Day 2 — SQL Server Basics + Core Querying

1) Introduction to SQL Server (brief)

What is SQL Server?
A Relational Database Management System (RDBMS) from Microsoft used to store, retrieve and manage relational data.

SQL Server Editions (short): Express (free, limited), Standard, Enterprise (full features), Developer (full features but non-production).

SSMS overview: Microsoft SQL Server Management Studio — GUI for connecting, running queries, designing tables, viewing execution plans.

Databases, Schemas, Tables:

- Database = logical container.
- Schema = namespace inside DB (default `dbo`).
- Table = rows & columns.

DDL vs DML vs DQL

- DDL (Data Definition Language): `CREATE`, `ALTER`, `DROP`.
- DML (Data Manipulation Language): `INSERT`, `UPDATE`, `DELETE`.
- DQL (Data Query Language): `SELECT`.

---

2) Creating Database & Tables (banking schema)

Create database

```
CREATE DATABASE BankDemo;
GO

USE BankDemo;
GO
```

Tables with constraints (Customers, Branches, Accounts, Transactions)

```
-- Branches
CREATE TABLE dbo.Branches (
    BranchID INT IDENTITY(1,1) PRIMARY KEY,
    BranchName NVARCHAR(100) NOT NULL,
    City NVARCHAR(50) NOT NULL,
    IFSC CHAR(11) NOT NULL UNIQUE  -- Unique constraint
```

```
);

-- Customers
CREATE TABLE dbo.Customers (
    CustomerID INT IDENTITY(1,1) PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    Email NVARCHAR(255) NULL UNIQUE,      -- Unique email, allow NULL
    DateOfBirth DATE NULL,
    CreatedAt DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()
);

-- Accounts
CREATE TABLE dbo.Accounts (
    AccountID BIGINT IDENTITY(1000000000,1) PRIMARY KEY,
    CustomerID INT NOT NULL,
    BranchID INT NOT NULL,
    AccountType CHAR(1) NOT NULL CHECK (AccountType IN ('S','C')), -- S =
Savings, C = Current
    Balance MONEY NOT NULL DEFAULT (0),
    IsActive BIT NOT NULL DEFAULT (1),
    OpenedOn DATE NOT NULL DEFAULT CONVERT(date, GETDATE()),
    CONSTRAINT FK_Accounts_Customers FOREIGN KEY (CustomerID) REFERENCES
dbo.Customers(CustomerID),
    CONSTRAINT FK_Accounts_Branches FOREIGN KEY (BranchID) REFERENCES
dbo.Branches(BranchID)
);

-- Transactions
CREATE TABLE dbo.Transactions (
    TransactionID BIGINT IDENTITY(1,1) PRIMARY KEY,
    AccountID BIGINT NOT NULL,
    TranDate DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    Amount MONEY NOT NULL CHECK (Amount <> 0),
    TranType CHAR(1) NOT NULL CHECK (TranType IN ('D','W','T')), --
D=Deposit, W=Withdrawal, T=Transfer
    Description NVARCHAR(250) NULL,
    RelatedAccountID BIGINT NULL, -- used for transfers
    CONSTRAINT FK_Trans_Account FOREIGN KEY (AccountID) REFERENCES
dbo.Accounts(AccountID)
);
```

Notes on constraints:

- `PRIMARY KEY` enforces uniqueness + not null.
- `FOREIGN KEY` enforces referential integrity.
- `UNIQUE` enforces uniqueness but allows NULL by default (SQL Server allows one NULL).
- `CHECK` enforces custom rule.
- `DEFAULT` sets default values on insert.

---

3) SELECT Queries — basics & filtering

Insert sample data

```
INSERT INTO dbo.Branches (BranchName, City, IFSC)
VALUES
('Central', 'Mumbai', 'BKID0000001'),
('Andheri', 'Mumbai', 'BKID0000002');

INSERT INTO dbo.Customers (FirstName, LastName, Email, DateOfBirth)
VALUES
('Amit','Kumar','amit.k@example.com','1985-06-15'),
('Neha','Sharma','neha.s@example.com','1990-02-21'),
('Ravi','Patel', NULL, '1979-11-03');

INSERT INTO dbo.Accounts (CustomerID, BranchID, AccountType, Balance)
VALUES
(1,1,'S', 15000),
(2,2,'C', 500000),
(3,1,'S', 2500);
```

Simple SELECT and DISTINCT

```
-- All customers
SELECT * FROM dbo.Customers;

-- Distinct example: distinct cities for branches
SELECT DISTINCT City FROM dbo.Branches;
```

WHERE, ORDER BY

```
-- Customers born after 1985, ordered by last name
SELECT CustomerID, FirstName, LastName, DateOfBirth
FROM dbo.Customers
WHERE DateOfBirth > '1985-01-01'
ORDER BY LastName ASC;
```

TOP, OFFSET-FETCH (pagination)

```
-- Top 2 accounts with highest balance
SELECT TOP (2) AccountID, Balance
FROM dbo.Accounts
ORDER BY Balance DESC;

-- Pagination: page 2 with page size 2 (OFFSET requires ORDER BY)
DECLARE @PageSize INT = 2, @PageNumber INT = 2;
SELECT AccountID, CustomerID, Balance
FROM dbo.Accounts
ORDER BY AccountID
OFFSET (@PageNumber-1)*@PageSize ROWS
FETCH NEXT @PageSize ROWS ONLY;
```

Filtering operators: IN, BETWEEN, LIKE, AND/OR/NOT

```
-- IN
SELECT * FROM dbo.Accounts WHERE AccountType IN ('S','C');

-- BETWEEN
SELECT * FROM dbo.Accounts WHERE Balance BETWEEN 1000 AND 20000;

-- LIKE
SELECT * FROM dbo.Customers WHERE Email LIKE '%@example.com';

-- AND / OR / NOT
SELECT * FROM dbo.Customers
WHERE (FirstName = 'Amit' OR FirstName = 'Neha')
  AND NOT Email IS NULL;
```

---

4) JOINS (practical banking examples)

INNER JOIN: accounts with customer info

```
SELECT a.AccountID, a.Balance, a.AccountType, c.FirstName, c.LastName
FROM dbo.Accounts a
INNER JOIN dbo.Customers c ON a.CustomerID = c.CustomerID;
```

LEFT JOIN: all customers and their accounts (if any)

```
SELECT c.CustomerID, c.FirstName, c.LastName, a.AccountID, a.Balance
FROM dbo.Customers c
LEFT JOIN dbo.Accounts a ON c.CustomerID = a.CustomerID;
```

RIGHT JOIN: all accounts and branch info (mirror of left)

```
SELECT a.AccountID, a.Balance, b.BranchName
FROM dbo.Accounts a
RIGHT JOIN dbo.Branches b ON a.BranchID = b.BranchID;
-- Note: RIGHT JOIN less common; prefer LEFT JOIN by switching tables.
```

FULL OUTER JOIN: rows that don't match in either side

```
SELECT c.CustomerID, c.FirstName, a.AccountID
FROM dbo.Customers c
FULL OUTER JOIN dbo.Accounts a ON c.CustomerID = a.CustomerID;
```

CROSS JOIN: Cartesian product (use rarely)

```
-- Example: generate interest scenarios combining two rates and two terms
SELECT b.BranchName, r.Rate
FROM dbo.Branches b
CROSS JOIN (VALUES (3.5),(4.0),(4.5)) AS r(Rate);
```

SELF JOIN: find customers with same last name (example)

```
SELECT c1.CustomerID AS Cust1, c1.FirstName, c2.CustomerID AS Cust2,
c2.FirstName
FROM dbo.Customers c1
INNER JOIN dbo.Customers c2 ON c1.LastName = c2.LastName AND c1.CustomerID <>
c2.CustomerID;
```

---

5) Built-In Functions

String functions: LEFT, RIGHT, LEN, CONCAT

```
SELECT FirstName + ' ' + LastName AS FullName,
       LEFT(FirstName,1) + '.' + LastName AS ShortName,
       LEN(Email) AS EmailLen,
       CONCAT(FirstName,' ',LastName) AS ConcatName
FROM dbo.Customers;
```

Date/Time: GETDATE, DATEADD, DATEDIFF

```
SELECT GETDATE() AS Now,
       DATEADD(day, 30, GETDATE()) AS In30Days,
       DATEDIFF(year, DateOfBirth, GETDATE()) AS Age
FROM dbo.Customers;
```

Numeric: ROUND, CEILING, FLOOR

```
SELECT Balance,
       ROUND(CAST(Balance AS DECIMAL(18,2)), -2) AS RoundedToHundreds,
       CEILING(Balance) AS CeilingVal,
       FLOOR(Balance) AS FloorVal
FROM dbo.Accounts;
```

---

6) Subqueries & Derived Tables

Single-row subquery

```
-- Find accounts opened at same branch as customer 1 (single value subquery
for branch)
SELECT a.AccountID, a.Balance
FROM dbo.Accounts a
WHERE a.BranchID = (SELECT TOP 1 BranchID FROM dbo.Accounts WHERE CustomerID
= 1);
```

Multi-row subquery (IN)

```
-- Accounts for customers who have email at example.com
```

```
SELECT AccountID, CustomerID, Balance
FROM dbo.Accounts
WHERE CustomerID IN (SELECT CustomerID FROM dbo.Customers WHERE Email LIKE
'%@example.com');
```

Derived table (subquery in FROM)

```
SELECT dt.BranchID, COUNT(*) AS AccountCount, AVG(dt.Balance) AS AvgBalance
FROM (
    SELECT BranchID, Balance
    FROM dbo.Accounts
    WHERE IsActive = 1
) AS dt
GROUP BY dt.BranchID;
```

Stored Procedures, Indexing, Views

1) Views

What are views?
Named queries saved as virtual tables. Can simplify complex queries and encapsulate logic.

Simple view

```
CREATE VIEW dbo.v_CustomerAccounts
AS
SELECT c.CustomerID, c.FirstName, c.LastName, a.AccountID, a.Balance,
b.BranchName
FROM dbo.Customers c
JOIN dbo.Accounts a ON c.CustomerID = a.CustomerID
JOIN dbo.Branches b ON a.BranchID = b.BranchID;
GO

-- Use view
SELECT * FROM dbo.v_CustomerAccounts WHERE Balance > 10000;
```

Complex view (aggregations)

```
CREATE VIEW dbo.v_BranchSummary
AS
SELECT b.BranchID, b.BranchName, COUNT(a.AccountID) AS TotalAccounts,
SUM(a.Balance) AS TotalDeposits
FROM dbo.Branches b
LEFT JOIN dbo.Accounts a ON b.BranchID = a.BranchID
GROUP BY b.BranchID, b.BranchName;
GO
```

---

2) Stored Procedures

CREATE PROCEDURE, parameters, control flow, error handling.

Simple stored procedure (input param)

```
CREATE PROCEDURE dbo.usp_GetCustomerAccounts
    @CustomerID INT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT a.AccountID, a.Balance, a.AccountType, b.BranchName
    FROM dbo.Accounts a
    JOIN dbo.Branches b ON a.BranchID = b.BranchID
    WHERE a.CustomerID = @CustomerID;
END;
GO

-- Execute
EXEC dbo.usp_GetCustomerAccounts @CustomerID = 1;
```

Procedure with output parameter

```
CREATE PROCEDURE dbo.usp_GetCustomerBalance
    @CustomerID INT,
    @TotalBalance MONEY OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT @TotalBalance = SUM(Balance)
    FROM dbo.Accounts
    WHERE CustomerID = @CustomerID;
END;
GO

-- Call it
DECLARE @bal MONEY;
EXEC dbo.usp_GetCustomerBalance @CustomerID = 1, @TotalBalance = @bal OUTPUT;
SELECT @bal AS CustomerTotalBalance;
```

Control flow (IF, WHILE) and TRY…CATCH error handling

```
CREATE PROCEDURE dbo.usp_DepositToAccount
    @AccountID BIGINT,
    @Amount MONEY
AS
BEGIN
    SET NOCOUNT ON;

    IF @Amount <= 0
    BEGIN
        THROW 50001, 'Amount must be greater than zero.', 1;
    END
```

```
    BEGIN TRY
        BEGIN TRAN;

        UPDATE dbo.Accounts
        SET Balance = Balance + @Amount
        WHERE AccountID = @AccountID;

        INSERT INTO dbo.Transactions(AccountID, Amount, TranType,
Description)
        VALUES (@AccountID, @Amount, 'D', 'Deposit');

        COMMIT TRAN;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRAN;

        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrNo INT = ERROR_NUMBER();
        THROW @ErrNo, @ErrMsg, 1;
    END CATCH
END;
GO
```

Best practices for stored procedures:

- Use `SET NOCOUNT ON`.
- Keep procedures focused (single responsibility).
- Validate inputs and use transactions where needed.
- Return meaningful error messages / codes.
- Avoid dynamic SQL where possible; if used, parameterize to prevent SQL injection.

---

3) User-Defined Functions (UDFs)

Scalar function (avoid for heavy row-by-row work — can be slow)

```
CREATE FUNCTION dbo.ufn_GetCustomerFullName(@CustomerID INT)
RETURNS NVARCHAR(101)
AS
BEGIN
    DECLARE @name NVARCHAR(101);
    SELECT @name = CONCAT(FirstName,' ',LastName) FROM dbo.Customers WHERE
CustomerID = @CustomerID;
    RETURN @name;
END;
GO

-- Usage
SELECT dbo.ufn_GetCustomerFullName(1) AS FullName;
```

Inline Table-Valued Function (preferred over scalar UDFs where possible)

```
CREATE FUNCTION dbo.ufn_GetAccountsByBranch(@BranchID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT AccountID, CustomerID, Balance
    FROM dbo.Accounts
    WHERE BranchID = @BranchID
);
GO

-- Usage
SELECT * FROM dbo.ufn_GetAccountsByBranch(1);
```

When to use vs avoid UDFs:

- Use inline TVF for set-based reusable logic — optimizer can inline them.
- Avoid scalar UDFs for large-scale row processing (they can be performance bottlenecks). Consider applying logic directly in queries or using inline TVFs.

---

4) Indexing

What is an index? Data structure (B-tree) to speed up lookups, ordering, and seek operations.

Clustered vs Non-Clustered

- **Clustered index:** determines physical order of rows in table. One per table.
- **Non-clustered index:** separate structure containing keys and row locators (pointer to clustered key or RID).

Create indexes examples

```
-- Create clustered index (if not already using PK clustered)
CREATE CLUSTERED INDEX IX_Accounts_AccountID ON dbo.Accounts(AccountID);

-- Create non-clustered index on CustomerID + Balance (useful for queries
filtered by CustomerID)
CREATE NONCLUSTERED INDEX IX_Accounts_CustomerID_Balance
ON dbo.Accounts(CustomerID)
INCLUDE (Balance);
```

Covering index example

If you frequently run:

```
SELECT AccountID, Balance FROM dbo.Accounts WHERE CustomerID = 1;
```

A nonclustered index on `CustomerID` that **INCLUDEs** `Balance` makes the index "cover" the query (no lookup required).

Index maintenance & considerations

- Indexes speed reads, but slow writes (INSERT/UPDATE/DELETE) due to maintenance cost.
- Avoid too many indexes on write-heavy tables.
- Use `STATISTICS` and `EXEC sp_updatestats` / maintenance plans for large DBs.
- Consider filtered indexes for sparse conditions (e.g., only active accounts).

```
CREATE NONCLUSTERED INDEX IX_Accounts_Active
ON dbo.Accounts(CustomerID)
WHERE IsActive = 1;
```

Example: Money transfer stored procedure (putting things together)

```
CREATE PROCEDURE dbo.usp_TransferFunds
    @FromAccount BIGINT,
    @ToAccount BIGINT,
    @Amount MONEY
AS
BEGIN
    SET NOCOUNT ON;

    IF @Amount <= 0
        THROW 50002, 'Transfer amount must be > 0', 1;

    BEGIN TRY
        BEGIN TRAN;

        -- Ensure sufficient balance (pessimistic locking)
        UPDATE dbo.Accounts
        SET Balance = Balance - @Amount
        WHERE AccountID = @FromAccount AND Balance >= @Amount;

        IF @@ROWCOUNT = 0
        BEGIN
            THROW 50003, 'Insufficient funds or invalid source account.', 1;
        END

        UPDATE dbo.Accounts
        SET Balance = Balance + @Amount
        WHERE AccountID = @ToAccount;
```

```
        IF @@ROWCOUNT = 0
        BEGIN
            THROW 50004, 'Destination account not found.', 1;
        END

        INSERT INTO dbo.Transactions (AccountID, Amount, TranType,
Description, RelatedAccountID)
        VALUES (@FromAccount, -@Amount, 'T', 'Transfer out', @ToAccount);

        INSERT INTO dbo.Transactions (AccountID, Amount, TranType,
Description, RelatedAccountID)
        VALUES (@ToAccount, @Amount, 'T', 'Transfer in', @FromAccount);

        COMMIT TRAN;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRAN;

        THROW;
    END CATCH
END;
GO
```