SQL Notes - Comprehensive Guide

> **Note**: This is a cleaned and formatted version of SQL learning notes covering fundamentals to advanced topics.

Table of Contents

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
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:

```
Unstructured Data → Semi-Structured Data → Structured Data
(Videos, Images)    (JSON, XML, Logs)      (Relational Tables)
      ↓                   ↓                      ↓
Hard to query       Some querying          Full SQL power
```

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

📊 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:	Choose NoSQL when:
✓ Data is structured	✓ Data is unstructured/semi-structured
✓ Complex queries needed	✓ Simple key-based lookups
✓ ACID compliance required	✓ Massive scale needed
✓ Data relationships exist	✓ Schema changes frequently
✓ Reporting/analytics focus	✓ High write throughput needed

Database Hierarchy

```
Server (Physical/Virtual Machine)
│
├── System Databases (master, tempdb, model, msdb)
│
└── User Databases
    │
    └── Database (e.g., SalesDB)
        │
        ├── Schema (e.g., sales, hr, finance)
        │   │
        │   └── Tables
        │       │
        │       ├── Columns (Fields/Attributes)
        │       ├── Rows (Records/Tuples)
        │       ├── Constraints (PK, FK, CHECK)
        │       └── Indexes
        │
        ├── Views (Virtual Tables)
        ├── Stored Procedures
        ├── Functions
        └── Triggers
```

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

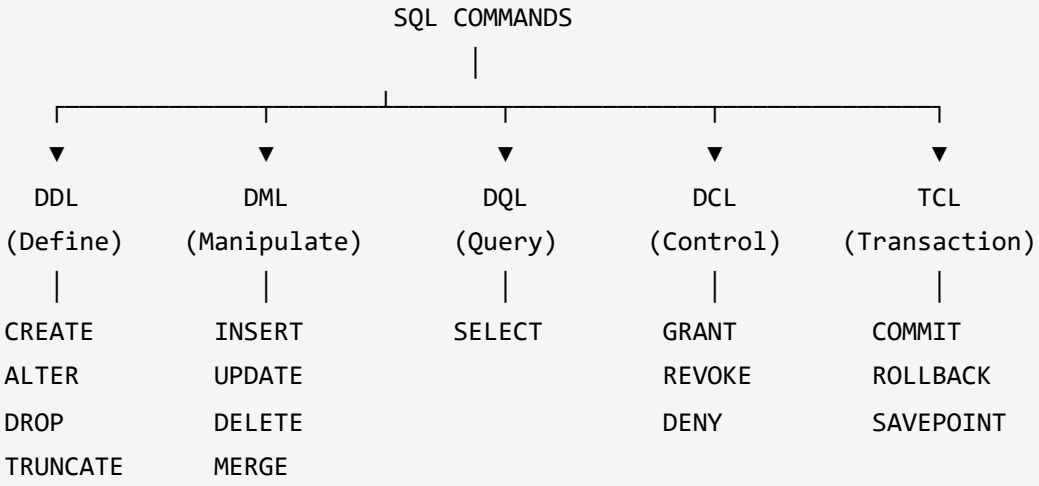
Data Type	Description	Format	Example
DATE	Date only	YYYY-MM-DD	'2024-01-15'
TIME	Time only	HH:MM:SS	'14:30:00'
DATETIME	Date and time	YYYY-MM-DD HH:MM:SS	'2024-01-15 14:30:00'
DATETIME2	High precision	100 nanoseconds	Audit timestamps

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

Aspect	Primary Key	Unique Key
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

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

Component	Purpose	Description
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

🔧 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

```sql
-- 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

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

Multi-Line Comment

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

🎯 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

Edition	Use Case
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

🔑 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

Issue	Solution
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)

Query Components (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

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

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

Using Semicolons

```
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

```sql
-- Add a static column to table data
SELECT
    id,
    first_name,
    'New Customer' AS customer_type
FROM customers;
```

Naming Columns (Aliases)

```sql
-- Use AS to give columns custom names
SELECT
    123 AS static_number,
    'Hello' AS greeting;
```

🎯 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
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).

🔑 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 SELECT *
Use meaningful aliases	Makes results clearer
Format queries on multiple lines	Easier to read and maintain
Avoid SELECT * in production	Returns unnecessary data
Always specify FROM	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.

```
SELECT id, first_name
FROM customers;
```

Exercise 2:

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

```
SELECT order_id, sales
FROM orders;
```

Exercise 3:

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

```
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

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

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

⚠️ Important Rules

String Values Require Single Quotes

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

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

Numbers Don't Need Quotes

```sql
-- ✅ 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)

```sql
-- These may give different results:
WHERE country = 'germany'
WHERE country = 'Germany'
```

🎯 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

```sql
-- 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.

```sql
-- ❌ 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:

```sql
-- 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)

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

-- Pattern match
WHERE name LIKE 'John%'   -- John, Johnson, Johnny...
WHERE name LIKE 'J_hn'    -- John, Jahn, Jehn...
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`
<> 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-NULL	`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!
```

🔑 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.

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

Exercise 2:

Get customer ID and name for customers from the UK.

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

Exercise 3:

Get all customers who don't have zero score.

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

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.

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...

Data Type	ASC Order	DESC Order
Dates	Oldest → Newest	Newest → Oldest
NULL	First (varies by DB)	Last (varies by DB)

📝 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

```sql
-- Sort by country A-Z
SELECT *
FROM customers
ORDER BY country ASC;
-- Result: Germany, Germany, UK, USA, USA
```

Example 4: Sort with WHERE

```sql
-- 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:

```sql
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)

🎯 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:

```sql
-- 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):

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

SQL Server workaround:

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

Q4: 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)
```

🔑 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.

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

Exercise 2:

Get orders sorted by sales (highest first).

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

Exercise 3:

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

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

Exercise 4:

Get the lowest scoring customer.

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

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	`SELECT score FROM customers`	Each row is separate
Group Level	`SELECT country, SUM(score) GROUP BY country`	Rows combined by country
Table Level	`SELECT SUM(score) FROM customers`	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**

```sql
-- 🧠 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

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

Aggregate Functions

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

💻 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

⚠️ Important Rules

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

```sql
-- ❌ 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

```sql
-- 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"
```

🎯 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

```sql
-- 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: 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!

🔑 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.

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

Exercise 4:

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

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

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

Aspect	WHERE	HAVING
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)

```sql
-- 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)
```

 Basic Syntax

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

💻 Practical Examples

Example 1: Filter by Total Score

```sql
-- 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

```sql
-- 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

```sql
-- 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

Step	Clause	Description
4	HAVING	Filter groups (after aggregation)
5	SELECT	Select columns
6	ORDER BY	Sort results

🔧 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

🎯 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

Database	Alias in HAVING?
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: What's the performance difference between WHERE and HAVING?

Answer:

Filter	Performance	Why
WHERE	✅ Better	Filters BEFORE aggregation = fewer rows to process
HAVING	❌ 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;
```

🔑 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.

```sql
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.

```sql
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;
```

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)
```

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 */
```

💡 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.

🎯 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)

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:

```sql
-- ❌ 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;
```

🔑 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.

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

Exercise 2:

Find all unique country and city combinations.

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

Exercise 3:

Count how many unique countries we have.

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

Exercise 4:

Get unique order dates.

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

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

Step	Clause	Action
2	WHERE	Filter rows
3	GROUP BY	Aggregate
4	HAVING	Filter groups
5	SELECT	Choose columns
6	ORDER BY	Sort results
7	TOP/LIMIT	Limit rows (LAST!)

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

🎯 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

```
-- ❌ 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
-- 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;
```

🔑 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;
```

🔧 Database-Specific Syntax

SQL Server

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

MySQL

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

PostgreSQL

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

Oracle

```sql
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

The Fundamental Difference

CODING ORDER (What you type):

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

EXECUTION ORDER (What SQL does):

```
1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT ← aliases born here
6. ORDER BY
7. TOP/LIMIT
```

SELECT is written FIRST but executed FIFTH!

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)

```sql
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)

```sql
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

Step	Clause	What It Does
4	HAVING	Filters groups
5	SELECT	Chooses columns
6	ORDER BY	Sorts the result
7	TOP/LIMIT	Limits rows returned

💻 Visual Example

Query:

```sql
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

```sql
-- ❌ 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

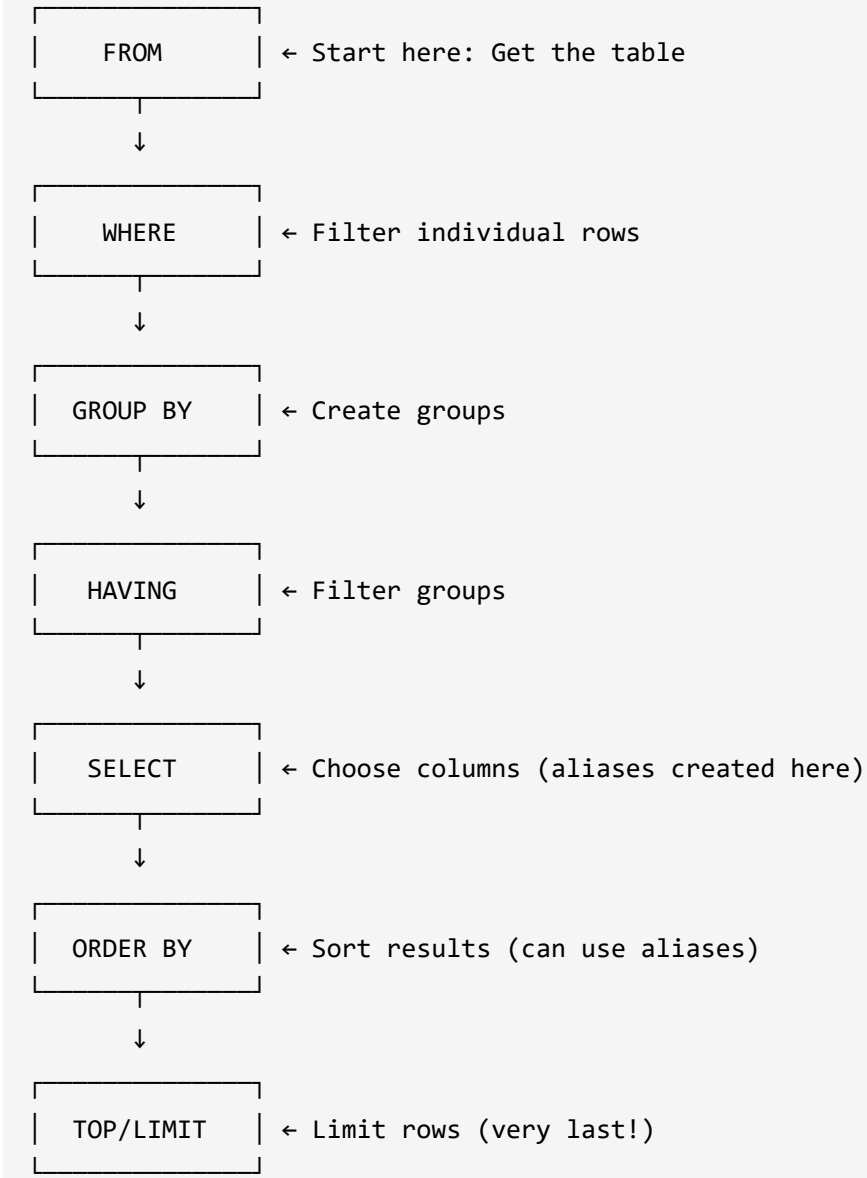
```sql
-- ❌ This FAILS
SELECT department, COUNT(*) AS emp_count
FROM employees
WHERE COUNT(*) > 5     -- Error! Can't use aggregate
GROUP BY department;

-- ✅ 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	❌ No	Executes before SELECT
WHERE	❌ No	Executes before SELECT
GROUP BY	❌ No	Executes before SELECT
HAVING	❌ No	Executes before SELECT

Clause	Can Use Alias?	Reason
SELECT	✅ Creating	This is where aliases are defined
ORDER BY	✅ Yes	Executes after SELECT

Where Aggregates Work

Clause	Can Use Aggregate?
WHERE	❌ No (use HAVING)
HAVING	✅ Yes
SELECT	✅ Yes
ORDER BY	✅ Yes

🎯 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

Database	GROUP BY Alias?	Reason
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 runs     |     GROUP BY     |     HAVING runs
   (rows exist)    |   (grouping)     |   (groups exist)
_____________________________________________________________
                   |                  |
                   |     SUM()        |
                   |     created      |
                   |     here         |
                   ▼                  ▼
```

- **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.

🔑 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

```sql
-- 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

```sql
-- 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;
```

💡 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**ELECT
- **O**RDER BY
- **T**OP

This document is a work in progress. More notes (11-28) covering String Functions, Date Functions, NULL Functions, CASE Statements, Window Functions, Subqueries, CTEs, Views, and Temporary Tables will be added.

Note 11: String Functions in SQL

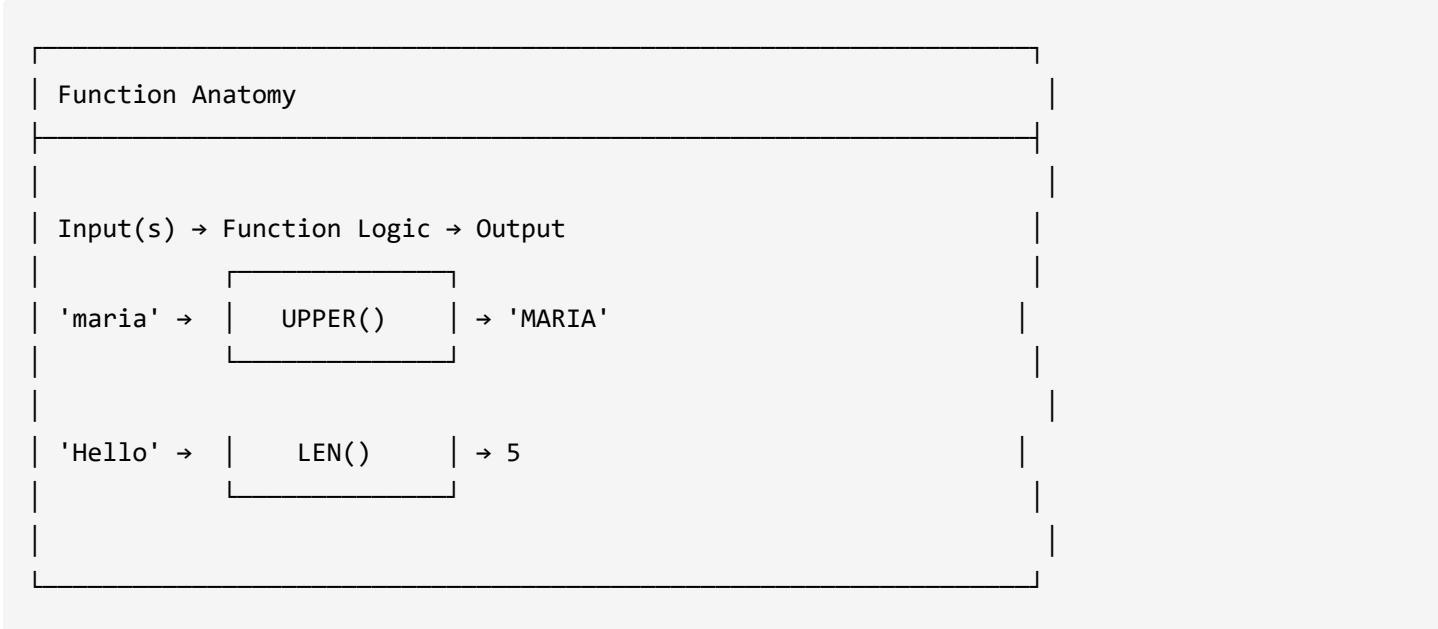
Overview

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

Theory

What are Functions?

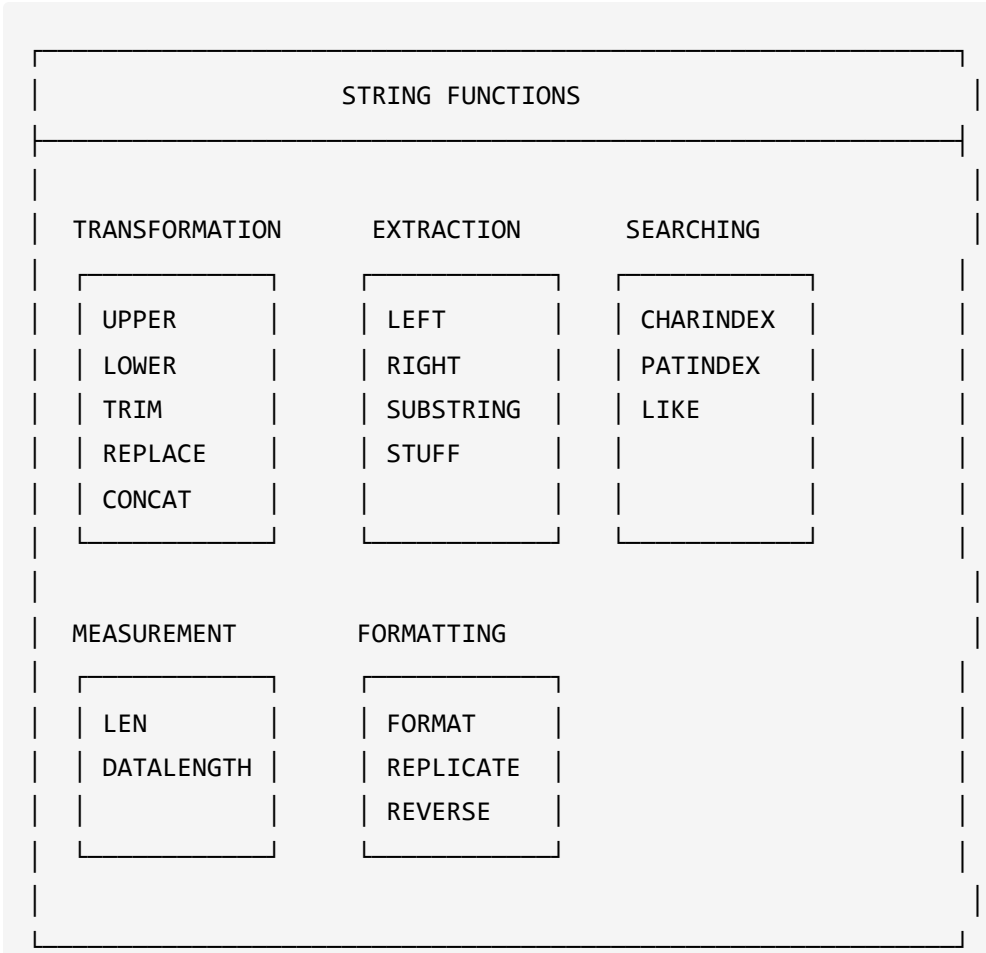
A function is a reusable code block that transforms input into output:



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

```
-- 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"
```

2. UPPER - Convert to Uppercase

```
UPPER(string_value)

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

3. LOWER - Convert to Lowercase

```
LOWER(string_value)

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

4. TRIM - Remove Spaces

```
TRIM(string_value)

SELECT TRIM(first_name) AS clean_name FROM customers;
-- Input: "  John  " → Output: "John"

-- Find values with spaces
SELECT * FROM customers WHERE first_name <> TRIM(first_name);
```

5. REPLACE - Replace Characters

```
REPLACE(value, old_string, new_string)

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

String Calculation Functions

LEN - Calculate String Length

```
LEN(string_value)

SELECT first_name, LEN(first_name) AS name_length FROM customers;
-- "Maria" → 5
```

String Extraction Functions

1. LEFT - Extract from Start

```
LEFT(string_value, number_of_characters)

SELECT LEFT(first_name, 2) AS first_two FROM customers;
-- "Maria" → "Ma"
```

2. RIGHT - Extract from End

```
RIGHT(string_value, number_of_characters)

SELECT RIGHT(first_name, 2) AS last_two FROM customers;
-- "Maria" → "ia"
```

3. SUBSTRING - Extract from Middle

```
SUBSTRING(string_value, start_position, length)

SELECT SUBSTRING('Maria', 3, 2) AS result;
-- Output: "ri" (starts at position 3, takes 2 characters)
```

Nesting Functions

```
-- 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
```

Interview Questions

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

Aspect	CONCAT()	+ Operator
NULL handling	Ignores NULL (treats as '')	NULL propagates (result = NULL)
Type safety	Auto-converts to string	Requires same types

Q2: What is the difference between LEN and DATALENGTH?

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

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

```
-- 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
```

Q4: What is the difference between TRIM, LTRIM, and RTRIM?

Function	Removes
LTRIM()	Leading spaces only (left)
RTRIM()	Trailing spaces only (right)

Function	Removes
TRIM()	Both leading and trailing

Q5: What is STUFF and when would you use it?

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

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

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'
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'

Note 12: Date and Time Functions in SQL

Overview

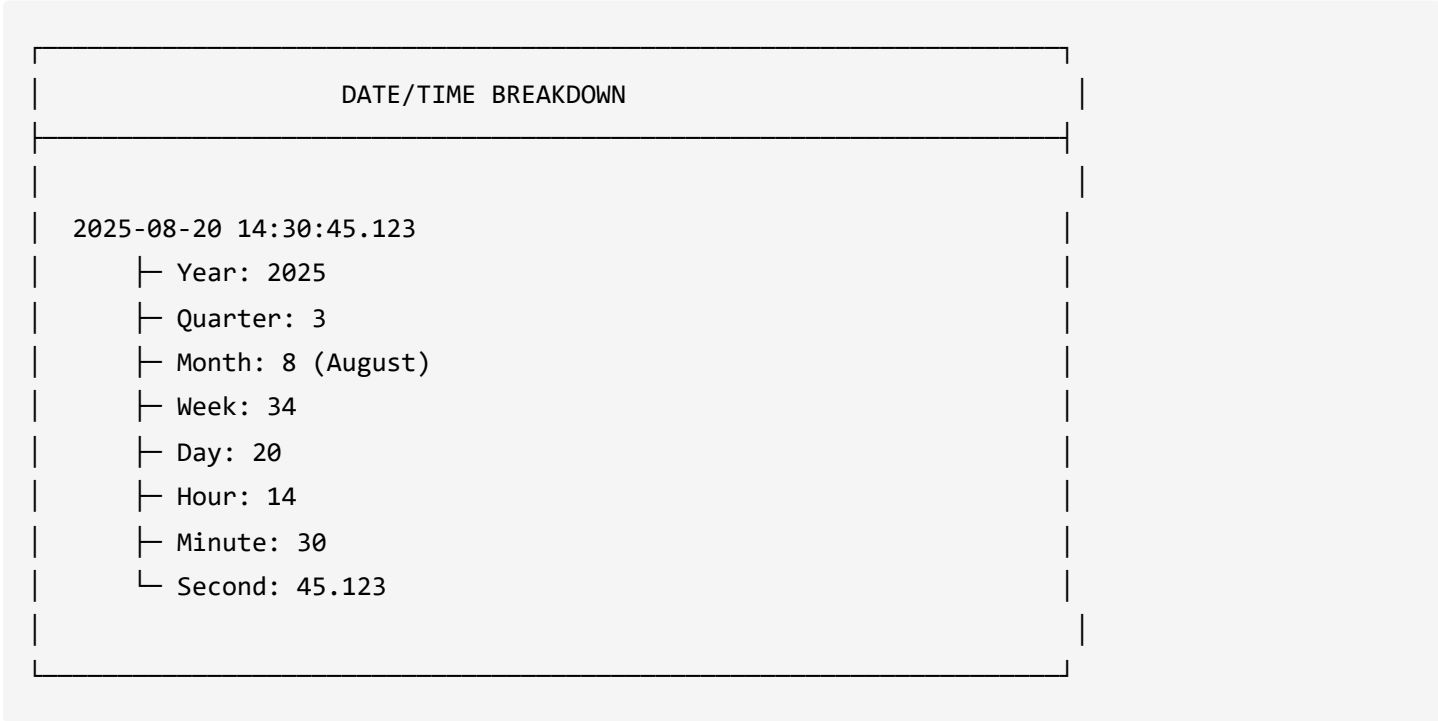
This note covers date and time functions in SQL - essential tools for extracting parts, formatting, calculating, and validating date/time values.

Theory

Date/Time Data Types

Type	Storage	Range	Precision	Example
DATE	3 bytes	0001-9999	Day	'2025-08-20'
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



Part Extraction Functions

1. YEAR, MONTH, DAY - Basic Extraction

```
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

```
DATEPART(part, date)

SELECT
    DATEPART(year, order_date) AS year,
    DATEPART(month, order_date) AS month,
    DATEPART(quarter, order_date) AS quarter,
    DATEPART(week, order_date) AS week,
    DATEPART(weekday, order_date) AS weekday
FROM orders;
```

3. DATENAME - Extract Part as Text

```
DATENAME(part, date)

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

```
DATETRUNC(part, date)

-- Truncate to month level
SELECT DATETRUNC(month, '2025-08-20 14:30:45');
-- Output: 2025-08-01 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

```
EOMONTH(date)

SELECT EOMONTH('2025-08-20'); -- Output: 2025-08-31
SELECT EOMONTH('2025-02-15'); -- Output: 2025-02-28
```

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

```
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

-- Calculate delivery date
SELECT order_date, DATEADD(day, 7, order_date) AS expected_delivery
FROM orders;
```

3. DATEDIFF - Difference Between Dates

```
DATEDIFF(part, start_date, end_date)

-- Days between dates
SELECT DATEDIFF(day, '2025-01-01', '2025-01-15'); -- Output: 14

-- Calculate age
SELECT DATEDIFF(year, birth_date, GETDATE()) AS age FROM employees;

-- Shipping duration
SELECT order_id, DATEDIFF(day, order_date, ship_date) AS days_to_ship
FROM orders;
```

Date Formatting

FORMAT - Custom Date Formatting

```
FORMAT(date, format_string)

SELECT FORMAT(order_date, 'dd/MM/yyyy'); -- "20/08/2025"
SELECT FORMAT(order_date, 'MMMM dd, yyyy'); -- "August 20, 2025"
SELECT FORMAT(order_date, 'dddd'); -- "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)
yyyy	Year (2025)

Interview Questions

Q1: What is the difference between DATETIME and DATETIME2?

Aspect	DATETIME	DATETIME2
Range	1753-9999	0001-9999
Precision	3.33 ms	100 ns
Storage	8 bytes fixed	6-8 bytes
Recommended	Legacy	✅ New development

Q2: How do you calculate age accurately?

```
-- DATEDIFF(year, ...) counts year boundaries, not actual years
-- ✅ 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;
```

Q3: How do you get the first day of the current month?

```
-- Method 1: DATETRUNC (SQL Server 2022+)
SELECT DATETRUNC(month, GETDATE());

-- Method 2: DATEFROMPARTS
SELECT DATEFROMPARTS(YEAR(GETDATE()), MONTH(GETDATE()), 1);
```

Q4: Why should you avoid using functions on columns in WHERE?

Functions prevent index usage (not SARGable):

```
-- ❌ Non-SARGable: Full table scan
SELECT * FROM orders WHERE YEAR(order_date) = 2025;

-- ✅ SARGable: Can use index
SELECT * FROM orders
WHERE order_date >= '2025-01-01' AND order_date < '2026-01-01';
```

Key Takeaways

Function	Purpose
YEAR/MONTH/DAY	Quick extraction

Function	Purpose
DATEPART	Any part as integer
DATENAME	Part as text
DATETRUNC	Reset to precision level
DATEADD	Add/subtract time units
DATEDIFF	Calculate time between dates
FORMAT	Custom display formatting
ISDATE	Validate date values

Note 13: NULL Functions in SQL

Overview

This note covers NULL handling functions in SQL - essential tools for managing missing or unknown values in your data.

Theory

What is NULL?

NULL represents missing, unknown, or inapplicable data. It is NOT:

- Zero (0)
- Empty string ('')
- False

```
┌──────────────────────────────────────────────────────────────────────┐
│                    NULL BEHAVIOR                                     │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  NULL = NULL    →  UNKNOWN (not TRUE!)                               │
│  NULL <> NULL   →  UNKNOWN                                           │
│  NULL + 5       →  NULL                                              │
│  NULL AND TRUE  →  UNKNOWN                                           │
│  NULL OR TRUE   →  TRUE                                              │
│                                                                      │
│  Always use IS NULL / IS NOT NULL for comparisons!                   │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

NULL Functions

1. ISNULL - Replace NULL with Default

```
ISNULL(value, replacement)

-- Replace NULL with 0
SELECT ISNULL(score, 0) AS score FROM customers;

-- Replace NULL with 'Unknown'
SELECT ISNULL(country, 'Unknown') AS country FROM customers;
```

2. COALESCE - First Non-NULL Value

```
COALESCE(value1, value2, value3, ...)

-- Returns first non-NULL value
SELECT COALESCE(phone, mobile, email, 'No contact') AS contact
FROM customers;
```

ISNULL vs COALESCE:

Aspect	ISNULL	COALESCE
Parameters	2 only	Multiple
Standard	SQL Server only	ANSI Standard
Return type	First argument's type	Highest precedence type

3. NULLIF - Return NULL if Equal

```
NULLIF(value1, value2)

-- Returns NULL if values are equal, otherwise returns value1
SELECT NULLIF(discount, 0) AS discount FROM orders;
-- If discount = 0, returns NULL; else returns discount

-- Use case: Avoid division by zero
SELECT sales / NULLIF(quantity, 0) AS unit_price FROM orders;
```

4. IS NULL / IS NOT NULL

```
-- Find NULL values
SELECT * FROM customers WHERE phone IS NULL;

-- Find non-NULL values
SELECT * FROM customers WHERE phone IS NOT NULL;

-- ❌ Never use = NULL or <> NULL
SELECT * FROM customers WHERE phone = NULL;  -- Returns nothing!
```

Practical Examples

```
-- Handle NULL in calculations
SELECT
    product_name,
    price,
    ISNULL(discount, 0) AS discount,
    price - ISNULL(discount, 0) AS final_price
FROM products;

-- Conditional logic with NULL
SELECT
    customer_name,
    CASE
        WHEN phone IS NOT NULL THEN 'Has phone'
        WHEN email IS NOT NULL THEN 'Email only'
        ELSE 'No contact'
    END AS contact_status
FROM customers;
```

Interview Questions

Q1: What is the difference between ISNULL and COALESCE?

```
-- COALESCE accepts multiple arguments
SELECT COALESCE(phone, mobile, home_phone, 'N/A');

-- ISNULL only accepts 2
SELECT ISNULL(ISNULL(phone, mobile), 'N/A'); -- Nested for same effect
```

Q2: How do you count NULL values?

```
SELECT
    COUNT(*) AS total_rows,
    COUNT(phone) AS rows_with_phone,
    COUNT(*) - COUNT(phone) AS rows_without_phone
FROM customers;
```

Q3: How does NULL affect aggregations?

```
-- Aggregates ignore NULL values
SELECT AVG(score) FROM students; -- NULLs not counted in average
SELECT SUM(score) FROM students; -- NULLs not added
SELECT COUNT(score) FROM students; -- NULLs not counted
SELECT COUNT(*) FROM students; -- Counts all rows including NULL
```

Note 14: CASE Statements in SQL

Overview

CASE statements provide conditional logic in SQL queries - allowing you to create different outputs based on conditions.

Theory

Two Types of CASE

```
┌─────────────────────────────────────────────────────────────────────┐
│                    CASE STATEMENT TYPES                             │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  SIMPLE CASE (equality check):                                      │
│  CASE column                                                        │
│      WHEN value1 THEN result1                                       │
│      WHEN value2 THEN result2                                       │
│      ELSE default                                                   │
│  END                                                                │
│                                                                     │
│  SEARCHED CASE (any condition):                                     │
│  CASE                                                               │
│      WHEN condition1 THEN result1                                   │
│      WHEN condition2 THEN result2                                   │
│      ELSE default                                                   │
│  END                                                                │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

Simple CASE

```sql
-- Check equality to a column
SELECT
    customer_name,
    country,
    CASE country
        WHEN 'USA' THEN 'Domestic'
        WHEN 'Canada' THEN 'North America'
        WHEN 'UK' THEN 'Europe'
        ELSE 'International'
    END AS region
FROM customers;
```

Searched CASE

```
-- Any condition (more flexible)
SELECT
    customer_name,
    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;
```

CASE in Different Clauses

```
-- In SELECT (create calculated column)
SELECT name, CASE WHEN salary > 50000 THEN 'High' ELSE 'Low' END AS salary_level
FROM employees;

-- In ORDER BY (custom sorting)
SELECT * FROM orders
ORDER BY CASE status
    WHEN 'Urgent' THEN 1
    WHEN 'Normal' THEN 2
    ELSE 3
END;

-- In WHERE (conditional filtering)
SELECT * FROM products
WHERE CASE WHEN category = 'Electronics' THEN price > 100 ELSE price > 50 END;
```

Conditional Aggregation

```sql
-- Count by condition
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 by condition
SELECT
    SUM(CASE WHEN region = 'East' THEN sales ELSE 0 END) AS east_sales,
    SUM(CASE WHEN region = 'West' THEN sales ELSE 0 END) AS west_sales
FROM orders;
```

Interview Questions

Q1: What is the difference between Simple and Searched CASE?

Aspect	Simple CASE	Searched CASE
Syntax	`CASE column WHEN value...`	`CASE WHEN condition...`
Comparison	Equality only (=)	Any condition (<, >, LIKE, etc.)
Flexibility	Limited	Highly flexible

Q2: What happens if no WHEN matches and there's no ELSE?

```sql
-- Returns NULL if no match and no ELSE
SELECT CASE WHEN 1=2 THEN 'Yes' END; -- Returns NULL
```

Q3: Can CASE be nested?

```sql
SELECT
    CASE
        WHEN category = 'Electronics' THEN
            CASE WHEN price > 1000 THEN 'Premium' ELSE 'Standard' END
        ELSE 'Other'
    END AS classification
FROM products;
```

Note 15-18: Window Functions

Overview

Window functions perform calculations across a set of rows related to the current row, without collapsing rows like GROUP BY.

Basic Syntax

```
function_name() OVER (
    [PARTITION BY column]
    [ORDER BY column]
    [ROWS/RANGE frame_clause]
)
```

Key Window Functions

Aggregate Window Functions

```
-- Running total
SELECT
    order_date,
    sales,
    SUM(sales) OVER (ORDER BY order_date) AS running_total
FROM orders;

-- Partition totals
SELECT
    department,
    employee_name,
    salary,
    SUM(salary) OVER (PARTITION BY department) AS dept_total,
    AVG(salary) OVER (PARTITION BY department) AS dept_avg
FROM employees;
```

Ranking Functions

```sql
SELECT
    employee_name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
    RANK() OVER (ORDER BY salary DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank,
    NTILE(4) OVER (ORDER BY salary DESC) AS quartile
FROM employees;
```

Ranking Function Differences:

Function	Handles Ties	Gaps
ROW_NUMBER()	Arbitrary order	No gaps
RANK()	Same rank	Has gaps
DENSE_RANK()	Same rank	No gaps

Value Functions

```sql
SELECT
    order_date,
    sales,
    LAG(sales, 1) OVER (ORDER BY order_date) AS prev_sales,
    LEAD(sales, 1) OVER (ORDER BY order_date) AS next_sales,
    FIRST_VALUE(sales) OVER (ORDER BY order_date) AS first_sale,
    LAST_VALUE(sales) OVER (ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_sale
FROM orders;
```

Frame Clause

```
ROWS BETWEEN frame_start AND frame_end

-- Frame boundaries:
-- UNBOUNDED PRECEDING (first row)
-- n PRECEDING (n rows before)
-- CURRENT ROW
-- n FOLLOWING (n rows after)
-- UNBOUNDED FOLLOWING (last row)

-- 3-row moving average
SELECT
    order_date,
    sales,
    AVG(sales) OVER (ORDER BY order_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS moving_avg
FROM orders;
```

Interview Questions

Q1: What is the difference between RANK and DENSE_RANK?

```
Data: 100, 100, 90, 80

RANK():       1, 1, 3, 4  (skips 2)
DENSE_RANK(): 1, 1, 2, 3  (no gaps)
```

Q2: How do you get the top N per group?

```
WITH ranked AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rn
    FROM employees
)
SELECT * FROM ranked WHERE rn <= 3;
```

Q3: What is the difference between ROWS and RANGE?

- ROWS: Physical rows (exact count)
- RANGE: Logical range (includes ties)

Note 19-23: Advanced Window Functions

NTILE - Divide into Buckets

```sql
-- Divide into quartiles
SELECT
    employee_name,
    salary,
    NTILE(4) OVER (ORDER BY salary DESC) AS quartile
FROM employees;
```

PERCENT_RANK and CUME_DIST

```sql
SELECT
    employee_name,
    salary,
    PERCENT_RANK() OVER (ORDER BY salary) AS percent_rank,
    CUME_DIST() OVER (ORDER BY salary) AS cumulative_dist
FROM employees;
```

LAG and LEAD

```sql
-- Compare to previous/next row
SELECT
    order_date,
    sales,
    LAG(sales) OVER (ORDER BY order_date) AS prev_sales,
    sales - LAG(sales) OVER (ORDER BY order_date) AS change
FROM orders;
```

FIRST_VALUE and LAST_VALUE

```sql
SELECT
    department,
    employee_name,
    salary,
    FIRST_VALUE(employee_name) OVER (
        PARTITION BY department ORDER BY salary DESC
    ) AS top_earner
FROM employees;
```

Note 24-25: Subqueries

Overview

Subqueries are queries nested inside other queries. They can appear in SELECT, FROM, WHERE, and HAVING clauses.

Types of Subqueries

Non-Correlated (Independent)

```
-- Runs once, independently
SELECT * FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Correlated (Dependent)

```
-- References outer query, runs per row
SELECT * FROM employees e
WHERE salary > (
    SELECT AVG(salary) FROM employees
    WHERE department_id = e.department_id
);
```

EXISTS and NOT EXISTS

```
-- Find customers with orders
SELECT * FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);

-- Find customers without orders
SELECT * FROM customers c
WHERE NOT EXISTS (
    SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);
```

Subquery Locations

```sql
-- In SELECT (scalar subquery)
SELECT
    customer_name,
    (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id) AS order_count
FROM customers c;

-- In FROM (derived table)
SELECT * FROM (
    SELECT customer_id, SUM(sales) AS total FROM orders GROUP BY customer_id
) AS customer_totals
WHERE total > 1000;

-- In WHERE
SELECT * FROM products
WHERE category_id IN (SELECT id FROM categories WHERE active = 1);
```

Interview Questions

Q1: What is a correlated subquery?

A correlated subquery references the outer query and executes once per row of the outer query.

Q2: Why is EXISTS preferred over IN with NULLs?

```sql
-- NOT IN fails with NULL in subquery
SELECT * FROM customers WHERE id NOT IN (1, 2, NULL);
-- Returns 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);
```

Note 26: Common Table Expressions (CTEs)

Overview

CTEs create temporary, named result sets that exist only within a single query.

Basic Syntax

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

Multiple CTEs

```
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.*, 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;
```

Nested CTEs

```
WITH
    totals AS (
        SELECT customer_id, SUM(sales) AS total_sales
        FROM orders GROUP BY customer_id
    ),
    ranked AS (
        SELECT *, RANK() OVER(ORDER BY total_sales DESC) AS sales_rank
        FROM totals
    )
SELECT * FROM ranked WHERE sales_rank <= 5;
```

Recursive CTE

```sql
-- Organization hierarchy
WITH org_chart AS (
    -- Anchor: Top-level
    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
)
SELECT * FROM org_chart;
```

CTE vs Subquery

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

Note 27: SQL Views

Overview

Views are virtual tables based on the result of a query, without storing data.

Creating Views

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

-- Example
CREATE VIEW v_monthly_summary AS
SELECT
    DATE_TRUNC('month', order_date) AS order_month,
    SUM(sales) AS total_sales,
    COUNT(order_id) AS total_orders
FROM orders
GROUP BY DATE_TRUNC('month', order_date);
```

Using Views

```sql
SELECT * FROM v_monthly_summary;

-- With additional operations
SELECT order_month, total_sales,
       SUM(total_sales) OVER (ORDER BY order_month) AS running_total
FROM v_monthly_summary;
```

Views vs Tables

Aspect	Table	View
Storage	Stores actual data	Stores only query
Data freshness	Point-in-time	Always current
Performance	Direct access	Executes query each time

Interview Questions

Q1: Can you UPDATE data through a view?

Sometimes, if the view is "updatable" (simple, single table, no aggregations).

Q2: What is a Materialized View?

Aspect	Regular View	Materialized View
Data storage	None	Stores result set
Freshness	Always current	May be stale
Performance	Slower	Faster (pre-computed)

Note 28: CTAS and Temporary Tables

Overview

CTAS (Create Table As Select) creates tables from query results. Temporary tables store intermediate results.

CTAS Syntax

```
-- SQL Server
SELECT columns INTO new_table FROM existing_table WHERE condition;

-- PostgreSQL/MySQL
CREATE TABLE new_table AS SELECT columns FROM existing_table;
```

Temporary Tables

```
-- Local temp table (session-scoped)
SELECT * INTO #orders_temp FROM orders;

-- Use like regular table
SELECT * FROM #orders_temp;

-- Automatically dropped when session ends
```

Comparison

Aspect	View	CTAS Table	Temp Table
Storage	None	Permanent	tempdb
Data freshness	Always current	Snapshot	Snapshot

Aspect	View	CTAS Table	Temp Table
Lifetime	Until dropped	Until dropped	Session
Auto cleanup	No	No	Yes

Use Cases

```
-- ETL Process with Temp Tables
-- Step 1: Extract
SELECT * INTO #staging FROM source.orders;

-- Step 2: Transform
DELETE FROM #staging WHERE amount IS NULL;
UPDATE #staging SET status = 'Unknown' WHERE status = '';

-- Step 3: Load
INSERT INTO warehouse.orders SELECT * FROM #staging;
```

Note 29: Stored Procedures

Overview

Stored Procedures are pre-compiled SQL programs stored in the database that can be executed on demand.

Basic Syntax

```
CREATE PROCEDURE procedure_name
    @parameter1 datatype = default_value
AS
BEGIN
    -- SQL statements
END;

-- Execute
EXEC procedure_name @parameter1 = value;
```

Complete Example

```
CREATE PROCEDURE get_customer_summary
    @country VARCHAR(50) = 'USA'
AS
BEGIN
    BEGIN TRY
        -- Data cleanup
        IF EXISTS (SELECT 1 FROM customers WHERE score IS NULL AND country = @country)
        BEGIN
            UPDATE customers SET score = 0 WHERE score IS NULL AND country = @country;
        END

        -- Generate report
        DECLARE @total INT, @avg_score FLOAT;

        SELECT @total = COUNT(*), @avg_score = AVG(score)
        FROM customers WHERE country = @country;

        PRINT 'Total: ' + CAST(@total AS VARCHAR);
        PRINT 'Average: ' + CAST(@avg_score AS VARCHAR);

    END TRY
    BEGIN CATCH
        PRINT 'Error: ' + ERROR_MESSAGE();
    END CATCH
END;
```

Key Components

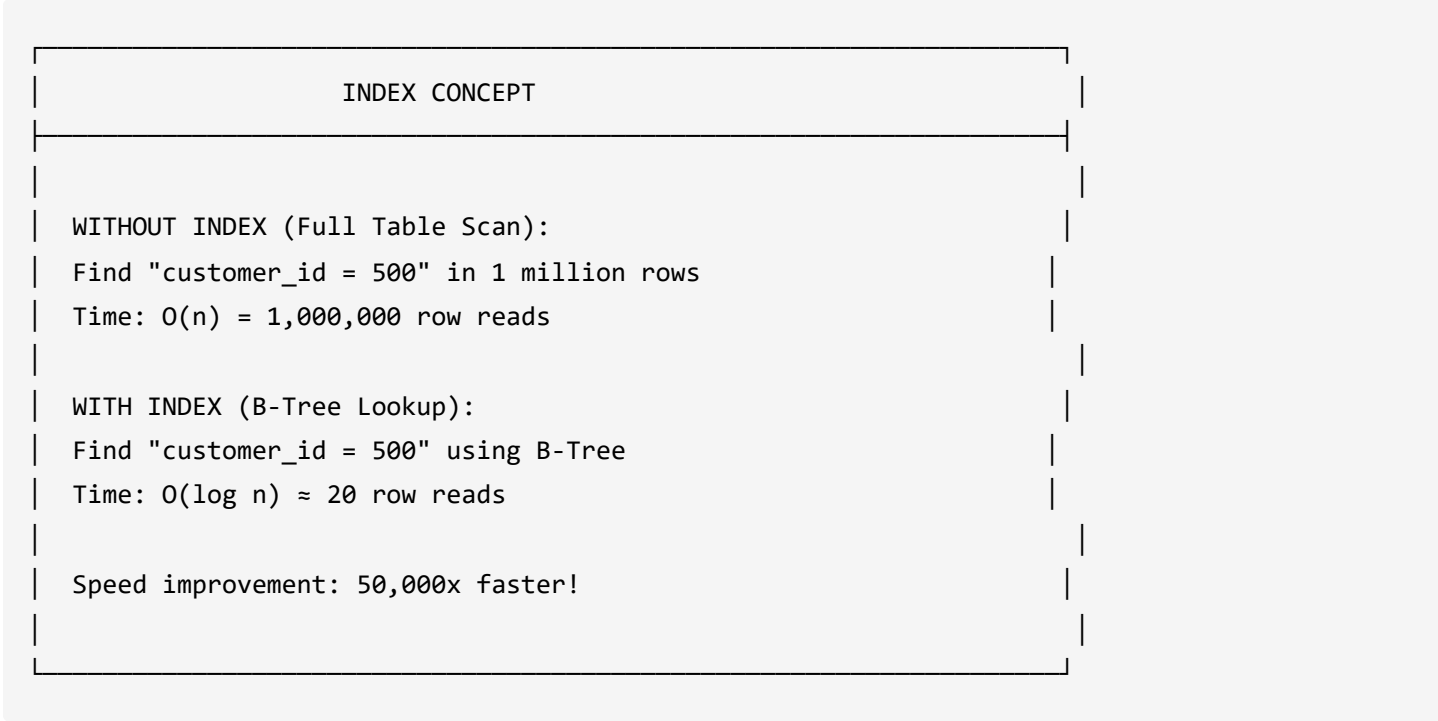
Component	Purpose	Example
Parameters	Accept input values	`@country VARCHAR(50)`
Variables	Store intermediate values	`DECLARE @total INT`
Control Flow	Conditional logic	`IF...ELSE`, `WHILE`
Error Handling	Graceful failure	`TRY...CATCH`

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.

Index Types



Clustered vs Non-Clustered

Aspect	Clustered	Non-Clustered
Per Table	Maximum 1	Multiple allowed
Data Storage	Contains actual rows	Contains pointers
Read Performance	Fastest	Slightly slower
Write Performance	Slower	Faster
Best For	Primary keys, ranges	WHERE, JOINs

Creating Indexes

```sql
-- Clustered Index
CREATE CLUSTERED INDEX idx_orders_orderid ON orders(order_id);

-- Non-Clustered Index
CREATE NONCLUSTERED INDEX idx_customers_lastname ON customers(last_name);

-- Composite Index
CREATE INDEX idx_orders_date_status ON orders(order_date, order_status);
```

Leftmost Prefix Rule

For composite indexes, the index can only be used if the query uses columns starting from the left:

```sql
-- Index on (country, city, zipcode)
CREATE INDEX idx_location ON customers(country, city, zipcode);

-- ✅ Uses index
WHERE country = 'USA'
WHERE country = 'USA' AND city = 'NYC'

-- ❌ Cannot use index
WHERE city = 'NYC'
WHERE zipcode = '10001'
```

Row Store vs Column Store

Aspect	Row Store	Column Store
Storage	Row by row	Column by column
Compression	Limited	High
Best For	OLTP (transactions)	OLAP (analytics)

Best Practices

Do's:

1. Index primary keys
2. Index foreign keys used in JOINs
3. Index columns in frequent WHERE clauses
4. Use composite indexes for multi-column filters

Don'ts:

1. Don't over-index (slows writes)
2. Don't index frequently updated columns
3. Don't create duplicate indexes

Quick Reference Card

SQL Query Template

```
SELECT [DISTINCT] columns
FROM table_name
[WHERE row_filter]
[GROUP BY columns]
[HAVING aggregate_filter]
[ORDER BY columns [ASC|DESC]]
[TOP n / LIMIT n]
```

Execution Order

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

Common Aggregate Functions

- `COUNT(*)` - Count all rows
- `COUNT(column)` - Count non-NULL values
- `SUM(column)` - Total of values
- `AVG(column)` - Average value
- `MAX(column)` - Maximum value
- `MIN(column)` - Minimum value

WHERE vs HAVING

WHERE	HAVING
Before GROUP BY	After GROUP BY
Filters rows	Filters groups
No aggregates	Can use aggregates

DISTINCT vs GROUP BY

DISTINCT	GROUP BY
Remove duplicates	Group for aggregation
Simple	Allows aggregates