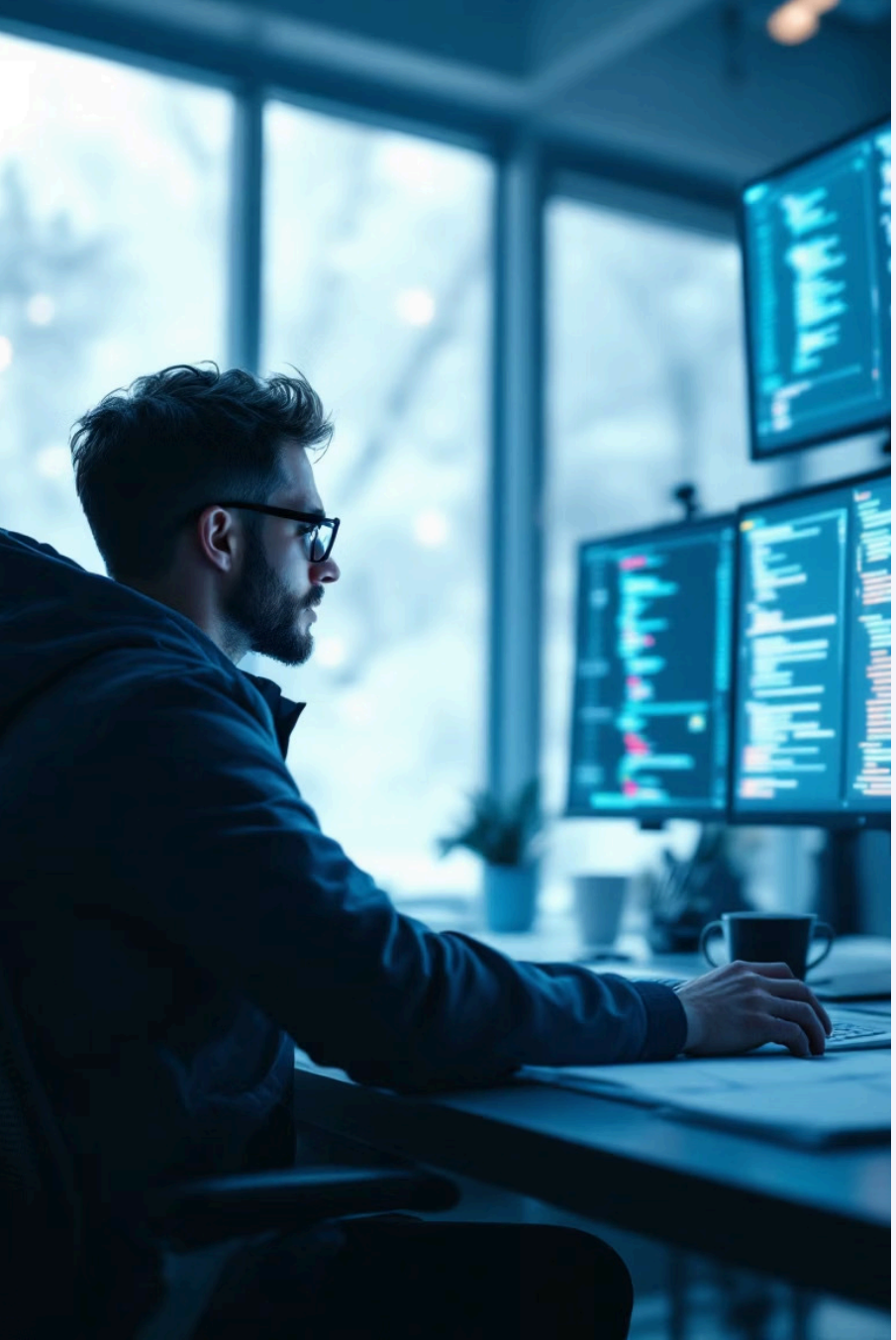

SQL FUNDAMENTALS

Complete Bootcamp for Data Analysts

- 26 Core Concepts
- Interview-Ready Patterns
- Production Best Practices
- Hands-On Examples

Session 3 | 2 Hours | Live Coding

Session 3: SQL for Data Analysis

SQL FUNDAMENTALS

Complete Bootcamp for Data Analysts

26 Core Concepts

From basics to advanced analytics

Interview-Ready Patterns

Real questions, proven solutions

Production Best Practices

Write code that scales

Hands-On Examples

Practice with real scenarios

Session 3 | 2 Hours | Live Coding

The Foundation

SQL EXECUTION ORDER

THE MOST CRITICAL CONCEPT

⚡ Query Execution Order (Memorize This!)

01

FROM + JOINs

Get tables

02

WHERE

Filter rows

03

GROUP BY

Create groups

04

HAVING

Filter groups

05

SELECT

Choose columns

06

DISTINCT

Remove duplicates

07

ORDER BY

Sort results

08

TOP/OFFSET-FETCH

Limit rows

Memory Trick: From Where Group Having Select Distinctly Order The-Top

Why This Matters:

- WHERE executes BEFORE SELECT
- Can't use aliases in WHERE (they don't exist yet!)
- HAVING filters groups, WHERE filters rows
- ORDER BY can use aliases (executes last)

EXECUTION ORDER

Common Mistake

❌ WRONG (Most Common Beginner Error):

```
SELECT
  TotalAmount * 1.1 AS TotalWithTax
FROM Orders
WHERE TotalWithTax > 100;
-- ERROR! Alias doesn't exist yet
```

The alias TotalWithTax doesn't exist when WHERE executes!

✅ CORRECT Option 1 - Repeat Calculation:

```
SELECT
  TotalAmount * 1.1 AS TotalWithTax
FROM Orders
WHERE TotalAmount * 1.1 > 100;
```

✅ CORRECT Option 2 - Use CTE:

```
WITH OrdersWithTax AS (
 SELECT
 OrderID,
 TotalAmount * 1.1 AS TotalWithTax
 FROM Orders
)
SELECT *
FROM OrdersWithTax
WHERE TotalWithTax > 100;
```

SQL SERVER HIERARCHY

Understanding the Structure

```
Server Instance
├── Databases
│   ├── Tables
│   │   ├── Columns (with data types)
│   │   └── Rows (actual data)
│   ├── Views (saved queries)
│   ├── Stored Procedures (saved logic)
│   └── Functions
└── Security (logins, permissions)
```

Key Concepts:

- **Schema** = Organizational container (sales.Customers, hr.Employees)
- **Database** = Collection of related objects
- **Table** = Structured data storage
- **Column** = Single attribute (CustomerName, OrderDate)
- **Row** = Single record

DDL vs DML vs DQL

Three Types of SQL Commands

DDL

Define structure

Commands: CREATE, ALTER, DROP

Example: CREATE TABLE Customers

DML

Modify data

Commands: INSERT, UPDATE, DELETE

Example: INSERT INTO Customers VALUES (...)

DQL

Query data

Commands: SELECT

Example: SELECT * FROM Customers

Production Rule:

- DDL = Database Admin
- DML = Application Code
- DQL = Data Analysts (YOU!)

DATA TYPES

Quick Reference - Choose the Right Type

Numbers:

- **INT** → IDs, counts (-2B to +2B)
- **BIGINT** → Large numbers
- **DECIMAL(10,2)** → Money (exact)
- **FLOAT** → Scientific calculations (approximate)

Text:

- **VARCHAR(n)** → Variable text (English)
- **NVARCHAR(n)** → Unicode text (all languages) ← **Use this**
- **CHAR(n)** → Fixed length (avoid)

Dates:

- **DATE** → Date only (2024-01-15)
- **DATETIME2** → Date + Time ← **Recommended**
- **DATETIME** → Older, less precise

Other:

- **BIT** → True/False (0 or 1)

SELECT

THE FOUNDATION

Basic Syntax

```
SELECT column1, column2
FROM table
WHERE condition
ORDER BY column;
```

Best Practices:

```
--
```

WHERE CLAUSE

FILTERING ROWS

Comparison Operators:

= Equal
<> Not equal (also !=)
> Greater than
< Less than
>= Greater or equal
<= Less or equal

Logical Operators:

```
-- AND (all conditions must be true)
WHERE Age >= 18 AND Country = 'USA'

-- OR (any condition can be true)
WHERE Status = 'Active' OR Status = 'Pending'

-- NOT (reverse condition)
WHERE NOT Status = 'Cancelled'
```

Special Operators:

```
-- IN (match list)
WHERE Status IN ('Active', 'Pending', 'Shipped')

-- BETWEEN (inclusive range)
WHERE OrderDate BETWEEN '2024-01-01' AND '2024-12-31'

-- LIKE (pattern matching)
WHERE Email LIKE '%@gmail.com'
WHERE FirstName LIKE 'J%' -- Starts with J
WHERE LastName LIKE '%son' -- Ends with son
WHERE Phone LIKE '555-____' -- _ = single character

-- IS NULL / IS NOT NULL
WHERE Email IS NULL
WHERE Phone IS NOT NULL
```

NULL HANDLING

CRITICAL CONCEPT

⚠️ **NULL is NOT a value - it's the ABSENCE of a value**

Common Mistakes:

```
--
```

AGGREGATE FUNCTIONS

Summarizing Data

COUNT(*)

Count all rows

`SELECT COUNT(*) FROM Orders`

COUNT(column)

Count non-NULL values

`SELECT COUNT(Email) FROM Customers`

SUM(column)

Total of numbers

`SELECT SUM(TotalAmount) FROM Orders`

AVG(column)

Average

`SELECT AVG(TotalAmount) FROM Orders`

MIN(column)

Minimum value

`SELECT MIN(OrderDate) FROM Orders`

MAX(column)

Maximum value

`SELECT MAX(TotalAmount) FROM Orders`

Real Example:

```
SELECT
 COUNT(*) AS TotalOrders,
 COUNT(DISTINCT CustomerID) AS UniqueCustomers,
 SUM(TotalAmount) AS TotalRevenue,
 AVG(TotalAmount) AS AvgOrderValue,
 MIN(TotalAmount) AS SmallestOrder,
 MAX(TotalAmount) AS LargestOrder
FROM Orders
WHERE OrderDate >= '2024-01-01';
```

GROUP BY

AGGREGATING BY CATEGORY

The Golden Rule:

Every column in SELECT must be:

1. In the GROUP BY clause, OR
2. Inside an aggregate function

Example:

```
SELECT
  CustomerID,        -- In GROUP BY ✓
  COUNT(*) AS OrderCount,  -- Aggregate ✓
  SUM(TotalAmount) AS TotalSpent -- Aggregate ✓
FROM Orders
GROUP BY CustomerID; -- Must match SELECT
```

❌ Common Mistake:

```
SELECT
 CustomerID,
 CustomerName, --
```

HAVING

FILTERING GROUPS

WHERE vs HAVING

WHERE

Filters: Individual rows

When: BEFORE grouping

Can use aggregates? NO ❌

Execution order: Step 2

HAVING

Filters: Grouped results

When: AFTER grouping

Can use aggregates? YES ✅

Execution order: Step 4

Example:

```
SELECT
  CustomerID,
  COUNT(*) AS OrderCount,
  SUM(TotalAmount) AS TotalSpent
FROM Orders
WHERE OrderDate >= '2024-01-01'  -- Filter rows first
GROUP BY CustomerID
HAVING COUNT(*) >= 5             -- Filter groups after
  AND SUM(TotalAmount) > 1000
ORDER BY TotalSpent DESC;
```

Use WHERE when: Filtering individual rows (customer, date, status)

Use HAVING when: Filtering aggregated results (count > 5, sum > 1000)

JOINS

THE 5 TYPES

Visual Representation

- **INNER JOIN:** Only matches from both
- **LEFT JOIN:** All from left + matches from right
- **RIGHT JOIN:** All from right + matches from left
- **FULL JOIN:** Everything from both
- **CROSS JOIN:** Every combination (Cartesian product)

When to Use:

INNER

Only customers WHO HAVE orders

LEFT

ALL customers, even WITHOUT orders

RIGHT

Rarely used (rewrite as LEFT)

FULL

Everything, including orphans (rare)

CROSS

Combinations (size × color = all variants)

INNER JOIN

ONLY MATCHES

Syntax:

```
SELECT
  c.CustomerID,
  c.FirstName,
  c.LastName,
  o.OrderID,
  o.OrderDate,
  o.TotalAmount
FROM Customers c
INNER JOIN Orders o
  ON c.CustomerID = o.CustomerID;
```

Result:

- Only customers who HAVE placed orders
- Customers without orders: EXCLUDED
- Orders without valid customer: EXCLUDED

Use Case: "Show me all active customers with their purchase history"

LEFT JOIN

ALL FROM LEFT

Syntax:

```
SELECT
  c.CustomerID,
  c.FirstName,
  c.LastName,
  COUNT(o.OrderID) AS OrderCount,
  ISNULL(SUM(o.TotalAmount), 0) AS TotalSpent
FROM Customers c
LEFT JOIN Orders o
  ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName;
```

Result:

- ALL customers included
- Customers without orders show 0 orders, 0 spent
- NULL values from right table become 0 with ISNULL

Use Case: "Show me ALL customers and their purchase totals, including customers who haven't ordered yet"

SUBQUERIES

QUERIES WITHIN QUERIES

Three Types:

1. Scalar Subquery (returns single value):

```
SELECT *
FROM Orders
WHERE TotalAmount > (
  SELECT AVG(TotalAmount)
  FROM Orders
);
```

2. List Subquery (returns list of values):

```
SELECT *
FROM Customers
WHERE CustomerID IN (
  SELECT CustomerID
  FROM Orders
  WHERE OrderDate >= '2024-01-01'
);
```

3. Table Subquery (returns entire table):

```
SELECT *
FROM (
 SELECT
 CustomerID,
 SUM(TotalAmount) AS Total
 FROM Orders
 GROUP BY CustomerID
) AS CustomerTotals
WHERE Total > 1000;
```

CTEs

READABLE SUBQUERIES

Common Table Expression = Temporary Named Result Set

Why Use CTEs:

- More readable than nested subqueries
- Can reference multiple times
- Can be recursive
- Better for complex queries

Syntax:

```
WITH CustomerSummary AS (
  SELECT
    CustomerID,
    COUNT(*) AS OrderCount,
    SUM(TotalAmount) AS TotalSpent
  FROM Orders
  GROUP BY CustomerID
)
SELECT
  c.CustomerName,
  cs.OrderCount,
  cs.TotalSpent
FROM Customers c
INNER JOIN CustomerSummary cs
  ON c.CustomerID = cs.CustomerID
WHERE cs.TotalSpent > 5000
ORDER BY cs.TotalSpent DESC;
```

Multiple CTEs:

```
WITH HighValueOrders AS (
 SELECT * FROM Orders WHERE TotalAmount > 1000
),
ActiveCustomers AS (
 SELECT DISTINCT CustomerID
 FROM Orders
 WHERE OrderDate >= '2024-01-01'
)
SELECT *
FROM HighValueOrders
WHERE CustomerID IN (SELECT CustomerID FROM ActiveCustomers);
```

WINDOW FUNCTIONS

ADVANCED ANALYTICS

What Are Window Functions?

- Perform calculations ACROSS rows
- Don't collapse rows (unlike GROUP BY)
- Each row keeps its identity

The 4 Most Important:

1. ROW_NUMBER() - Unique sequential number

```
SELECT
  CustomerID,
  OrderDate,
  TotalAmount,
  ROW_NUMBER() OVER (
    PARTITION BY CustomerID
    ORDER BY OrderDate
  ) AS OrderSequence
FROM Orders;
```

2. RANK() - Rank with gaps

```
SELECT
  ProductID,
  TotalRevenue,
  RANK() OVER (ORDER BY TotalRevenue DESC) AS RevenueRank
FROM ProductSales;
-- If tie at rank 2, next is rank 4 (gap)
```

3. DENSE_RANK() - Rank without gaps

```
DENSE_RANK() OVER (ORDER BY TotalRevenue DESC)
-- If tie at rank 2, next is rank 3 (no gap)
```

4. LAG/LEAD - Previous/Next row values

```
SELECT
 OrderDate,
 TotalAmount,
 LAG(TotalAmount) OVER (ORDER BY OrderDate) AS PreviousOrder,
 LEAD(TotalAmount) OVER (ORDER BY OrderDate) AS NextOrder
FROM Orders;
```

WINDOW FUNCTIONS

PARTITIONING

PARTITION BY = GROUP BY for Window Functions

Running Total Per Customer:

```
SELECT
  CustomerID,
  OrderDate,
  TotalAmount,
  SUM(TotalAmount) OVER (
    PARTITION BY CustomerID
    ORDER BY OrderDate
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS RunningTotal
FROM Orders;
```

Rank Within Each Category:

```
SELECT
  Category,
  ProductName,
  TotalRevenue,
  RANK() OVER (
    PARTITION BY Category
    ORDER BY TotalRevenue DESC
  ) AS CategoryRank
FROM Products;
```

Key Insight:

- Without PARTITION BY: Calculates across ALL rows
- With PARTITION BY: Resets calculation for each partition

STRING FUNCTIONS

Most Useful String Functions:

```
-- CONCAT: Join strings (handles NULL)
SELECT CONCAT(FirstName, ' ', LastName) AS FullName;

-- String concatenation with +
SELECT FirstName + ' ' + LastName AS FullName;

-- UPPER/LOWER: Change case
SELECT UPPER(Email), LOWER(ProductName);

-- TRIM/LTRIM/RTRIM: Remove spaces
SELECT TRIM(' hello ') → 'hello'

-- SUBSTRING: Extract part of string
SELECT SUBSTRING(Email, 1, 10); -- First 10 characters

-- LEN: String length
SELECT LEN(ProductName);

-- LEFT/RIGHT: Get characters from start/end
SELECT LEFT(Phone, 3); -- Area code
SELECT RIGHT(OrderID, 4); -- Last 4 digits

-- REPLACE: Replace text
SELECT REPLACE(Email, '@gmail.com', '@company.com');

-- CHARINDEX: Find position
SELECT CHARINDEX('@', Email); -- Position of @
```

DATE FUNCTIONS

Essential Date Functions:

```
-- GETDATE: Current date and time
SELECT GETDATE(); -- 2024-02-01 14:30:45

-- DATEADD: Add interval
SELECT DATEADD(DAY, 7, GETDATE()); -- 7 days from now
SELECT DATEADD(MONTH, -3, GETDATE()); -- 3 months ago
SELECT DATEADD(YEAR, 1, OrderDate); -- 1 year after order

-- DATEDIFF: Calculate difference
SELECT DATEDIFF(DAY, OrderDate, ShipDate); -- Days between
SELECT DATEDIFF(MONTH, HireDate, GETDATE()); -- Months employed
SELECT DATEDIFF(YEAR, DateOfBirth, GETDATE()); -- Age in years

-- YEAR/MONTH/DAY: Extract parts
SELECT YEAR(OrderDate), MONTH(OrderDate), DAY(OrderDate);

-- EOMONTH: End of month
SELECT EOMONTH(GETDATE()); -- Last day of current month
SELECT EOMONTH(GETDATE(), 1); -- Last day of next month

-- FORMAT: Custom date format
SELECT FORMAT(GETDATE(), 'yyyy-MM-dd'); -- 2024-02-01
SELECT FORMAT(GETDATE(), 'MMMM dd, yyyy'); -- February 01, 2024
SELECT FORMAT(GETDATE(), 'dd/MM/yyyy HH:mm'); -- 01/02/2024 14:30
```

CASE EXPRESSIONS

CONDITIONAL LOGIC

Simple CASE:

```
SELECT
  OrderID,
  Status,
  CASE Status
    WHEN 'Pending' THEN 'Awaiting Processing'
    WHEN 'Shipped' THEN 'In Transit'
    WHEN 'Delivered' THEN 'Complete'
    ELSE 'Unknown'
  END AS StatusDescription
FROM Orders;
```

Searched CASE (more flexible):

```
SELECT
  ProductName,
  Price,
  CASE
    WHEN Price < 10 THEN 'Budget'
    WHEN Price BETWEEN 10 AND 50 THEN 'Standard'
    WHEN Price BETWEEN 51 AND 100 THEN 'Premium'
    ELSE 'Luxury'
  END AS PriceCategory
FROM Products;
```

Use Cases:

- Categorization (price ranges, age groups)
- Custom sorting
- Conditional aggregation
- Data transformation

COMMON INTERVIEW QUESTIONS

Question 1: Find Duplicate Records

```
-- Find customers with duplicate emails
SELECT Email, COUNT(*) AS DuplicateCount
FROM Customers
GROUP BY Email
HAVING COUNT(*) > 1;

-- Get full records of duplicates
WITH Duplicates AS (
  SELECT Email, COUNT(*) AS cnt
  FROM Customers
  GROUP BY Email
  HAVING COUNT(*) > 1
)
SELECT c.*
FROM Customers c
INNER JOIN Duplicates d ON c.Email = d.Email
ORDER BY c.Email;
```

Question 2: Top N Per Group

```
-- Top 3 products per category by revenue
WITH RankedProducts AS (
 SELECT
 Category,
 ProductName,
 TotalRevenue,
 ROW_NUMBER() OVER (
 PARTITION BY Category
 ORDER BY TotalRevenue DESC
 ) AS rn
 FROM Products
)
SELECT Category, ProductName, TotalRevenue
FROM RankedProducts
WHERE rn <= 3;
```

COMMON INTERVIEW QUESTIONS

Question 3: Month-over-Month Growth

```
WITH MonthlySales AS (
 SELECT
 YEAR(OrderDate) AS Year,
 MONTH(OrderDate) AS Month,
 SUM(TotalAmount) AS Revenue
 FROM Orders
 GROUP BY YEAR(OrderDate), MONTH(OrderDate)
)
SELECT
 Year,
 Month,
 Revenue,
 LAG(Revenue) OVER (ORDER BY Year, Month) AS PrevMonthRevenue,
 Revenue - LAG(Revenue) OVER (ORDER BY Year, Month) AS Growth,
 CAST(
 (Revenue - LAG(Revenue) OVER (ORDER BY Year, Month)) * 100.0
 / LAG(Revenue) OVER (ORDER BY Year, Month)
 AS DECIMAL(5,2)
 ) AS GrowthPercent
FROM MonthlySales;
```

COMMON MISTAKES TO AVOID

Mistake 1: Forgetting WHERE in UPDATE/DELETE

```
--
```

COMMON MISTAKES TO AVOID

Mistake 4: Missing Columns in GROUP BY

```
--
```

PERFORMANCE BEST PRACTICES

Query Optimization:

✅ DO:

- Select only needed columns (not SELECT *)
- Filter early with WHERE
- Use EXISTS instead of IN for large subqueries
- Create indexes on foreign keys
- Use appropriate data types
- Avoid functions on indexed columns in WHERE

❌ DON'T:

- SELECT * (retrieves unnecessary data)
- Use LIKE '%value' (can't use index)
- Put functions in WHERE on indexed columns
- Use DISTINCT when not needed
- Over-index (slows INSERT/UPDATE)

Index Best Practices:

```
-- Create index on foreign key
CREATE INDEX IX_Orders_CustomerID ON Orders(CustomerID);

-- Create index on frequently filtered columns
CREATE INDEX IX_Orders_OrderDate ON Orders(OrderDate);

-- Composite index for common filter combinations
CREATE INDEX IX_Orders_Customer_Date ON Orders(CustomerID, OrderDate);
```

NEXT STEPS & PRACTICE

What You've Learned:

- ✅ SQL execution order (CRITICAL!)
- ✅ SELECT, WHERE, GROUP BY, HAVING
- ✅ All 5 JOIN types
- ✅ Subqueries and CTEs
- ✅ Window functions (ROW_NUMBER, RANK, LAG/LEAD)
- ✅ String and Date functions
- ✅ CASE expressions
- ✅ Common interview patterns
- ✅ Performance best practices

Practice Projects:

1. **Customer Analysis:** RFM segmentation using window functions
2. **Sales Trends:** Month-over-month growth with LAG
3. **Product Performance:** Top N per category with ROW_NUMBER
4. **Data Cleaning:** Find and remove duplicates
5. **Cohort Analysis:** Customer retention by signup month

Resources:

- Practice SQL: SQLZoo, HackerRank, LeetCode
- Documentation: docs.microsoft.com/sql
- Next Session: Medallion Architecture (Bronze → Silver → Gold)

Remember: The execution order is EVERYTHING. Master that and the rest follows!

BACKUP SLIDES

DATA TYPE CONVERSION

CAST vs CONVERT vs TRY_CONVERT:

```
-- CAST (ANSI standard)
SELECT CAST('123' AS INT);
SELECT CAST(OrderDate AS DATE);

-- CONVERT (SQL Server specific, allows format)
SELECT CONVERT(VARCHAR, GETDATE(), 101); -- MM/DD/YYYY
SELECT CONVERT(VARCHAR, GETDATE(), 103); -- DD/MM/YYYY

-- TRY_CONVERT (safe - returns NULL on error)
SELECT TRY_CONVERT(INT, 'abc');       -- Returns NULL instead of error
SELECT TRY_CONVERT(DATE, '2024-13-45'); -- Returns NULL (invalid date)
```

When to Use:

- User input: Use TRY_CONVERT
- Guaranteed valid data: Use CAST or CONVERT
- Date formatting: Use CONVERT with style codes

TRANSACTIONS – DATA CONSISTENCY

ACID Properties:

- **Atomicity:** All or nothing
- **Consistency:** Valid state always
- **Isolation:** Transactions don't interfere
- **Durability:** Committed = permanent

Syntax:

```
BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

IF @@ERROR = 0
  COMMIT TRANSACTION;
ELSE
  ROLLBACK TRANSACTION;
```

Use When:

- Multiple related updates must succeed together
- Financial transactions
- Inventory management
- Critical data modifications

INDEXES – PERFORMANCE BASICS

What is an Index?

- Like a book index - helps find data faster
- Speeds up SELECT queries
- Slows down INSERT/UPDATE/DELETE

Types:

```
-- Clustered Index (one per table - physical order)
CREATE CLUSTERED INDEX IX_Orders_OrderID ON Orders(OrderID);

-- Non-Clustered Index (can have many)
CREATE NONCLUSTERED INDEX IX_Orders_CustomerID ON Orders(CustomerID);

-- Composite Index (multiple columns)
CREATE INDEX IX_Orders_Customer_Date ON Orders(CustomerID, OrderDate);

-- Unique Index (enforces uniqueness)
CREATE UNIQUE INDEX IX_Customers_Email ON Customers(Email);
```

When to Index:

- Primary keys (automatic)
- Foreign keys
- Frequently filtered columns (WHERE, JOIN)
- Frequently sorted columns (ORDER BY)

When NOT to Index:

- Small tables (< 1000 rows)
- Columns rarely queried
- High-write, low-read tables