) AS lowest_price
FROM products;

-- Earliest order date
SELECT MIN(order_date) AS first_order
FROM orders;
```

5. MAX - Maximum Value

```
-- Highest sales
SELECT MAX(sales) AS highest_sales
FROM orders;

-- Most recent order
SELECT MAX(order_date) AS latest_order
FROM orders;
```

Using with GROUP BY

Aggregate functions become powerful when combined with GROUP BY:

```
-- Sales summary by product
SELECT
    product_id,
    COUNT(*) AS order_count,
    SUM(sales) AS total_sales,
    AVG(sales) AS avg_sales,
    MIN(sales) AS min_sale,
    MAX(sales) AS max_sale
FROM orders
GROUP BY product_id;
```

Output:

product_id	order_count	total_sales	avg_sales	min_sale	max_sale
101	4	140	35	10	90
102	3	105	35	15	60



Multi-Level Aggregation

You can group by multiple columns:

```
SELECT
    product_id,
    order_status,
    SUM(sales) AS total_sales
FROM orders
GROUP BY product_id, order_status
ORDER BY product_id, order_status;
```

🎯 Practical Examples

Example 1: Business Overview

```
SELECT
    COUNT(*) AS total_orders,
    SUM(sales) AS total_revenue,
    AVG(sales) AS avg_order_value,
    MAX(sales) AS largest_order,
    MIN(sales) AS smallest_order
FROM orders;
```

Example 2: Customer Analysis

```
SELECT
    customer_id,
    COUNT(*) AS orders,
    SUM(sales) AS total_spent,
    AVG(sales) AS avg_order
FROM orders
GROUP BY customer_id
ORDER BY total_spent DESC;
```

Example 3: Monthly Report

```
SELECT  
    MONTH(order_date) AS month,  
    COUNT(*) AS orders,  
    SUM(sales) AS revenue  
FROM orders  
GROUP BY MONTH(order_date)  
ORDER BY month;
```

⚠ NULL Handling

All aggregate functions (except COUNT(*)) ignore NULLs:

```
-- Example data: 10, 20, NULL, 30  
SELECT  
    COUNT(*) AS count_all,          -- Returns: 4  
    COUNT(value) AS count_values,   -- Returns: 3  
    SUM(value) AS total,           -- Returns: 60  
    AVG(value) AS average         -- Returns: 20 (60/3, not 60/4)  
FROM sample_table;
```

To include NULLs as zero:

```
SELECT AVG(COALESCE(score, 0)) AS avg_including_nulls  
FROM customers;
```

❓ Practice Questions

Q1: Find total number of orders.

```
SELECT COUNT(*) AS total_orders FROM orders;
```

Q2: Find total sales amount.

```
SELECT SUM(sales) AS total_sales FROM orders;
```

Q3: Find average sales per customer.

```
SELECT customer_id, AVG(sales) AS avg_sales  
FROM orders  
GROUP BY customer_id;
```

Q4: Find highest and lowest scores.

```
SELECT MAX(score) AS highest, MIN(score) AS lowest  
FROM customers;
```

Q5: Count unique products ordered.

```
SELECT COUNT(DISTINCT product_id) AS unique_products  
FROM orders;
```



Interview Questions

Q1: What is the difference between COUNT(*), COUNT(column), and COUNT(DISTINCT)?

Answer:

Syntax	Counts	NULL Handling
COUNT(*)	All rows	Includes NULLs
COUNT(column)	Values in column	Ignores NULLs
COUNT(DISTINCT col)	Unique values	Ignores NULLs

```
-- Table: id=1,name='A' | id=2,name='A' | id=3,name=NULL

SELECT COUNT(*) AS all_rows;          -- 3
SELECT COUNT(name) AS non_null;       -- 2
SELECT COUNT(DISTINCT name) AS unique; -- 1 ('A' counted once)
```

Q2: Why does AVG ignore NULLs and is this always desired?

Answer: AVG calculates sum/count of non-NULL values only.

```
-- Scores: [80, NULL, 100]
SELECT AVG(score); -- Returns 90 (180/2, not 180/3)

-- When NULLs should be treated as 0:
SELECT AVG(COALESCE(score, 0)); -- Returns 60 (180/3)
```

Context matters:

- NULL = "unknown score" → Ignore (default behavior)
- NULL = "zero score" → Use COALESCE

Q3: Can you use aggregate functions in WHERE clause?

Answer: No! Aggregates require grouped data, but WHERE runs before GROUP BY.

```
-- ✗ Error: Cannot use aggregate in WHERE
SELECT * FROM orders WHERE SUM(sales) > 1000;

-- ✓ Correct: Use HAVING
SELECT customer_id, SUM(sales)
FROM orders
GROUP BY customer_id
HAVING SUM(sales) > 1000;
```

Q4: What happens when you use an aggregate on an empty set?

Answer:

Function	Empty Set Result
COUNT(*)	0
COUNT(col)	0
SUM()	NULL
AVG()	NULL
MIN()	NULL
MAX()	NULL

```
SELECT
    COUNT(*) AS cnt,    -- 0
    SUM(sales)          -- NULL
FROM orders
WHERE 1 = 0;    -- No rows match
```

Q5: Can you use aggregate functions with DISTINCT?

Answer: Yes, for SUM, AVG, and COUNT:

```
-- Data: sales = [100, 100, 200, 200, 300]

SELECT SUM(sales);           -- 900 (all values)
SELECT SUM(DISTINCT sales); -- 600 (100+200+300)

SELECT AVG(sales);          -- 180 (900/5)
SELECT AVG(DISTINCT sales); -- 200 (600/3)
```

Q6: What is STRING_AGG and when would you use it?

Answer: STRING_AGG concatenates values from multiple rows into one string:

```
-- Combine product names per order
SELECT order_id, STRING_AGG(product_name, ', ') AS products
FROM order_details
GROUP BY order_id;

-- Result: 'Apple, Banana, Cherry' (one row per order)

-- With ordering (SQL Server 2017+)
SELECT STRING_AGG(product_name, ', ') WITHIN GROUP (ORDER BY product_name)
FROM order_details;
```

Q7: How do you calculate percentage of total using aggregates?

Answer:

```
-- Method 1: Subquery
SELECT
    category,
    SUM(sales) AS category_sales,
    SUM(sales) * 100.0 / (SELECT SUM(sales) FROM orders) AS percent_of_total
FROM orders
GROUP BY category;

-- Method 2: Window function
SELECT
    category,
    SUM(sales) AS category_sales,
    SUM(sales) * 100.0 / SUM(SUM(sales)) OVER () AS percent_of_total
FROM orders
GROUP BY category;
```

Q8: Can you nest aggregate functions?

Answer: Not directly, but you can use subqueries or window functions:

```
-- ✗ Error: Cannot nest aggregates
SELECT MAX(SUM(sales)) FROM orders GROUP BY customer_id;

-- ✓ Correct: Use subquery
SELECT MAX(total_sales) FROM (
    SELECT customer_id, SUM(sales) AS total_sales
    FROM orders
    GROUP BY customer_id
) t;

-- ✓ Alternative: Window function
SELECT DISTINCT MAX(SUM(sales)) OVER ()
FROM orders
GROUP BY customer_id;
```

Q9: What is the difference between STDEV and STDEVP?

Answer:

Function	Type	Use When
STDEV()	Sample	Data is a sample of population
STDEVP()	Population	Data is the complete population

SELECT

```
STDEV(score) AS sample_std,      -- Divides by (n-1)
STDEVP(score) AS pop_std        -- Divides by n
FROM students;
```

Q10: How do you find the second highest value?

Answer:

```
-- Method 1: Subquery with MAX
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Method 2: OFFSET-FETCH
SELECT salary FROM employees
ORDER BY salary DESC
OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY;

-- Method 3: DENSE_RANK (handles ties)
SELECT salary FROM (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
    FROM employees
) t
WHERE rnk = 2;
```



Key Takeaways

1. **COUNT(*)** - Counts all rows
 2. **SUM** - Totals numeric values
 3. **AVG** - Calculates average
 4. **MIN/MAX** - Finds extremes
 5. **NULLs ignored** - Except COUNT(*)
 6. **GROUP BY** - Creates categories
-



Best Practices

Practice	Reason
Use COUNT(*) for row counts	Includes NULLs
Use COUNT(column) for values	Excludes NULLs
Handle NULLs before AVG	Accurate results
Combine with GROUP BY	Meaningful insights
Use HAVING for filtering	Filter aggregated results



Exercises

Exercise 1:

Find total customers and average score.

```

SELECT
    COUNT(*) AS total_customers,
    AVG(score) AS avg_score
FROM customers;
  
```

Exercise 2:

Find sales statistics by product.

```
SELECT
    product_id,
    COUNT(*) AS orders,
    SUM(sales) AS total,
    AVG(sales) AS average
FROM orders
GROUP BY product_id;
```

Exercise 3:

Find customers with more than 3 orders.

```
SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 3;
```

Note 16: Window Functions Introduction

Overview

Window Functions (also called Analytical Functions) allow you to perform calculations across subsets of data **without losing row-level details** - unlike GROUP BY which collapses rows.



Theory

What are Window Functions?

Window Functions (also called Analytical or OLAP Functions) perform calculations across a "window" of rows while **preserving all original rows** in the output.

GROUP BY vs WINDOW FUNCTION

GROUP BY:

Rows → Collapse → Summary

10	
20	
30	

→

60	
----	--

3 rows → 1 row

WINDOW FUNCTION:

Rows → Calculate → Same Rows

+ New Column

10		10		60
20		20		60
30		30		60

3 rows → 3 rows + SUM

The "Window" Concept

A **window** is a subset of rows that the function operates on:

Full Data

Caps	10
Caps	30
Gloves	5
Gloves	20
Hats	15

PARTITION BY product

→

Window 1: Caps	10, 30	SUM = 40
Window 2: Gloves	5, 20	SUM = 25
Window 3: Hats	15	SUM = 15

Window Function Categories

Category	Functions	Purpose
Aggregate	SUM, AVG, COUNT, MIN, MAX	Summarize over window
Ranking	ROW_NUMBER, RANK, DENSE_RANK, NTILE	Assign positions
Value	LAG, LEAD, FIRST_VALUE, LAST_VALUE	Access other rows
Distribution	PERCENT_RANK, CUME_DIST	Statistical distribution

Group By vs Window Functions

Feature	GROUP BY	Window Functions
Output rows	Collapsed (fewer rows)	Same as input
Detail preservation	Lost	Retained <input checked="" type="checkbox"/>
Can show original values	No	Yes
Ranking functions	Not available	Available
Access other rows	Not possible	LAG/LEAD/FIRST/LAST
Use case	Final summary	Analytics + details



Visual Comparison

GROUP BY Behavior:

Input: 4 orders → Output: 2 product rows

OrderID	Product	Sales		Product	Total Sales
1	Caps	10	→	Caps	40
2	Caps	30		Gloves	25
3	Gloves	5			
4	Gloves	20			

Window Function Behavior:

Input: 4 orders

→ Output: 4 orders + calculation

OrderID	Product	Sales		OrderID	Product	Sales	Total Sales
1	Caps	10	→	1	Caps	10	40
2	Caps	30		2	Caps	30	40
3	Gloves	5		3	Gloves	5	25
4	Gloves	20		4	Gloves	20	25



Why Use Window Functions?

Problem with GROUP BY:

-- Task: Find total sales per product WITH order details

```
SELECT order_id, order_date, product_id, SUM(sales) AS total_sales
FROM orders
GROUP BY product_id; -- ERROR! order_id and order_date not in GROUP BY
```

Adding all columns to GROUP BY breaks aggregation:

```
SELECT order_id, order_date, product_id, SUM(sales) AS total_sales
FROM orders
GROUP BY order_id, order_date, product_id; -- NOT aggregated properly!
```

Solution with Window Functions:

```
SELECT
    order_id,
    order_date,
    product_id,
    SUM(sales) OVER(PARTITION BY product_id) AS total_sales
FROM orders; -- Works! All details + aggregation
```



Basic Syntax

```
SELECT
    column1,
    column2,
    FUNCTION() OVER(
        [PARTITION BY column]
        [ORDER BY column]
        [ROWS BETWEEN ... AND ...]
    ) AS alias
FROM table;
```

Syntax Components:

Component	Purpose	Required
FUNCTION()	The calculation (SUM, AVG, RANK, etc.)	Yes
OVER()	Declares window function	Yes
PARTITION BY	Divides data into groups/windows	No
ORDER BY	Sorts within partition	Depends on function

Component	Purpose	Required
ROWS BETWEEN	Defines frame boundaries	No



Function Categories

1. Aggregate Functions

Same as GROUP BY but over window:

- `COUNT()` - Count rows
- `SUM()` - Total values
- `AVG()` - Average value
- `MIN()` - Minimum value
- `MAX()` - Maximum value

2. Ranking Functions

Assign ranks to rows:

- `ROW_NUMBER()` - Unique sequential number
- `RANK()` - Rank with gaps for ties
- `DENSE_RANK()` - Rank without gaps
- `NTILE(n)` - Divide into n buckets
- `PERCENT_RANK()` - Relative rank as percentage
- `CUME_DIST()` - Cumulative distribution

3. Value/Analytical Functions

Access values from other rows:

- `LAG()` - Previous row value
- `LEAD()` - Next row value
- `FIRST_VALUE()` - First value in window
- `LAST_VALUE()` - Last value in window

🎯 Practical Examples

Example 1: Simple Window (Entire Dataset)

```
-- Total sales across ALL orders on each row
SELECT
    order_id,
    sales,
    SUM(sales) OVER() AS total_all_sales
FROM orders;
```

Example 2: Partitioned Window

```
-- Total sales per product, keeping row details
SELECT
    order_id,
    product_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total
FROM orders;
```

Example 3: Adding Order Details

```
-- Compare individual sale to product total
SELECT
    order_id,
    order_date,
    product_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,
    sales * 100.0 / SUM(sales) OVER(PARTITION BY product_id) AS pct_of_product
FROM orders;
```



Key Differences Summary

Aspect	GROUP BY	Window Functions
Use Case	Simple aggregation	Advanced analytics
Row Count	Reduced	Same
Extra Columns	Not allowed	Allowed freely
Complexity	Simple	More complex
Performance	Generally faster	May be slower

❓ Practice Questions

Q1: What's the main difference between GROUP BY and Window Functions?

Answer: GROUP BY collapses rows into groups, losing row-level detail. Window functions keep all original rows while adding calculations.

Q2: When should you use Window Functions instead of GROUP BY?

Answer: When you need both aggregated values AND row-level details in the same result set.

Q3: What does the OVER() clause indicate?

Answer: OVER() tells SQL that you're using a window function, not a regular aggregate.



Interview Questions

Q1: Can you explain the difference between GROUP BY and window functions?

Answer:

Aspect	GROUP BY	Window Function
Rows returned	One per group	All original rows
Detail data	Lost	Preserved
Calculation scope	Each group	Each row in context

```
-- GROUP BY: 2 rows (one per product)
SELECT product, SUM(sales) FROM orders GROUP BY product;

-- Window: All rows + product totals
SELECT *, SUM(sales) OVER(PARTITION BY product)
FROM orders;
```

Q2: When does a window function execute in query order?

Answer: Window functions execute **after** WHERE, GROUP BY, and HAVING, but **before** ORDER BY and DISTINCT.

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT (Window functions calculated here!)
6. DISTINCT
7. ORDER BY
8. TOP/LIMIT

This is why you can't use window function results in WHERE (use subquery instead).

Q3: Can you use window functions in WHERE or HAVING?

Answer: No! Window functions execute during SELECT, after WHERE/HAVING.

```
-- ✗ Error: Window function in WHERE
SELECT * FROM orders WHERE ROW_NUMBER() OVER(ORDER BY date) <= 10;

-- ✓ Correct: Use subquery
SELECT * FROM (
    SELECT *, ROW_NUMBER() OVER(ORDER BY date) AS rn FROM orders
) t WHERE rn <= 10;
```

Q4: What happens if you omit PARTITION BY?

Answer: The window becomes the **entire result set**:

```
-- With PARTITION BY: sum per product
SUM(sales) OVER(PARTITION BY product) -- Window = each product group

-- Without PARTITION BY: sum of all rows
SUM(sales) OVER() -- Window = entire result set
```

Q5: Can you use multiple window functions with different partitions?

Answer: Yes! Each OVER() defines its own window:

```
SELECT
    order_id,
    product_id,
    customer_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,
    SUM(sales) OVER(PARTITION BY customer_id) AS customer_total,
    SUM(sales) OVER() AS grand_total
FROM orders;
```

Q6: What is a named window (WINDOW clause)?

Answer: You can define a window once and reuse it:

```
-- Without named window (repetitive)
SELECT
    SUM(sales) OVER(PARTITION BY product ORDER BY date),
    AVG(sales) OVER(PARTITION BY product ORDER BY date),
    COUNT(*) OVER(PARTITION BY product ORDER BY date)
FROM orders;

-- With named window (SQL Server 2022+, PostgreSQL, etc.)
SELECT
    SUM(sales) OVER w,
    AVG(sales) OVER w,
    COUNT(*) OVER w
FROM orders
WINDOW w AS (PARTITION BY product ORDER BY date);
```

Q7: Do window functions affect query performance?

Answer: Yes, window functions can be resource-intensive:

Performance Factor	Impact
Sorting	ORDER BY in OVER requires sorting
Memory	Keeps all rows in memory for window
Multiple windows	Different partitions = multiple passes

Optimization tips:

- Create indexes on PARTITION BY and ORDER BY columns
- Limit result set with WHERE before applying windows
- Use same window definition where possible

Q8: Can aggregate and window functions coexist in the same query?

Answer: Yes, but they operate at different levels:

```

SELECT
    product_id,
    SUM(sales) AS product_total,                                -- Aggregate (GROUP BY level)
    SUM(SUM(sales)) OVER() AS grand_total                      -- Window over aggregated result
FROM orders
GROUP BY product_id;

```

Note: The inner SUM is the aggregate, the outer SUM OVER() is the window function on the grouped result.

Q9: What is the difference between ROWS and RANGE in frame clause?

Answer:

Clause	Operates On	Boundary
ROWS	Physical position	Exact row count
RANGE	Logical value	All rows with same value

```
-- ROWS: Exactly 2 preceding rows
SUM(sales) OVER(ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
```

```
-- RANGE: All rows with same or lesser date value
SUM(sales) OVER(ORDER BY date RANGE BETWEEN 2 PRECEDING AND CURRENT ROW)
```

Q10: How do you calculate running totals and moving averages?

Answer:

```
-- Running total (cumulative sum)
SELECT
    date,
    sales,
    SUM(sales) OVER(ORDER BY date ROWS UNBOUNDED PRECEDING) AS running_total
FROM orders;

-- 3-day moving average
SELECT
    date,
    sales,
    AVG(sales) OVER(ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS movir
FROM orders;
```



🔑 Key Takeaways

1. **Window functions preserve row details** - no data is lost
2. **OVER()** is required to declare a window function
3. **PARTITION BY** divides data into windows (like GROUP BY but different)
4. **Same functions** work for both GROUP BY and windows
5. **More functions available** for windows (ranking, value access)
6. **Combine details + aggregates** in one query

📌 Best Practices

Practice	Reason
Use window when you need details	Preserves row-level data
Use GROUP BY for simple totals	Simpler, often faster
Name window results clearly	Improves readability

Practice	Reason
Start simple, add complexity	Easier debugging
Consider performance	Windows can be resource-intensive



Exercises

Exercise 1: Basic Window

Show all orders with total sales across entire dataset.

```
SELECT *, SUM(sales) OVER() AS total_sales  
FROM orders;
```

Exercise 2: Partitioned Window

Show all orders with total sales per customer.

```
SELECT  
    order_id,  
    customer_id,  
    sales,  
    SUM(sales) OVER(PARTITION BY customer_id) AS customer_total  
FROM orders;
```

Exercise 3: Multiple Windows

Show sales with both product total and grand total.

```
SELECT
    product_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,
    SUM(sales) OVER() AS grand_total
FROM orders;
```

Note 17: PARTITION BY and ORDER BY in Windows

Overview

This note covers the PARTITION BY and ORDER BY clauses within window functions - essential for defining how data is grouped and sorted for calculations.

PARTITION BY Clause

What is PARTITION BY?

PARTITION BY divides the entire dataset into smaller groups (partitions/windows) for separate calculations.

PARTITION BY Visualization

Without PARTITION BY:

All rows = 1 window
A 10
A 20
B 15
B 15

SUM=60

All rows get 60

With PARTITION BY product:

Product A = Window 1
A 10
A 20
Product B = Window 2
B 15
B 15

SUM=30

SUM=30

Each partition gets its own SUM

Think of it as: "*Calculate this separately for each group*"

Without PARTITION BY:

The entire dataset is treated as **one window**.

```
-- Total sales across ALL orders
SELECT
    order_id,
    sales,
    SUM(sales) OVER() AS total_all -- Entire table = one window
FROM orders;
```

Result: Every row shows the same grand total.

With PARTITION BY:

Data is divided into **multiple windows** based on column values.

```
-- Total sales per PRODUCT
SELECT
    order_id,
    product_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total
FROM orders;
```

Result: Each row shows total for its specific product.



Visual Understanding

Partitioning by Month:

Original Data:

Month	Product	Sales
January	A	20
January	B	10
February	A	30
February	B	20

Two Separate Windows:

Window 1 (January):

Month	Product	Sales
January	A	20
January	B	10

Window 2 (February):

Month	Product	Sales
February	A	30
February	B	20



PARTITION BY Options

Option 1: No Partition (Entire Dataset)

```
-- Grand total for all rows  
SELECT *, SUM(sales) OVER() AS grand_total  
FROM orders;
```

Option 2: Single Column Partition

```
-- Total per product  
SELECT *, SUM(sales) OVER(PARTITION BY product_id) AS product_total  
FROM orders;
```

Option 3: Multiple Column Partition

```
-- Total per product AND status combination  
SELECT *,  
    SUM(sales) OVER(PARTITION BY product_id, order_status) AS combo_total  
FROM orders;
```



ORDER BY in Windows

What is ORDER BY in Windows?

ORDER BY within OVER() sorts rows **within each partition** for calculations.

Why is ORDER BY Important?

- Determines sequence for ranking functions
- Enables running totals and moving calculations
- Required for ROW_NUMBER, RANK, etc.



ORDER BY Behavior

Without ORDER BY:

Aggregate applies to **entire partition at once**.

```
SELECT
    order_date,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS total
FROM orders;
-- Every row in partition shows same total
```

With ORDER BY:

Creates **cumulative/running calculation**.

```
SELECT
    order_date,
    sales,
    SUM(sales) OVER(PARTITION BY product_id ORDER BY order_date) AS running_total
FROM orders;
-- Each row shows total UP TO that point
```



Running Total Example

Ordered Data with Running Total:

Date	Sales	Running Total	
Jan 1	10	10	← 10
Jan 5	20	30	← 10 + 20
Jan 10	15	45	← 10 + 20 + 15
Jan 15	25	70	← 10 + 20 + 15 + 25



ORDER BY Options

Ascending Order (Default):

```
SUM(sales) OVER(ORDER BY order_date ASC) AS running_total
```

Descending Order:

```
SUM(sales) OVER(ORDER BY order_date DESC) AS reverse_running
```

Multiple Sort Columns:

```
SUM(sales) OVER(ORDER BY order_date, order_id) AS running_total
```

🎯 Combined Examples

Example 1: Running Total per Customer

```
SELECT
    customer_id,
    order_date,
    sales,
    SUM(sales) OVER(
        PARTITION BY customer_id
        ORDER BY order_date
    ) AS customer_running_total
FROM orders;
```

Example 2: Ranking within Product

```
SELECT
    product_id,
    order_date,
    sales,
    ROW_NUMBER() OVER(
        PARTITION BY product_id
        ORDER BY sales DESC
    ) AS sales_rank
FROM orders;
```

Example 3: Multiple Aggregations

```
SELECT
    order_date,
    product_id,
    sales,
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,
    SUM(sales) OVER(PARTITION BY product_id ORDER BY order_date) AS running_total,
    SUM(sales) OVER() AS grand_total
FROM orders;
```



Syntax Rules Summary

Function Type	PARTITION BY	ORDER BY
Aggregate (SUM, AVG, etc.)	Optional	Optional
Ranking (RANK, ROW_NUMBER)	Optional	Required
Value (LAG, LEAD)	Optional	Required

⚠ Common Patterns

Pattern 1: Percentage of Total

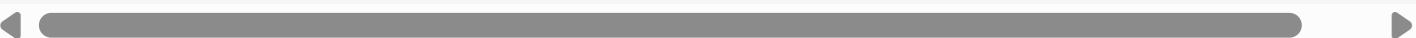
```
SELECT
    product_id,
    sales,
    sales * 100.0 / SUM(sales) OVER() AS pct_of_grand_total,
    sales * 100.0 / SUM(sales) OVER(PARTITION BY product_id) AS pct_of_product
FROM orders;
```

Pattern 2: Compare to Average

```
SELECT
    order_id,
    sales,
    AVG(sales) OVER() AS avg_sales,
    sales - AVG(sales) OVER() AS diff_from_avg
FROM orders;
```

Pattern 3: Find First and Last

```
SELECT
    product_id,
    order_date,
    sales,
    FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY order_date) AS first_sales,
    LAST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY order_date
                           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_sales
FROM orders;
```



❓ Practice Questions

Q1: Partition orders by customer and show customer total.

```
SELECT
    customer_id,
    order_id,
    sales,
    SUM(sales) OVER(PARTITION BY customer_id) AS customer_total
FROM orders;
```

Q2: Create running total of sales ordered by date.

```
SELECT
    order_date,
    sales,
    SUM(sales) OVER(ORDER BY order_date) AS running_total
FROM orders;
```

Q3: Rank products within each category by sales.

```
SELECT
```

```
category_id,
product_id,
sales,
RANK() OVER(PARTITION BY category_id ORDER BY sales DESC) AS category_rank
FROM products;
```

Interview Questions

Q1: What is the difference between PARTITION BY and GROUP BY?

Answer:

Aspect	GROUP BY	PARTITION BY
Output rows	One per group	All original rows
Usage	With aggregate functions	With window functions
Clause location	After FROM/WHERE	Inside OVER()
Row detail	Lost	Preserved

```
-- GROUP BY: Returns 2 rows (one per product)
```

```
SELECT product_id, SUM(sales) FROM orders GROUP BY product_id;
```

```
-- PARTITION BY: Returns all rows with product total added
```

```
SELECT product_id, sales, SUM(sales) OVER(PARTITION BY product_id)
FROM orders;
```

Q2: What happens when ORDER BY is added to an aggregate window function?

Answer: It creates a **running/cumulative calculation** instead of a total:

```
-- Without ORDER BY: Total for entire partition
SUM(sales) OVER(PARTITION BY product_id)

-- Every row shows the same product total

-- With ORDER BY: Running total
SUM(sales) OVER(PARTITION BY product_id ORDER BY order_date)

-- Each row shows cumulative sum up to that date
```

Q3: Why is ORDER BY required for ranking functions?

Answer: Ranking functions assign positions based on order. Without ORDER BY, there's no defined sequence:

```
-- ✗ Error in strict mode (or undefined behavior)
ROW_NUMBER() OVER(PARTITION BY product_id)

-- ✓ Correct: Defines ranking order
ROW_NUMBER() OVER(PARTITION BY product_id ORDER BY sales DESC)
```

Q4: Can you PARTITION BY multiple columns?

Answer: Yes! Each unique combination becomes a separate partition:

```
SELECT
    year, region, product,
    SUM(sales) OVER(PARTITION BY year, region) AS year_region_total
FROM orders;
-- Separate partition for: 2024-East, 2024-West, 2025-East, etc.
```

Q5: How does PARTITION BY handle NULLs?

Answer: All NULL values are grouped into the **same partition**:

```
-- If region has NULLs, all NULL region rows form one partition
SELECT region, sales,
       SUM(sales) OVER(PARTITION BY region) AS region_total
FROM orders;
-- NULL region rows get their own partition total
```

Q6: Can you use an expression in PARTITION BY?

Answer: Yes:

```
-- Partition by year extracted from date
SELECT
    order_date,
    sales,
    SUM(sales) OVER(PARTITION BY YEAR(order_date)) AS year_total
FROM orders;

-- Partition by calculated tier
SELECT
    sales,
    SUM(sales) OVER(PARTITION BY CASE WHEN sales > 100 THEN 'High' ELSE 'Low' END)
FROM orders;
```

Q7: How do you create a running total that resets each month?

Answer: PARTITION BY month, ORDER BY date:

```
SELECT
    order_date,
    sales,
    SUM(sales) OVER(
        PARTITION BY YEAR(order_date), MONTH(order_date)
        ORDER BY order_date
    ) AS monthly_running_total
FROM orders;
```

Q8: What's the difference between these two queries?

```
-- Query A
SELECT SUM(sales) OVER(ORDER BY date) FROM orders;

-- Query B
SELECT SUM(sales) OVER(ORDER BY date ROWS UNBOUNDED PRECEDING) FROM orders;
```

Answer: They produce the same result, but:

- Query A uses **implicit default frame** (RANGE UNBOUNDED PRECEDING)
- Query B uses **explicit ROWS frame**

The difference appears with tied values: RANGE includes all ties, ROWS counts exact physical rows.

Q9: How do you calculate percentage of partition total?

Answer:

```
SELECT
    region,
    sales,
    sales * 100.0 / SUM(sales) OVER(PARTITION BY region) AS pct_of_region,
    sales * 100.0 / SUM(sales) OVER() AS pct_of_total
FROM orders;
```

Q10: Can you mix aggregates and window functions in the same query?

Answer: Yes, but they operate at different levels:

```
SELECT
```

```
    region,
    SUM(sales) AS region_total,          -- Aggregate (GROUP BY level)
    SUM(SUM(sales)) OVER() AS grand_total,  -- Window on aggregated result
    SUM(sales) * 100.0 / SUM(SUM(sales)) OVER() AS pct_of_total
  FROM orders
 GROUP BY region;
```

Key Takeaways

1. **PARTITION BY** = GROUP BY equivalent for windows
2. **Without PARTITION BY** = entire dataset is one window
3. **ORDER BY** sorts within partitions
4. **ORDER BY** enables running/cumulative calculations
5. **ORDER BY is required** for ranking and value functions
6. **Can combine both** for sophisticated analysis

Best Practices

Practice	Reason
Use meaningful partitions	Logical grouping for analysis
Always specify ORDER BY for rankings	Prevents unpredictable results
Consider NULL handling	NULLs can affect partitioning
Test with small datasets first	Verify logic before scaling
Use aliases for clarity	Window results need good names



Exercises

Exercise 1: Product Analysis

Show each order with product total and grand total.

```
SELECT  
    order_id,  
    product_id,  
    sales,  
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,  
    SUM(sales) OVER() AS grand_total  
FROM orders;
```

Exercise 2: Customer Running Total

Show running total for each customer.

```
SELECT  
    customer_id,  
    order_date,  
    sales,  
    SUM(sales) OVER(  
        PARTITION BY customer_id  
        ORDER BY order_date  
    ) AS running_total  
FROM orders;
```

Exercise 3: Multi-Level Analysis

Show total by customer, by product, and overall.

```
SELECT
```

```
    customer_id,  
    product_id,  
    sales,  
    SUM(sales) OVER(PARTITION BY customer_id) AS customer_total,  
    SUM(sales) OVER(PARTITION BY product_id) AS product_total,  
    SUM(sales) OVER() AS grand_total  
FROM orders;
```

Note 18: Frame Clause (ROWS BETWEEN)

Overview

The Frame Clause defines a **subset of rows within a partition** for window function calculations. It gives fine-grained control over which rows are included in each calculation.

Theory

What is a Frame?

A **Frame** is a subset of rows relative to the current row that the window function uses for its calculation.

FRAME CLAUSE Concept

Full Partition: [Row1] [Row2] [Row3] [Row4] [Row5] [Row6]

↑
Current Row

Different Frame Examples for Current Row (Row3):

UNBOUNDED PRECEDING to CURRENT ROW (Running Total):

[Row1] [Row2] [Row3]
|—— Frame ——|

2 PRECEDING to CURRENT ROW (3-row Moving Average):

[Row1] [Row2] [Row3]
|—— Frame ——|

1 PRECEDING to 1 FOLLOWING (Centered Window):

[Row2] [Row3] [Row4]
|—— Frame ——|

UNBOUNDED PRECEDING to UNBOUNDED FOLLOWING (Entire Partition):

[Row1] [Row2] [Row3] [Row4] [Row5] [Row6]
|————— Frame —————|

Frame Boundary Keywords

Keyword	Meaning	Example Use
UNBOUNDED PRECEDING	First row of partition	Running totals
n PRECEDING	n rows before current	Moving averages
CURRENT ROW	The current row	Frame boundary
n FOLLOWING	n rows after current	Lead calculations

Keyword	Meaning	Example Use
UNBOUNDED FOLLOWING	Last row of partition	LAST_VALUE

ROWS vs RANGE vs GROUPS

Type	Based On	Behavior
ROWS	Physical position	Exact row count
RANGE	Value equality	All rows with same value
GROUPS	Peer groups	Groups of equal ORDER BY values

-- ROWS: Exactly 2 rows before current

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

-- RANGE: All rows with values within 2 of current (for numeric ORDER BY)

RANGE BETWEEN 2 PRECEDING AND CURRENT ROW

Best Practice: Use ROWS for predictable, consistent results.

Default Frame Behavior

When ORDER BY is present but no frame is specified:

- **Default is:** RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- This can cause unexpected behavior with tied values

-- These are equivalent:

SUM(sales) OVER(ORDER BY date)

SUM(sales) OVER(ORDER BY date RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

-- For predictable behavior, be explicit:

SUM(sales) OVER(ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)



Frame Clause Syntax

```
FUNCTION() OVER(  
    PARTITION BY column  
    ORDER BY column  
    ROWS BETWEEN start_point AND end_point  
)
```

Frame Keywords:

Keyword	Meaning
UNBOUNDED PRECEDING	First row of partition
n PRECEDING	n rows before current
CURRENT ROW	The current row
n FOLLOWING	n rows after current
UNBOUNDED FOLLOWING	Last row of partition



Visual Frame Examples

Example 1: UNBOUNDED PRECEDING to CURRENT ROW

All rows from start to current (Running Total):

Partition: [10] [20] [30] [40] [50]
 ↑
 Current Row

Frame: [10] [20] [30]
 |—— Included ——|

$$\text{SUM} = 10 + 20 + 30 = 60$$

Example 2: 2 PRECEDING to CURRENT ROW

Last 3 rows including current (Moving Average):

Partition: [10] [20] [30] [40] [50]
 ↑
 Current Row

Frame: [20] [30] [40]
 |—— Included ——|

$$\text{AVG} = (20 + 30 + 40) / 3 = 30$$

Example 3: UNBOUNDED PRECEDING to UNBOUNDED FOLLOWING

All rows in partition:

Partition: [10] [20] [30] [40] [50]



Current Row

Frame: [10] [20] [30] [40] [50]

|———— All Rows ———|

SUM = 10 + 20 + 30 + 40 + 50 = 150



Common Frame Patterns

1. Running Total (Cumulative Sum)

```
SUM(sales) OVER(
    ORDER BY order_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
```

Includes all rows from the start up to current row.

2. Rolling Total (Fixed Window)

```
SUM(sales) OVER(
    ORDER BY order_date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
)
```

Includes last 3 rows (2 before + current).

3. Moving Average

```
AVG(sales) OVER(  
    ORDER BY order_date  
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
)
```

Average of last 3 rows.

4. Entire Partition

```
SUM(sales) OVER(  
    PARTITION BY product_id  
    ORDER BY order_date  
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
)
```

All rows in the partition (required for LAST_VALUE).



Practical Examples

Example 1: Running Total

```
SELECT  
    order_date,  
    sales,  
    SUM(sales) OVER(  
        ORDER BY order_date  
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
    ) AS running_total  
FROM orders;
```

Output:

order_date	sales	running_total
Jan 1	100	100
Jan 2	150	250
Jan 3	200	450
Jan 4	50	500

Example 2: 3-Day Moving Average

```

SELECT
    order_date,
    sales,
    AVG(sales) OVER(
        ORDER BY order_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_3day
FROM orders;

```

Output:

order_date	sales	moving_avg_3day
Jan 1	100	100.00
Jan 2	150	125.00
Jan 3	200	150.00
Jan 4	50	133.33

Example 3: Running Total per Product

```

SELECT
    product_id,
    order_date,
    sales,
    SUM(sales) OVER(
        PARTITION BY product_id
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS product_running_total
FROM orders;

```

⚠ Default Frame Behavior

Important Rule:

When you use ORDER BY in a window function, SQL Server applies a **default frame**:

```

-- This:
SUM(sales) OVER(ORDER BY order_date)

-- Is equivalent to:
SUM(sales) OVER(
    ORDER BY order_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)

```

Without ORDER BY:

No frame is applied - entire partition is used.

```

-- This uses entire partition:
SUM(sales) OVER(PARTITION BY product_id)

```



Frame Types: ROWS vs RANGE

ROWS (Recommended)

- Physical rows based on position
- More predictable behavior

```
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

-- Always exactly 3 rows (if available)

RANGE

- Logical grouping based on values
- Can include multiple rows with same value

```
RANGE BETWEEN 2 PRECEDING AND CURRENT ROW
```

-- Rows with values within range

Tip: Use ROWS for predictable results. RANGE is rarely needed.



Use Cases

Use Case 1: Track Progress Over Time

```
-- Show cumulative revenue by month
SELECT
    month,
    revenue,
    SUM(revenue) OVER(
        ORDER BY month
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_revenue
FROM monthly_sales;
```

Use Case 2: Smoothing Data

```
-- 7-day moving average to smooth daily fluctuations
SELECT
    date,
    visitors,
    AVG(visitors) OVER(
        ORDER BY date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS weekly_avg
FROM daily_traffic;
```

Use Case 3: Compare to Window Stats

```
-- Compare current to min/max in rolling window
SELECT
    date,
    price,
    MIN(price) OVER(
        ORDER BY date
        ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
    ) AS min_5day,
    MAX(price) OVER(
        ORDER BY date
        ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
    ) AS max_5day
FROM stock_prices;
```

❓ Practice Questions

Q1: Calculate running total of orders.

```
SELECT
    order_date,
    amount,
    SUM(amount) OVER(
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM orders;
```

Q2: Calculate 5-day moving average.

```

SELECT
    date,
    value,
    AVG(value) OVER(
        ORDER BY date
        ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
    ) AS moving_avg_5
FROM daily_data;

```

Q3: Calculate rolling sum of last 3 orders per customer.

```

SELECT
    customer_id,
    order_date,
    amount,
    SUM(amount) OVER(
        PARTITION BY customer_id
        ORDER BY order_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS rolling_3_total
FROM orders;

```

🎯 Interview Questions

Q1: What is the difference between ROWS and RANGE?

Answer:

Aspect	ROWS	RANGE
Based on	Physical row position	Logical value
With ties	Counts each row separately	Includes all ties

Aspect	ROWS	RANGE
Predictability	<input checked="" type="checkbox"/> Always predictable	Can vary with data

-- Data: date=Jan-1, Jan-1, Jan-2, Jan-3 (two rows have same date)

-- Current row: first Jan-1

-- ROWS BETWEEN 1 PRECEDING AND CURRENT ROW

-- Returns: exactly 2 rows

-- RANGE BETWEEN 1 PRECEDING AND CURRENT ROW

-- Returns: all rows with same date value (may be 2+ rows)

Q2: What is the default frame when ORDER BY is specified?

Answer: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

This means:

- All preceding rows with values less than current
- All rows with the same value as current (ties)

Important: This default can cause unexpected results! Always specify explicit frame.

Q3: How do you calculate a 7-day moving average?

Answer:

```

SELECT
    date,
    sales,
    AVG(sales) OVER(
        ORDER BY date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7day
FROM daily_sales;

-- Note: 6 PRECEDING + CURRENT ROW = 7 rows
-- First 6 rows will have fewer than 7 values

```

Q4: How do you calculate Year-to-Date (YTD) totals?

Answer:

```

SELECT
    month,
    sales,
    SUM(sales) OVER(
        PARTITION BY YEAR(date)
        ORDER BY month
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS ytd_sales
FROM monthly_sales;

```

Q5: Why does LAST_VALUE often return the current row's value?

Answer: Because the default frame ends at CURRENT ROW:

```
-- ❌ Often returns current row (not truly last)
LAST_VALUE(sales) OVER(ORDER BY date)
-- Default frame is UNBOUNDED PRECEDING to CURRENT ROW

-- ✅ Correct: Extend frame to end of partition
LAST_VALUE(sales) OVER(
    ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)
```

Q6: How do you create a centered moving average?

Answer:

```
-- 5-point centered average (2 before, current, 2 after)
SELECT
    date,
    value,
    AVG(value) OVER(
        ORDER BY date
        ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
    ) AS centered_avg_5
FROM data;
```

Q7: What happens at partition boundaries?

Answer: Frames are automatically bounded by partition edges:

```
-- For first row in partition:  
-- "2 PRECEDING" only includes available rows (may be 0 or 1)  
  
SELECT  
    customer_id,  
    order_num,  
    SUM(amount) OVER(  
        PARTITION BY customer_id  
        ORDER BY order_num  
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
    )  
FROM orders;  
  
-- First order: only 1 row (current)  
-- Second order: 2 rows  
-- Third+ orders: 3 rows
```

Q8: How do you calculate running total that resets monthly?

Answer:

```
SELECT  
    order_date,  
    amount,  
    SUM(amount) OVER(  
        PARTITION BY YEAR(order_date), MONTH(order_date)  
        ORDER BY order_date  
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
    ) AS monthly_running_total  
FROM orders;
```

Q9: Can you use frames with ranking functions?

Answer: No, ranking functions don't support frame clauses:

```
-- ✗ Error: RANK doesn't accept frame
RANK() OVER(ORDER BY sales ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

-- ✓ Correct: No frame needed
RANK() OVER(ORDER BY sales)
```

Ranking functions always operate on entire partition.

Q10: How do you compare current value to 3-period moving average?

Answer:

```
SELECT
    date,
    value,
    AVG(value) OVER(
        ORDER BY date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg,
    value - AVG(value) OVER(
        ORDER BY date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS diff_from_avg
FROM data;
```

🔑 Key Takeaways

1. **Frame Clause** defines subset of rows for calculation
2. **UNBOUNDED PRECEDING** = first row of partition
3. **CURRENT ROW** = the row being processed
4. **UNBOUNDED FOLLOWING** = last row of partition
5. **n PRECEDING/FOLLOWING** = specific number of rows
6. **Default frame** with ORDER BY is UNBOUNDED PRECEDING to CURRENT ROW
7. **Use ROWS over RANGE** for predictable results



Best Practices

Practice	Reason
Be explicit with frames	Avoid relying on defaults
Use ROWS not RANGE	More predictable behavior
Test edge cases	First/last rows may have fewer rows
Choose appropriate window size	Business logic dependent
Document frame logic	Helps future maintenance



Quick Reference

Frame Specification	Description
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	Running/cumulative
ROWS BETWEEN n PRECEDING AND CURRENT ROW	Moving/rolling (n+1 rows)
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Entire partition
ROWS BETWEEN CURRENT ROW AND n FOLLOWING	Current + next n rows
ROWS BETWEEN n PRECEDING AND n FOLLOWING	Centered window



Exercises

Exercise 1: Monthly Running Total

```
SELECT
    month,
    sales,
    SUM(sales) OVER(
        ORDER BY month
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS ytd_sales
FROM monthly_data;
```

Exercise 2: 3-Month Moving Average per Product

```
SELECT
    product_id,
    month,
    sales,
    AVG(sales) OVER(
        PARTITION BY product_id
        ORDER BY month
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_3m
FROM product_sales;
```

Exercise 3: Compare to Rolling Min/Max

```
SELECT
    date,
    price,
    MIN(price) OVER(ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS week_min,
    MAX(price) OVER(ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS week_max,
    price - MIN(price) OVER(ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
FROM prices;
```

Note 19: Ranking Functions (ROW_NUMBER, RANK, DENSE_RANK)

Overview

Ranking functions assign a position/rank to each row within a window based on specified ordering. Each function handles ties differently.

Theory

What are Ranking Functions?

Ranking Functions assign sequential integers to rows based on ORDER BY values:

RANKING FUNCTIONS Comparison

Data (ordered by score DESC):

Alice=95, Bob=90, Carol=90, Dave=85, Eve=80

ROW_NUMBER() (unique)	RANK() (gaps)	DENSE_RANK() (no gaps)
Alice: 1	Alice: 1	Alice: 1
Bob: 2	Bob: 2	Bob: 2
Carol: 3	Carol: 2	Carol: 2
Dave: 4	Dave: 4	Dave: 3
Eve: 5	Eve: 5	Eve: 4

← Ties get same rank
← Gap vs no gap

Key differences:

- ROW_NUMBER: Bob & Carol get different numbers (arbitrary)
- RANK: Bob & Carol both get 2, Dave gets 4 (skips 3)
- DENSE_RANK: Bob & Carol both get 2, Dave gets 3 (no skip)

Key Characteristics:

- **ORDER BY is required** - must sort to determine rank
- **Empty parentheses** - no expression inside (except NTILE)
- **Frame clause not allowed** - applies to entire partition
- **PARTITION BY optional** - resets ranking per group



Comparison of Three Main Ranking Functions

Function	Ties	Gaps	Use Case
ROW_NUMBER()	No (unique)	N/A	Unique sequential number
RANK()	Yes (same)	Yes	Traditional ranking with gaps
DENSE_RANK()	Yes (same)	No	Ranking without gaps



ROW_NUMBER()

Purpose:

Assigns a **unique sequential integer** to each row. Ties get different numbers.

Syntax:

```
ROW_NUMBER() OVER(ORDER BY column [DESC])
```

Example:

```
SELECT
    name,
    score,
    ROW_NUMBER() OVER(ORDER BY score DESC) AS row_num
FROM students;
```

Result:

name	score	row_num
Alice	95	1

name	score	row_num
Bob	90	2
Carol	90	3
Dave	85	4

Note: Bob and Carol have same score but different row numbers.

Key Points:

- Always produces **unique numbers** (1, 2, 3, 4...)
- Tie-breaking is arbitrary** unless you add more ORDER BY columns
- Great for **generating IDs** or **removing duplicates**



RANK()

Purpose:

Assigns same rank to ties, then **skips numbers** (creates gaps).

Syntax:

```
RANK() OVER(ORDER BY column [DESC])
```

Example:

```
SELECT
    name,
    score,
    RANK() OVER(ORDER BY score DESC) AS rank
FROM students;
```

Result:

name	score	rank
Alice	95	1
Bob	90	2
Carol	90	2
Dave	85	4

Note: Bob and Carol both get rank 2, then Dave gets rank 4 (skips 3).

Key Points:

- Ties share the same rank
- Gaps appear after ties
- Traditional competition ranking (1st, 2nd, 2nd, 4th...)



DENSE_RANK()

Purpose:

Assigns same rank to ties but **no gaps** in sequence.

Syntax:

```
DENSE_RANK() OVER(ORDER BY column [DESC])
```

Example:

```
SELECT
    name,
    score,
    DENSE_RANK() OVER(ORDER BY score DESC) AS dense_rank
FROM students;
```

Result:

name	score	dense_rank
Alice	95	1
Bob	90	2
Carol	90	2
Dave	85	3

Note: Bob and Carol both get rank 2, Dave gets rank 3 (no gap).

Key Points:

- **Ties share the same rank**
- **No gaps** - continuous sequence
- Useful when you need **count of distinct ranks**

 **Side-by-Side Comparison**

```
SELECT
    name,
    score,
    ROW_NUMBER() OVER(ORDER BY score DESC) AS row_num,
    RANK() OVER(ORDER BY score DESC) AS rank,
    DENSE_RANK() OVER(ORDER BY score DESC) AS dense_rank
FROM students;
```

Result:

name	score	row_num	rank	dense_rank
Alice	95	1	1	1
Bob	90	2	2	2

name	score	row_num	rank	dense_rank
Carol	90	3	2	2
Dave	85	4	4	3
Eve	80	5	5	4



With PARTITION BY

Ranking restarts for each partition:

```
SELECT
    department,
    name,
    salary,
    RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS dept_rank
FROM employees;
```

Result:

department	name	salary	dept_rank
Sales	John	80000	1
Sales	Jane	75000	2
IT	Bob	90000	1
IT	Alice	85000	2

🎯 Practical Use Cases

Use Case 1: Top N Analysis

```
-- Top 3 products by sales
SELECT * FROM (
    SELECT
        product_name,
        total_sales,
        ROW_NUMBER() OVER(ORDER BY total_sales DESC) AS rn
    FROM products
) ranked
WHERE rn <= 3;
```

Use Case 2: Top N per Category

```
-- Top 2 products in each category
SELECT * FROM (
    SELECT
        category,
        product_name,
        sales,
        ROW_NUMBER() OVER(PARTITION BY category ORDER BY sales DESC) AS rn
    FROM products
) ranked
WHERE rn <= 2;
```

Use Case 3: Remove Duplicates

```
-- Keep only first occurrence of each customer
SELECT * FROM (
    SELECT
        *,
        ROW_NUMBER() OVER(PARTITION BY customer_email ORDER BY created_date) AS rn
    FROM customers
) ranked
WHERE rn = 1;
```

Use Case 4: Bottom N Analysis

```
-- Bottom 5 performers
SELECT * FROM (
    SELECT
        employee_name,
        performance_score,
        RANK() OVER(ORDER BY performance_score ASC) AS rank
    FROM employees
) ranked
WHERE rank <= 5;
```

⚠ Choosing the Right Function

Scenario	Best Function
Need unique numbers	ROW_NUMBER()
Traditional ranking (like sports)	RANK()
Dense ranking without gaps	DENSE_RANK()
Removing duplicates	ROW_NUMBER()
Top N including all ties	RANK() or DENSE_RANK()

Scenario	Best Function
Generating sequential IDs	ROW_NUMBER()

❓ Practice Questions

Q1: Rank employees by salary within department.

```
SELECT
    department_id,
    employee_name,
    salary,
    RANK() OVER(PARTITION BY department_id ORDER BY salary DESC) AS salary_rank
FROM employees;
```

Q2: Find top 3 customers by total orders.

```
SELECT * FROM (
    SELECT
        customer_id,
        COUNT(*) AS order_count,
        ROW_NUMBER() OVER(ORDER BY COUNT(*) DESC) AS rn
    FROM orders
    GROUP BY customer_id
) ranked
WHERE rn <= 3;
```

Q3: Identify duplicate orders and keep first.

```

SELECT * FROM (
    SELECT
        *,
        ROW_NUMBER() OVER(PARTITION BY order_id ORDER BY created_at) AS rn
    FROM orders
) deduped
WHERE rn = 1;

```

Q4: Show all ranking types for products.

```

SELECT
    product_name,
    sales,
    ROW_NUMBER() OVER(ORDER BY sales DESC) AS row_num,
    RANK() OVER(ORDER BY sales DESC) AS rank,
    DENSE_RANK() OVER(ORDER BY sales DESC) AS dense_rank
FROM products;

```

Interview Questions

Q1: When would you use ROW_NUMBER vs RANK vs DENSE_RANK?

Answer:

Use Case	Best Function
Unique ID/sequence	ROW_NUMBER()
Traditional competition ranking	RANK()
Finding distinct rank levels	DENSE_RANK()
Removing duplicates	ROW_NUMBER()

Use Case	Best Function
Top N including all ties	RANK() or DENSE_RANK()
Pagination	ROW_NUMBER()

Q2: How do you get Top 3 performers INCLUDING all ties?

Answer: Use RANK() or DENSE_RANK(), not ROW_NUMBER():

```
-- Top 3 including ties
SELECT * FROM (
    SELECT *, DENSE_RANK() OVER(ORDER BY score DESC) AS rnk
    FROM employees
) t
WHERE rnk <= 3;

-- If scores are: 100, 95, 95, 90, 85
-- Returns: 100(1), 95(2), 95(2), 90(3) = 4 rows

-- ROW_NUMBER would return exactly 3 rows (arbitrary tie-break)
```

Q3: How do you find duplicate rows using ranking?

Answer:

```
-- Find duplicates
SELECT * FROM (
    SELECT
        *,
        ROW_NUMBER() OVER(
            PARTITION BY email -- Columns that define duplicates
            ORDER BY created_at -- Keep which one?
        ) AS rn
    FROM customers
) t
WHERE rn > 1; -- Returns all duplicates (except first)

-- To keep only first and delete rest:
DELETE FROM customers WHERE id IN (
    SELECT id FROM (
        SELECT id, ROW_NUMBER() OVER(PARTITION BY email ORDER BY id) AS rn
        FROM customers
    ) t WHERE rn > 1
);

```

Q4: Why should you add a secondary ORDER BY with ROW_NUMBER?

Answer: Without it, tie-breaking is **non-deterministic**:

```
-- ❌ Non-deterministic: May return different results each run
ROW_NUMBER() OVER(ORDER BY score DESC)
-- If two people have same score, who gets 1 vs 2?

-- ✅ Deterministic: Same result every time
ROW_NUMBER() OVER(ORDER BY score DESC, id ASC)
-- Ties broken by id
```

Q5: How do you find the Nth highest salary?

Answer:

```
-- Find 3rd highest salary
SELECT salary FROM (
    SELECT DISTINCT salary, DENSE_RANK() OVER(ORDER BY salary DESC) AS rnk
    FROM employees
) t
WHERE rnk = 3;

-- Using DENSE_RANK ensures:
-- If salaries are: 100, 100, 90, 80
-- DENSE_RANK: 1, 1, 2, 3 → 3rd highest = 80
-- RANK: 1, 1, 3, 4 → There is no rank 2!
```

Q6: Can you filter by rank in the WHERE clause directly?

Answer: No! Window functions execute during SELECT, after WHERE:

```
-- ✗ Error: Cannot use window function in WHERE
SELECT * FROM employees
WHERE ROW_NUMBER() OVER(ORDER BY salary DESC) <= 5;

-- ✓ Correct: Use subquery or CTE
SELECT * FROM (
    SELECT *, ROW_NUMBER() OVER(ORDER BY salary DESC) AS rn
    FROM employees
) t
WHERE rn <= 5;

-- ✓ Alternative: CTE
WITH ranked AS (
    SELECT *, ROW_NUMBER() OVER(ORDER BY salary DESC) AS rn
    FROM employees
)
SELECT * FROM ranked WHERE rn <= 5;
```

Q7: How do you get Top N per group?

Answer: Use PARTITION BY:

```
-- Top 2 employees per department by salary
SELECT * FROM (
    SELECT
        department,
        employee_name,
        salary,
        ROW_NUMBER() OVER(
            PARTITION BY department
            ORDER BY salary DESC
        ) AS dept_rank
    FROM employees
) t
WHERE dept_rank <= 2;
```

Q8: What happens to NULLs in ranking ORDER BY?

Answer: NULLs are typically sorted first (ASC) or last (DESC):

```
-- SQL Server: NULLs first in ASC, last in DESC
SELECT name, score,
    RANK() OVER(ORDER BY score ASC) AS rank_asc
FROM students;
-- NULL scores get lowest rank numbers

-- To control NULL position:
ORDER BY CASE WHEN score IS NULL THEN 1 ELSE 0 END, score ASC
```

Q9: How do you create pagination with ROW_NUMBER?

Answer:

```
-- Page 3, 10 items per page (rows 21-30)
SELECT * FROM (
    SELECT
        *,
        ROW_NUMBER() OVER(ORDER BY id) AS row_num
    FROM products
) t
WHERE row_num BETWEEN 21 AND 30;

-- Or using OFFSET-FETCH (simpler):
SELECT * FROM products
ORDER BY id
OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY;
```

Q10: What is the performance impact of ranking functions?

Answer:

Factor	Impact
Sorting	ORDER BY requires sort operation
Partition count	More partitions = more sorting passes
Index	Index on ORDER BY columns helps significantly
Large datasets	May spill to tempdb

Optimization tips:

- Create index on (PARTITION BY columns, ORDER BY columns)
- Filter data with WHERE before ranking
- Use TOP/LIMIT if you only need top N results



Key Takeaways

1. **ROW_NUMBER()** - Always unique, arbitrary tie-breaking
 2. **RANK()** - Ties same rank, creates gaps
 3. **DENSE_RANK()** - Ties same rank, no gaps
 4. **ORDER BY required** for all ranking functions
 5. **PARTITION BY** resets ranking for each group
 6. **Use subqueries** to filter by rank (WHERE rn <= N)
-



Best Practices

Practice	Reason
Add secondary ORDER BY for ROW_NUMBER	Deterministic tie-breaking
Use subquery to filter ranks	Can't use in WHERE directly
Choose function based on ties	Different behavior for ties
Consider performance	Rankings can be expensive
Test with sample data	Verify expected behavior



Syntax Quick Reference

```
-- ROW_NUMBER  
ROW_NUMBER() OVER(ORDER BY column)  
  
-- RANK  
RANK() OVER(ORDER BY column)  
  
-- DENSE_RANK  
DENSE_RANK() OVER(ORDER BY column)  
  
-- With partition  
RANK() OVER(PARTITION BY category ORDER BY value DESC)  
  
-- Filter top N  
SELECT * FROM (  
    SELECT *, ROW_NUMBER() OVER(ORDER BY col) AS rn  
    FROM table  
) t WHERE rn <= N
```



Exercises

Exercise 1: Department Rankings

Rank employees by salary in each department.

```
SELECT  
    department,  
    name,  
    salary,  
    DENSE_RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS dept_rank  
FROM employees;
```

Exercise 2: Top Products

Find top 5 products by revenue.

```
SELECT * FROM (
    SELECT
        product_name,
        revenue,
        ROW_NUMBER() OVER(ORDER BY revenue DESC) AS rn
    FROM products
) ranked
WHERE rn <= 5;
```

Exercise 3: Deduplication

Keep most recent order per customer.

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

Note 20: NTILE Function

Overview

NTILE divides rows into a specified number of equal groups (buckets/tiles) and assigns a bucket number to each row.



Theory

What is NTILE?

NTILE(n) distributes rows of an ordered partition into **n approximately equal groups** (buckets/tiles) and assigns a group number (1 to n) to each row.

NTILE(4) DISTRIBUTION			
Ordered Data: [100, 95, 90, 85, 80, 75, 70, 65] (8 employees)			
Bucket 1 (Top)	Bucket 2 (High)	Bucket 3 (Low)	Bucket 4 (Bottom)
100 → 1	90 → 2	80 → 3	70 → 4
95 → 1	85 → 2	75 → 3	65 → 4

Formula: 8 rows ÷ 4 buckets = 2 rows per bucket (even!)

Distribution Algorithm (Key Interview Concept!)

Rule: When rows don't divide evenly, **earlier buckets get the extra rows**.

UNEVEN DISTRIBUTION FORMULA

Total Rows \div N buckets = Base size + Remainder

Remainder rows go to FIRST buckets

10 rows \div 3 buckets = 3 base + 1 remainder

Bucket 1	Bucket 2	Bucket 3	
4 rows	3 rows	3 rows	\leftarrow First bucket gets extra

10 rows \div 4 buckets = 2 base + 2 remainder

Bucket 1	Bucket 2	Bucket 3	Bucket 4	
3 rows	3 rows	2 rows	2 rows	\leftarrow First 2 get extra

7 rows \div 3 buckets = 2 base + 1 remainder

Bucket 1	Bucket 2	Bucket 3
3 rows	2 rows	2 rows

NTILE vs Percentile vs PERCENT_RANK

Function	What It Returns	Value Range
NTILE(4)	Bucket number	1, 2, 3, 4 (integers)
PERCENT_RANK()	Relative position	0.0 to 1.0 (decimal)
NTILE(100)	Approximate percentile	1 to 100 (integers)

Edge Case: More Buckets Than Rows

5 rows with NTILE(10):

```
| Row 1 → Bucket 1  
| Row 2 → Bucket 2  
| Row 3 → Bucket 3  
| Row 4 → Bucket 4  
| Row 5 → Bucket 5  
  
| Buckets 6-10 will NEVER be assigned (no rows for them!)
```



Syntax

```
NTILE(number_of_groups) OVER(  
    [PARTITION BY column]  
    ORDER BY column  
)
```

Parameters:

Parameter	Description	Required
number_of_groups	How many buckets to create	Yes
PARTITION BY	Divide data before grouping	No
ORDER BY	Determine row order for grouping	Yes



Example

Basic NTILE:

```
SELECT  
    customer_id,  
    total_sales,  
    NTILE(4) OVER(ORDER BY total_sales DESC) AS quartile  
FROM customers;
```

Result:

customer_id	total_sales	quartile
101	5000	1
102	4500	1
103	4000	2
104	3500	2
105	3000	3
106	2500	3
107	2000	4
108	1500	4



Distribution Logic

Even Distribution:

$12 \text{ rows} \div 3 \text{ groups} = 4 \text{ rows each}$

Group 1: Rows 1-4 (4 rows)
Group 2: Rows 5-8 (4 rows)
Group 3: Rows 9-12 (4 rows)

Uneven Distribution:

$$10 \text{ rows} \div 3 \text{ groups} = 4, 3, 3$$

Group 1: Rows 1-4 (4 rows) ← gets extra
Group 2: Rows 5-7 (3 rows)
Group 3: Rows 8-10 (3 rows)

Another Example:

$$7 \text{ rows} \div 3 \text{ groups} = 3, 2, 2$$

Group 1: Rows 1-3 (3 rows) ← gets extra
Group 2: Rows 4-5 (2 rows)
Group 3: Rows 6-7 (2 rows)

Common Use Cases

Use Case 1: Data Segmentation (Quartiles)

```
-- Divide customers into 4 segments by spending
SELECT
    customer_name,
    total_spending,
    NTILE(4) OVER(ORDER BY total_spending DESC) AS spending_quartile,
    CASE NTILE(4) OVER(ORDER BY total_spending DESC)
        WHEN 1 THEN 'High Spender'
        WHEN 2 THEN 'Medium-High'
        WHEN 3 THEN 'Medium-Low'
        WHEN 4 THEN 'Low Spender'
    END AS segment
FROM customers;
```

Use Case 2: Percentile Groups

```
-- Divide into deciles (10 groups)
SELECT
    product_name,
    revenue,
    NTILE(10) OVER(ORDER BY revenue DESC) AS decile
FROM products;
```

Use Case 3: Load Balancing for ETL

```
-- Split data into 4 batches for parallel processing
SELECT
    *,
    NTILE(4) OVER(ORDER BY order_id) AS batch_number
FROM orders;

-- Then export each batch separately:
-- WHERE batch_number = 1 for first batch
-- WHERE batch_number = 2 for second batch, etc.
```

Use Case 4: Employee Performance Tiers

-- Divide employees into 3 performance tiers

SELECT

```
employee_name,
performance_score,
NTILE(3) OVER(ORDER BY performance_score DESC) AS tier,
CASE NTILE(3) OVER(ORDER BY performance_score DESC)
    WHEN 1 THEN 'Top Performer'
    WHEN 2 THEN 'Average'
    WHEN 3 THEN 'Needs Improvement'
END AS tier_label
FROM employees;
```



With PARTITION BY

-- Quartiles within each region

SELECT

```
region,
salesperson,
sales,
NTILE(4) OVER(PARTITION BY region ORDER BY sales DESC) AS regional_quartile
FROM sales_team;
```

Result:

region	salesperson	sales	regional_quartile
East	John	50000	1
East	Jane	45000	2
East	Bob	40000	3
East	Alice	35000	4
West	Mary	60000	1

region	salesperson	sales	regional_quartile
West	Tom	55000	2
West	Sue	50000	3
West	Jim	45000	4

⚠️ Important Notes

1. NTILE vs Other Rankings:

Function	Purpose
ROW_NUMBER	Sequential unique number
RANK	Position with gaps
DENSE_RANK	Position without gaps
NTILE	Divide into equal groups

2. Number Must Be Positive:

```
NTILE(0) -- Error!
NTILE(-1) -- Error!
NTILE(4) -- Valid ✓
```

3. More Groups Than Rows:

```
-- 5 rows with NTILE(10)
-- Only groups 1-5 will be assigned (one row each)
-- Groups 6-10 won't exist
```

❓ Practice Questions

Q1: Divide products into 3 price tiers.

```
SELECT
    product_name,
    price,
    NTILE(3) OVER(ORDER BY price DESC) AS price_tier,
    CASE NTILE(3) OVER(ORDER BY price DESC)
        WHEN 1 THEN 'Premium'
        WHEN 2 THEN 'Standard'
        WHEN 3 THEN 'Budget'
    END AS tier_name
FROM products;
```

Q2: Create 5 equal batches for data export.

```
SELECT
    *,
    NTILE(5) OVER(ORDER BY id) AS batch
FROM large_table;
```

Q3: Quartiles within each category.

```
SELECT
    category,
    product_name,
    sales,
    NTILE(4) OVER(PARTITION BY category ORDER BY sales DESC) AS quartile
FROM products;
```



Interview Questions

Q1: If you have 10 rows and NTILE(3), how many rows in each bucket?

Answer: 4, 3, 3 - Earlier buckets get the extra rows.

```
10 ÷ 3 = 3 remainder 1
Base: 3 rows per bucket
Remainder: 1 extra row goes to first bucket

Bucket 1: 4 rows (3 + 1 extra)
Bucket 2: 3 rows
Bucket 3: 3 rows
```

Formula: If remainder is R, first R buckets get (base + 1) rows.

Q2: What happens with NTILE(10) on only 5 rows?

Answer: Only buckets 1-5 are assigned. **Buckets 6-10 are never created.**

```
SELECT name, NTILE(10) OVER(ORDER BY id) AS bucket
FROM five_row_table;

-- Result: Each row gets a unique bucket 1, 2, 3, 4, 5
-- No bucket 6, 7, 8, 9, or 10 exists
```

This is critical for ETL when expecting specific bucket counts!

Q3: How is NTILE different from PERCENT_RANK?

Answer:

Aspect	NTILE(4)	PERCENT_RANK()
Returns	Integer (1-4)	Decimal (0.0-1.0)
Equal groups	Tries to make equal buckets	Relative position

Aspect	NTILE(4)	PERCENT_RANK()
Ties	Put in same bucket sequentially	Get same rank
Use case	Segmentation	Actual percentile

```
-- 4 rows with scores: 100, 90, 80, 70
SELECT score,
       NTILE(4) OVER(ORDER BY score DESC) AS tile,
       PERCENT_RANK() OVER(ORDER BY score DESC) AS pct_rank
FROM students;

-- Result:
-- 100: tile=1, pct_rank=0.00 (top)
-- 90: tile=2, pct_rank=0.33
-- 80: tile=3, pct_rank=0.67
-- 70: tile=4, pct_rank=1.00 (bottom)
```

Q4: How do you identify outliers using NTILE?

Answer:

```
-- Find bottom 10% performers (outliers)
SELECT * FROM (
    SELECT
        employee_name,
        performance_score,
        NTILE(10) OVER(ORDER BY performance_score) AS decile
    FROM employees
) t
WHERE decile = 1; -- Bottom 10%

-- Top 10%
WHERE decile = 10; -- Top 10%
```

Q5: Why doesn't NTILE work well with very small datasets?

Answer: NTILE approximates, it doesn't calculate true percentiles:

```
-- 4 employees with NTILE(4):  
-- Each gets a different bucket (1, 2, 3, 4)  
-- Even if 3 have same score!  
  
-- True quartile analysis needs more data:  
-- Rule of thumb: Need at least 4x your NTILE value  
-- NTILE(4) → need at least 16 rows for meaningful results
```

Q6: How do you use NTILE for batch processing?

Answer:

```
-- Split 1 million rows into 4 parallel batches  
SELECT  
    *,  
    NTILE(4) OVER(ORDER BY id) AS batch_id  
INTO #batched_data  
FROM large_table;  
  
-- Process each batch in parallel:  
-- Job 1: WHERE batch_id = 1  
-- Job 2: WHERE batch_id = 2  
-- Job 3: WHERE batch_id = 3  
-- Job 4: WHERE batch_id = 4  
  
-- Benefits:  
-- 1. Equal distribution of work  
-- 2. Deterministic (same rows in same batch)  
-- 3. No overlap between batches
```

Q7: Can NTILE create true percentiles?

Answer: No! NTILE(100) ≠ percentiles

```
-- NTILE(100) divides into 100 equal groups
-- But it doesn't calculate actual percentile values

-- Problem: If you have 50 rows, NTILE(100) gives 1-50 only
-- Problem: Ties are split across buckets

-- For true percentiles, use PERCENT_RANK() or PERCENTILE_CONT()
```

SELECT

```
score,
NTILE(100) OVER(ORDER BY score) AS ntile_100, -- Bucket number
PERCENT_RANK() OVER(ORDER BY score) * 100 AS true_percentile
FROM students;
```

Q8: What's the difference between NTILE(4) and dividing PERCENT_RANK by 0.25?

Answer:

```
-- NTILE(4): Forces exactly 4 buckets with ~equal row counts
-- PERCENT_RANK ranges: May have unequal row counts per bucket
```

```
SELECT score,
NTILE(4) OVER(ORDER BY score) AS ntile_bucket,
CASE
  WHEN PERCENT_RANK() OVER(ORDER BY score) < 0.25 THEN 1
  WHEN PERCENT_RANK() OVER(ORDER BY score) < 0.50 THEN 2
  WHEN PERCENT_RANK() OVER(ORDER BY score) < 0.75 THEN 3
  ELSE 4
END AS percentile_bucket
FROM students;
```

```
-- With 12 rows:
-- NTILE(4): 3, 3, 3, 3 rows per bucket
-- PERCENT_RANK: May not be equal if ties exist!
```

Q9: How do you combine NTILE with CASE for labeling?

Answer:

```
-- Customer segmentation with labels

SELECT
    customer_name,
    total_spending,
    NTILE(4) OVER(ORDER BY total_spending DESC) AS quartile,
    CASE NTILE(4) OVER(ORDER BY total_spending DESC)
        WHEN 1 THEN 'Premium'
        WHEN 2 THEN 'Gold'
        WHEN 3 THEN 'Silver'
        WHEN 4 THEN 'Bronze'
    END AS tier
FROM customers;

-- Note: Window function calculated twice (optimization: use subquery)
SELECT *, 
    CASE quartile
        WHEN 1 THEN 'Premium'
        WHEN 2 THEN 'Gold'
        WHEN 3 THEN 'Silver'
        WHEN 4 THEN 'Bronze'
    END AS tier
FROM (
    SELECT customer_name, total_spending,
        NTILE(4) OVER(ORDER BY total_spending DESC) AS quartile
    FROM customers
) t;
```

Q10: Does NTILE guarantee equal bucket sizes?

Answer: No! NTILE only guarantees **approximately** equal sizes.

Guarantees:

- Largest bucket - smallest bucket ≤ 1 row difference
- Earlier buckets get extra rows (if uneven)
- Each row assigned to exactly one bucket

Does NOT guarantee:

- Exactly equal bucket sizes (impossible if rows % n $\neq 0$)
- Buckets based on value ranges (only position-based)
- All bucket numbers exist (if fewer rows than buckets)

🔑 Key Takeaways

1. **NTILE(n)** creates n equal groups
2. **Uneven rows** go to earlier groups
3. **ORDER BY required** to determine grouping
4. Great for **segmentation** and **percentiles**
5. Useful for **load balancing** in ETL
6. **Cannot use frame clause** with NTILE

📌 Best Practices

Practice	Reason
Use meaningful n values	4 (quartiles), 5 (quintiles), 10 (deciles)
Combine with CASE	Create readable labels
Use for batch processing	Equal distribution for parallelism
Consider edge cases	Very few rows, more groups than rows
Partition when needed	Segment within categories



Exercises

Exercise 1: Customer Segmentation

Divide customers into 4 groups by total purchases.

```
SELECT
    customer_id,
    total_purchases,
    NTILE(4) OVER(ORDER BY total_purchases DESC) AS customer_segment
FROM customers;
```

Exercise 2: Salary Percentiles

Create deciles for employee salaries.

```
SELECT
    employee_name,
    department,
    salary,
    NTILE(10) OVER(ORDER BY salary) AS salary_percentile
FROM employees;
```

Exercise 3: Batch Export with Labels

Create labeled batches for data migration.

```
SELECT
    *,
    NTILE(4) OVER(ORDER BY created_date) AS batch_num,
    'Batch_' + CAST(NTILE(4) OVER(ORDER BY created_date) AS VARCHAR) AS batch_name
FROM records;
```

Note 21: PERCENT_RANK and

CUME_DIST

Overview

These percentage-based ranking functions calculate relative positions as decimals between 0 and 1, useful for distribution analysis.

Theory

What are Percentage-Based Rankings?

Unlike integer rankings (1, 2, 3...), these functions return **decimal values (0 to 1)** showing where each row stands relative to the entire dataset.

PERCENT_RANK vs CUME_DIST VISUALIZATION

5 Students: Alice(100), Bob(80), Carol(80), Dave(50), Eve(30)

Score Scale: 30 ————— 100

PERCENT_RANK (How many below you / total - 1):

Eve	Dave	Bob/Carol	Alice
1.00	0.75	0.25	0.00
(last)			(first)

CUME_DIST (Your position / total):

Eve	Dave	Bob/Carol	Alice
1.00	0.80	0.60	0.20
(100%)	(80%)	(60%)	(20%)

The Key Formulas (Interview Must-Know!)

FORMULA COMPARISON

PERCENT_RANK = (Rank - 1) / (N - 1)

- First row ALWAYS = 0.00
- Last row ALWAYS = 1.00
- "What percentage of rows are BELOW me?"

CUME_DIST = Number of rows \leq current value / N

- First row = 1/N (never 0!)
- Last row ALWAYS = 1.00
- "What percentage of rows are AT or BELOW this value?"

Tie Handling Comparison

Data: Alice=100, Bob=80, Carol=80, Dave=50, Eve=30 (5 students)

PERCENT_RANK
(Uses first rank)

CUME_DIST
(Uses last position)

Alice (100): $(1-1)/(5-1) = 0.00$	$1/5 = 0.20$
Bob (80): $(2-1)/(5-1) = 0.25$	$3/5 = 0.60$ (last of 3)
Carol (80): $(2-1)/(5-1) = 0.25$ (tie!)	$3/5 = 0.60$ (same)
Dave (50): $(4-1)/(5-1) = 0.75$	$4/5 = 0.80$
Eve (30): $(5-1)/(5-1) = 1.00$	$5/5 = 1.00$

Function Summary

Function	Full Name	Range	First Row	Ties Use	Meaning
PERCENT_RANK()	Percent Rank	0 to 1	0	First position	% of rows below
CUME_DIST()	Cumulative Distribution	>0 to 1	1/N	Last position	% at or below



CUME_DIST() - Cumulative Distribution

What it Calculates:

The percentage of rows with values **less than or equal to** the current row.

Formula:

```
CUME_DIST = Position Number / Total Rows
```

For ties: Uses the **last position** of the tied value.

Syntax:

```
CUME_DIST() OVER(ORDER BY column)
```

Example:

```

SELECT
    name,
    score,
    CUME_DIST() OVER(ORDER BY score DESC) AS cume_dist
FROM students;

```

name	score	cume_dist
Alice	100	0.20
Bob	80	0.60
Carol	80	0.60
Dave	50	0.80
Eve	30	1.00

Step-by-Step Calculation:

Row 1 (Alice, 100): Position 1 → $1/5 = 0.20$

Row 2 (Bob, 80): Position 2, but tie → last position 3 → $3/5 = 0.60$

Row 3 (Carol, 80): Same value → same as Bob → $3/5 = 0.60$

Row 4 (Dave, 50): Position 4 → $4/5 = 0.80$

Row 5 (Eve, 30): Position 5 → $5/5 = 1.00$



PERCENT_RANK() - Percent Rank

What it Calculates:

The relative rank of a row, ranging from 0 (first) to 1 (last).

Formula:

$$\text{PERCENT_RANK} = (\text{Rank} - 1) / (\text{Total Rows} - 1)$$

For ties: Uses the **first position** of the tied value.

Syntax:

```
PERCENT_RANK() OVER(ORDER BY column)
```

Example:

```
SELECT
    name,
    score,
    PERCENT_RANK() OVER(ORDER BY score DESC) AS pct_rank
FROM students;
```

name	score	pct_rank
Alice	100	0.00
Bob	80	0.25
Carol	80	0.25
Dave	50	0.75
Eve	30	1.00

Step-by-Step Calculation:

Row 1 (Alice): Rank 1 $\rightarrow (1-1)/(5-1) = 0/4 = 0.00$
 Row 2 (Bob): Rank 2 (first occurrence) $\rightarrow (2-1)/(5-1) = 1/4 = 0.25$
 Row 3 (Carol): Same value, same rank 2 $\rightarrow (2-1)/(5-1) = 1/4 = 0.25$
 Row 4 (Dave): Rank 4 $\rightarrow (4-1)/(5-1) = 3/4 = 0.75$
 Row 5 (Eve): Rank 5 $\rightarrow (5-1)/(5-1) = 4/4 = 1.00$



Side-by-Side Comparison

```
SELECT
    name,
    score,
    CUME_DIST() OVER(ORDER BY score DESC) AS cume_dist,
    PERCENT_RANK() OVER(ORDER BY score DESC) AS pct_rank
FROM students;
```

name	score	cume_dist	pct_rank
Alice	100	0.20	0.00
Bob	80	0.60	0.25
Carol	80	0.60	0.25
Dave	50	0.80	0.75
Eve	30	1.00	1.00

Key Differences:

Aspect	CUME_DIST	PERCENT_RANK
First row	> 0	Always 0
Last row	Always 1	Always 1
Ties use	Last position	First position
Formula includes current row	Yes (inclusive)	No (exclusive)

🎯 Practical Use Cases

Use Case 1: Top Percentile Analysis

```
-- Find products in top 20% of sales
SELECT * FROM (
    SELECT
        product_name,
        sales,
        CUME_DIST() OVER(ORDER BY sales DESC) AS percentile
    FROM products
) ranked
WHERE percentile <= 0.20;
```

Use Case 2: Distribution Analysis

```
-- See where each employee falls in salary distribution
```

```
SELECT
```

```
    employee_name,  
    salary,  
    CAST(CUME_DIST() OVER(ORDER BY salary) * 100 AS INT) AS salary_percentile  
FROM employees;
```

Use Case 3: Relative Performance

```
-- How does each store rank relative to others?
```

```
SELECT
```

```
    store_name,  
    revenue,  
    FORMAT(PERCENT_RANK() OVER(ORDER BY revenue DESC), 'P0') AS relative_position  
FROM stores;
```

Use Case 4: Filter by Percentile Range

```
-- Find middle 50% of prices (25th to 75th percentile)
```

```
SELECT * FROM (
```

```
    SELECT
```

```
        product_name,  
        price,  
        PERCENT_RANK() OVER(ORDER BY price) AS pct_rank  
    FROM products  
) ranked
```

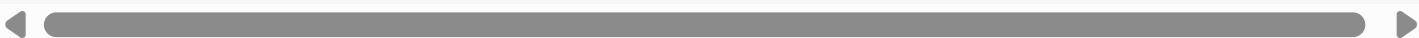
```
WHERE pct_rank BETWEEN 0.25 AND 0.75;
```



Formatting Output

Convert to Percentage:

```
SELECT
    name,
    score,
    CAST(CUME_DIST() OVER(ORDER BY score DESC) * 100 AS DECIMAL(5,1)) AS percentile
FROM students;
```



Add Percentage Symbol:

```
SELECT
    name,
    score,
    CONCAT(CAST(CUME_DIST() OVER(ORDER BY score DESC) * 100 AS INT), '%') AS percent
FROM students;
```



⚠️ When to Use Which?

Use CUME_DIST when...	Use PERCENT_RANK when...
Need distribution of data points	Need relative position
Inclusive calculation preferred	Exclusive calculation preferred
Working with percentiles	Working with relative rankings
Measuring "at or below this value"	Measuring "rank relative to others"

?

Practice Questions

Q1: Find products in top 40% by price.

```
SELECT * FROM (
    SELECT
        product_name,
        price,
        CUME_DIST() OVER(ORDER BY price DESC) AS cume_pct
    FROM products
) ranked
WHERE cume_pct <= 0.40;
```

Q2: Show employees with their salary percentile.

```
SELECT
    employee_name,
    salary,
    CONCAT(CAST(PERCENT_RANK() OVER(ORDER BY salary) * 100 AS INT), '%') AS salary_p
FROM employees;
```

Q3: Compare both functions for scores.

```
SELECT
    student_name,
    score,
    ROUND(CUME_DIST() OVER(ORDER BY score DESC) * 100, 1) AS cumulative_pct,
    ROUND(PERCENT_RANK() OVER(ORDER BY score DESC) * 100, 1) AS relative_rank_pct
FROM students;
```



Interview Questions

Q1: What is the difference between PERCENT_RANK and CUME_DIST?

Answer:

Aspect	PERCENT_RANK	CUME_DIST
Formula	$(\text{Rank}-1)/(\text{N}-1)$	Position/N
First row	Always 0	$1/\text{N}$ (never 0)
Meaning	"% of rows BELOW me"	"% of rows AT or BELOW"
Ties use	First position (RANK)	Last position
Statistical analog	Exclusive percentile	Inclusive percentile

```
-- 3 rows with values: A, B, C
SELECT val,
       PERCENT_RANK() OVER(ORDER BY val) AS pct_rank,    -- 0.00, 0.50, 1.00
       CUME_DIST() OVER(ORDER BY val) AS cume_dist      -- 0.33, 0.67, 1.00
FROM table3;
```

Q2: How do you find the top 10% of employees?

Answer: Use CUME_DIST (inclusive is more intuitive for "top N%"):

```
-- Find top 10% performers
SELECT * FROM (
    SELECT
        employee_name,
        performance_score,
        CUME_DIST() OVER(ORDER BY performance_score DESC) AS pct
    FROM employees
) ranked
WHERE pct <= 0.10;

-- Why CUME_DIST?
-- CUME_DIST = 0.10 means "this person and all above = 10%"
-- PERCENT_RANK = 0.10 means "10% are BELOW this person"
```

Q3: Calculate what percentile a specific score falls into?

Answer:

```
-- What percentile is a score of 85?
SELECT
    score,
    CAST(PERCENT_RANK() OVER(ORDER BY score) * 100 AS INT) AS percentile
FROM all_scores
WHERE score = 85;

-- If result is 0.75, this score beats 75% of all scores
```

Q4: Why does PERCENT_RANK start at 0 but CUME_DIST never equals 0?

Answer: Due to their formulas:

PERCENT_RANK: (Rank - 1) / (N - 1)

First row: $(1 - 1) / (N - 1) = 0 / \text{something} = 0$

CUME_DIST: Position / N

First row: $1 / N = \text{some positive number (e.g., } 1/5 = 0.20\text{)}$

The philosophical difference:

- PERCENT_RANK asks: "What % are BELOW me?" First has no one below $\rightarrow 0$
- CUME_DIST asks: "What % includes me and below?" First includes self $\rightarrow 1/N$

Q5: Find employees in the middle 50% (25th to 75th percentile)?

Answer:

```

SELECT * FROM (
    SELECT
        employee_name,
        salary,
        PERCENT_RANK() OVER(ORDER BY salary) AS pct_rank
    FROM employees
) ranked
WHERE pct_rank BETWEEN 0.25 AND 0.75;

-- Note: Using PERCENT_RANK because we want:
-- 25% below on one end, 25% above on other end

```

Q6: How do ties affect PERCENT_RANK and CUME_DIST differently?

Answer:

```
-- Scores: 100, 90, 90, 90, 80 (5 students, 3-way tie at 90)
SELECT score,
       RANK() OVER(ORDER BY score DESC) AS rank,
       PERCENT_RANK() OVER(ORDER BY score DESC) AS pct_rank,
       CUME_DIST() OVER(ORDER BY score DESC) AS cume_dist
FROM students;

-- Results:
-- 100: rank=1, pct_rank=0.00, cume_dist=0.20
-- 90: rank=2, pct_rank=0.25, cume_dist=0.80 (all 3 get same!)
-- 90: rank=2, pct_rank=0.25, cume_dist=0.80
-- 90: rank=2, pct_rank=0.25, cume_dist=0.80
-- 80: rank=5, pct_rank=1.00, cume_dist=1.00

-- CUME_DIST jumps to 0.80 (4/5) because it includes ALL ties
-- PERCENT_RANK stays at 0.25 using first rank position
```

Q7: What happens with only 1 row?

Answer:

```
-- Single row edge case
SELECT score,
       PERCENT_RANK() OVER(ORDER BY score) AS pct_rank,
       CUME_DIST() OVER(ORDER BY score) AS cume_dist
FROM single_row_table;

-- PERCENT_RANK: (1-1)/(1-1) = 0/0 → SQL returns 0
-- CUME_DIST: 1/1 = 1.00

-- Result: pct_rank = 0, cume_dist = 1.00
```

Q8: How do you find the median using these functions?

Answer:

```
-- Find value closest to 50th percentile
SELECT TOP 1 * FROM (
    SELECT
        value,
        ABS(PERCENT_RANK() OVER(ORDER BY value) - 0.5) AS distance_from_median
    FROM data_table
) t
ORDER BY distance_from_median;

-- Better approach: Use PERCENTILE_CONT for true median
SELECT DISTINCT
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY value) OVER() AS median
FROM data_table;
```

Q9: Convert PERCENT_RANK to a readable percentile label?

Answer:

```
SELECT
    student_name,
    score,
    CASE
        WHEN PERCENT_RANK() OVER(ORDER BY score) >= 0.90 THEN 'Top 10%'
        WHEN PERCENT_RANK() OVER(ORDER BY score) >= 0.75 THEN 'Top 25%'
        WHEN PERCENT_RANK() OVER(ORDER BY score) >= 0.50 THEN 'Top 50%'
        WHEN PERCENT_RANK() OVER(ORDER BY score) >= 0.25 THEN 'Below Average'
        ELSE 'Bottom 25%'
    END AS performance_tier
FROM students;
```

Q10: PERCENT_RANK vs NTILE(100) - which gives true percentiles?

Answer: PERCENT_RANK gives true percentiles, NTILE(100) does not:

```
-- NTILE(100): Just divides into 100 buckets
-- Problems:
-- 1. With 50 rows, only buckets 1-50 exist
-- 2. Doesn't return decimal percentages
-- 3. Ties are split across buckets

-- PERCENT_RANK: True relative percentile
-- 1. Works with any row count
-- 2. Returns 0.00 to 1.00
-- 3. Ties get same percentile

SELECT score,
    NTILE(100) OVER(ORDER BY score) AS ntile_100,
    PERCENT_RANK() OVER(ORDER BY score) AS true_percentile
FROM students;
```

🔑 Key Takeaways

1. **CUME_DIST** - Cumulative distribution (inclusive)
2. **PERCENT_RANK** - Relative position (exclusive)
3. Both return values **0 to 1** (multiply by 100 for percentage)
4. **CUME_DIST** first row > 0, **PERCENT_RANK** first row = 0
5. **Both handle ties** by sharing same percentage
6. Great for **percentile analysis** and **distribution studies**

📌 Best Practices

Practice	Reason
Multiply by 100 for readability	Easier to understand as percentages
Use subquery for filtering	Can't filter in same SELECT
Consider formula differences	Choose based on inclusive/exclusive need

Practice	Reason
Format output appropriately	Add % symbol for clarity
Test with ties	Verify expected behavior



Quick Reference

```
-- Cumulative Distribution
CUME_DIST() OVER(ORDER BY column)

-- Percent Rank
PERCENT_RANK() OVER(ORDER BY column)

-- With partition
CUME_DIST() OVER(PARTITION BY category ORDER BY value)

-- Format as percentage
CAST(CUME_DIST() OVER(ORDER BY col) * 100 AS INT)
```



Exercises

Exercise 1: Sales Percentile

Show each product's sales percentile.

```
SELECT
    product_name,
    sales,
    ROUND(CUME_DIST() OVER(ORDER BY sales) * 100, 0) AS sales_percentile
FROM products;
```

Exercise 2: Top Performers

Find employees in top 10% of performance scores.

```
SELECT * FROM (
    SELECT
        employee_name,
        performance_score,
        CUME_DIST() OVER(ORDER BY performance_score DESC) AS pct
    FROM employees
) ranked
WHERE pct <= 0.10;
```

Exercise 3: Distribution by Category

Show percentile within each category.

```
SELECT
    category,
    product_name,
    price,
    ROUND(PERCENT_RANK() OVER(PARTITION BY category ORDER BY price) * 100, 0) AS cat
FROM products;
```

Note 22: LAG and LEAD Functions

Overview

LAG and LEAD allow you to access values from **previous** or **next** rows without self-joins, enabling powerful time-series and sequential analysis.



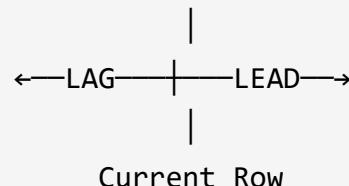
Theory

What are LAG and LEAD?

- **LAG()** - Access value from a **previous** row in the ordered result set
- **LEAD()** - Access value from a **next** row in the ordered result set

LAG and LEAD VISUALIZATION

Ordered Data:	Jan	Feb	Mar	Apr	May
	\$100	\$150	\$120	\$180	\$200



From Mar's perspective:

LAG(sales, 1) = \$150 (Feb - 1 row back)
LAG(sales, 2) = \$100 (Jan - 2 rows back)
LEAD(sales, 1) = \$180 (Apr - 1 row ahead)
LEAD(sales, 2) = \$200 (May - 2 rows ahead)

Why LAG/LEAD Instead of Self-Joins?

OLD WAY vs NEW WAY

OLD WAY (Self-Join) - Slow, complex:

```
SELECT a.month, a.sales, b.sales AS prev_sales  
FROM sales a  
LEFT JOIN sales b ON a.month = b.month + 1  
-- Requires join, handles months/dates awkwardly
```

NEW WAY (LAG) - Fast, simple:

```
SELECT month, sales, LAG(sales) OVER(ORDER BY month) AS prev_sales  
FROM sales  
-- Single table scan, cleaner syntax
```

Edge Case Handling

BOUNDARY BEHAVIOR

First Row: LAG returns NULL (no previous row exists)
 Last Row: LEAD returns NULL (no next row exists)

Row	Sales	LAG(sales)	LEAD(sales)
Jan	100	NULL ← First	150
Feb	150	100	120
Mar	120	150	180
Apr	180	120	NULL ← Last

Solution: Provide default value as 3rd parameter
`LAG(sales, 1, 0) → Returns 0 instead of NULL`

Parameters Summary

Parameter	Description	Required	Default	Example
expression	Column/value to retrieve	Yes	-	sales
offset	Rows back (LAG) or forward (LEAD)	No	1	2 (two rows away)
default	Value if no row exists	No	NULL	0, 'N/A'



Syntax

LAG - Previous Row Value:

```
LAG(column, offset, default) OVER(ORDER BY column)
```

LEAD - Next Row Value:

```
LEAD(column, offset, default) OVER(ORDER BY column)
```

Parameters:

Parameter	Description	Required	Default
column	Value to retrieve	Yes	-
offset	Number of rows back/forward	No	1
default	Value if no row exists	No	NULL



LAG Examples

Basic LAG (Previous Row):

```
SELECT
    month,
    sales,
    LAG(sales) OVER(ORDER BY month) AS prev_month_sales
FROM monthly_sales;
```

month	sales	prev_month_sales
Jan	100	NULL

month	sales	prev_month_sales
Feb	150	100
Mar	120	150
Apr	180	120

LAG with Offset:

```
-- Get sales from 2 months ago
LAG(sales, 2) OVER(ORDER BY month)
```

month	sales	2_months_ago
Jan	100	NULL
Feb	150	NULL
Mar	120	100
Apr	180	150

LAG with Default Value:

```
-- Replace NULL with 0
LAG(sales, 1, 0) OVER(ORDER BY month)
```

month	sales	prev_sales
Jan	100	0
Feb	150	100
Mar	120	150



LEAD Examples

Basic LEAD (Next Row):

```
SELECT
    month,
    sales,
    LEAD(sales) OVER(ORDER BY month) AS next_month_sales
FROM monthly_sales;
```

month	sales	next_month_sales
Jan	100	150
Feb	150	120
Mar	120	180
Apr	180	NULL

LEAD with Offset:

```
-- Get sales from 2 months ahead
LEAD(sales, 2) OVER(ORDER BY month)
```

LEAD with Default:

```
-- Replace NULL with 0
LEAD(sales, 1, 0) OVER(ORDER BY month)
```



Practical Use Cases

Use Case 1: Month-over-Month Change

```

SELECT
    month,
    sales,
    LAG(sales) OVER(ORDER BY month) AS prev_month,
    sales - LAG(sales) OVER(ORDER BY month) AS change,
    ROUND((sales - LAG(sales) OVER(ORDER BY month)) * 100.0 /
          LAG(sales) OVER(ORDER BY month), 2) AS pct_change
FROM monthly_sales;
  
```

Result:

month	sales	prev_month	change	pct_change
Jan	100	NULL	NULL	NULL
Feb	150	100	50	50.00
Mar	120	150	-30	-20.00
Apr	180	120	60	50.00

Use Case 2: Year-over-Year Comparison

```

SELECT
    year,
    month,
    sales,
    LAG(sales, 12) OVER(ORDER BY year, month) AS same_month_last_year,
    sales - LAG(sales, 12) OVER(ORDER BY year, month) AS yoy_change
FROM monthly_sales;
  
```

Use Case 3: Days Between Orders (Customer Retention)

```

SELECT
    customer_id,
    order_date,
    LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date) AS prev_order
    DATEDIFF(DAY,
        LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date),
        order_date) AS days_between_orders
FROM orders;

```



Use Case 4: Growth Direction

```

SELECT
    month,
    sales,
    CASE
        WHEN sales > LAG(sales) OVER(ORDER BY month) THEN '📈 Up'
        WHEN sales < LAG(sales) OVER(ORDER BY month) THEN '📉 Down'
        WHEN sales = LAG(sales) OVER(ORDER BY month) THEN '➡ Flat'
        ELSE 'N/A'
    END AS trend
FROM monthly_sales;

```

Use Case 5: Forecast Comparison

```

-- Compare actual to what we predicted (if forecast was made month ahead)
SELECT
    month,
    actual_sales,
    LEAD(forecast, 1) OVER(ORDER BY month) AS forecast_for_next,
    actual_sales - LAG(forecast) OVER(ORDER BY month) AS variance_from_forecast
FROM sales_data;

```



With PARTITION BY

Restart for each group:

```
SELECT
    department,
    month,
    revenue,
    LAG(revenue) OVER(PARTITION BY department ORDER BY month) AS prev_month_rev
FROM department_sales;
```

department	month	revenue	prev_month_rev
IT	Jan	100	NULL
IT	Feb	120	100
Sales	Jan	200	NULL
Sales	Feb	180	200

⚠️ Important Notes

1. ORDER BY is Required:

```
-- This will error:
LAG(sales) OVER() -- ✗ No ORDER BY

-- Correct:
LAG(sales) OVER(ORDER BY month) -- ✓
```

2. Frame Clause Not Allowed:

```
-- Cannot use with LAG/LEAD:
LAG(sales) OVER(ORDER BY month ROWS BETWEEN...) -- ✗
```

3. NULL Handling:

First row for LAG and last row for LEAD return NULL unless default specified.

4. Data Types:

Expression can be any data type - numbers, strings, dates, etc.

❓ Practice Questions

Q1: Show previous day's stock price.

```
SELECT
    date,
    price,
    LAG(price) OVER(ORDER BY date) AS yesterday_price,
    price - LAG(price) OVER(ORDER BY date) AS daily_change
FROM stock_prices;
```

Q2: Calculate average days between customer orders.

```
SELECT
    customer_id,
    AVG(days_between) AS avg_days_between_orders
FROM (
    SELECT
        customer_id,
        order_date,
        DATEDIFF(DAY,
            LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date),
            order_date) AS days_between
    FROM orders
) t
WHERE days_between IS NOT NULL
GROUP BY customer_id;
```

Q3: Month-over-month growth rate by product.

SELECT

```
product_id,
month,
sales,
LAG(sales) OVER(PARTITION BY product_id ORDER BY month) AS prev_sales,
ROUND((sales - LAG(sales) OVER(PARTITION BY product_id ORDER BY month)) * 100.0
      NULLIF(LAG(sales) OVER(PARTITION BY product_id ORDER BY month), 0), 2) AS
FROM product_sales;
```



🎯 Interview Questions

Q1: How would you calculate month-over-month growth percentage?

Answer:

SELECT

```
month,
revenue,
LAG(revenue) OVER(ORDER BY month) AS prev_month,
ROUND(
    (revenue - LAG(revenue) OVER(ORDER BY month)) * 100.0 /
    NULLIF(LAG(revenue) OVER(ORDER BY month), 0)
    , 2) AS mom_growth_pct
FROM monthly_revenue;
```

-- Key points:

- 1. NULLIF prevents divide by zero
- 2. Multiply by 100.0 (not 100) for decimal division
- 3. First month will have NULL growth

Q2: How do you handle the NULL in the first row from LAG?

Answer: Three options:

```
-- Option 1: Default value (3rd parameter)
LAG(sales, 1, 0) OVER(ORDER BY month) -- Returns 0 instead of NULL

-- Option 2: COALESCE
COALESCE(LAG(sales) OVER(ORDER BY month), 0)

-- Option 3: ISNULL (SQL Server)
ISNULL(LAG(sales) OVER(ORDER BY month), 0)

-- Option 4: Filter out NULL rows
SELECT * FROM (
    SELECT month, sales,
        LAG(sales) OVER(ORDER BY month) AS prev_sales
    FROM sales
) t
WHERE prev_sales IS NOT NULL;
```

Q3: Find customers whose orders are more than 30 days apart?

Answer:

```

SELECT DISTINCT customer_id
FROM (
    SELECT
        customer_id,
        order_date,
        LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date) AS prev_order_date,
        DATEDIFF(DAY,
            LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date),
            order_date) AS days_gap
    FROM orders
) t
WHERE days_gap > 30;

-- This finds customers with at-risk retention (long gaps)

```

Q4: Why can't you use a frame clause with LAG/LEAD?

Answer: LAG and LEAD are **value functions**, not aggregate functions:

```

-- ❌ This doesn't make sense and will error:
LAG(sales) OVER(ORDER BY month ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)

-- LAG/LEAD have a fixed definition:
-- LAG looks at a SPECIFIC row (offset rows back)
-- It doesn't aggregate over a range

-- For range-based operations, use FIRST_VALUE or LAST_VALUE
FIRST_VALUE(sales) OVER(ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)

```

Q5: How do you do year-over-year comparison?

Answer:

```
-- Method 1: Using LAG with offset 12 (monthly data)
SELECT
    year,
    month,
    sales,
    LAG(sales, 12) OVER(ORDER BY year, month) AS same_month_last_year,
    sales - LAG(sales, 12) OVER(ORDER BY year, month) AS yoy_difference
FROM monthly_sales;

-- Method 2: Using PARTITION BY month (more reliable)
SELECT
    year,
    month,
    sales,
    LAG(sales) OVER(PARTITION BY month ORDER BY year) AS last_year,
    sales - LAG(sales) OVER(PARTITION BY month ORDER BY year) AS yoy_diff
FROM monthly_sales;
```

Q6: Find the running difference between consecutive rows?

Answer:

```
SELECT
    date,
    value,
    value - LAG(value, 1, value) OVER(ORDER BY date) AS diff_from_prev
FROM time_series;

-- Using LAG(value, 1, value) means:
-- - Get previous value
-- - If NULL (first row), use current value
-- - So first row shows 0 difference
```

Q7: How would you detect if a value increased, decreased, or stayed the same?

Answer:

```

SELECT
    date,
    price,
    CASE
        WHEN price > LAG(price) OVER(ORDER BY date) THEN 'UP'
        WHEN price < LAG(price) OVER(ORDER BY date) THEN 'DOWN'
        WHEN price = LAG(price) OVER(ORDER BY date) THEN 'UNCHANGED'
        ELSE 'FIRST DAY'
    END AS price_movement
FROM stock_prices;

```

Q8: Compare LAG/LEAD with FIRST_VALUE/LAST_VALUE?

Answer:

Aspect	LAG/LEAD	FIRST_VALUE/LAST_VALUE
Access	Specific offset from current	First/last in window
Frame clause	Not allowed	Required for proper use
Use case	Row-to-row comparison	Get boundary values
Default value	Built-in parameter	Use IGNORE NULLS (if supported)

```

-- LAG: "What was the value 1 row before?"
LAG(sales) OVER(ORDER BY month)

-- FIRST_VALUE: "What was the first value in my window?"
FIRST_VALUE(sales) OVER(ORDER BY month ROWS UNBOUNDED PRECEDING)

```

Q9: How to get both previous AND next value in one query?

Answer:

```
SELECT
    date,
    price,
    LAG(price) OVER(ORDER BY date) AS prev_price,
    LEAD(price) OVER(ORDER BY date) AS next_price,
    -- Calculate if current is local maximum
    CASE
        WHEN price > LAG(price) OVER(ORDER BY date)
            AND price > LEAD(price) OVER(ORDER BY date)
        THEN 'Local Peak'
        WHEN price < LAG(price) OVER(ORDER BY date)
            AND price < LEAD(price) OVER(ORDER BY date)
        THEN 'Local Valley'
        ELSE 'Normal'
    END AS pattern
FROM stock_prices;
```

Q10: What's the performance impact of multiple LAG/LEAD calls?

Answer:

-- Each LAG/LEAD with same OVER clause shares the sort:

SELECT

```
month,  
LAG(sales, 1) OVER(ORDER BY month) AS prev_1,  
LAG(sales, 2) OVER(ORDER BY month) AS prev_2,  
LAG(sales, 3) OVER(ORDER BY month) AS prev_3
```

FROM sales;

-- Optimizer typically shares the sort operation

-- Different OVER clauses = different sorts:

SELECT

```
month,  
LAG(sales) OVER(ORDER BY month) AS by_month,  
LAG(sales) OVER(ORDER BY region) AS by_region -- Different sort!
```

FROM sales;

-- May require two separate sort operations

-- Best practice: Create index on ORDER BY columns

```
CREATE INDEX idx_sales_month ON sales(month);
```

🔑 Key Takeaways

1. **LAG** = Previous row value
2. **LEAD** = Next row value
3. **Default offset is 1** (immediate prev/next)
4. **Specify default** to avoid NULLs
5. **ORDER BY required** - determines sequence
6. **PARTITION BY optional** - restarts for groups
7. Perfect for **time-series analysis**



Best Practices

Practice	Reason
Handle first/last row NULLs	Use default value or COALESCE
Use meaningful ORDER BY	Correct sequence is critical
Partition when needed	Separate analysis by group
Use NULLIF in divisions	Avoid divide by zero
Name columns clearly	prev_month, next_day, etc.



Quick Reference

-- Previous value

`LAG(column) OVER(ORDER BY col)`

-- Previous with offset

`LAG(column, 2) OVER(ORDER BY col)`

-- With default

`LAG(column, 1, 0) OVER(ORDER BY col)`

-- Next value

`LEAD(column) OVER(ORDER BY col)`

-- With partition

`LAG(column) OVER(PARTITION BY group ORDER BY col)`

-- Growth calculation

`(value - LAG(value) OVER(...)) * 100.0 / LAG(value) OVER(...)`



Exercises

Exercise 1: Sales Trend

Show monthly sales with previous month and change.

```
SELECT
    month,
    sales,
    LAG(sales) OVER(ORDER BY month) AS prev_month,
    sales - LAG(sales) OVER(ORDER BY month) AS change
FROM monthly_sales;
```

Exercise 2: Customer Order Gaps

Find days between each customer's orders.

```
SELECT
    customer_id,
    order_date,
    LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date) AS prev_order
    DATEDIFF(DAY,
        LAG(order_date) OVER(PARTITION BY customer_id ORDER BY order_date),
        order_date) AS days_gap
FROM orders;
```

Exercise 3: Price Movement Direction

Show if price went up, down, or stayed same.

```
SELECT
    date,
    stock_price,
    CASE
        WHEN stock_price > LAG(stock_price) OVER(ORDER BY date) THEN 'UP'
        WHEN stock_price < LAG(stock_price) OVER(ORDER BY date) THEN 'DOWN'
        ELSE 'SAME'
    END AS movement
FROM daily_prices;
```

Note 23: FIRST_VALUE and LAST_VALUE

Overview

FIRST_VALUE and LAST_VALUE retrieve the first or last value in a window, enabling comparisons to extremes (minimum/maximum) within partitions.

Theory

What are FIRST_VALUE and LAST_VALUE?

- **FIRST_VALUE()** - Returns the first value in the window frame
- **LAST_VALUE()** - Returns the last value in the window frame

FIRST_VALUE and LAST_VALUE VISUALIZATION

Ordered Window: \$100 \$150 \$200 \$180 \$250

↑

↑

FIRST_VALUE

LAST_VALUE

Common Use Cases:

- Compare current value to baseline (first)
- Compare current value to final/latest (last)
- Find gap from minimum or maximum
- Track growth from starting point

⚠ CRITICAL: LAST_VALUE Default Frame Trap!

This is the #1 interview gotcha for LAST_VALUE:

THE LAST_VALUE TRAP

Default Window Frame: ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW ← Problem!

Date	Sales	Expected LAST	Actual LAST (default)
------	-------	---------------	-----------------------

Jan 1	100	80	100 ← Only sees itself!
Jan 5	150	80	150 ← Same problem
Jan 10	200	80	200 ← Frame ends at self
Jan 15	80	80	80 ← Finally correct

FIX: Always add full frame for LAST_VALUE
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Comparison: FIRST_VALUE vs LAST_VALUE

Aspect	FIRST_VALUE	LAST_VALUE
Default behavior	Works as expected	Needs frame clause fix!
Gets	First in ordered sequence	Last in ordered sequence
Frame clause	Optional	REQUIRED for correct result
Alternative	-	Use FIRST_VALUE + ORDER BY DESC

Pro Tip: Avoid LAST_VALUE Problems

```
-- Instead of LAST_VALUE with frame clause:
```

```
LAST_VALUE(col) OVER(ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

```
-- Use FIRST_VALUE with reversed order:
```

```
FIRST_VALUE(col) OVER(ORDER BY date DESC)
```

```
-- Same result, no frame clause needed!
```



FIRST_VALUE Syntax

```
FIRST_VALUE(column) OVER(  
    [PARTITION BY column]  
    ORDER BY column  
    [ROWS BETWEEN ...]  
)
```

Example:

```
SELECT  
    order_date,  
    sales,  
    FIRST_VALUE(sales) OVER(ORDER BY order_date) AS first_sale  
FROM orders;
```

order_date	sales	first_sale
Jan 1	100	100
Jan 5	150	100
Jan 10	200	100
Jan 15	80	100



LAST_VALUE Syntax

```
LAST_VALUE(column) OVER(  
    [PARTITION BY column]  
    ORDER BY column  
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
)
```

⚠ Critical: Frame Clause Required!

By default, LAST_VALUE only sees up to current row, NOT the actual last row!

Wrong (common mistake):

```
LAST_VALUE(sales) OVER(ORDER BY order_date) -- Only sees to current row!
```

Correct:

```
LAST_VALUE(sales) OVER(  
    ORDER BY order_date  
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
)
```



LAST_VALUE Default Behavior Problem

Without Frame Clause:

```
SELECT  
    order_date,  
    sales,  
    LAST_VALUE(sales) OVER(ORDER BY order_date) AS last_sale  
FROM orders;
```

order_date	sales	last_sale
Jan 1	100	100
Jan 5	150	150
Jan 10	200	200
Jan 15	80	80

Problem: Default frame is UNBOUNDED PRECEDING to CURRENT ROW

With Proper Frame Clause:

```
SELECT
    order_date,
    sales,
    LAST_VALUE(sales) OVER(
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_sale
FROM orders;
```

order_date	sales	last_sale
Jan 1	100	80
Jan 5	150	80
Jan 10	200	80
Jan 15	80	80

🎯 Practical Use Cases

Use Case 1: Compare to Lowest (First Sorted Value)

```
-- Compare each sale to the lowest sale
SELECT
    product_id,
    sales,
    FIRST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY sales ASC
    ) AS lowest_sale,
    sales - FIRST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY sales ASC
    ) AS above_lowest
FROM orders;
```

Use Case 2: Compare to Highest (Last Sorted Value)

```
-- Compare each sale to the highest sale
SELECT
    product_id,
    sales,
    LAST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY sales ASC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS highest_sale,
    LAST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY sales ASC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) - sales AS below_highest
FROM orders;
```

Use Case 3: Alternative - Use FIRST_VALUE with DESC

```
-- Get highest using FIRST_VALUE (avoids frame clause issue)
SELECT
    product_id,
    sales,
    FIRST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY sales DESC
    ) AS highest_sale
FROM orders;
```

Use Case 4: First and Last Order Dates

```
SELECT
    customer_id,
    order_date,
    FIRST_VALUE(order_date) OVER(
        PARTITION BY customer_id
        ORDER BY order_date
    ) AS first_order,
    LAST_VALUE(order_date) OVER(
        PARTITION BY customer_id
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_order
FROM orders;
```

💡 Alternative Approaches

Using MIN/MAX Instead:

```
-- Often simpler for just getting min/max values
SELECT
    product_id,
    sales,
    MIN(sales) OVER(PARTITION BY product_id) AS min_sales,
    MAX(sales) OVER(PARTITION BY product_id) AS max_sales
FROM orders;
```

When to Use FIRST_VALUE vs MIN:

Use FIRST_VALUE	Use MIN/MAX
Need first in ORDER BY sequence	Need lowest/highest value
Accessing non-numeric columns	Simple aggregations
Need value at specific position	Order doesn't matter



With PARTITION BY

```
SELECT
    department,
    employee_name,
    salary,
    FIRST_VALUE(employee_name) OVER(
        PARTITION BY department
        ORDER BY salary DESC
    ) AS highest_paid_employee,
    FIRST_VALUE(salary) OVER(
        PARTITION BY department
        ORDER BY salary DESC
    ) AS highest_salary
FROM employees;
```

⚠️ Important Rules

1. ORDER BY is Required

Must sort to define "first" and "last."

2. LAST_VALUE Needs Frame Clause

Always specify `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` for `LAST_VALUE`.

3. Expression Can Be Any Data Type

Works with numbers, strings, dates, etc.

4. Tip: Use FIRST_VALUE with DESC

To get "last" value, often easier to use `FIRST_VALUE` with descending order.

?

Practice Questions

Q1: Get first and last sales per product.

```
SELECT
    product_id,
    order_date,
    sales,
    FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY order_date) AS first_sale,
    LAST_VALUE(sales) OVER(
        PARTITION BY product_id
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_sale
FROM orders;
```

Q2: Compare each employee's salary to department highest.

```
SELECT
    department_id,
    employee_name,
    salary,
    FIRST_VALUE(salary) OVER(
        PARTITION BY department_id
        ORDER BY salary DESC
    ) AS dept_highest,
    FIRST_VALUE(salary) OVER(
        PARTITION BY department_id
        ORDER BY salary DESC
    ) - salary AS diff_from_highest
FROM employees;
```

Q3: Find each customer's first purchase amount.

```

SELECT
    customer_id,
    order_date,
    amount,
    FIRST_VALUE(amount) OVER(
        PARTITION BY customer_id
        ORDER BY order_date
    ) AS first_purchase
FROM orders;

```

Interview Questions

Q1: Why does LAST_VALUE return the current row's value by default?

Answer: Due to the default window frame:

```

-- Default frame (implicit):
LAST_VALUE(sales) OVER(ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    ^^^^^^^^^^^^^^^^^^^^
                    Frame ends HERE!

-- The "last" value in this frame IS the current row
-- Solution: Extend frame to include all rows:
LAST_VALUE(sales) OVER(ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

```

Q2: What's the difference between MAX() OVER() and FIRST_VALUE() OVER(ORDER BY col DESC)?

Answer:

Aspect	MAX() OVER()	FIRST_VALUE(ORDER BY DESC)
Returns	Maximum value	First value after sorting
Ties	Any max value	First in sorted order
Gets name too?	Only value	Can get related columns
Use case	Just need max value	Need max + related data

```
-- Get highest salary
MAX(salary) OVER() -- Returns: 100000

-- Get highest paid employee's NAME (not possible with MAX!)
FIRST_VALUE(employee_name) OVER(ORDER BY salary DESC) -- Returns: 'Alice'
```

Q3: How do you calculate percentage of maximum?

Answer:

```
SELECT
    product_name,
    sales,
    FIRST_VALUE(sales) OVER(ORDER BY sales DESC) AS max_sales,
    ROUND(sales * 100.0 / FIRST_VALUE(sales) OVER(ORDER BY sales DESC), 2) AS pct_of
FROM products;

-- Result: Shows each product as % of best performer
```

Q4: When would you use FIRST_VALUE instead of MIN?

Answer: When you need related column values:

-- Need employee name with lowest salary (can't do with MIN alone!)

SELECT

```
department,
employee_name,
salary,
FIRST_VALUE(employee_name) OVER(
    PARTITION BY department
    ORDER BY salary ASC
) AS lowest_paid_name,
FIRST_VALUE(salary) OVER(
    PARTITION BY department
    ORDER BY salary ASC
) AS lowest_salary
FROM employees;
```

-- MIN can only get the value, not the name:

-- MIN(salary) = 50000, but WHO has that salary?

Q5: How do you compare each row to the first AND last values?

Answer:

SELECT

```
date,
stock_price,
FIRST_VALUE(stock_price) OVER(ORDER BY date) AS opening_price,
FIRST_VALUE(stock_price) OVER(ORDER BY date DESC) AS closing_price, -- Trick!
stock_price - FIRST_VALUE(stock_price) OVER(ORDER BY date) AS change_from_open
FROM stock_prices;
```

-- Notice: Using FIRST_VALUE with DESC order to get "last" value

-- Avoids the LAST_VALUE frame clause trap!

Q6: Find the difference between current value and department minimum?

Answer:

```
SELECT
```

```
    department_id,
    employee_name,
    salary,
    FIRST_VALUE(salary) OVER(
        PARTITION BY department_id
        ORDER BY salary ASC
    ) AS dept_min_salary,
    salary - FIRST_VALUE(salary) OVER(
        PARTITION BY department_id
        ORDER BY salary ASC
    ) AS above_minimum
FROM employees;
```

Q7: FIRST_VALUE vs LAG - when to use which?

Answer:

Use Case	Best Function
Previous row value	LAG(col)
First value in sequence	FIRST_VALUE(col)
Value N rows back	LAG(col, N)
Always same starting point	FIRST_VALUE(col)

-- LAG: Relative to current position (changes per row)

-- FIRST_VALUE: Absolute first (same for all rows)

```
SELECT date, price,
    LAG(price) OVER(ORDER BY date) AS yesterday,      -- Changes per row
    FIRST_VALUE(price) OVER(ORDER BY date) AS day_1   -- Always first day
FROM prices;
```

Q8: How do you get the first non-NULL value?

Answer:

```
-- Method 1: IGNORE NULLS (if supported - Oracle, PostgreSQL 11+)
FIRST_VALUE(column IGNORE NULLS) OVER(ORDER BY date)

-- Method 2: Filter NULLs first (works everywhere)
SELECT * FROM (
    SELECT *,
        FIRST_VALUE(column) OVER(ORDER BY date) AS first_val
    FROM (SELECT * FROM table WHERE column IS NOT NULL) t
) result;

-- Method 3: Use MIN for simple cases
MIN(column) OVER() -- Automatically ignores NULLs
```

Q9: Calculate growth from first value in each category?

Answer:

```
SELECT
    category,
    month,
    sales,
    FIRST_VALUE(sales) OVER(
        PARTITION BY category
        ORDER BY month
    ) AS baseline,
    ROUND(
        (sales - FIRST_VALUE(sales) OVER(PARTITION BY category ORDER BY month)) * 100
        / NULLIF(FIRST_VALUE(sales) OVER(PARTITION BY category ORDER BY month), 0)
        , 2) AS growth_pct_from_baseline
FROM monthly_sales;
```

Q10: Get both the earliest and latest hire in each department?

Answer:

```
SELECT DISTINCT
    department_id,
    FIRST_VALUE(employee_name) OVER(
        PARTITION BY department_id
        ORDER BY hire_date
    ) AS first_hired,
    FIRST_VALUE(hire_date) OVER(
        PARTITION BY department_id
        ORDER BY hire_date
    ) AS first_hire_date,
    FIRST_VALUE(employee_name) OVER(
        PARTITION BY department_id
        ORDER BY hire_date DESC -- Trick: reverse order!
    ) AS latest_hired,
    FIRST_VALUE(hire_date) OVER(
        PARTITION BY department_id
        ORDER BY hire_date DESC
    ) AS latest_hire_date
FROM employees;
```

🔑 Key Takeaways

1. **FIRST_VALUE** - Gets first value in ordered window
2. **LAST_VALUE** - Gets last value (needs frame clause!)
3. **ORDER BY required** - Defines first/last sequence
4. **Frame clause critical** for LAST_VALUE
5. **Alternative:** Use MIN/MAX for simple extremes
6. **Tip:** FIRST_VALUE with DESC avoids frame issues



Best Practices

Practice	Reason
Always use frame with LAST_VALUE	Avoid unexpected results
Consider FIRST_VALUE with DESC	Simpler than LAST_VALUE
Use MIN/MAX when appropriate	Often simpler for aggregates
Test with sample data	Verify expected behavior
Document frame clauses	Helps future maintenance



Quick Reference

```
-- First value (works as expected)
FIRST_VALUE(column) OVER(ORDER BY col)

-- Last value (needs frame!)
LAST_VALUE(column) OVER(
    ORDER BY col
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)

-- Get "last" using FIRST_VALUE + DESC
FIRST_VALUE(column) OVER(ORDER BY col DESC)

-- With partition
FIRST_VALUE(column) OVER(PARTITION BY group ORDER BY col)
```



Exercises

Exercise 1: Compare to Lowest Price

Show product prices with difference from lowest.

```
SELECT
    product_name,
    price,
    FIRST_VALUE(price) OVER(ORDER BY price) AS lowest_price,
    price - FIRST_VALUE(price) OVER(ORDER BY price) AS above_lowest
FROM products;
```

Exercise 2: First and Last Employee by Hire Date

```
SELECT
    department,
    employee_name,
    hire_date,
    FIRST_VALUE(employee_name) OVER(
        PARTITION BY department ORDER BY hire_date
    ) AS first_hired,
    LAST_VALUE(employee_name) OVER(
        PARTITION BY department
        ORDER BY hire_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_hired
FROM employees;
```

Exercise 3: Sales Distance from Extremes

```
SELECT
```

```
product_id,  
sales,  
FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY sales) AS min_sale,  
FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY sales DESC) AS max_sale  
sales - FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY sales) AS from_  
FIRST_VALUE(sales) OVER(PARTITION BY product_id ORDER BY sales DESC) - sales AS  
FROM orders;
```

Note 24: Subqueries - Fundamentals

Overview

A subquery is a query nested inside another query. It allows you to break down complex problems into smaller, manageable steps.

Theory

What is a Subquery?

A **subquery** (inner query/nested query) is a query embedded within another query (outer/main query). It executes first, and its result is used by the outer query.

SUBQUERY EXECUTION FLOW

```
|| SELECT * FROM products  
|| WHERE price > (SELECT AVG(price) FROM products);  
||
```

INNER QUERY

```
|| Step 1: Execute inner query → AVG(price) = $50  
|| Step 2: Substitute result → WHERE price > 50  
|| Step 3: Execute outer query → Return all products with price > 50  
||
```

Subquery Types by Return Value

SUBQUERY RETURN TYPES

SCALAR SUBQUERY (1 row × 1 column)

`50` → Used with: `=, >, <, >=, <=, <>`

Example: `WHERE price > (SELECT AVG(price) FROM products)`

ROW SUBQUERY (N rows × 1 column)

`1`

`2` → Used with: `IN, NOT IN, ANY, ALL`

`3`

Example: `WHERE id IN (SELECT id FROM active_users)`

TABLE SUBQUERY (N rows × M columns)

A	B	C
D	E	F

→ Used in: `FROM clause (derived table)`

→ MUST have an alias!

Example: `FROM (SELECT id, SUM(sales) FROM orders GROUP BY id) AS t`

Subquery Placement Locations

Location	Purpose	Example
WHERE	Filter rows based on subquery result	<code>WHERE price > (SELECT AVG(price)...)</code>

Location	Purpose	Example
SELECT	Add calculated column to each row	<code>SELECT *, (SELECT COUNT(*) FROM...)</code>
FROM	Create a derived/virtual table	<code>FROM (SELECT...GROUP BY) AS t</code>
HAVING	Filter groups based on subquery	<code>HAVING SUM(x) > (SELECT AVG...)</code>
JOIN	Join with derived data	<code>JOIN (SELECT...) AS t ON...</code>

Correlated vs Non-Correlated

Type	Behavior	Performance
Non-Correlated	Runs ONCE, independent of outer query	Faster
Correlated	Runs ONCE PER ROW of outer query	Slower (see Note 25)



Scalar Subqueries

Returns **one value** (one row, one column).

Example: Compare to Average

```
-- Find products priced above average
SELECT product_name, price
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

In SELECT Clause:

```
SELECT
    product_name,
    price,
    (SELECT AVG(price) FROM products) AS avg_price,
    price - (SELECT AVG(price) FROM products) AS diff_from_avg
FROM products;
```

How It Works:

1. Subquery: `SELECT AVG(price) FROM products` → Returns 20
2. Main Query becomes: `SELECT ... WHERE price > 20`
3. Execute main query with the value



Row Subqueries (IN Operator)

Returns **multiple rows, single column** - works with IN operator.

Example: Orders from German Customers

```
-- Find orders from customers in Germany
SELECT *
FROM orders
WHERE customer_id IN (
    SELECT customer_id
    FROM customers
    WHERE country = 'Germany'
);
```

NOT IN:

```
-- Orders NOT from German customers
SELECT *
FROM orders
WHERE customer_id NOT IN (
    SELECT customer_id
    FROM customers
    WHERE country = 'Germany'
);
```



Table Subqueries (FROM Clause)

Returns **multiple rows and columns** - acts as a derived table.

Example: Aggregate then Query

```
-- Find customers with above-average total sales
SELECT *
FROM (
    SELECT
        customer_id,
        SUM(sales) AS total_sales
    FROM orders
    GROUP BY customer_id
) AS customer_totals
WHERE total_sales > 100;
```

Note: Subquery in FROM must have an alias!

🎯 Subqueries by Location

In WHERE Clause:

```
-- Products priced above average
SELECT *
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

In SELECT Clause:

```
-- Add calculated column
SELECT
    order_id,
    sales,
    (SELECT SUM(sales) FROM orders) AS total_all_sales
FROM orders;
```

In FROM Clause:

```
-- Query aggregated results
SELECT customer_id, order_count
FROM (
    SELECT customer_id, COUNT(*) AS order_count
    FROM orders
    GROUP BY customer_id
) AS counts
WHERE order_count > 5;
```

In HAVING Clause:

```
-- Groups with above-average totals
SELECT customer_id, SUM(sales) AS total
FROM orders
GROUP BY customer_id
HAVING SUM(sales) > (SELECT AVG(sales) FROM orders);
```



Comparison Operators with Subqueries

Operator	Subquery Requirement
= > < >= <= <>	Scalar (one value)
IN, NOT IN	Row (multiple values)
ANY, SOME	Row (true if any match)
ALL	Row (true if all match)
EXISTS	Any (checks if rows exist)

ANY/ALL Examples:

```
-- Salary greater than ANY (at least one) male salary
SELECT *
FROM employees
WHERE gender = 'F'
AND salary > ANY (SELECT salary FROM employees WHERE gender = 'M');

-- Salary greater than ALL male salaries
SELECT *
FROM employees
WHERE gender = 'F'
AND salary > ALL (SELECT salary FROM employees WHERE gender = 'M');
```

⚠ Common Rules & Best Practices

1. Scalar Subquery in SELECT Must Return One Value

-- Wrong (multiple values):

```
SELECT *, (SELECT price FROM products) AS p FROM orders; -- Error!
```

-- Right (aggregated to one value):

```
SELECT *, (SELECT AVG(price) FROM products) AS avg_p FROM orders; -- ✓
```

2. FROM Subquery Needs Alias

-- Wrong:

```
SELECT * FROM (SELECT * FROM orders); -- Error!
```

-- Right:

```
SELECT * FROM (SELECT * FROM orders) AS o; -- ✓
```

3. ORDER BY Not Allowed in Subqueries

-- Wrong:

```
SELECT * FROM (SELECT * FROM orders ORDER BY date) AS o; -- Error!
```

-- Right (order in main query):

```
SELECT * FROM (SELECT * FROM orders) AS o ORDER BY date; -- ✓
```

❓ Practice Questions

Q1: Find products above average price.

```
SELECT product_name, price
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

Q2: Find customers who have placed orders.

```
SELECT *
FROM customers
WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders);
```

Q3: Show orders with total sales column.

```
SELECT
    order_id,
    sales,
    (SELECT SUM(sales) FROM orders) AS grand_total,
    sales * 100.0 / (SELECT SUM(sales) FROM orders) AS pct_of_total
FROM orders;
```

Q4: Find top spending customers.

```
SELECT *
FROM (
    SELECT customer_id, SUM(sales) AS total_spent
    FROM orders
    GROUP BY customer_id
) AS spending
WHERE total_spent > 200;
```



Interview Questions

Q1: What is a subquery and when would you use one?

Answer: A subquery is a query nested inside another query. Use it when:

- Need to filter based on aggregated values (WHERE price > AVG)
- Need results of one query as input to another
- Breaking complex problems into steps
- Creating derived tables for further analysis

```
-- Filter based on aggregate (can't do in WHERE directly!)
```

```
SELECT * FROM products WHERE price > (SELECT AVG(price) FROM products);
```

Q2: Subquery vs JOIN - which should you use?

Answer:

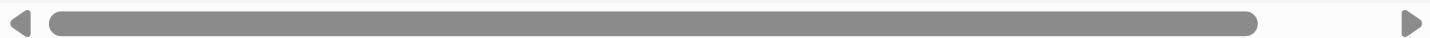
Scenario	Better Choice
Checking existence	EXISTS (subquery)
Filtering by list	IN (subquery) or JOIN
Getting single value	Scalar subquery
Need related columns	JOIN
Performance-critical	Test both (often JOIN is faster)

```
-- Subquery approach
```

```
SELECT * FROM orders WHERE customer_id IN (SELECT id FROM customers WHERE country =
```

```
-- JOIN approach (often faster)
```

```
SELECT o.* FROM orders o JOIN customers c ON o.customer_id = c.id WHERE c.country =
```



Q3: What happens if a scalar subquery returns multiple rows?

Answer: Error! Scalar context requires exactly one value:

```
-- ✗ ERROR: Subquery returns more than 1 row
SELECT * FROM products
WHERE price = (SELECT price FROM products WHERE category = 'Electronics');

-- ✓ Fix: Use IN for multiple values
SELECT * FROM products
WHERE price IN (SELECT price FROM products WHERE category = 'Electronics');

-- ✓ Or aggregate to single value
SELECT * FROM products
WHERE price = (SELECT MAX(price) FROM products WHERE category = 'Electronics');
```

Q4: Explain the difference between ANY and ALL?

Answer:

```
-- ANY: True if comparison matches AT LEAST ONE value
WHERE salary > ANY (SELECT salary FROM dept WHERE name = 'IT')
-- Means: Salary greater than the MINIMUM IT salary

-- ALL: True if comparison matches EVERY value
WHERE salary > ALL (SELECT salary FROM dept WHERE name = 'IT')
-- Means: Salary greater than the MAXIMUM IT salary

-- Memory trick:
-- > ANY = greater than minimum
-- > ALL = greater than maximum
-- < ANY = less than maximum
-- < ALL = less than minimum
```

Q5: Why do FROM subqueries require an alias?

Answer: SQL treats derived tables as virtual tables, and all tables need a name for column qualification:

```
-- ✗ Error: Every derived table must have its own alias
SELECT * FROM (SELECT id, SUM(sales) FROM orders GROUP BY id);

-- ✓ Correct: Alias provided
SELECT * FROM (SELECT id, SUM(sales) AS total FROM orders GROUP BY id) AS order_total
^~~~~~
Required!
```

Q6: Can you use ORDER BY in a subquery?

Answer: Generally **NO**, with exceptions:

```
-- ✗ Error in most databases
SELECT * FROM (SELECT * FROM products ORDER BY price) AS t;

-- ✓ Allowed with TOP/LIMIT (SQL Server)
SELECT * FROM (SELECT TOP 10 * FROM products ORDER BY price) AS t;

-- ✓ Allowed in MySQL
SELECT * FROM (SELECT * FROM products ORDER BY price LIMIT 10) AS t;

-- Best practice: ORDER BY in outer query
SELECT * FROM (SELECT * FROM products) AS t ORDER BY price;
```

Q7: How do you find the second highest salary using a subquery?

Answer:

-- Method 1: Nested subquery

```
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

-- Method 2: Using OFFSET (cleaner)

```
SELECT salary FROM employees ORDER BY salary DESC OFFSET 1 ROW FETCH NEXT 1 ROW ONLY
```

-- Method 3: DENSE_RANK

```
SELECT salary FROM (
    SELECT salary, DENSE_RANK() OVER(ORDER BY salary DESC) AS rnk
    FROM employees
) t WHERE rnk = 2;
```



Q8: When does a subquery run - before or after the WHERE clause?

Answer: Non-correlated subqueries run ONCE before the main query:

```
SELECT * FROM orders WHERE amount > (SELECT AVG(amount) FROM orders);
```

-- Execution order:

- 1. Subquery calculates AVG = 500
- 2. Main query becomes: WHERE amount > 500
- 3. Main query executes

-- Correlated subqueries run PER ROW (different story - see Note 25)

Q9: How do you use a subquery in the SELECT clause?

Answer:

```
SELECT
```

```
    product_name,
    price,
    (SELECT AVG(price) FROM products) AS avg_price,
    price - (SELECT AVG(price) FROM products) AS diff_from_avg,
    price * 100.0 / (SELECT SUM(price) FROM products) AS pct_of_total
FROM products;
```

-- ! Warning: Same subquery executes multiple times!

-- Better: Use window functions

```
SELECT
```

```
    product_name,
    price,
    AVG(price) OVER() AS avg_price,
    price - AVG(price) OVER() AS diff_from_avg
FROM products;
```

Q10: What is a derived table vs a CTE?

Answer:

Aspect	Derived Table (Subquery)	CTE (WITH clause)
Syntax	In FROM clause	Before main query
Reuse	Must repeat if used again	Define once, use many times
Recursion	Not possible	Supports recursion
Readability	Can get messy	Cleaner, top-down logic

```
-- Derived table
SELECT * FROM (SELECT id, SUM(x) AS total FROM t GROUP BY id) AS derived;

-- CTE (cleaner, reusable)
WITH totals AS (
    SELECT id, SUM(x) AS total FROM t GROUP BY id
)
SELECT * FROM totals;
```

🔑 Key Takeaways

1. **Subquery** = Query inside another query
2. **Executes first**, result used by main query
3. **Scalar** = One value (for = > < comparisons)
4. **Row** = Multiple values (for IN, ANY, ALL)
5. **Table** = Multiple columns/rows (needs alias)
6. **Location varies** - SELECT, FROM, WHERE, HAVING

📌 Best Practices

Practice	Reason
Use aliases for FROM subqueries	Required by SQL
Consider JOINs for performance	Sometimes faster
Keep subqueries simple	Easier to debug
Test subqueries separately	Verify intermediate results
Avoid deep nesting	Hard to read and maintain

🧪 Exercises

Exercise 1: Above Average Salary

Find employees with salary above average.

```
SELECT employee_name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Exercise 2: Customers with Orders

Find customers who have placed at least one order.

```
SELECT *
FROM customers
WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders);
```

Exercise 3: Department Totals Above Threshold

Find departments with total salary above \$500,000.

```
SELECT *
FROM (
    SELECT department_id, SUM(salary) AS total_salary
    FROM employees
    GROUP BY department_id
) AS dept_totals
WHERE total_salary > 500000;
```

Note 25: Correlated Subqueries and EXISTS



Overview

Correlated subqueries depend on the outer query and execute once per row. The EXISTS operator checks if a subquery returns any rows.



Non-Correlated vs Correlated

Visual Comparison

NON-CORRELATED vs CORRELATED SUBQUERY

NON-CORRELATED (Independent):

```
SELECT * FROM products WHERE price > (SELECT AVG(price)...)
```

Runs ONCE = 50

Becomes: WHERE price > 50

CORRELATED (Dependent):

```
SELECT * FROM orders o
WHERE sales > (SELECT AVG(sales) FROM orders
                WHERE customer_id = o.customer_id)
```

References outer!

For Customer 1: AVG = 100 → Is sales > 100?

For Customer 2: AVG = 200 → Is sales > 200?

For Customer 3: AVG = 150 → Is sales > 150?

...runs for EACH row!

Key Difference Summary

Aspect	Non-Correlated	Correlated
Dependency	Independent	References outer query
Execution	Runs ONCE	Runs PER ROW

Aspect	Non-Correlated	Correlated
Performance	Faster	Slower (N executions)
Can run alone?	Yes	No (needs outer values)
Use case	Static comparison	Row-specific comparison



EXISTS Operator Deep Dive

HOW EXISTS WORKS

```
SELECT * FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.cust_id = c.cust_id);
```

For each customer:

Customer 1: SELECT 1 FROM orders WHERE cust_id = 1
Returns rows? YES → Include Customer 1

Customer 2: SELECT 1 FROM orders WHERE cust_id = 2
Returns rows? NO → Exclude Customer 2

Customer 3: SELECT 1 FROM orders WHERE cust_id = 3
Returns rows? YES → Include Customer 3

Key Points:

- Returns TRUE/FALSE only (not data)
- SELECT 1 is convention (value doesn't matter)
- STOPS at first match (efficient!)
- Handles NULLs properly (unlike IN)

EXISTS vs IN Comparison

Scenario	EXISTS	IN
Subquery has NULLs	✓ Works correctly	⚠ May return wrong results
Large subquery result	✓ Faster (stops early)	✗ Slower (loads all)
Small subquery result	Similar	Similar
Anti-join (NOT)	✓ NOT EXISTS works	⚠ NOT IN fails with NULLs



Correlated Subquery Examples

Example 1: Compare to Group Average

```
-- Find orders above customer's average
SELECT order_id, customer_id, sales
FROM orders o
WHERE sales > (
    SELECT AVG(sales)
    FROM orders
    WHERE customer_id = o.customer_id
);
```

Example 2: Total Per Customer (in SELECT)

```
-- Add customer order count to each row
SELECT
    c.customer_id,
    c.first_name,
    (SELECT COUNT(*)
     FROM orders o
     WHERE o.customer_id = c.customer_id) AS order_count
FROM customers c;
```

Example 3: Latest Order Per Customer

```
-- Get latest order for each customer
SELECT *
FROM orders o1
WHERE order_date = (
    SELECT MAX(order_date)
    FROM orders o2
    WHERE o2.customer_id = o1.customer_id
);
```



EXISTS Operator

What is EXISTS?

EXISTS checks **if a subquery returns any rows** - returns TRUE/FALSE.

```
SELECT *
FROM table1
WHERE EXISTS (
    SELECT 1
    FROM table2
    WHERE table2.id = table1.id
);
```

Key Points:

- Returns TRUE if subquery has **at least one row**
- Often uses `SELECT 1` (value doesn't matter)
- Typically used with correlated subqueries
- Very efficient - stops at first match



EXISTS Example

Find Customers Who Have Orders:

```
SELECT *
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
);
```

How It Works:

For customer 1:

- Check: `SELECT 1 FROM orders WHERE customer_id = 1`
- Result: Has rows? YES → Include customer 1

For customer 2:

- Check: `SELECT 1 FROM orders WHERE customer_id = 2`
- Result: Has rows? NO → Exclude customer 2

...continues for each customer



NOT EXISTS

Find Customers Without Orders:

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



EXISTS vs IN

Both can solve similar problems:

Using IN:

```
SELECT *
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);
```

Using EXISTS:

```
SELECT *
FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);
```

When to Use Which:

Use EXISTS when...	Use IN when...
Subquery is large	Subquery is small
Just checking existence	Need actual values
Complex correlation	Simple list matching
Better for NULLs	Avoid with NULLs

🎯 Practical Use Cases

Use Case 1: Orders from German Customers (EXISTS)

```
SELECT *
FROM orders o
WHERE EXISTS (
    SELECT 1
    FROM customers c
    WHERE c.customer_id = o.customer_id
        AND c.country = 'Germany'
);
```

Use Case 2: Products Never Ordered

```
SELECT *
FROM products p
WHERE NOT EXISTS (
    SELECT 1
    FROM order_items oi
    WHERE oi.product_id = p.product_id
);
```

Use Case 3: Employees with Above-Average Department Salary

```
SELECT *
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department_id = e.department_id
);
```

Use Case 4: First Order Per Customer

```
SELECT *
FROM orders o1
WHERE order_date = (
    SELECT MIN(order_date)
    FROM orders o2
    WHERE o2.customer_id = o1.customer_id
);
```

⚠️ Performance Considerations

Correlated Subquery Performance:

- Executes **once per row** (can be slow)
- Consider using JOINS or window functions as alternatives

Alternative using JOIN:

```
-- Instead of correlated subquery
SELECT o.*
FROM orders o
INNER JOIN (
    SELECT customer_id, AVG(sales) AS avg_sales
    FROM orders
    GROUP BY customer_id
) avgs ON o.customer_id = avgs.customer_id
WHERE o.sales > avgs.avg_sales;
```

Alternative using Window Function:

```
SELECT *
FROM (
    SELECT *,
        AVG(sales) OVER(PARTITION BY customer_id) AS cust_avg
    FROM orders
) sub
WHERE sales > cust_avg;
```



Comparison Summary

Feature	Non-Correlated	Correlated
Dependency	Independent	Depends on outer
Execution	Once	Per row
Performance	Faster	Slower
Can run alone	Yes	No
Complexity	Simpler	More complex
Use case	Static comparison	Row-by-row comparison

❓ Practice Questions

Q1: Find customers who have placed orders.

```
SELECT *
FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);
```

Q2: Find employees earning above their department average.

```
SELECT *
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department_id = e.department_id
);
```

Q3: Count orders per customer using correlated subquery.

```
SELECT
    c.*,
    (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id) AS order_count
FROM customers c;
```



Q4: Find products never ordered.

```
SELECT *
FROM products p
WHERE NOT EXISTS (
    SELECT 1 FROM order_items oi WHERE oi.product_id = p.product_id
);
```

Interview Questions

Q1: What is a correlated subquery and when would you use one?

Answer: A correlated subquery references the outer query and executes once per row.

Use when:

- Comparing each row to its group's aggregate
- Finding rows based on related table conditions
- Row-by-row comparisons that differ per record

```
-- Find employees earning above their department average
SELECT * FROM employees e
WHERE salary > (
    SELECT AVG(salary) FROM employees
    WHERE department_id = e.department_id -- References outer 'e'!
);
```

Q2: Why is EXISTS preferred over IN when dealing with NULLs?

Answer: NOT IN returns no results if subquery contains NULL:

```
-- Suppose subquery returns: (1, 2, NULL)

-- NOT IN fails with NULL!
SELECT * FROM customers WHERE id NOT IN (1, 2, NULL);
-- Logically: id != 1 AND id != 2 AND id != NULL
-- id != NULL is UNKNOWN → entire condition becomes UNKNOWN
-- Result: 0 rows! (unexpected)

-- NOT EXISTS handles NULLs correctly
SELECT * FROM customers c
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id);
-- Works as expected!
```

Q3: How does EXISTS differ from JOIN for checking relationships?

Answer:

```
-- EXISTS: Returns customer data only, no duplicates
SELECT * FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id);
-- If customer has 10 orders, still 1 row returned

-- JOIN: May create duplicates
SELECT c.* FROM customers c
JOIN orders o ON c.id = o.customer_id;
-- If customer has 10 orders, 10 rows returned!

-- For existence check, EXISTS is cleaner
-- For needing order data too, use JOIN
```

Q4: What is the performance implication of correlated subqueries?

Answer: They can be slow because they execute once per row:

```
-- 10,000 employees = 10,000 subquery executions!
SELECT * FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE dept_id = e.dept_id);

-- Alternative: JOIN with pre-aggregated data (runs twice only)
SELECT e.* FROM employees e
JOIN (SELECT dept_id, AVG(salary) AS avg_sal FROM employees GROUP BY dept_id) d
ON e.dept_id = d.dept_id
WHERE e.salary > d.avg_sal;

-- Alternative: Window function (single pass)
SELECT * FROM (
    SELECT *, AVG(salary) OVER(PARTITION BY dept_id) AS dept_avg
    FROM employees
) t WHERE salary > dept_avg;
```

Q5: How do you find duplicates using a correlated subquery?

Answer:

```
-- Find all rows that have duplicates (based on email)
SELECT * FROM users u1
WHERE EXISTS (
    SELECT 1 FROM users u2
    WHERE u2.email = u1.email
    AND u2.id <> u1.id -- Different row with same email
);

-- Find one copy of each duplicate to delete
SELECT * FROM users u1
WHERE id > (SELECT MIN(id) FROM users u2 WHERE u2.email = u1.email);
```

Q6: What is the difference between WHERE EXISTS and HAVING EXISTS?

Answer:

```
-- WHERE EXISTS: Filter individual rows
SELECT * FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id);

-- HAVING EXISTS: Filter groups (after GROUP BY)
SELECT department_id, COUNT(*) AS emp_count
FROM employees e
GROUP BY department_id
HAVING EXISTS (
    SELECT 1 FROM projects p
    WHERE p.department_id = e.department_id
);
```

Q7: How do you find the Nth highest salary per department?

Answer:

```
-- 3rd highest salary in each department
SELECT * FROM employees e1
WHERE 3 = (
    SELECT COUNT(DISTINCT salary) FROM employees e2
    WHERE e2.department_id = e1.department_id
    AND e2.salary >= e1.salary
);
```

Q8: When should you use a correlated subquery in SELECT clause?

Answer:

```
-- Add calculated column per row based on related data
SELECT
    c.customer_name,
    c.city,
    (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.id) AS order_count,
    (SELECT MAX(order_date) FROM orders o WHERE o.customer_id = c.id) AS last_order
FROM customers c;

-- ⚠ Caution: Each subquery runs per row!
-- Better alternative: LEFT JOIN with aggregation
SELECT c.customer_name, c.city,
    COALESCE(o.order_count, 0) AS order_count,
    o.last_order
FROM customers c
LEFT JOIN (
    SELECT customer_id, COUNT(*) AS order_count, MAX(order_date) AS last_order
    FROM orders GROUP BY customer_id
) o ON c.id = o.customer_id;
```

Q9: Explain the execution order of EXISTS vs the outer query

Answer:

```
-- EXISTS is evaluated for EACH row of outer query:
SELECT * FROM customers c          -- Step 1: Get first customer row
WHERE EXISTS (                      -- Step 2: Run this for that row
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.id       -- Using value from Step 1
);
                                         -- Step 3: Include row if TRUE
                                         -- Step 4: Repeat for next row

-- The subquery doesn't run completely first!
-- It's interleaved with outer query processing
```

Q10: Convert a correlated subquery to a window function

Answer:

```
-- Original: Correlated subquery (slow)
SELECT order_id, amount,
       (SELECT SUM(amount) FROM orders o2
        WHERE o2.customer_id = o1.customer_id) AS customer_total
  FROM orders o1;

-- Converted: Window function (fast)
SELECT order_id, amount,
       SUM(amount) OVER(PARTITION BY customer_id) AS customer_total
  FROM orders;

-- Window function benefits:
-- ✓ Single table scan
-- ✓ Cleaner syntax
-- ✓ Better performance
```

🔑 Key Takeaways

1. **Correlated subqueries** reference outer query
2. **Execute per row** (can be slower)
3. **EXISTS** checks if rows exist (TRUE/FALSE)
4. **NOT EXISTS** finds missing relationships
5. **SELECT 1** is common in EXISTS (value irrelevant)
6. Consider **JOINS or window functions** for performance



Best Practices

Practice	Reason
Use EXISTS for existence checks	More efficient than IN for large data
Consider performance	Correlated = slower
Test alternatives	JOINS may be faster
Use aliases clearly	o, c, e for tables
Use SELECT 1 with EXISTS	Signals "just checking"



Exercises

Exercise 1: Customers with Orders (EXISTS)

```
SELECT c.customer_id, c.first_name
FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);
```

Exercise 2: Above Department Average (Correlated)

```
SELECT employee_name, salary, department_id
FROM employees e
WHERE salary > (
    SELECT AVG(salary) FROM employees WHERE department_id = e.department_id
);
```

Exercise 3: Products Without Sales (NOT EXISTS)

```
SELECT product_name
FROM products p
WHERE NOT EXISTS (
    SELECT 1 FROM sales s WHERE s.product_id = p.product_id
);
```

Note 26: Common Table Expressions (CTEs)

Overview

CTEs (Common Table Expressions) create temporary, named result sets that exist only within a single query. They make complex queries more readable and maintainable.

Theory

What is a CTE?

A **CTE** (Common Table Expression) is a temporary named result set defined at the beginning of a query using the `WITH` keyword. It exists only for the duration of that single query.

CTE STRUCTURE AND FLOW

```
|| WITH cte_name AS (           ← Name the result set
||     SELECT ...               ← Define the query
|| )
|| SELECT * FROM cte_name;      ← Use it like a table
```

EXECUTION

- || Step 1: CTE query executes
- || Step 2: Result stored temporarily as "cte_name"
- || Step 3: Main query uses cte_name like a real table
- || Step 4: Final result returned
- || Step 5: CTE vanishes (exists only during query)

CTE Types: Standalone vs Nested

STANDALONE vs NESTED CTEs

STANDALONE (Independent):

```
WITH
    sales AS (SELECT ... FROM orders),      ← Queries database
    customers AS (SELECT ... FROM customers) ← Queries database
SELECT ... FROM sales JOIN customers ...
```

Both CTEs are independent, can run in parallel

NESTED (Dependent):

```
WITH
    totals AS (SELECT ... FROM orders),      ← Step 1: Run this
    ranked AS (SELECT ... FROM totals)        ← Step 2: Uses ^^^
SELECT ... FROM ranked
```

Second CTE depends on first, must run sequentially

CTE vs Subquery Comparison

Aspect	CTE	Subquery
Readability	High (named, top-level)	Lower (nested inline)
Reusability	Reference multiple times	Repeat for each use
Debugging	Easy (run CTE alone)	Hard (buried in query)
Recursion	Supports recursive queries	Not possible
Position	Before main query	Inline within query
Naming	Required (self-documenting)	Optional



Basic CTE Syntax

```
WITH cte_name AS (
    -- CTE query (definition)
    SELECT column1, column2
    FROM table
    WHERE condition
)
-- Main query (uses the CTE)
SELECT *
FROM cte_name;
```

Key Rules:

- Starts with `WITH` keyword
- CTE name followed by `AS`
- Query wrapped in parentheses
- Main query immediately follows
- CTE only exists for that query



Simple CTE Example

Task: Find total sales per customer, then filter high spenders

```
WITH customer_totals AS (
    SELECT
        customer_id,
        SUM(sales) AS total_sales
    FROM orders
    GROUP BY customer_id
)
SELECT *
FROM customer_totals
WHERE total_sales > 100;
```

Execution Flow:

1. CTE executes: Calculates totals per customer
2. Result stored temporarily as "customer_totals"
3. Main query filters the CTE results
4. Final result returned
5. CTE destroyed (no longer exists)



Multiple CTEs (Standalone)

Multiple independent CTEs separated by commas:

```
WITH
```

```
    cte_sales AS (
        SELECT customer_id, SUM(sales) AS total_sales
        FROM orders
        GROUP BY customer_id
    ),
    cte_orders AS (
        SELECT customer_id, MAX(order_date) AS last_order
        FROM orders
        GROUP BY customer_id
    )
SELECT
    c.customer_id,
    c.first_name,
    s.total_sales,
    o.last_order
FROM customers c
LEFT JOIN cte_sales s ON c.customer_id = s.customer_id
LEFT JOIN cte_orders o ON c.customer_id = o.customer_id;
```

Rules for Multiple CTEs:

- Only **first CTE** uses `WITH`
- Subsequent CTEs use **comma** separator
- No comma after last CTE (before main query)
- Each CTE is independent (standalone)



Nested CTEs

One CTE references another CTE:

```
WITH
```

```
    cte_totals AS (
        SELECT customer_id, SUM(sales) AS total_sales
        FROM orders
        GROUP BY customer_id
    ),
    cte_ranked AS (
        SELECT
            customer_id,
            total_sales,
            RANK() OVER(ORDER BY total_sales DESC) AS sales_rank
        FROM cte_totals -- References first CTE!
    )
SELECT *
FROM cte_ranked
WHERE sales_rank <= 5;
```

Nested vs Standalone:

Standalone CTE	Nested CTE
Goes directly to database	References another CTE
Can run independently	Depends on previous CTE
No dependencies	Must run in sequence

🎯 Practical Examples

Example 1: Customer Report

WITH

```
sales_summary AS (
    SELECT
        customer_id,
        SUM(sales) AS total_sales,
        COUNT(*) AS order_count
    FROM orders
    GROUP BY customer_id
),
last_order AS (
    SELECT
        customer_id,
        MAX(order_date) AS last_order_date
    FROM orders
    GROUP BY customer_id
)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    ss.total_sales,
    ss.order_count,
    lo.last_order_date
FROM customers c
LEFT JOIN sales_summary ss ON c.customer_id = ss.customer_id
LEFT JOIN last_order lo ON c.customer_id = lo.customer_id;
```

Example 2: Segmentation with Nested CTE

```

WITH
    customer_totals AS (
        SELECT customer_id, SUM(sales) AS total_sales
        FROM orders
        GROUP BY customer_id
    ),
    customer_segments AS (
        SELECT
            customer_id,
            total_sales,
            CASE
                WHEN total_sales > 500 THEN 'High'
                WHEN total_sales > 200 THEN 'Medium'
                ELSE 'Low'
            END AS segment
        FROM customer_totals
    )
SELECT *
FROM customer_segments
ORDER BY total_sales DESC;

```

Example 3: Top N Analysis

```

WITH ranked_products AS (
    SELECT
        product_id,
        SUM(sales) AS total_revenue,
        ROW_NUMBER() OVER(ORDER BY SUM(sales) DESC) AS rank
    FROM orders
    GROUP BY product_id
)
SELECT *
FROM ranked_products
WHERE rank <= 10;

```

⚠️ CTE Restrictions

NOT Allowed in CTEs:

```
-- Cannot use ORDER BY in CTE
WITH cte AS (
    SELECT * FROM orders ORDER BY date -- Error!
)
SELECT * FROM cte;
```

ORDER BY Restriction:

- **Not allowed** in CTE definition
- **Allowed** in main query

```
-- Correct approach
WITH cte AS (
    SELECT * FROM orders
)
SELECT * FROM cte ORDER BY date; -- ORDER BY here!
```



CTE vs Subquery

Feature	CTE	Subquery
Readability	High	Lower
Reusability	Can reference multiple times	Defined each time
Named	Yes	Usually not
Position	Before main query	Inline
Debugging	Easier	Harder

Same Query - Two Approaches:

Subquery:

```
SELECT *
FROM (
    SELECT customer_id, SUM(sales) AS total
    FROM orders
    GROUP BY customer_id
) AS totals
WHERE total > 100;
```

CTE:

```
WITH totals AS (
    SELECT customer_id, SUM(sales) AS total
    FROM orders
    GROUP BY customer_id
)
SELECT *
FROM totals
WHERE total > 100;
```

📌 Best Practices

1. Limit Number of CTEs

Recommended: 3-5 CTEs per query
 More than 5: Consider splitting into separate queries

2. Use Descriptive Names

```
WITH customer_sales_totals AS (...) -- Good ✓
WITH cte1 AS (...) -- Bad X
```

3. One Purpose Per CTE

```
-- Good: Each CTE has one job
```

```
WITH sales AS (...),  
     orders AS (...),  
     rankings AS (...)
```

4. Refactor to Reduce Redundancy

Merge CTEs when logic overlaps.

❓ Practice Questions

Q1: Calculate customer totals using CTE.

```
WITH customer_totals AS (  
    SELECT customer_id, SUM(sales) AS total_sales  
    FROM orders  
    GROUP BY customer_id  
)  
SELECT * FROM customer_totals ORDER BY total_sales DESC;
```

Q2: Create customer report with multiple CTEs.

```

WITH
sales AS (
    SELECT customer_id, SUM(sales) AS total, COUNT(*) AS orders
    FROM orders GROUP BY customer_id
),
last_orders AS (
    SELECT customer_id, MAX(order_date) AS last_order
    FROM orders GROUP BY customer_id
)
SELECT c.*, s.total, s.orders, l.last_order
FROM customers c
LEFT JOIN sales s ON c.customer_id = s.customer_id
LEFT JOIN last_orders l ON c.customer_id = l.customer_id;

```

Q3: Rank and filter using nested CTE.

```

WITH
totals AS (
    SELECT product_id, SUM(sales) AS revenue
    FROM orders GROUP BY product_id
),
ranked AS (
    SELECT *, RANK() OVER(ORDER BY revenue DESC) AS rnk
    FROM totals
)
SELECT * FROM ranked WHERE rnk <= 5;

```

Interview Questions

Q1: What is a CTE and when would you use one?

Answer: A CTE (Common Table Expression) is a temporary named result set that exists only within a single query execution.

Use cases:

- Breaking complex queries into readable steps
- Reusing the same subquery multiple times
- Creating recursive queries (hierarchical data)
- Improving code maintainability

```
WITH sales_totals AS (
    SELECT customer_id, SUM(amount) AS total
    FROM orders GROUP BY customer_id
)
SELECT c.name, s.total
FROM customers c JOIN sales_totals s ON c.id = s.customer_id
WHERE s.total > 1000;
```

Q2: Can you reference a CTE multiple times in the same query?

Answer: Yes! This is one of the main advantages over subqueries:

```
WITH sales_summary AS (
    SELECT customer_id, SUM(amount) AS total
    FROM orders GROUP BY customer_id
)
-- Referenced 3 times!
SELECT
    (SELECT COUNT(*) FROM sales_summary) AS total_customers,
    (SELECT AVG(total) FROM sales_summary) AS avg_sales,
    (SELECT MAX(total) FROM sales_summary) AS max_sales;

-- With subquery, you'd repeat the same SELECT 3 times!
```

Q3: What are the differences between CTE and Temp Table?

Answer:

Aspect	CTE	Temp Table
Scope	Single query only	Entire session
Storage	Memory (usually)	TempDB on disk
Indexing	Not possible	Can add indexes
Persistence	Vanishes after query	Exists until dropped
Statistics	Uses base table stats	Has own statistics
Use case	Readability, single query	Large datasets, multiple queries

Q4: Can you have ORDER BY inside a CTE?

Answer: Generally No, except with TOP/OFFSET:

```
-- ✗ Error: ORDER BY not allowed in CTE
WITH cte AS (
    SELECT * FROM orders ORDER BY date
)
SELECT * FROM cte;

-- ✓ Allowed with TOP (SQL Server)
WITH cte AS (
    SELECT TOP 100 * FROM orders ORDER BY date
)
SELECT * FROM cte;

-- ✓ ORDER BY in main query
WITH cte AS (
    SELECT * FROM orders
)
SELECT * FROM cte ORDER BY date;
```

Q5: What is a recursive CTE?

Answer: A CTE that references itself, used for hierarchical data:

```
-- Organization hierarchy
WITH org_chart AS (
    -- Anchor: Top-level employees (no manager)
    SELECT id, name, manager_id, 0 AS level
    FROM employees WHERE manager_id IS NULL

    UNION ALL

    -- Recursive: Each level's reports
    SELECT e.id, e.name, e.manager_id, oc.level + 1
    FROM employees e
    JOIN org_chart oc ON e.manager_id = oc.id -- References itself!
)
SELECT * FROM org_chart;

-- Results: Full org chart with hierarchy levels
```

Q6: How many CTEs can you have in one query?

Answer: No hard limit, but best practice is 3-5:

```
WITH
    cte1 AS (...),
    cte2 AS (...),
    cte3 AS (...),
    cte4 AS (...),
    cte5 AS (...) -- No comma after last one!
SELECT ...;

-- Rules:
-- Only first CTE has WITH
-- Comma between CTEs
-- No comma before main SELECT
```

Q7: CTE vs Subquery - which performs better?

Answer: Usually the same - the optimizer treats them similarly:

-- These typically have identical execution plans:

-- CTE

```
WITH totals AS (
    SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id
)
SELECT * FROM totals WHERE total > 1000;
```

-- Subquery

```
SELECT * FROM (
    SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id
) AS totals WHERE total > 1000;
```

-- Performance difference comes from:

-- 1. CTE referenced multiple times (may re-execute)

-- 2. Temp table might be better for very large intermediate results

Q8: How do you convert a nested subquery to CTEs?

Answer:

```
-- Nested subquery (hard to read)
SELECT * FROM (
    SELECT *, RANK() OVER(ORDER BY total DESC) AS rnk
    FROM (
        SELECT customer_id, SUM(amount) AS total
        FROM orders GROUP BY customer_id
    ) inner_totals
) ranked WHERE rnk <= 10;

-- Converted to CTEs (readable)
WITH
    totals AS (
        SELECT customer_id, SUM(amount) AS total
        FROM orders GROUP BY customer_id
    ),
    ranked AS (
        SELECT *, RANK() OVER(ORDER BY total DESC) AS rnk
        FROM totals
    )
SELECT * FROM ranked WHERE rnk <= 10;
```

Q9: Can you use DML (INSERT, UPDATE, DELETE) with CTEs?

Answer: Yes! CTEs can be used with DML statements:

```
-- CTE with DELETE
WITH duplicates AS (
    SELECT id, ROW_NUMBER() OVER(PARTITION BY email ORDER BY id) AS rn
    FROM users
)
DELETE FROM duplicates WHERE rn > 1;

-- CTE with UPDATE
WITH ranked_employees AS (
    SELECT id, salary, DENSE_RANK() OVER(ORDER BY salary DESC) AS rnk
    FROM employees
)
UPDATE ranked_employees SET salary = salary * 1.1 WHERE rnk = 1;
```

Q10: What happens if a CTE is referenced but never used?

Answer: It's still executed (in most databases):

```
WITH
    used_cte AS (SELECT * FROM orders),
    unused_cte AS (SELECT * FROM large_table) -- Still runs!
SELECT * FROM used_cte;

-- Best practice: Remove unused CTEs for performance
-- Some optimizers may skip unused CTEs, but don't rely on it
```

🔑 Key Takeaways

1. **WITH** keyword starts CTE
2. CTEs exist **only for one query**
3. **Multiple CTEs** separated by commas
4. **Nested CTEs** reference previous CTEs
5. **ORDER BY** not allowed inside CTE
6. More readable than subqueries
7. Limit to **3-5 CTEs** per query



Quick Reference

```
-- Single CTE
WITH cte_name AS (
    SELECT ...
)
SELECT * FROM cte_name;

-- Multiple CTEs
WITH
    cte1 AS (...),
    cte2 AS (...),
    cte3 AS ...
SELECT ...;

-- Nested CTE (references another)
WITH
    base AS (SELECT ...),
    derived AS (SELECT * FROM base)
SELECT * FROM derived;
```

Exercises

Exercise 1: Simple Aggregation CTE

```
WITH monthly_sales AS (
    SELECT
        MONTH(order_date) AS month,
        SUM(sales) AS total
    FROM orders
    GROUP BY MONTH(order_date)
)
SELECT * FROM monthly_sales ORDER BY month;
```

Exercise 2: Multiple Standalone CTEs

```

WITH
    customer_stats AS (
        SELECT customer_id, SUM(sales) AS total, COUNT(*) AS orders
        FROM orders GROUP BY customer_id
    ),
    product_stats AS (
        SELECT product_id, SUM(sales) AS total, COUNT(*) AS orders
        FROM orders GROUP BY product_id
    )
SELECT 'Customers' AS type, COUNT(*) AS count FROM customer_stats
UNION ALL
SELECT 'Products', COUNT(*) FROM product_stats;

```

Exercise 3: Nested CTE for Ranking

```

WITH
    emp_totals AS (
        SELECT department_id, SUM(salary) AS dept_salary
        FROM employees GROUP BY department_id
    ),
    emp_ranked AS (
        SELECT *, DENSE_RANK() OVER(ORDER BY dept_salary DESC) AS rnk
        FROM emp_totals
    )
SELECT * FROM emp_ranked WHERE rnk <= 3;

```

Note 27: SQL Views

Overview

Views are virtual tables in SQL that are based on the result of a query without actually storing data in the database. They provide abstraction, reusability, and simplify complex queries for end users.



Theory

What is a View?

VIEW vs TABLE ARCHITECTURE

TABLE (Physical):

DISK STORAGE

Row 1	Row 2	Row 3	Row 4
			← Actual data stored

VIEW (Virtual):

SYSTEM CATALOG

Query: `SELECT a.*, b.* FROM a JOIN b ON a.id = b.id`

↓

When accessed: Execute query → Return result → No storage

View Execution Flow

HOW VIEWS EXECUTE

User Query: `SELECT * FROM v_sales_summary WHERE region = 'West'`

- Step 1: Database finds view definition in System Catalog
- Step 2: View query merged/substituted into user query
- Step 3: Combined query executes against base tables
- Step 4: Result returned to user

View Definition:

```
SELECT region, SUM(sales) AS total FROM orders GROUP BY region
```

Expanded Query:

```
SELECT * FROM (
    SELECT region, SUM(sales) AS total
    FROM orders GROUP BY region
) AS v_sales_summary
WHERE region = 'West'
```

Views vs Tables Comparison

Aspect	Table	View
Storage	Stores actual data on disk	Stores only the query (no data)
Data freshness	Point-in-time	Always current (queries base tables)
Write operations	INSERT, UPDATE, DELETE	Generally read-only*

Aspect	Table	View
Performance	Direct data access (fast)	Executes query each time (slower)
Maintenance	Schema changes = migration	Just modify the query
Indexes	Can be indexed	Cannot be indexed (usually)

*Some simple views are updatable

Creating Views

Basic Syntax

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

With Schema

```
CREATE VIEW schema_name.view_name AS
(
    SELECT column1, column2, ...
    FROM table_name
);
```

Example: Monthly Summary View

```
CREATE VIEW sales.v_monthly_summary AS
(
    SELECT
        DATE_TRUNC('month', order_date) AS order_month,
        SUM(sales) AS total_sales,
        COUNT(order_id) AS total_orders,
        SUM(quantity) AS total_quantities
    FROM sales.orders
    GROUP BY DATE_TRUNC('month', order_date)
);
```

Using Views

Query a View

```
-- Query view like any table
SELECT * FROM sales.v_monthly_summary;

-- Use view with window functions
SELECT
    order_month,
    total_sales,
    SUM(total_sales) OVER (ORDER BY order_month) AS running_total
FROM sales.v_monthly_summary;
```

Managing Views

Drop a View

```
DROP VIEW view_name;

-- With schema
DROP VIEW sales.v_monthly_summary;
```

Update/Replace a View (SQL Server)

```
-- Method 1: Drop and recreate
DROP VIEW sales.v_monthly_summary;
CREATE VIEW sales.v_monthly_summary AS
(
    -- new query here
);

-- Method 2: Using T-SQL (if exists, drop first)
IF OBJECT_ID('sales.v_monthly_summary', 'V') IS NOT NULL
    DROP VIEW sales.v_monthly_summary;
GO
CREATE VIEW sales.v_monthly_summary AS
(
    -- query here
);
```

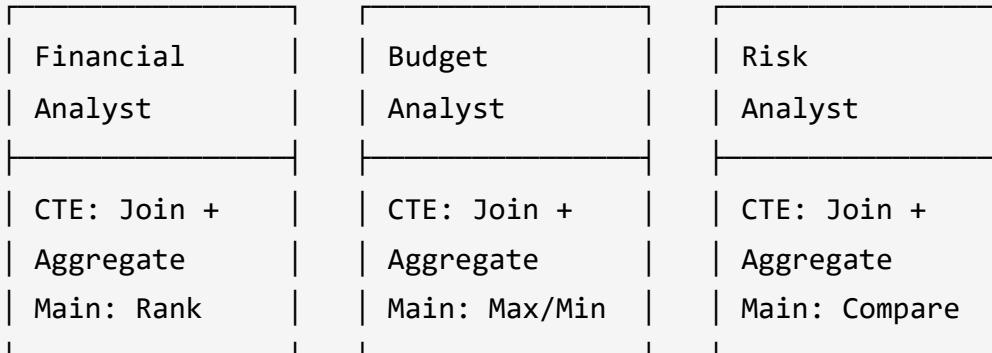
In PostgreSQL/Other Databases

```
CREATE OR REPLACE VIEW view_name AS
(
    SELECT ...
);
```

Why Use Views?

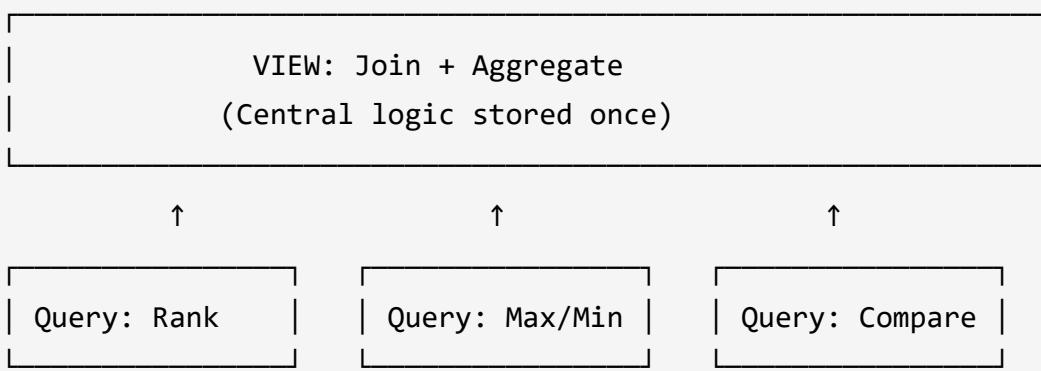
Use Case 1: Store Central Logic (Reusability)

Without Views:



↓ ↓ ↓
Same CTE repeated 3 times = REDUNDANCY

With Views:



Use Case 2: Hide Complexity (Abstraction)

- Complex database with technical table names
- Multiple joins required for simple analysis
- Create friendly views for end users

```
-- Create a user-friendly view combining multiple tables
CREATE VIEW v_order_details AS
(
  SELECT
    o.order_id,
    p.product_name,
    p.category,
    c.first_name + ' ' + c.last_name AS customer_name,
    c.country AS customer_country,
    e.first_name + ' ' + e.last_name AS employee_name,
    o.order_date,
    o.sales,
    o.quantity
  FROM sales.orders o
  LEFT JOIN sales.products p ON p.product_id = o.product_id
  LEFT JOIN sales.customers c ON c.customer_id = o.customer_id
  LEFT JOIN sales.employees e ON e.employee_id = o.salesperson_id
);

```

Views vs CTEs Comparison

Aspect	Views	CTEs
Scope	Multiple queries	Single query
Persistence	Persisted in database	Temporary (on-the-fly)
Reusability	Project-wide	Within one query
Maintenance	Need to create/drop	No maintenance
When to Use	Important logic for many users	Logic for one specific scenario

Rule of Thumb

- **View:** Logic is important and will be reused across multiple queries
- **CTE:** Logic is only needed within one query, not worth persisting

How Database Executes Views

Creating a View

1. Data Engineer → CREATE VIEW command
2. Database Engine → Stores metadata + query in System Catalog
3. No actual data stored in disk/cache

Querying a View

1. User → SELECT FROM view
2. Database Engine → Retrieves query from System Catalog
3. Database Engine → Executes view query against physical tables
4. Result → Returned to user

Dropping a View

1. User → DROP VIEW command
2. Database Engine → Removes metadata + query from Catalog
3. Original table data → NOT affected

Practice Questions

Q1: Create a Monthly Sales Summary View

```
CREATE VIEW v_monthly_sales AS
(
    SELECT
        DATE_TRUNC('month', order_date) AS order_month,
        SUM(sales) AS total_sales,
        COUNT(*) AS total_orders
    FROM sales.orders
    GROUP BY DATE_TRUNC('month', order_date)
);
```

Q2: Create a Customer Orders View

```
CREATE VIEW v_customer_orders AS
(
    SELECT
        c.customer_id,
        c.first_name + ' ' + c.last_name AS full_name,
        c.country,
        COUNT(o.order_id) AS order_count,
        SUM(o.sales) AS total_spent
    FROM sales.customers c
    LEFT JOIN sales.orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.first_name, c.last_name, c.country
);
```

Q3: Use the View for Analysis

```
-- Calculate running total from view
SELECT
    order_month,
    total_sales,
    SUM(total_sales) OVER (ORDER BY order_month) AS running_total
FROM v_monthly_sales
ORDER BY order_month;
```

Key Takeaways

1. **Views are virtual tables** - They store queries, not data
2. **Data is always fresh** - Query executes each time view is accessed
3. **Reduce redundancy** - Central logic accessible by all users
4. **Improve abstraction** - Hide complex joins from end users
5. **Read-only** - Cannot write data through views
6. **Slower than tables** - Extra step to execute the view query
7. **Easy to maintain** - Just modify the query, no data migration

Best Practices

1. **Naming Convention:** Use prefix like `v_` for views
2. **Specify Schema:** Always include schema name for organization
3. **Document Views:** Add comments explaining the view's purpose
4. **Don't Over-Use:** Use tables for performance-critical operations
5. **Review Regularly:** Clean up unused views
6. **Test Changes:** Always test view modifications before deploying

Exercises

1. Create a view that shows all products with their total sales
 2. Create a view combining employees, orders, and customers
 3. Write a query that uses your view to calculate rankings
 4. Practice dropping and recreating views with schema
 5. Compare query performance: direct table vs through view
-

Interview Questions

Q1: What is a view and how is it different from a table?

Answer: A view is a **virtual table** based on a stored SELECT query. Unlike tables, views don't store data - they execute the query each time accessed.

Difference	Table	View
Data storage	On disk	None (query stored)
Freshness	Point-in-time	Always current
Updates	Direct	Limited/None
Speed	Faster	Slower (runs query)

Q2: Can you UPDATE data through a view?

Answer: Sometimes, if the view is "updatable":

```
--  Updatable view (simple, single table)
CREATE VIEW v_employees AS SELECT id, name, salary FROM employees;
UPDATE v_employees SET salary = 50000 WHERE id = 1; -- Works!
```

-- Non-updatable view (aggregation, joins, DISTINCT)

```
CREATE VIEW v_dept_summary AS
SELECT dept, AVG(salary) AS avg_sal FROM employees GROUP BY dept;
UPDATE v_dept_summary SET avg_sal = 60000; -- Error!
```

Non-updatable if view contains: GROUP BY, DISTINCT, aggregates, JOINs (usually), UNION, subqueries in SELECT.

Q3: What is the difference between View and Materialized View?

Answer:

Aspect	Regular View	Materialized View
Data storage	None (virtual)	Stores result set
Execution	Every access	Once, then cached
Freshness	Always current	May be stale
Refresh	N/A	Manual or scheduled
Performance	Slower	Faster (pre-computed)
Use case	Light queries	Complex aggregations

```
-- Materialized View (PostgreSQL)
CREATE MATERIALIZED VIEW mv_sales_summary AS
SELECT region, SUM(sales) FROM orders GROUP BY region;

-- Refresh when needed
REFRESH MATERIALIZED VIEW mv_sales_summary;
```

Q4: When would you use a view vs a CTE?

Answer:

Use View When	Use CTE When
Logic reused across queries	Logic for single query
Shared by multiple users	Only you need it
Permanent abstraction needed	Temporary, one-time use
Security/access control	No permission concerns

Q5: How can views provide security?

Answer: Views can hide sensitive columns and rows:

```
-- Full table (has salary, SSN)
-- employees: id, name, department, salary, ssn

-- View for general users (hides sensitive data)
CREATE VIEW v_employees_public AS
SELECT id, name, department FROM employees;

-- View for HR only (shows salary, not SSN)
CREATE VIEW v_employees_hr AS
SELECT id, name, department, salary FROM employees;

-- GRANT access to views, not base table
GRANT SELECT ON v_employees_public TO public_role;
GRANT SELECT ON v_employees_hr TO hr_role;
```

Q6: Can you create an index on a view?

Answer: Not on regular views in most databases. But **Indexed Views** (SQL Server) / **Materialized Views** support indexes:

```
-- SQL Server Indexed View
CREATE VIEW v_order_totals
WITH SCHEMABINDING -- Required for indexed view
AS
SELECT customer_id, SUM(amount) AS total, COUNT_BIG(*) AS cnt
FROM dbo.orders
GROUP BY customer_id;
GO

-- Create index on the view
CREATE UNIQUE CLUSTERED INDEX idx_v_order_totals
ON v_order_totals(customer_id);
```

Q7: What happens to views when base table changes?

Answer:

Change	Effect on View
Add column	View still works (doesn't see new column)
Drop column used by view	View breaks!
Rename column used by view	View breaks!
Drop table	View breaks!
Add/remove rows	View reflects changes

```
-- Best practice: Use SCHEMABINDING (SQL Server)
CREATE VIEW v_orders WITH SCHEMABINDING AS
SELECT order_id, customer_id FROM dbo.orders;
-- Now you CANNOT drop or alter orders table columns!
```

Q8: Why might a query through a view be slow?

Answer:

1. **View query complexity** - Complex JOINs, aggregations
2. **No indexes on view** - Regular views can't be indexed

3. **Nested views** - View calling view calling view...
4. **No query optimization** - Optimizer may not push predicates

```
-- Slow: Filters applied AFTER view executes  
SELECT * FROM v_all_orders WHERE customer_id = 1;  
  
-- Optimizer should push down, but may not always  
-- Check execution plan!
```

Q9: What is WITH CHECK OPTION in views?

Answer: Ensures INSERT/UPDATE through view doesn't create rows invisible to the view:

```
CREATE VIEW v_active_employees AS  
SELECT * FROM employees WHERE status = 'active'  
WITH CHECK OPTION;  
  
-- This would fail!  
INSERT INTO v_active_employees (name, status) VALUES ('John', 'inactive');  
-- Error: Row would be invisible to view  
  
-- This succeeds  
INSERT INTO v_active_employees (name, status) VALUES ('Jane', 'active');
```

Q10: How do you troubleshoot a broken view?

Answer:

```
-- 1. Check if view exists
SELECT * FROM INFORMATION_SCHEMA.VIEWS WHERE TABLE_NAME = 'v_sales';

-- 2. Get view definition
EXEC sp_helptext 'v_sales'; -- SQL Server
-- or
SELECT definition FROM sys.sql_modules WHERE object_id = OBJECT_ID('v_sales');

-- 3. Check for dependency issues
SELECT * FROM sys.dm_sql_referencing_entities('dbo.v_sales', 'OBJECT');

-- 4. Refresh view metadata (after base table changes)
EXEC sp_refreshview 'v_sales'; -- SQL Server
```

Note 28: CTAS and Temporary Tables

Overview

This note covers two important methods for creating tables from queries: **CTAS (Create Table As Select)** and **Temporary Tables**. These are essential techniques for data transformation, ETL processes, and data analysis.



Theory

CTAS vs Traditional Create + Insert

TABLE CREATION METHODS

TRADITIONAL (Two Steps):

```
| Step 1: CREATE TABLE orders (id INT, name VARCHAR(100), ...)  
| Step 2: INSERT INTO orders SELECT ... FROM source
```

Pros: Define exact data types, constraints, indexes

Cons: Two statements, more verbose

CTAS (One Step):

```
| SQL Server: SELECT ... INTO new_table FROM source  
| Others:      CREATE TABLE new_table AS SELECT ... FROM source
```

Pros: One statement, infers data types from query

Cons: No constraints/indexes (must add after)

Temporary Table Types (SQL Server)

TEMPORARY TABLE TYPES

LOCAL TEMP TABLE (#):

```
| SELECT * INTO #orders FROM sales.orders
```

- Visible ONLY to current session
- Dropped when session ends
- Stored in tempdb
- Most common type

GLOBAL TEMP TABLE (##):

```
| SELECT * INTO ##orders FROM sales.orders
```

- Visible to ALL sessions
- Dropped when LAST session using it ends
- Use with caution (naming conflicts!)

TABLE VARIABLE (@):

```
| DECLARE @orders TABLE (id INT, name VARCHAR(100))
```

```
| INSERT INTO @orders SELECT ...
```

- Scoped to batch/stored procedure
- Minimal logging (faster for small sets)
- No statistics (bad for large sets)

View vs CTAS vs Temp Table Comparison

Aspect	View	CTAS Table	Temp Table
Storage	None (virtual)	Permanent on disk	tempdb (temporary)
Data freshness	Always current	Snapshot at creation	Snapshot at creation
Lifetime	Until dropped	Until dropped	Session/connection
Auto cleanup	No	No	Yes
Can modify data	Rarely	Yes	Yes
Use case	Abstraction, security	Performance, snapshots	ETL, experimentation

CTAS Syntax by Database

MySQL, PostgreSQL, Oracle

```
CREATE TABLE new_table AS
(
    SELECT columns
    FROM existing_table
    WHERE condition
);
```

SQL Server (SELECT INTO)

```
SELECT columns
INTO new_table
FROM existing_table
WHERE condition;
```

CTAS Examples

Basic Example

```
-- SQL Server syntax
SELECT
    DATENAME(month, order_date) AS order_month,
    COUNT(order_id) AS total_orders
INTO sales.monthly_orders
FROM sales.orders
GROUP BY DATENAME(month, order_date);
```

Verify the Table

```
-- Check the new table
SELECT * FROM sales.monthly_orders;
```

Drop the Table

```
DROP TABLE sales.monthly_orders;
```

Refreshing CTAS Tables

Problem: Table Already Exists

```
-- This will fail if table exists
SELECT * INTO sales.monthly_orders FROM ...
-- Error: "Table already exists"
```

Solution: Drop First, Then Recreate

```
-- Using T-SQL (SQL Server)
IF OBJECT_ID('sales.monthly_orders', 'U') IS NOT NULL
    DROP TABLE sales.monthly_orders;
GO

SELECT
    DATENAME(month, order_date) AS order_month,
    COUNT(order_id) AS total_orders
INTO sales.monthly_orders
FROM sales.orders
GROUP BY DATENAME(month, order_date);
```

Views vs CTAS Tables

Aspect	Views	CTAS Tables
Data Storage	Virtual (no data)	Physical (data stored)
Query Execution	Every time accessed	Once at creation
Data Freshness	Always current	Snapshot at creation time
Performance	Slower (executes query)	Faster (data pre-computed)
Maintenance	Easy (change query)	Need to refresh/recreate
Storage	No additional storage	Requires storage space

Analogy

- **View:** Ordering pizza at restaurant - made fresh each time
- **CTAS Table:** Frozen pizza - prepared earlier, just heat up

When to Use CTAS vs Views

Use Views When:

- You need always-fresh data
- Query is simple and fast
- Multiple users need real-time access
- Flexibility is more important than speed

Use CTAS When:

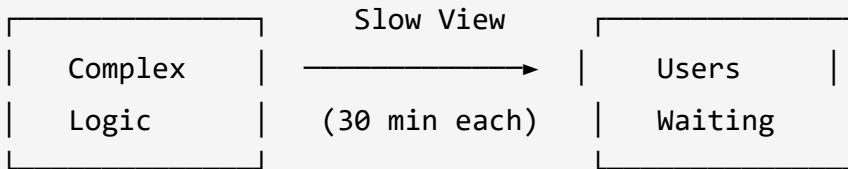
- Query is complex and slow (30+ minutes)
- Performance at query time matters
- Data doesn't need real-time updates
- You can prepare data overnight

Common CTAS Use Cases

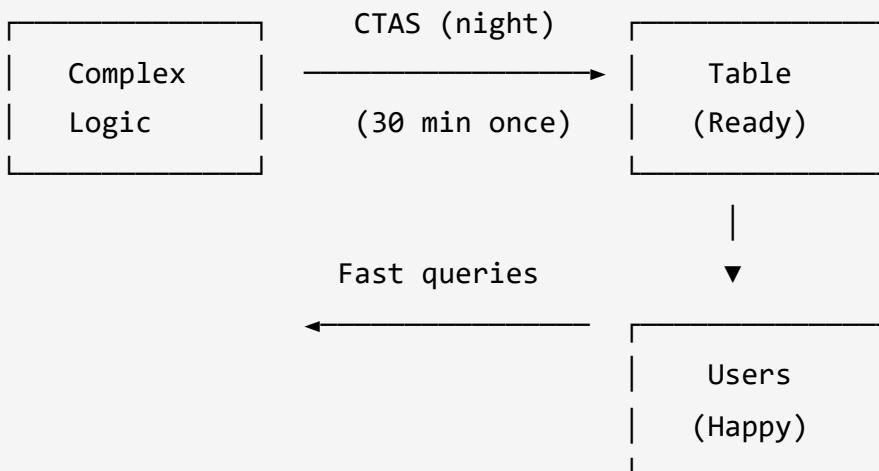
1. Performance Optimization

Scenario: View query takes 30 minutes

Solution: Use CTAS to pre-compute overnight



vs.



2. Data Quality Analysis (Snapshot)

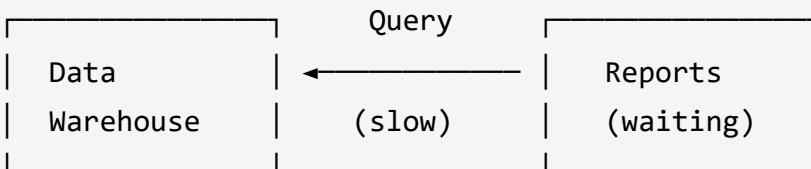
```

-- Create snapshot to analyze a bug
SELECT *
INTO debug.orders_snapshot_20240115
FROM sales.orders
WHERE order_date = '2024-01-15';

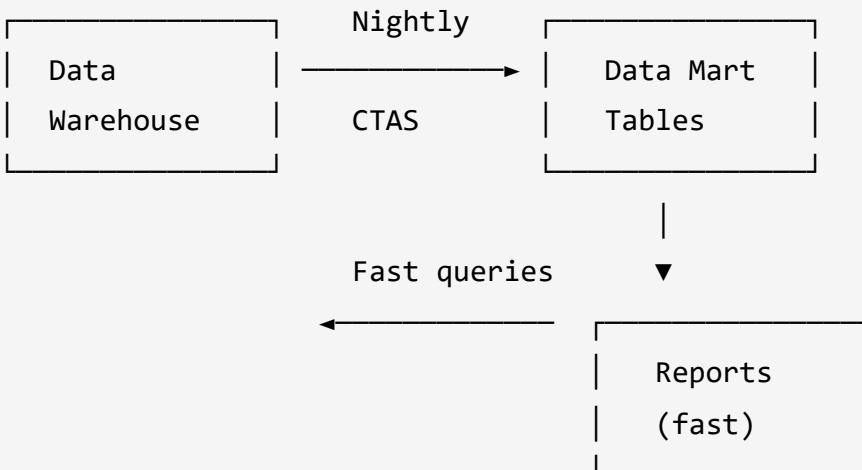
-- Analyze without worrying about data changes
SELECT * FROM debug.orders_snapshot_20240115;
  
```

3. Physical Data Marts

Virtual Data Mart (Views):



Physical Data Mart (CTAS):



Temporary Tables

What Are Temporary Tables?

- Store intermediate results in temporary storage
- Automatically dropped when session ends
- Perfect for ETL workflows and data manipulation

Syntax

```
-- Add # before table name to make it temporary
SELECT columns
INTO #temp_table
FROM source_table
WHERE condition;
```

Key Characteristics

- Created in **tempdb** database
 - Prefixed with **#** (single hash)
 - **Session-scoped**: Dropped automatically when you disconnect
 - Can be queried like regular tables during session
-

Temporary Table Examples

Create a Temporary Table

```
SELECT *
INTO #orders_temp
FROM sales.orders;

-- Verify
SELECT * FROM #orders_temp;
```

Modify Temporary Data

```
-- Delete some records
DELETE FROM #orders_temp
WHERE order_status = 'Delivered';

-- Check result
SELECT * FROM #orders_temp;
```

Save Results Back to Permanent Table

```
-- After manipulation, save to permanent table
SELECT *
INTO sales.orders_test
FROM #orders_temp;
```

Where Are Temporary Tables Stored?

Location in SQL Server

```
System Databases
  └── tempdb
    └── Temporary Tables
      └── #orders_temp (your table here)
```

Session Lifecycle

1. Connect to database → Session starts
2. CREATE #temp_table → Table created in tempdb
3. Query/Modify → Use like normal table
4. Disconnect → Session ends
5. Automatic cleanup → tempdb removes table

Permanent vs Temporary Tables

Aspect	Permanent Tables	Temporary Tables
Prefix	No prefix	# prefix
Location	User database	tempdb
Lifetime	Until explicitly dropped	Until session ends
Visibility	All sessions	Current session only
Use Case	Persistent data	Intermediate results

Use Case: ETL Process with Temp Tables

Without Temp Tables (Not Allowed)

Source Table (Cannot modify directly)

↓

Remove Duplicates X

Handle Nulls X

Filter Data X

↓

Data Warehouse

With Temp Tables (Best Practice)

-- Step 1: Extract to temp table

```
SELECT * INTO #staging FROM source.orders;
```

-- Step 2: Transform (clean data)

```
DELETE FROM #staging WHERE duplicate = 1;  
UPDATE #staging SET value = 0 WHERE value IS NULL;  
DELETE FROM #staging WHERE status = 'Invalid';
```

-- Step 3: Load to warehouse

```
INSERT INTO warehouse.orders  
SELECT * FROM #staging;
```

Tables Summary

Table Types

1. **Permanent Tables:** Live forever until dropped

- Create Insert method
- CTAS method

2. **Temporary Tables:** Dropped when session ends

- Use # prefix
- Good for intermediate results

Comparison Chart

	Permanent Tables	Temporary Tables
Method		
Create Insert	Manual structure	N/A
CTAS	From query (no # prefix)	From query (# prefix)
Lifetime	Until DROP	Until session end
Auto Cleanup	No	Yes

Practice Questions

Q1: Create Table from Query

```
-- Create table of customer order summaries
SELECT
    c.customer_id,
    c.first_name,
    c.country,
    COUNT(o.order_id) AS total_orders,
    SUM(o.sales) AS total_sales
INTO sales.customer_summary
FROM sales.customers c
LEFT JOIN sales.orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.country;
```

Q2: Create Temporary Table for Analysis

```
-- Copy orders to temp table for manipulation
SELECT * INTO #orders_analysis
FROM sales.orders
WHERE YEAR(order_date) = 2024;

-- Perform analysis
SELECT order_status, COUNT(*)
FROM #orders_analysis
GROUP BY order_status;
```

Q3: Refresh CTAS Table

```
-- Drop if exists and recreate
IF OBJECT_ID('sales.daily_summary', 'U') IS NOT NULL
    DROP TABLE sales.daily_summary;
GO

SELECT
    CAST(order_date AS DATE) AS order_date,
    SUM(sales) AS daily_sales
INTO sales.daily_summary
FROM sales.orders
GROUP BY CAST(order_date AS DATE);
```

Key Takeaways

1. **CTAS creates physical tables** from query results in one step
2. **Views are virtual**, CTAS tables are physical - different use cases
3. **Temporary tables (#)** auto-cleanup when session ends
4. **Use CTAS** when view performance is too slow
5. **Use temp tables** for safe data manipulation without affecting source
6. **SQL Server uses SELECT INTO** instead of CREATE TABLE AS

Best Practices

1. **Start with Views:** Only switch to CTAS when performance demands
 2. **Naming Convention:** Clear names indicating table purpose and date
 3. **Document CTAS Jobs:** Track what runs nightly vs. on-demand
 4. **Clean Up:** Don't leave unnecessary permanent tables
 5. **Use Temp Tables Wisely:** Great for ETL, debugging, experimentation
 6. **Monitor tempdb:** Too many temp tables can fill up tempdb
-

Exercises

1. Create a CTAS table summarizing monthly sales
 2. Write a script to refresh the table (drop if exists + recreate)
 3. Create a temporary table, modify it, then save results
 4. Compare query performance: View vs CTAS table
 5. Implement a simple ETL using temporary tables
-

Interview Questions

Q1: What is CTAS and when would you use it?

Answer: CTAS (Create Table As Select) creates a new table from query results in one statement.

Use cases:

- Creating summary/aggregate tables for performance
- Taking data snapshots for debugging
- Building data marts from data warehouse
- Migrating/copying data between schemas

```
-- SQL Server syntax
SELECT customer_id, SUM(amount) AS total
INTO sales.customer_totals
FROM orders
GROUP BY customer_id;
```

```
-- PostgreSQL/MySQL syntax
CREATE TABLE sales.customer_totals AS
SELECT customer_id, SUM(amount) AS total
FROM orders
GROUP BY customer_id;
```

Q2: What is the difference between # and ## temporary tables?

Answer:

Aspect	Local (#)	Global (##)
Visibility	Current session only	All sessions
Naming	#table_name	##table_name
Dropped when	Session ends	Last session using it ends
Naming conflict	Safe (session-scoped)	Possible (be careful!)

```
-- Only I can see this
SELECT * INTO #my_temp FROM orders;

-- Everyone can see this (until last user disconnects)
SELECT * INTO ##shared_temp FROM orders;
```

Q3: Temp Table vs Table Variable - when to use which?

Answer:

Scenario	Use Temp Table (#)	Use Table Variable (@)
Small data (<1000 rows)	✗	✓ Faster
Large data	✓ Has statistics	✗ No statistics
Need indexes	✓ Supported	⚠ Limited
Transaction rollback	✓ Rolled back	✗ NOT rolled back
Scope needed	Session	Batch/procedure

```
-- Table variable (small sets, stays after rollback)
DECLARE @temp TABLE (id INT, name VARCHAR(50));
INSERT INTO @temp SELECT TOP 100 * FROM orders;

-- Temp table (large sets, has statistics)
SELECT * INTO #temp FROM orders; -- Full table
```

Q4: How do you refresh a CTAS table (update with new data)?

Answer:

```
-- Method 1: Drop and recreate
IF OBJECT_ID('sales.daily_summary', 'U') IS NOT NULL
    DROP TABLE sales.daily_summary;
GO
SELECT * INTO sales.daily_summary FROM ...;

-- Method 2: TRUNCATE and INSERT (preserves structure/indexes)
TRUNCATE TABLE sales.daily_summary;
INSERT INTO sales.daily_summary SELECT * FROM ...;

-- Method 3: SQL Server 2016+ (DROP IF EXISTS)
DROP TABLE IF EXISTS sales.daily_summary;
SELECT * INTO sales.daily_summary FROM ...;
```

Q5: What constraints/indexes does CTAS copy from source?

Answer: Almost nothing!

```
-- Source table has:  
-- • PRIMARY KEY  
-- • FOREIGN KEYS  
-- • INDEXES  
-- • DEFAULTS  
-- • CHECK constraints  
  
SELECT * INTO new_table FROM source_table;  
  
-- new_table has:  
-- • Column names and data types ✓  
-- • NULL/NOT NULL (usually) ✓  
-- • NOTHING ELSE! ✗  
  
-- Must add manually:  
ALTER TABLE new_table ADD PRIMARY KEY (id);  
CREATE INDEX idx_name ON new_table(column);
```

Q6: Where are temporary tables stored and why does it matter?

Answer: In **tempdb** database.

Why it matters:

- tempdb is shared by all databases on server
- Too many/large temp tables can fill tempdb
- tempdb performance affects all queries using temp tables
- tempdb is recreated on SQL Server restart (tables lost)

```
-- Check tempdb space usage
SELECT
    SUM(size * 8.0 / 1024) AS size_mb
FROM tempdb.sys.database_files;
```

Q7: Can you create indexes on temporary tables?

Answer: Yes! This is a key advantage over table variables:

```
-- Create temp table
SELECT * INTO #orders FROM sales.orders;

-- Add index for better performance
CREATE INDEX idx_temp_customer ON #orders(customer_id);
CREATE INDEX idx_temp_date ON #orders(order_date);

-- Now queries on #orders can use these indexes
SELECT * FROM #orders WHERE customer_id = 100;
```

Q8: How do you use temp tables in ETL processes?

Answer:

```
-- Extract: Copy source data to temp table
SELECT * INTO #staging FROM source_system.orders;

-- Transform: Clean and modify safely
DELETE FROM #staging WHERE amount IS NULL;
UPDATE #staging SET status = 'Unknown' WHERE status = '';
DELETE FROM #staging WHERE duplicate_flag = 1;

-- Load: Insert clean data to destination
INSERT INTO warehouse.orders
SELECT * FROM #staging;

-- Cleanup: Automatic when session ends!
```

Q9: CTAS performance tips?

Answer:

```
-- 1. Use SELECT INTO with minimal logging (SQL Server)
-- Requires simple/bulk-logged recovery model for full benefit

-- 2. Add indexes AFTER loading data (faster)
SELECT * INTO new_table FROM big_query; -- Load first
CREATE INDEX idx ON new_table(col); -- Then index

-- 3. Use parallel insert (SQL Server 2016+)
ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ADAPTIVE_JOINS = ON;

-- 4. Consider partitioning for very large tables
```

Q10: Can you use temp tables across stored procedures?

Answer: Yes for # tables, with conditions:

```
-- Procedure 1 creates temp table
CREATE PROCEDURE sp_create_temp AS
BEGIN
    SELECT * INTO #shared FROM orders;
    -- #shared exists in calling session
    EXEC sp_use_temp; -- Can access #shared!
END

-- Procedure 2 uses it (if called from same session)
CREATE PROCEDURE sp_use_temp AS
BEGIN
    SELECT * FROM #shared; -- Works if sp_create_temp called first
END

-- ❌ But this won't work:
-- Inner procedure creating temp table for outer procedure
-- Temp tables are visible DOWN the call stack, not UP
```

Note 29: Stored Procedures

Overview

Stored Procedures are pre-compiled SQL programs stored in the database that can be executed on demand. They allow you to bundle multiple SQL statements into a single reusable unit with parameters, variables, and control flow logic.



Theory

What is a Stored Procedure?

STORED PROCEDURE CONCEPT

Without Stored Procedures:

Application

- SQL Query 1 (50 lines)
- SQL Query 2 (30 lines)
- SQL Query 3 (40 lines)
- ...duplicated in every app that needs this logic

With Stored Procedures:

Database

- PROCEDURE: sp_generate_report
- All logic centralized here
- Pre-compiled execution plan
- Controlled access

App 1 App 2 App 3
EXEC sp_generate_report (all just call it)

Stored Procedure Execution Flow

EXECUTION LIFECYCLE

FIRST EXECUTION:

1. Parse SQL syntax
2. Compile query plan
3. Cache compiled plan
4. Execute

SUBSEQUENT EXECUTIONS:

1. Retrieve cached plan ← Skip parsing/compiling!
2. Execute

Benefits:

- Faster execution (cached plan)
- Reduced network traffic (only EXEC command sent)
- Consistent execution

Stored Procedure Components

Component	Purpose	Example
Parameters	Accept input values	@country VARCHAR(50)
Variables	Store intermediate values	DECLARE @total INT
Control Flow	Conditional logic	IF...ELSE , WHILE
Error Handling	Graceful failure	TRY...CATCH
Return Values	Status codes	RETURN 0 (success)

Component	Purpose	Example
Output Parameters	Return multiple values	@result INT OUTPUT

Creating Stored Procedures

Basic Syntax

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    -- SQL statements here
END;
```

With Schema

```
CREATE PROCEDURE schema_name.procedure_name
AS
BEGIN
    -- SQL statements here
END;
```

Example: Customer Summary

```
CREATE PROCEDURE get_customer_summary
AS
BEGIN
    SELECT
        COUNT(*) AS total_customers,
        AVG(score) AS average_score
    FROM sales.customers;
END;
```

Executing Stored Procedures

Execute Command

```
-- Method 1: EXECUTE keyword  
EXECUTE get_customer_summary;  
  
-- Method 2: EXEC shorthand  
EXEC get_customer_summary;
```

Managing Stored Procedures

Alter (Modify)

```
ALTER PROCEDURE get_customer_summary  
AS  
BEGIN  
    -- Modified SQL statements  
    SELECT  
        COUNT(*) AS total_customers,  
        AVG(score) AS average_score,  
        MAX(score) AS highest_score  
    FROM sales.customers;  
END;
```

Drop (Delete)

```
DROP PROCEDURE get_customer_summary;
```

Parameters

Input Parameters

Parameters make stored procedures dynamic and reusable.

Syntax

```
CREATE PROCEDURE procedure_name
    @parameter_name data_type
AS
BEGIN
    -- Use @parameter_name in queries
END;
```

Example: Filter by Country

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50)
AS
BEGIN
    SELECT
        COUNT(*) AS total_customers,
        AVG(score) AS average_score
    FROM sales.customers
    WHERE country = @country;
END;
```

Execute with Parameter

```
-- Pass parameter value
EXEC get_customer_summary @country = 'Germany';
EXEC get_customer_summary @country = 'USA';
```

Default Parameter Values

Syntax

```
CREATE PROCEDURE procedure_name
    @parameter_name data_type = default_value
AS
BEGIN
    -- Use parameter
END;
```

Example

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    SELECT
        COUNT(*) AS total_customers,
        AVG(score) AS average_score
    FROM sales.customers
    WHERE country = @country;
END;
```

Execute with Default

```
-- Uses default value 'USA'
EXEC get_customer_summary;

-- Override default with specific value
EXEC get_customer_summary @country = 'Germany';
```

Multiple SQL Statements

Multiple Queries in One Procedure

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    -- Report 1: Customer counts
    SELECT
        COUNT(*) AS total_customers,
        AVG(score) AS average_score
    FROM sales.customers
    WHERE country = @country;

    -- Report 2: Order summary
    SELECT
        COUNT(o.order_id) AS total_orders,
        SUM(o.sales) AS total_sales
    FROM sales.orders o
    JOIN sales.customers c ON c.customer_id = o.customer_id
    WHERE c.country = @country;
END;
```

Best Practice: Use Semicolons

```
-- End each statement with semicolon for clarity
SELECT ... FROM ... WHERE ...;
SELECT ... FROM ... WHERE ...;
```

Variables

What Are Variables?

Variables are placeholders that store values within a stored procedure for reuse.

Three Steps to Use Variables

1. Declare

```
DECLARE @variable_name data_type;
```

2. Assign Value

```
SET @variable_name = value;
-- or assign from query
SELECT @variable_name = column FROM table;
```

3. Use

```
-- Use in queries, conditions, or PRINT statements
SELECT * FROM table WHERE column = @variable_name;
PRINT @variable_name;
```

Complete Example

```

CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    -- Declare variables
    DECLARE @total_customers INT;
    DECLARE @average_score FLOAT;

    -- Assign values from query
    SELECT
        @total_customers = COUNT(*),
        @average_score = AVG(score)
    FROM sales.customers
    WHERE country = @country;

    -- Use variables in PRINT
    PRINT 'Total customers from ' + @country + ': '
        + CAST(@total_customers AS VARCHAR);
    PRINT 'Average score from ' + @country + ': '
        + CAST(@average_score AS VARCHAR);
END;

```

Parameters vs Variables

Aspect	Parameters	Variables
Source	Input from user	Defined inside procedure
Purpose	Make procedure flexible	Store intermediate values
Scope	Passed when executing	Internal to procedure
Syntax	After procedure name	DECLARE inside BEGIN/END

Control Flow: IF-ELSE

Syntax

```
IF condition
BEGIN
    -- Statements if TRUE
END
ELSE
BEGIN
    -- Statements if FALSE
END
```

Example: Data Cleanup

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    -- Check if nulls exist
    IF EXISTS (
        SELECT 1 FROM sales.customers
        WHERE score IS NULL AND country = @country
    )
    BEGIN
        PRINT 'Updating null scores to zero';
        UPDATE sales.customers
        SET score = 0
        WHERE score IS NULL AND country = @country;
    END
    ELSE
    BEGIN
        PRINT 'No null scores found';
    END
    -- Generate report
    SELECT
        COUNT(*) AS total_customers,
        AVG(score) AS average_score
    FROM sales.customers
    WHERE country = @country;
END;
```

Why Use IF-ELSE?

- **Efficiency:** Only run operations when needed
- **Resource saving:** Skip unnecessary updates
- **Control:** Different logic for different conditions

Error Handling: TRY-CATCH

Syntax

```
BEGIN TRY  
    -- Code that might fail  
END TRY  
  
BEGIN CATCH  
    -- Handle error  
END CATCH
```

Error Functions

Function	Description
ERROR_MESSAGE()	Error description text
ERROR_NUMBER()	Error ID number
ERROR_LINE()	Line number where error occurred
ERROR_PROCEDURE()	Name of stored procedure
ERROR_SEVERITY()	Error severity level
ERROR_STATE()	Error state number

Complete Example

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    BEGIN TRY
        -- =====
        -- Step 1: Prepare and Clean Up Data
        -- =====

        IF EXISTS (
            SELECT 1 FROM sales.customers
            WHERE score IS NULL AND country = @country
        )
        BEGIN
            PRINT 'Updating null scores to zero';
            UPDATE sales.customers
            SET score = 0
            WHERE score IS NULL AND country = @country;
        END
        ELSE
        BEGIN
            PRINT 'No null scores found';
        END

        -- =====
        -- Step 2: Generate Summary Reports
        -- =====

        DECLARE @total_customers INT;
        DECLARE @average_score FLOAT;

        SELECT
            @total_customers = COUNT(*),
            @average_score = AVG(score)
        FROM sales.customers
        WHERE country = @country;

        PRINT 'Total customers from ' + @country + ': '
        + CAST(@total_customers AS VARCHAR);
        PRINT 'Average score: ' + CAST(@average_score AS VARCHAR);
    END TRY
    BEGIN CATCH
        PRINT 'An error occurred during the procedure execution';
        ROLLBACK TRANSACTION;
    END CATCH
END
```

```
-- Return data
SELECT
    COUNT(o.order_id) AS total_orders,
    SUM(o.sales) AS total_sales
FROM sales.orders o
JOIN sales.customers c ON c.customer_id = o.customer_id
WHERE c.country = @country;

END TRY
BEGIN CATCH
    -- =====
    -- Error Handling
    -- =====

    PRINT 'An error occurred';
    PRINT 'Error Message: ' + ERROR_MESSAGE();
    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS VARCHAR);
    PRINT 'Error Line: ' + CAST(ERROR_LINE() AS VARCHAR);
    PRINT 'Error Procedure: ' + ERROR_PROCEDURE();
END CATCH
END;
```

Code Organization Best Practices

Use Indentation

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    BEGIN TRY
        -- First level indent
        IF condition
            BEGIN
                -- Second level indent
                SELECT ...;
            END
    END TRY
    BEGIN CATCH
        -- Error handling
    END CATCH
END;
```

Use Comments and Sections

```
-- =====
-- Step 1: Data Preparation
-- =====
-- Description of what this section does

-- =====
-- Step 2: Generate Reports
-- =====
-- Calculate total customers and average score
```

Complete Stored Procedure Template

```
CREATE PROCEDURE schema_name.procedure_name
    @param1 VARCHAR(50) = 'default_value',
    @param2 INT = 0
AS
BEGIN
    BEGIN TRY
        -- =====
        -- Variable Declarations
        -- =====
        DECLARE @var1 INT;
        DECLARE @var2 VARCHAR(100);

        -- =====
        -- Step 1: Data Validation/Preparation
        -- =====
        IF EXISTS (SELECT 1 FROM table WHERE condition)
        BEGIN
            -- Action if condition met
            PRINT 'Condition met - taking action';
        END
        ELSE
        BEGIN
            PRINT 'No action needed';
        END

        -- =====
        -- Step 2: Main Logic
        -- =====
        SELECT @var1 = COUNT(*), @var2 = MAX(column)
        FROM table
        WHERE condition;

        -- =====
        -- Step 3: Output Results
        -- =====
        PRINT 'Result: ' + CAST(@var1 AS VARCHAR);

        SELECT column1, column2
```

```
    FROM table
    WHERE condition;

END TRY
BEGIN CATCH
    -- =====
    -- Error Handling
    -- =====
    PRINT 'Error: ' + ERROR_MESSAGE();
    PRINT 'Line: ' + CAST(ERROR_LINE() AS VARCHAR);
END CATCH
END;
```

Practice Questions

Q1: Create Basic Stored Procedure

```
CREATE PROCEDURE get_order_count
AS
BEGIN
    SELECT COUNT(*) AS total_orders FROM sales.orders;
END;
```

Q2: Add Parameter

```
CREATE PROCEDURE get_orders_by_status
    @status VARCHAR(50)
AS
BEGIN
    SELECT * FROM sales.orders
    WHERE order_status = @status;
END;

-- Execute
EXEC get_orders_by_status @status = 'Shipped';
```

Q3: Multiple Parameters with Default

```
CREATE PROCEDURE get_sales_report
    @year INT = 2024,
    @status VARCHAR(50) = 'Completed'
AS
BEGIN
    SELECT
        MONTH(order_date) AS month,
        SUM(sales) AS total_sales
    FROM sales.orders
    WHERE YEAR(order_date) = @year
        AND order_status = @status
    GROUP BY MONTH(order_date);
END;
```

Key Takeaways

1. **Stored Procedures bundle SQL** into reusable programs
2. **Parameters make them flexible** - same logic, different data
3. **Default values** provide convenience for common cases
4. **Variables store intermediate values** for reuse
5. **IF-ELSE provides control flow** for conditional logic
6. **TRY-CATCH handles errors** gracefully
7. **Good organization** makes code maintainable

Best Practices

1. **Use meaningful names:** `get_customer_summary` not `proc1`
2. **Include schema:** `sales.get_summary`
3. **Add comments:** Document purpose and parameters
4. **Use consistent indentation:** Improves readability
5. **Validate inputs:** Check parameters before processing

6. **Handle errors:** Always use TRY-CATCH for production code
 7. **End statements with semicolons:** Clear statement boundaries
-

Exercises

1. Create a stored procedure to get top 10 orders by sales amount
 2. Add a parameter to filter by date range
 3. Add error handling with custom error messages
 4. Create a procedure that updates data and returns a result
 5. Build a multi-step ETL procedure with variables
-

Interview Questions

Q1: What are the advantages of stored procedures over inline SQL?

Answer:

Advantage	Explanation
Performance	Pre-compiled, cached execution plan
Security	Grant EXECUTE without table access
Maintainability	Change logic in one place
Reduced network	Send "EXEC sp_name" vs. full query
Code reuse	Multiple apps use same procedure
Transaction control	Complex operations in one unit

Q2: What is the difference between a stored procedure and a function?

Answer:

Aspect	Stored Procedure	Function
Returns	0 or more result sets	Single value (scalar) or table
Call in SELECT	✗ Cannot	✓ Can be used in SELECT
Side effects	✓ Can INSERT/UPDATE/DELETE	✗ Should not (pure)
Output params	✓ Multiple outputs	✗ Only return value
Transaction	✓ Can manage transactions	⚠ Limited
Calling syntax	EXEC procedure	SELECT function()

```
-- Function can be used in SELECT
SELECT dbo.fn_calculate_tax(amount) FROM orders;

-- Procedure cannot
SELECT EXEC sp_calculate_tax(@amount); -- Error!
```

Q3: How do you pass OUTPUT parameters?

Answer:

```
-- Create procedure with OUTPUT parameter
CREATE PROCEDURE sp_get_customer_count
    @country VARCHAR(50),
    @total INT OUTPUT -- OUTPUT keyword
AS
BEGIN
    SELECT @total = COUNT(*)
    FROM customers
    WHERE country = @country;
END;

-- Execute and capture output
DECLARE @result INT;
EXEC sp_get_customer_count @country = 'USA', @total = @result OUTPUT;
PRINT 'Count: ' + CAST(@result AS VARCHAR);
```

Q4: How does TRY-CATCH work in stored procedures?

Answer:

```
CREATE PROCEDURE sp_safe_insert
    @id INT, @name VARCHAR(100)
AS
BEGIN
    BEGIN TRY
        INSERT INTO customers (id, name) VALUES (@id, @name);
        PRINT 'Insert successful';
    END TRY
    BEGIN CATCH
        -- Capture error details
        DECLARE @error_msg NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @error_num INT = ERROR_NUMBER();
        DECLARE @error_line INT = ERROR_LINE();

        -- Log or handle error
        PRINT 'Error ' + CAST(@error_num AS VARCHAR) + ': ' + @error_msg;

        -- Re-throw or handle gracefully
        THROW; -- Or return error code
    END CATCH
END;
```

Q5: How do you prevent SQL injection in stored procedures?

Answer:

```
-- ✗ UNSAFE: Dynamic SQL without parameterization
CREATE PROCEDURE sp_search_unsafe
    @search VARCHAR(100)
AS
BEGIN
    EXEC('SELECT * FROM products WHERE name = ''' + @search + ''');
    -- User could pass: ';' DROP TABLE products; --
END;

-- ✓ SAFE: Use sp_executesql with parameters
CREATE PROCEDURE sp_search_safe
    @search VARCHAR(100)
AS
BEGIN
    EXEC sp_executesql
        N'SELECT * FROM products WHERE name = @name',
        N'@name VARCHAR(100)',
        @name = @search;
END;

-- ✓ SAFEST: No dynamic SQL
CREATE PROCEDURE sp_search_safest
    @search VARCHAR(100)
AS
BEGIN
    SELECT * FROM products WHERE name = @search;
END;
```

Q6: What is parameter sniffing and how do you handle it?

Answer: Parameter sniffing is when SQL Server creates an execution plan based on the first parameter value, which may not be optimal for other values.

```
-- Problem: Plan optimized for 'USA' (1M rows) used for 'Vatican' (10 rows)
EXEC sp_get_customers @country = 'USA';      -- Plan cached
EXEC sp_get_customers @country = 'Vatican'; -- Bad plan used!

-- Solutions:
-- 1. OPTION (RECOMPILE) - new plan each time
CREATE PROCEDURE sp_get_customers @country VARCHAR(50)
AS
BEGIN
    SELECT * FROM customers WHERE country = @country
    OPTION (RECOMPILE);
END;

-- 2. Local variable (breaks sniffing)
CREATE PROCEDURE sp_get_customers @country VARCHAR(50)
AS
BEGIN
    DECLARE @local VARCHAR(50) = @country;
    SELECT * FROM customers WHERE country = @local;
END;

-- 3. OPTIMIZE FOR hint
SELECT * FROM customers WHERE country = @country
OPTION (OPTIMIZE FOR (@country = 'USA'));
```

Q7: How do you handle transactions in stored procedures?

Answer:

```

CREATE PROCEDURE sp_transfer_funds
    @from_account INT,
    @to_account INT,
    @amount DECIMAL(10,2)
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        -- Debit source account
        UPDATE accounts SET balance = balance - @amount
        WHERE account_id = @from_account;

        -- Credit destination account
        UPDATE accounts SET balance = balance + @amount
        WHERE account_id = @to_account;

        COMMIT TRANSACTION;
        PRINT 'Transfer successful';
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;

        THROW; -- Re-throw the error
    END CATCH
END;

```

Q8: What's the difference between RETURN and OUTPUT parameters?

Answer:

Aspect	RETURN	OUTPUT Parameter
Value count	One integer only	Multiple values
Data types	INT only	Any data type
Purpose	Status/error code	Return data

Aspect	RETURN	OUTPUT Parameter
Capture	EXEC @status = sp	@param OUTPUT

```
-- RETURN: Status code
CREATE PROCEDURE sp_validate @id INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM users WHERE id = @id)
        RETURN -1; -- Not found
    RETURN 0; -- Success
END;

DECLARE @status INT;
EXEC @status = sp_validate @id = 1;
IF @status = -1 PRINT 'Not found';
```

Q9: How do you debug stored procedures?

Answer:

```
-- 1. PRINT statements
PRINT 'Reached step 1';
PRINT 'Variable value: ' + CAST(@var AS VARCHAR);

-- 2. SELECT intermediate results
SELECT @variable AS debug_value;

-- 3. Use SET NOCOUNT ON (cleaner output)
SET NOCOUNT ON; -- Suppresses "X rows affected" messages

-- 4. Log to a debug table
INSERT INTO debug_log (procedure_name, step, message, timestamp)
VALUES ('sp_my_proc', 1, 'Started processing', GETDATE());

-- 5. SQL Server Management Studio debugger
-- Right-click procedure → Debug
```

Q10: Stored Procedure vs Ad-hoc Query - when to use which?

Answer:

Use Stored Procedure When	Use Ad-hoc Query When
Logic reused across apps	One-time analysis
Security control needed	Developer/DBA exploration
Complex multi-step process	Simple SELECT
Performance critical	Development/testing
Audit/logging required	Quick data check
Transaction management	Reporting tools that generate SQL

```
-- Stored procedure: Production, security, reuse
EXEC sp_generate_monthly_report @year = 2024, @month = 1;

-- Ad-hoc: Quick investigation
SELECT TOP 10 * FROM orders WHERE status = 'Error';
```

Note 30: SQL Indexes

Overview

Indexes are special database structures that dramatically improve query performance by allowing SQL to find data quickly without scanning entire tables. Understanding indexes is crucial for database optimization.



Theory

What is an Index?

INDEX CONCEPT: BOOK ANALOGY

WITHOUT INDEX (Full Table Scan):

```
| Find "customer_id = 500" in 1 million rows
| Row 1: Not it... Row 2: Not it... Row 3: Not it...
| ... check ALL 1 million rows ...
| Time: O(n) = 1,000,000 row reads
```

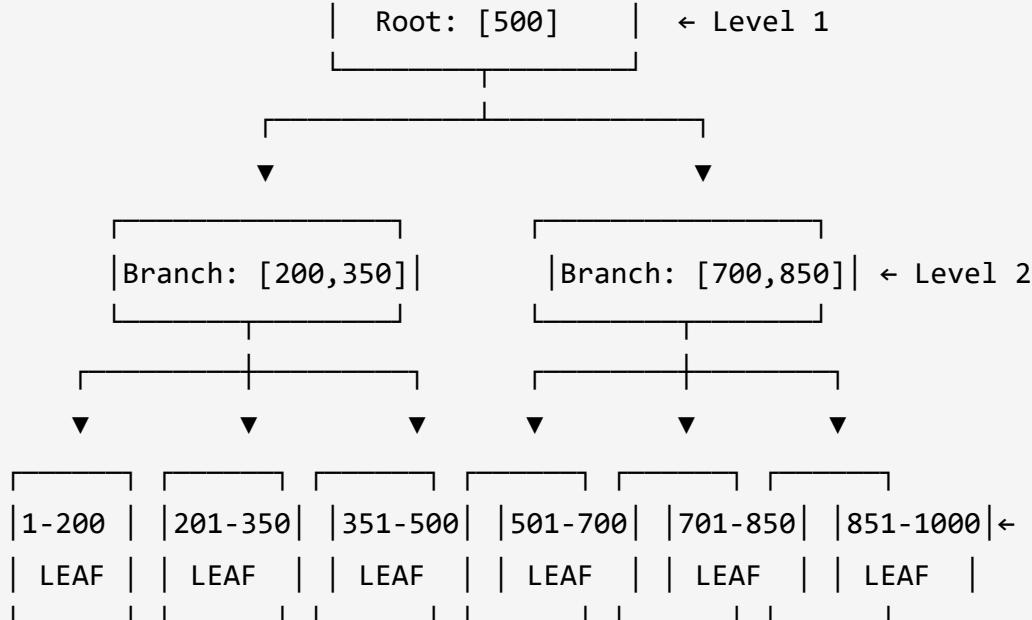
WITH INDEX (B-Tree Lookup):

```
| Find "customer_id = 500" using B-Tree
| Root: 500 > 250? → Go right
| Branch: 500 < 750? → Go left
| Leaf: Found! Here's the row pointer
| Time: O(log n) ≈ 20 row reads
```

Speed improvement: $1,000,000 / 20 = 50,000x$ faster!

B-Tree Index Structure

B-TREE INDEX VISUALIZATION



Finding ID = 275:

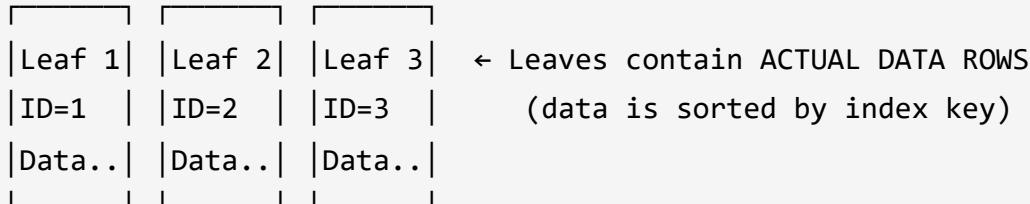
1. Root: 275 < 500 → Go LEFT
2. Branch: 200 < 275 < 350 → Middle pointer
3. Leaf: Scan 201-350, find 275 → Return row/pointer

Clustered vs Non-Clustered

CLUSTERED vs NON-CLUSTERED INDEX

CLUSTERED INDEX (1 per table):

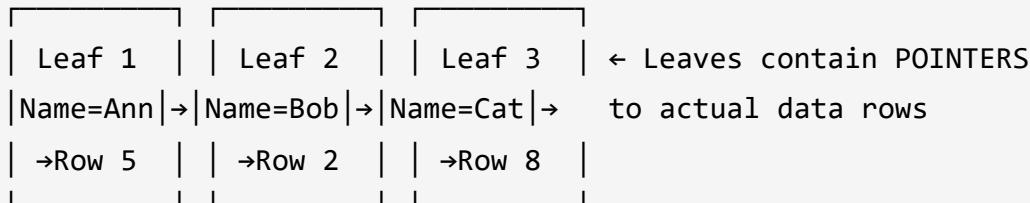
B-Tree



The index IS the table. Data physically sorted.

NON-CLUSTERED INDEX (many per table):

B-Tree



↓

Actual Data Table
(separate structure)

Index is separate. Extra lookup step to get full row.

Index Type Summary

Type	Per Table	Data	Read	Write	Best For
Clustered	1 only	Contains actual rows	Fastest	Slower	Primary keys, ranges
Non-Clustered	Unlimited	Contains pointers	Fast	Faster	WHERE, JOINs
Covering	Unlimited	Index + included cols	Fastest (no lookup)	Slower	Specific queries
Columnstore	Limited	Column-wise storage	Aggregations	Slow	Analytics, OLAP

Index Categories

Category 1: By Organization

- **Clustered Index:** Physically sorts data
- **Non-Clustered Index:** Separate structure with pointers

Category 2: By Storage

- **Row Store Index:** Traditional row-by-row storage
- **Column Store Index:** Column-by-column storage (for analytics)

Clustered Index

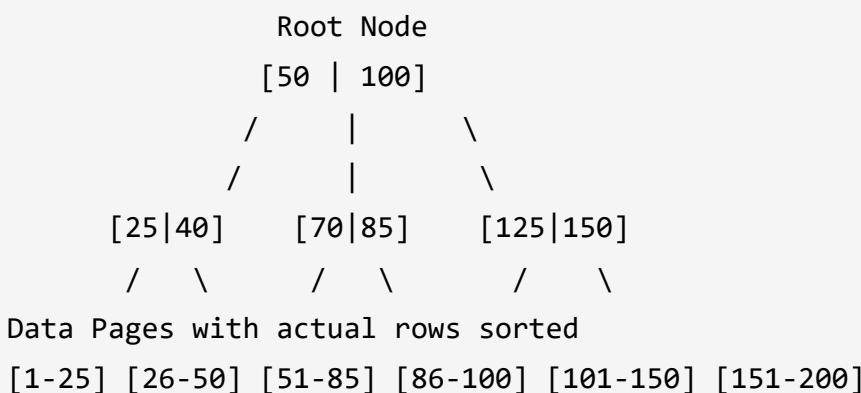
Definition

A **clustered index** physically sorts and stores the table data in the order of the index key.

Key Characteristics

- **ONE per table** (data can only be sorted one way)
- **IS the table** - reorganizes actual data
- **Fastest for reads** - data is already sorted
- **Slower for writes** - must maintain sort order
- **Primary keys** automatically get clustered index

B-Tree Structure (Clustered)



Key Point: Leaf nodes contain **actual data rows**

Non-Clustered Index

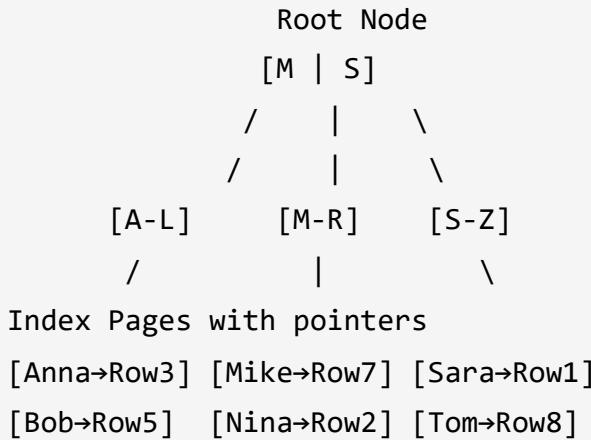
Definition

A **non-clustered index** creates a separate structure containing the index key and pointers to the actual data.

Key Characteristics

- **Multiple per table** (no limit)
- **Separate structure** - doesn't reorganize data
- **Slightly slower reads** - extra lookup step
- **Faster writes** - less reorganization needed

B-Tree Structure (Non-Clustered)



Key Point: Leaf nodes contain **pointers to data**, not actual rows

Clustered vs Non-Clustered Comparison

Aspect	Clustered	Non-Clustered
Per Table	Maximum 1	Multiple allowed
Data Storage	Stores actual data	Stores pointers
Read Performance	Faster	Slightly slower
Write Performance	Slower	Faster
Storage	Less (no extra layer)	More (extra structure)
Best For	Range queries, Primary Keys	Search conditions, Joins

Index Syntax

Create Clustered Index

```
CREATE CLUSTERED INDEX index_name  
ON schema.table_name (column_name);
```

Create Non-Clustered Index

```
CREATE NONCLUSTERED INDEX index_name  
ON schema.table_name (column_name);  
  
-- Or (NONCLUSTERED is default)  
CREATE INDEX index_name  
ON schema.table_name (column_name);
```

Drop Index

```
DROP INDEX index_name ON schema.table_name;
```

Index Examples

Create Table Copy for Testing

```
-- Create test table without indexes  
SELECT * INTO sales.tb_customers  
FROM sales.customers;
```

Create Clustered Index

```
CREATE CLUSTERED INDEX idx_tb_customers_customerid  
ON sales.tb_customers (customer_id);
```

Create Non-Clustered Index

```
CREATE NONCLUSTERED INDEX idx_tb_customers_lastname  
ON sales.tb_customers (last_name);
```

```
-- Using default (non-clustered)  
CREATE INDEX idx_tb_customers_firstname  
ON sales.tb_customers (first_name);
```

Naming Convention

idx_[table_name]_[column_name]

Examples:

- idx_customers_customerid
- idx_orders_orderdate
- idx_products_category

Composite Index (Multiple Columns)

Definition

An index built on **multiple columns** together.

Syntax

```
CREATE INDEX idx_customers_country_score  
ON sales.tb_customers (country, score);
```

Order Matters!

-- Index created on (country, score)

-- ✓ Uses index (starts with 'country')

```
SELECT * FROM customers WHERE country = 'USA';
```

-- ✓ Uses index (both columns, correct order)

```
SELECT * FROM customers WHERE country = 'USA' AND score > 500;
```

-- X Does NOT use index (skips 'country')

```
SELECT * FROM customers WHERE score > 500;
```

-- X Does NOT use index (wrong order in query matters less, but skipping columns ma

```
SELECT * FROM customers WHERE score > 500 AND country = 'USA';
```

Leftmost Prefix Rule

Index on columns: (A, B, C, D)

Query uses: Uses Index?

A ✓ Yes

A, B ✓ Yes

A, B, C ✓ Yes

A, B, C, D ✓ Yes

B X No (skipped A)

A, C X No (skipped B)

B, C, D X No (skipped A)

Row Store vs Column Store Index

Row Store (Traditional)

Data stored row by row:

ID	Name	Status
1	Anna	Active
2	Bob	Inactive
3	Carol	Active
4	David	Active

→ Data Page 1

→ Data Page 2

Column Store

Data stored column by column:

ID Column	→ Data Page 1
1, 2, 3, 4	

Name Column	→ Data Page 2
Anna, Bob...	

Status Column	→ Data Page 3
1, 2, 1, 1	(compressed with dictionary)

Column Store Process

1. **Row Groups:** Divide data into ~1 million row groups
2. **Column Segments:** Split each row group by column
3. **Compression:** Create dictionary, replace values with IDs
4. **Storage:** Store in LOB (Large Object) pages

Column Store Compression Example

Before Compression

Status Column: Active, Inactive, Active, Active, Inactive, Active...
(2 million rows of strings = lots of storage)

After Compression

Dictionary:

1 = Active

2 = Inactive

Data Stream: 1, 2, 1, 1, 2, 1...

(2 million small integers = minimal storage)

Why Column Store is Faster for Analytics

Query: Count Active Customers

```
SELECT COUNT(*) FROM customers WHERE status = 'Active';
```

Row Store Execution

1. Read Data Page 1 → Get all columns (ID, Name, Status)
2. Read Data Page 2 → Get all columns (ID, Name, Status)
3. ...read all pages...
4. Filter for Status = 'Active'
5. Count

Problem: Reading unnecessary columns (ID, Name)

Column Store Execution

1. Read ONLY Status column data page
2. Filter for value = 1 (Active in dictionary)
3. Count

Advantage: Read only the column needed!

Row Store vs Column Store Comparison

Aspect	Row Store	Column Store
Storage	Row by row	Column by column
Compression	Limited	High (dictionary)
Read Performance	Good for full rows	Excellent for aggregations
Write Performance	Fast	Slower
I/O Efficiency	Higher (reads all columns)	Lower (reads needed columns)
Best For	OLTP (transactions)	OLAP (analytics)
Use Cases	Banking, E-commerce	Data Warehouse, BI

Column Store Index Syntax

Clustered Column Store

```
-- Converts entire table to column store  
CREATE CLUSTERED COLUMNSTORE INDEX idx_name  
ON schema.table_name;  
  
-- Note: No column specification (includes all columns)
```

Non-Clustered Column Store

```
-- Creates additional column store structure  
CREATE NONCLUSTERED COLUMNSTORE INDEX idx_name  
ON schema.table_name (column1, column2);  
  
-- Can specify which columns to include
```

When to Use Each Index Type

Use Clustered Index When:

- Column has unique values
- Column rarely changes (not frequently updated)
- Range queries are common (BETWEEN , < , >)
- Primary key columns (automatic)

Use Non-Clustered Index When:

- Column used in WHERE conditions frequently
- Column used in JOINs (not primary key)
- Need exact match searches (=)
- Need multiple indexes on same table

Use Row Store When:

- OLTP systems (transactions)
- Frequent INSERT/UPDATE/DELETE
- Full row access is common
- Mixed read/write workloads

Use Column Store When:

- OLAP systems (analytics)
 - Large data sets (millions+ rows)
 - Complex aggregation queries
 - Read-heavy workloads
 - Data warehouse / BI scenarios
-

Index Best Practices

Do's ✓

1. **Index primary keys** (usually automatic)
2. **Index foreign keys** used in JOINs
3. **Index columns in WHERE clauses** used frequently
4. **Use composite indexes** for multi-column filters
5. **Follow leftmost prefix rule** for composite indexes
6. **Monitor index usage** and remove unused indexes

Don'ts X

1. **Don't over-index** - slows down writes
 2. **Don't index frequently updated columns** with clustered
 3. **Don't ignore index maintenance** - rebuild fragmented indexes
 4. **Don't use composite index in wrong order**
 5. **Don't create duplicate indexes**
-

Practice Questions

Q1: Create Clustered Index on Primary Key

```
CREATE CLUSTERED INDEX idx_orders_orderid  
ON sales.orders (order_id);
```

Q2: Create Non-Clustered Index for Search

```
CREATE INDEX idx_customers_country  
ON sales.customers (country);
```

Q3: Create Composite Index

```
CREATE INDEX idx_orders_date_status  
ON sales.orders (order_date, order_status);  
  
-- Query that uses this index:  
SELECT * FROM sales.orders  
WHERE order_date >= '2024-01-01'  
AND order_status = 'Completed';
```

Q4: Create Column Store for Analytics

```
CREATE CLUSTERED COLUMNSTORE INDEX idx_sales_columnstore  
ON analytics.sales_fact;
```

Key Takeaways

1. **Indexes speed up reads** at the cost of slower writes
2. **Clustered index = 1 per table**, physically sorts data
3. **Non-clustered = unlimited**, uses pointers to data

4. **Composite index column order matters** - follow leftmost prefix rule
 5. **Row store for transactions**, column store for analytics
 6. **Column store compresses data** dramatically
 7. **Primary keys get clustered index** automatically in SQL Server
-

Summary Table

Index Type	Count	Data Storage	Best For
Clustered	1	Physical sort	Primary keys, ranges
Non-Clustered	Many	Pointers	Searches, joins
Row Store	-	Row by row	OLTP
Column Store	-	Column by column	OLAP, Analytics

Exercises

1. Create a test table and add a clustered index
 2. Create multiple non-clustered indexes for different queries
 3. Create a composite index and test which queries use it
 4. Compare query performance with and without indexes
 5. Create a column store index on a large analytics table
-

Interview Questions

Q1: What is the difference between clustered and non-clustered index?

Answer:

Aspect	Clustered	Non-Clustered
Count	1 per table	Unlimited
Leaf nodes	Actual data rows	Pointers to rows
Data order	Physically sorted	Logical order only
Lookup	Direct (data in index)	Extra step (pointer lookup)
Insert cost	Higher (maintain sort)	Lower

```
-- Clustered: Data rows stored in order
CREATE CLUSTERED INDEX idx_id ON customers(id);
-- Now table is physically sorted by id

-- Non-clustered: Separate structure with pointers
CREATE NONCLUSTERED INDEX idx_name ON customers(name);
-- Table order unchanged, index has name→pointer mapping
```

Q2: What is a covering index and why is it important?

Answer: A covering index includes all columns needed by a query, eliminating the need to access the base table:

```
-- Query needs: name, email WHERE status = 'active'
SELECT name, email FROM users WHERE status = 'active';

-- Regular index (needs table lookup)
CREATE INDEX idx_status ON users(status);
-- Index finds rows → then goes to table for name, email

-- Covering index (no table lookup needed!)
CREATE INDEX idx_status_covering ON users(status) INCLUDE (name, email);
-- All data is IN the index → faster!
```

Key benefit: Avoids "key lookup" or "bookmark lookup" - major performance boost.

Q3: What is the leftmost prefix rule?

Answer: For composite indexes, the index can only be used if the query uses columns starting from the left:

```
-- Index on (country, city, zipcode)
CREATE INDEX idx_location ON customers(country, city, zipcode);

-- ✓ Uses index (starts with country)
WHERE country = 'USA'
WHERE country = 'USA' AND city = 'NYC'
WHERE country = 'USA' AND city = 'NYC' AND zipcode = '10001'

-- ✗ Cannot use index (skips country)
WHERE city = 'NYC'
WHERE zipcode = '10001'
WHERE city = 'NYC' AND zipcode = '10001'
```

Q4: When would you NOT want to create an index?

Answer:

Don't Index When	Reason
Small tables	Full scan is faster than index lookup
High write tables	Indexes slow down INSERT/UPDATE/DELETE
Low selectivity columns	Gender (M/F) - index doesn't help much
Rarely queried columns	Waste of storage
Already covered	Duplicate indexes waste resources

Q5: How do you identify missing indexes?

Answer:

```
-- SQL Server: DMV for missing indexes
```

```
SELECT
```

```
    d.statement AS table_name,  
    d.equality_columns,  
    d.inequality_columns,  
    d.included_columns,  
    s.avg_user_impact AS improvement_pct,  
    s.user_seeks + s.user_scans AS potential_uses  
FROM sys.dm_db_missing_index_details d  
JOIN sys.dm_db_missing_index_groups g ON d.index_handle = g.index_handle  
JOIN sys.dm_db_missing_index_group_stats s ON g.index_group_handle = s.group_handle  
ORDER BY s.avg_user_impact DESC;
```

```
-- Also check: Execution plan shows "Missing Index" warning
```



Q6: What is index fragmentation and how do you fix it?

Answer:

Fragmentation: When index pages become out of order due to
INSERT/UPDATE/DELETE operations.

```
-- Check fragmentation level
SELECT
    OBJECT_NAME(ips.object_id) AS table_name,
    i.name AS index_name,
    ips.avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'LIMITED') ips
JOIN sys.indexes i ON ips.object_id = i.object_id AND ips.index_id = i.index_id
WHERE ips.avg_fragmentation_in_percent > 10;

-- Fix fragmentation:
-- < 30%: REORGANIZE (online, faster)
ALTER INDEX idx_name ON table_name REORGANIZE;

-- > 30%: REBUILD (offline by default, more thorough)
ALTER INDEX idx_name ON table_name REBUILD;

-- Rebuild all indexes on a table
ALTER INDEX ALL ON table_name REBUILD;
```

Q7: What is a filtered index?

Answer: An index with a WHERE clause that only indexes a subset of rows:

```
-- Index ONLY active customers (not the entire table)
CREATE INDEX idx_active_customers
ON customers(last_name, first_name)
WHERE status = 'active';

-- Benefits:
-- ✓ Smaller index size
-- ✓ Faster maintenance
-- ✓ Better cache efficiency
-- ✓ Perfect for queries that always filter by same condition

-- Query that uses it:
SELECT * FROM customers WHERE status = 'active' AND last_name = 'Smith';
```

Q8: Explain index seek vs index scan

Answer:

Operation	Description	Performance	When Used
Index Seek	Jump directly to matching rows	✓ Excellent	Specific value lookup
Index Scan	Read entire index	⚠️ OK-Slow	Need all/most rows
Table Scan	Read entire table	✗ Slowest	No useful index

```
-- Index SEEK (good): Finds id=100 directly
```

```
SELECT * FROM orders WHERE id = 100;
```

```
-- Index SCAN (may be needed): Reads all index entries
```

```
SELECT * FROM orders WHERE description LIKE '%laptop%';
```

Q9: What is a columnstore index and when would you use it?

Answer:

Columnstore: Stores data column-by-column instead of row-by-row, with high compression.

Use Columnstore	Use Rowstore
Analytics/OLAP	Transactions/OLTP
Large aggregations	Point lookups
Read-heavy	Write-heavy
Millions+ rows	Smaller tables
Data warehouse	Line-of-business apps

```
-- Create clustered columnstore (entire table becomes columnstore)
CREATE CLUSTERED COLUMNSTORE INDEX cci_sales ON sales_fact;

-- Query that benefits (aggregate across millions of rows)
SELECT
    YEAR(order_date) AS year,
    SUM(amount) AS total_sales
FROM sales_fact
GROUP BY YEAR(order_date);
```

Q10: How do you analyze if an index is being used?

Answer:

```
-- Method 1: Check execution plan (SSMS)
SET SHOWPLAN_TEXT ON;
GO
SELECT * FROM orders WHERE customer_id = 100;
GO
SET SHOWPLAN_TEXT OFF;

-- Method 2: Index usage statistics
SELECT
    OBJECT_NAME(s.object_id) AS table_name,
    i.name AS index_name,
    s.user_seeks,
    s.user_scans,
    s.user_lookups,
    s.user_updates -- High updates + low seeks = candidate for removal
FROM sys.dm_db_index_usage_stats s
JOIN sys.indexes i ON s.object_id = i.object_id AND s.index_id = i.index_id
WHERE database_id = DB_ID()
ORDER BY s.user_seeks DESC;

-- Unused index (user_seeks = 0, user_updates > 0) = consider dropping
```