```
------------------------------------------------------------
      TOPIC : 1 DATABASE CREATION SCRIPTS
------------------------------------------------------------
```

```sql
/*============================================================
    DATABASE CREATION
    Syntax: CREATE DATABASE <DatabaseName>
============================================================*/
CREATE DATABASE TestDB;
GO


/*============================================================
    RENAME DATABASE
    Method 1: ALTER DATABASE (Recommended)
    Syntax: ALTER DATABASE <OldName> MODIFY NAME = <NewName>
============================================================*/
ALTER DATABASE TestDB MODIFY NAME = TestDatabase;
GO


/*============================================================
    RENAME DATABASE (Alternative Method)
    Using predefined stored procedure: sp_renamedb
    Syntax: EXEC sp_renamedb '<OldName>', '<NewName>'
============================================================*/
EXEC sp_renamedb 'TestDB', 'TestDatabase';
GO


/*============================================================
    IMPORTANT NOTE
    While renaming or dropping a database, SQL Server may
    block the operation if the MDF/LDF files are in use.
    To forcefully disconnect users:

    Set database to SINGLE_USER with ROLLBACK IMMEDIATE
============================================================*/

/* Put Database into SINGLE_USER Mode */
ALTER DATABASE TestDB SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
GO


/*============================================================
    DROP DATABASE
    Syntax: DROP DATABASE <DatabaseName>
============================================================*/
DROP DATABASE TestDB;
GO


/*============================================================
```

```
    After renaming a DB, set it back to MULTI_USER mode
    Syntax: ALTER DATABASE <DatabaseName> SET MULTI_USER
============================================================*/
ALTER DATABASE TestDatabase SET MULTI_USER;
GO


--Use Database Syntax: USE <DATA BASE NAME>
--Example
use TestDatabase;
```

TOPIC : 2 DDL QUERY

```
/*============================================================
    SQL SUB-TYPES
============================================================*/
-- 1) DDL (Data Definition Language) → CREATE, ALTER, DROP, TRUNCATE, sp_rename
-- 2) DML (Data Manipulation Language) → INSERT, UPDATE, DELETE
-- 3) DQL / DRL (Data Query / Retrieval Language) → SELECT
-- 4) TCL (Transaction Control Language) → COMMIT, ROLLBACK, SAVEPOINT
-- 5) DCL (Data Control Language) → GRANT, REVOKE

/*============================================================
    01. CREATE TABLE
============================================================*/
-- Syntax:
-- CREATE TABLE <TableName>
-- (
--      <ColumnName1> <DataType> [Size],
--      <ColumnName2> <DataType> [Size],
--      ...
-- );

-- Example:
CREATE TABLE Student
(
    StId    INT,
    SName   VARCHAR(MAX),
    Salary  DECIMAL(6,2)
);


 /*============================================================
    02. ALTER TABLE (MODIFY TABLE STRUCTURE)
============================================================*/

-- 1) Change the width (size) of an existing column
-- Syntax:
-- ALTER TABLE <TableName> ALTER COLUMN <ColumnName> <DataType>(NewSize);
ALTER TABLE Student ALTER COLUMN SName VARCHAR(100);
```

```sql
-- 2) Change data type of existing column
-- Syntax:
-- ALTER TABLE <TableName> ALTER COLUMN <ColumnName> <NewDataType>;
ALTER TABLE Student ALTER COLUMN SName NVARCHAR(100);


-- 3) Change NULL to NOT NULL (or vice versa)
-- Syntax:
-- ALTER TABLE <TableName> ALTER COLUMN <ColumnName> <DataType> NOT NULL;
-- ALTER TABLE <TableName> ALTER COLUMN <ColumnName> <DataType> NULL;
ALTER TABLE Student ALTER COLUMN StId INT NOT NULL;
ALTER TABLE Student ALTER COLUMN StId INT NULL;


-- 4) Add a new column to existing table
-- Syntax:
-- ALTER TABLE <TableName> ADD <ColumnName> <DataType>(Size);
ALTER TABLE Student ADD Branch VARCHAR(50);


-- 5) Delete (Drop) an existing column
-- Syntax:
-- ALTER TABLE <TableName> DROP COLUMN <ColumnName>;
ALTER TABLE Student DROP COLUMN Branch;


-- 6) Rename column or table using system stored procedure
-- Syntax:
-- EXEC sp_rename 'TableName.OldColumnName', 'NewColumnName';
-- EXEC sp_rename 'OldTableName', 'NewTableName';

EXEC sp_rename 'Student.SName', 'StudentName';


/*================================================================
    03. TRUNCATE TABLE
================================================================*/
-- Deletes ALL rows
-- Does NOT support WHERE clause
-- Structure remains
-- Syntax:
-- TRUNCATE TABLE <TableName>;
TRUNCATE TABLE Student;


/*================================================================
    04. DROP TABLE
================================================================*/
-- Permanently deletes the table from database
-- Syntax:
-- DROP TABLE <TableName>;
DROP TABLE Student;
```

TOPIC : 3 DDL QUERY EXAMPLE

```
create table international_teams
(
 team_id int,
 team_name varchar(50),
 team_rank int
)


select * from international_teams


insert into international_teams(team_id,team_name,team_rank)
Values(1,'India',1),
(2,'Australia',3),
(3,'England',2)


insert into international_teams values (4,'West Indians',5),(5,'South Africa',4)


alter table international_teams alter column team_name varchar(15)
alter table international_teams alter column team_name nvarchar(15)
alter table international_teams alter column team_name varchar(15) NOT NULL;
alter table international_teams alter column team_name NOT NULL;   -- Error
alter table international_teams alter column team_name nvarchar(20) NULL;
alter table international_teams add  column team_type varchar(50) NULL  -- Error
alter table international_teams add  team_type varchar(50) NOT NULL  -- Error
alter table international_teams add   team_type varchar(50) NULL  -- Correct
alter table international_teams drop column team_type

exec sp_rename 'international_teams','i_teams'
exec sp_rename 'i_teams.team_rank', 'world_rank'

truncate table i_teams

select * from i_teams

Drop table i_teams
```

TOPIC : 4 SQL DATA TYPES

```
/*=====================================================================
    SQL DATA TYPES – CLEAN NOTES
=====================================================================*/

-- 1) What are SQL data types?
```

```
--    SQL data types define what kind of data a column can store
--    (numbers, text, date/time, images, etc.).

/*=========================================================================
    1) INTEGER DATA TYPES
=========================================================================*/
-- Used to store whole numbers only (no decimal point).
-- Typical use: EmpId, ProductCode, BranchCode, etc.

-- TINYINT   : 1 byte   (0 to 255)
-- SMALLINT  : 2 bytes  (-32,768 to 32,767)
-- INT       : 4 bytes  (-2,147,483,648 to 2,147,483,647)
-- BIGINT    : 8 bytes  (very large integer range)

/*=========================================================================
    2) DECIMAL / NUMERIC DATA TYPES
=========================================================================*/
-- DECIMAL(p, s) and NUMERIC(p, s) are functionally the same.
-- p = Precision = total number of digits (both sides of the decimal).
-- s = Scale     = number of digits after the decimal point.

-- Example: 728.456
--          Total digits = 6  => Precision = 6
--          Digits after decimal = 3 => Scale = 3

-- Valid precision range : 1 to 38
-- Default (if not specified) often : DECIMAL(18, 0) or similar
-- (depends on SQL Server version, but idea: high precision integer)

/*=========================================================================
    3) MONEY DATA TYPES
=========================================================================*/
-- Used to store currency values.

-- SMALLMONEY : smaller range money type
-- MONEY      : larger range money type

-- Example: Salary, ProductPrice, InvoiceAmount, etc.

/*=========================================================================
    4) DATE / TIME DATA TYPES
=========================================================================*/
-- DATE      : Stores date only   =>  yyyy-mm-dd
-- TIME      : Stores time only   =>  hh:mm:ss[.fractional seconds]
-- DATETIME  : Stores date + time (older type in SQL Server)
-- (In newer SQL Server versions you also have DATETIME2, DATEONLY, etc.)

/*=========================================================================
    5) CHARACTER DATA TYPES
=========================================================================*/
-- Used to store letters, digits, and symbols (names, descriptions, etc.)

/* 5.1) Non-Unicode data types (single-byte per char) */
```

```
-- CHAR(size)        : Fixed-length (padded with spaces)
-- VARCHAR(size)     : Variable-length (up to given size)
-- VARCHAR(MAX)      : Variable-length, very large text
-- TEXT              : Old / deprecated (use VARCHAR(MAX) instead)

/* 5.2) Unicode data types (supports multiple languages) */
-- NCHAR(size)       : Fixed-length Unicode
-- NVARCHAR(size)    : Variable-length Unicode
-- NVARCHAR(MAX)     : Large Unicode text
-- NTEXT             : Old / deprecated (use NVARCHAR(MAX) instead)

-- NOTE: In SQL Server, Unicode string literals must be prefixed with N
-- Example:  N'', N'', N''

/*========================================================================
    6) BINARY DATA TYPES
========================================================================*/
-- Used to store raw binary data: images, audio, video, documents, etc.

-- BINARY(size)      : Fixed-length binary data
-- VARBINARY(size)   : Variable-length binary data
-- VARBINARY(MAX)    : Large binary data (e.g. file content)

/*========================================================================
    7) BOOLEAN DATA TYPE
========================================================================*/
-- In SQL Server, Boolean values are usually stored using BIT.
-- BIT : 0 = False, 1 = True, NULL = Unknown

/*========================================================================
    8) SPECIAL DATA TYPES (COMMON EXAMPLES)
========================================================================*/
-- UNIQUEIDENTIFIER : Stores a GUID.
-- XML              : Stores XML data.
-- GEOGRAPHY        : Spatial data (latitude/longitude etc.)
-- HIERARCHYID      : Represents hierarchical data (tree structures).

/************************************************************************
    REAL-TIME EXAMPLE TABLE USING THESE DATA TYPES
    (SQL Server style)
************************************************************************/

-- Drop table if exists (for re-run)
IF OBJECT_ID('dbo.EmployeeProfile', 'U') IS NOT NULL
    DROP TABLE dbo.EmployeeProfile;
GO

CREATE TABLE dbo.EmployeeProfile
(
    -- 1) Integer examples
    EmployeeId          INT IDENTITY(1,1)       NOT NULL,        -- Primary key
    DeptCode            SMALLINT                NULL,            -- Department code
```

```
    -- 2) Decimal / Numeric example
    BasicSalary         DECIMAL(18,2)           NOT NULL,        -- e.g. 50000.75

    -- 3) Money example
    Allowance           MONEY                   NULL,            -- e.g. 2000.00

    -- 4) Date / Time examples
    DateOfJoining       DATE                    NOT NULL,
    ShiftStartTime      TIME(0)                 NULL,
    LastLoginDateTime   DATETIME                NULL,

    -- 5) Character data types
    -- Non-Unicode (English only or limited charset)
    LoginUserName       VARCHAR(50)             NOT NULL,        -- Non-Unicode
    DepartmentShortName CHAR(5)                 NULL,            -- Fixed size code
    Remarks             VARCHAR(MAX)            NULL,            -- Long text (non-Unicode)

    -- Unicode (supports multiple languages)
    EmployeeFullName    NVARCHAR(100)           NOT NULL,        -- Unicode
    AddressLine         NVARCHAR(250)           NULL,            -- Unicode
    AboutEmployee       NVARCHAR(MAX)           NULL,            -- Large Unicode text

    -- 6) Binary data types
    ProfilePhoto        VARBINARY(MAX)          NULL,            -- Image file bytes
    SignatureBinary     VARBINARY(512)          NULL,            -- Small binary data

    -- 7) Boolean (Bit)
    IsActive            BIT                     NOT NULL
        CONSTRAINT DF_EmployeeProfile_IsActive DEFAULT(1),

    -- 8) Special Data Type
    RowGuid             UNIQUEIDENTIFIER        NOT NULL
        CONSTRAINT DF_EmployeeProfile_RowGuid DEFAULT NEWID(),

    CONSTRAINT PK_EmployeeProfile PRIMARY KEY (EmployeeId)
);
GO

/*********************************************************************
    INSERT EXAMPLES - NON-UNICODE vs UNICODE
*********************************************************************/

-- Example 1: Non-Unicode English data
INSERT INTO dbo.EmployeeProfile
(
    DeptCode,
    BasicSalary,
    Allowance,
    DateOfJoining,
    ShiftStartTime,
    LastLoginDateTime,
    LoginUserName,
    DepartmentShortName,
```

```
    Remarks,
    EmployeeFullName,
    AddressLine,
    AboutEmployee,
    ProfilePhoto,
    SignatureBinary,
    IsActive
)
VALUES
(
    10,                     -- DeptCode (SMALLINT)
    50000.75,               -- BasicSalary (DECIMAL(18,2))
    2500.00,                -- Allowance (MONEY)
    '2024-01-10',           -- DateOfJoining (DATE)
    '09:00:00',             -- ShiftStartTime (TIME)
    GETDATE(),              -- LastLoginDateTime (DATETIME)
    'john.doe',             -- LoginUserName (VARCHAR - Non-Unicode)
    'HR',                   -- DepartmentShortName (CHAR(5))
    'Regular full-time employee', -- Remarks (VARCHAR(MAX))
    N'John Doe',            -- EmployeeFullName (NVARCHAR - Unicode literal also ok)
    N'123, Main Street, Chennai', -- AddressLine (NVARCHAR)
    N'Good communication skills, works in HR.', -- AboutEmployee (NVARCHAR(MAX))
    NULL,                   -- ProfilePhoto (VARBINARY(MAX))
    NULL,                   -- SignatureBinary (VARBINARY)
    1                       -- IsActive (BIT)
);
GO

-- Example 2: Unicode data with local language (Tamil, Hindi, etc.)
INSERT INTO dbo.EmployeeProfile
(
    DeptCode,
    BasicSalary,
    Allowance,
    DateOfJoining,
    ShiftStartTime,
    LastLoginDateTime,
    LoginUserName,
    DepartmentShortName,
    Remarks,
    EmployeeFullName,
    AddressLine,
    AboutEmployee,
    ProfilePhoto,
    SignatureBinary,
    IsActive
)
VALUES
(
    20,
    65000.00,
    3000.00,
    '2024-02-15',
```

```
    '10:00:00',
    GETDATE(),
    't.shanmugam',                          -- Non-Unicode login (VARCHAR)
    'IT',
    'Handles core application development', -- Non-Unicode remark
    N' '              ,                      -- Unicode (Tamil)
    N', '             ,                     -- Unicode (Tamil address)
    N'   .'                                 , -- Unicode long text
    NULL,
    NULL,
    1
);
GO

-- Check inserted data
SELECT EmployeeId,
       LoginUserName,
       EmployeeFullName,
       AddressLine,
       BasicSalary,
       Allowance,
       IsActive
FROM dbo.EmployeeProfile;
GO

            ------------------------------------------------------------
                        TOPIC : 5 CONSTRIANTS
            ------------------------------------------------------------

/*============================================================
  TOPIC : SQL SERVER CONSTRAINTS (COMPLETE SYNTAX GUIDE)
  FILE  : constraints_practice.sql
============================================================*/


/*============================================================
  DEFAULT CONSTRAINT
============================================================*/

-- 1) DEFAULT during CREATE TABLE
CREATE TABLE car_default_create
(
    id INT,
    wheel_count SMALLINT DEFAULT 4
);

-- 2) DEFAULT using ALTER TABLE
CREATE TABLE car_default_alter
(
    id INT,
    wheel_count SMALLINT
);
```

```sql
ALTER TABLE car_default_alter
ADD CONSTRAINT df_wheel DEFAULT 4 FOR wheel_count;

DROP TABLE car_default_create;
DROP TABLE car_default_alter;


/*=========================================================
  NOT NULL CONSTRAINT
=========================================================*/

-- 1) NOT NULL during CREATE TABLE
CREATE TABLE notnull_create
(
    id INT,
    username VARCHAR(50) NOT NULL
);

-- 2) NOT NULL using ALTER TABLE
-- Works ONLY if column has no NULL values
ALTER TABLE notnull_create
ALTER COLUMN username VARCHAR(50) NOT NULL;

DROP TABLE notnull_create;


/*=========================================================
  NULL CONSTRAINT
=========================================================*/

-- 1) NULL during CREATE TABLE (DEFAULT behavior)
CREATE TABLE null_create
(
    id INT,
    remarks VARCHAR(100) NULL
);

-- 2) NULL using ALTER TABLE
ALTER TABLE null_create
ALTER COLUMN remarks VARCHAR(100) NULL;

DROP TABLE null_create;


/*=========================================================
  UNIQUE CONSTRAINT
=========================================================*/

-- UNIQUE during CREATE TABLE
CREATE TABLE unique_create
(
    id INT,
    remarks VARCHAR(100) UNIQUE
```

```
);

INSERT INTO unique_create VALUES (1, 'A');
-- INSERT INTO unique_create VALUES (2, 'A'); --  Error: duplicate value

DROP TABLE unique_create;


-- UNIQUE using ALTER TABLE
CREATE TABLE unique_create
(
    id INT,
    remarks VARCHAR(100)
);

-- Ensure NO duplicate values before adding UNIQUE
ALTER TABLE unique_create
ADD CONSTRAINT uq_unique_remarks UNIQUE (remarks);

DROP TABLE unique_create;


/*============================================================
  CHECK CONSTRAINT
============================================================*/

-- CHECK during CREATE TABLE
CREATE TABLE check_constraint
(
    id INT,
    age SMALLINT CONSTRAINT chk_age CHECK (age >= 18)
);

INSERT INTO check_constraint VALUES (1, 18);
INSERT INTO check_constraint VALUES (2, 20);
-- INSERT INTO check_constraint VALUES (3, 2); --  Error

DROP TABLE check_constraint;


-- CHECK using ALTER TABLE
CREATE TABLE check_constraint
(
    id INT,
    age SMALLINT
);

-- Ensure existing data satisfies condition
ALTER TABLE check_constraint
ADD CONSTRAINT chk_age CHECK (age >= 18);

DROP TABLE check_constraint;
```

```sql
/*=========================================================
  PRIMARY KEY CONSTRAINT
=========================================================*/

-- Correct PRIMARY KEY (only ONE per table)
CREATE TABLE branches
(
    bcode INT PRIMARY KEY,
    bname VARCHAR(40),
    bloc CHAR(6)
);

INSERT INTO branches VALUES (1, 'HDFC', '612001');
-- INSERT INTO branches VALUES (1, 'KVB', '612001'); --  Duplicate PK
-- INSERT INTO branches VALUES (NULL, 'IOB', '612002'); --  PK cannot be NULL

DROP TABLE branches;


-- PRIMARY KEY using ALTER TABLE
CREATE TABLE branches
(
    bcode INT NOT NULL,
    bname VARCHAR(40),
    bloc CHAR(6)
);

ALTER TABLE branches
ADD CONSTRAINT pk_branches PRIMARY KEY (bcode);

DROP TABLE branches;


-- COMPOSITE PRIMARY KEY
CREATE TABLE branches
(
    bcode INT,
    bname VARCHAR(40),
    bloc CHAR(6),
    CONSTRAINT pk_branch_comp PRIMARY KEY (bcode, bname)
);

INSERT INTO branches VALUES (1, 'HDFC', '612001'); --
INSERT INTO branches VALUES (1, 'KVB',  '612001'); --
-- INSERT INTO branches VALUES (1, 'HDFC', '612005'); --  Duplicate composite key

DROP TABLE branches;


/*=========================================================
  FOREIGN KEY CONSTRAINT
=========================================================*/
```

```sql
CREATE TABLE branches
(
    bcode INT PRIMARY KEY,
    bname VARCHAR(40)
);

CREATE TABLE bank
(
    id INT PRIMARY KEY,
    bc INT,
    income MONEY,
    CONSTRAINT fk_bank_branch
        FOREIGN KEY (bc) REFERENCES branches(bcode)
);

INSERT INTO branches VALUES (1, 'HDFC');
INSERT INTO bank VALUES (1, 1, 500000);
-- INSERT INTO bank VALUES (2, 5, 600000); --  No matching PK

DROP TABLE bank;
DROP TABLE branches;


/*============================================================
  FULL PRACTICE EXAMPLE
============================================================*/

DROP TABLE IF EXISTS carmodel;

CREATE TABLE carmodel
(
    id INT,
    car_brand VARCHAR(50),
    car_fuel_type VARCHAR(20),
    car_wheel SMALLINT DEFAULT 4,
    car_model VARCHAR(50) NOT NULL
);

INSERT INTO carmodel (id, car_brand, car_fuel_type, car_model)
VALUES
(1, 'TATA', 'PETROL', 'NEXON'),
(2, 'AUDI', 'PETROL', 'A6');

SELECT * FROM carmodel;


/*============================================================
  QUICK INTERVIEW SUMMARY
============================================================*/

-- DEFAULT
-- col datatype DEFAULT value
```

```
-- ALTER TABLE t ADD CONSTRAINT name DEFAULT value FOR col;

-- NOT NULL
-- col datatype NOT NULL
-- ALTER TABLE t ALTER COLUMN col datatype NOT NULL;

-- NULL
-- col datatype NULL
-- ALTER TABLE t ALTER COLUMN col datatype NULL;

-- UNIQUE
-- col datatype UNIQUE
-- ALTER TABLE t ADD CONSTRAINT name UNIQUE (col);

-- PRIMARY KEY
-- col datatype PRIMARY KEY
-- ALTER TABLE t ADD CONSTRAINT name PRIMARY KEY (col);

-- FOREIGN KEY
-- FOREIGN KEY (col) REFERENCES parent(col);
```

TOPIC : 6 PRIMARY KEY FOREIGN KEY RELATIONSHIP

```
/*=========================================================
  CUSTOMER TABLE
=========================================================*/

CREATE TABLE customer
(
    cid     INT PRIMARY KEY,
    cname   VARCHAR(30),
    cmobno  CHAR(10)
);

-- Use quotes for CHAR(10) mobile numbers
INSERT INTO customer VALUES (1, 'Thillai', '8675692630');
INSERT INTO customer VALUES (2, 'Senthil', '7598397824');
INSERT INTO customer VALUES (3, 'Tamil',   '9080323760');


/*=========================================================
  PRODUCTS TABLE
=========================================================*/

CREATE TABLE products
(
    pcode INT PRIMARY KEY,
    pname VARCHAR(40),
    price MONEY
);
```

```sql
INSERT INTO products VALUES (1, 'Camera',     24500);
INSERT INTO products VALUES (2, 'iPhone',    245000);
INSERT INTO products VALUES (3, 'Tab',        75500);
INSERT INTO products VALUES (4, 'PS5',        50000);
INSERT INTO products VALUES (5, 'HeadPhone',   2500);


/*=========================================================
  ORDERS TABLE
  (Relationship with TWO parent tables)
=========================================================*/

CREATE TABLE orders
(
    oid     INT PRIMARY KEY,
    ordate  DATETIME,
    quantity INT,

    cid INT,
    pcode INT,

    CONSTRAINT fk_orders_customer
        FOREIGN KEY (cid) REFERENCES customer(cid),

    CONSTRAINT fk_orders_product
        FOREIGN KEY (pcode) REFERENCES products(pcode)
);


/*=========================================================
  ADD CONSTRAINTS TO EXISTING TABLE
=========================================================*/

DROP TABLE IF EXISTS employee;

CREATE TABLE employee
(
    empid INT,
    ename VARCHAR(50),
    age   INT,
    pcode INT
);

-- 1) ADD PRIMARY KEY (column must be NOT NULL)
ALTER TABLE employee
ALTER COLUMN empid INT NOT NULL;

ALTER TABLE employee
ADD CONSTRAINT pk_employee PRIMARY KEY (empid);


-- 2) ADD UNIQUE CONSTRAINT
ALTER TABLE employee
```

```
ADD CONSTRAINT uq_employee_ename UNIQUE (ename);


-- 3) ADD CHECK CONSTRAINT
ALTER TABLE employee
ADD CONSTRAINT chk_employee_age CHECK (age >= 10);


-- 4) ADD FOREIGN KEY CONSTRAINT
ALTER TABLE employee
ADD CONSTRAINT fk_employee_pcode
FOREIGN KEY (pcode) REFERENCES products(pcode);


/*============================================================
  DROP CONSTRAINTS
============================================================*/

ALTER TABLE employee DROP CONSTRAINT chk_employee_age;
ALTER TABLE employee DROP CONSTRAINT uq_employee_ename;
```

TOPIC : 7 CASCADING

```
/*============================================================
 FILE : cascading_referential_integrity.sql
 DB   : SQL SERVER
 TOPIC: CASCADING REFERENTIAL INTEGRITY
 NOTE :
   • Cascading rules apply ONLY to FOREIGN KEY columns
   • Non-key columns can always be updated
============================================================*/


/*============================================================
  BASE TABLE : PRODUCTS (PARENT)
============================================================*/

DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS products;

CREATE TABLE products
(
    pcode INT PRIMARY KEY,
    pname VARCHAR(50),
    pcategory VARCHAR(50)
);

INSERT INTO products VALUES
(0,  'UNKNOWN PRODUCT',        'DEFAULT'),
(1,  'Samsung Galaxy S22',     'Mobile'),
(2,  'iPhone 15',              'Mobile'),
```

```
(3,  'Lenovo ThinkPad Gen3',    'Laptop'),
(4,  'Dell XPS 13',             'Laptop'),
(5,  'Sony VX100',              'Camera'),
(6,  'Canon EOS 1500D',         'Camera'),
(7,  'iPad Pro',                'Tablet'),
(8,  'Samsung Galaxy Tab',      'Tablet'),
(9,  'Apple Watch Series 9',    'Watch');

SELECT * FROM products;


/*============================================================
  IMPORTANT NOTE
============================================================
• Foreign Key protects ONLY:
      orders.pid  --->  products.pcode

• You CAN update:
      products.pname
      products.pcategory

• You CANNOT update/delete:
      products.pcode
  unless cascading rules are defined
============================================================*/


/*============================================================
  CASE 1 : NO ACTION (DEFAULT)
============================================================*/

DROP TABLE IF EXISTS orders;

CREATE TABLE orders
(
    oid INT PRIMARY KEY,
    ocname VARCHAR(50),
    odate DATETIME,
    pid INT,
    pcount INT,

    CONSTRAINT fk_orders_noaction
    FOREIGN KEY (pid) REFERENCES products(pcode)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
);

INSERT INTO orders VALUES
(1,'Thillai',GETDATE(),1,2),
(2,'Senthil',GETDATE(),2,1);

--  DELETE → ERROR
-- DELETE FROM products WHERE pcode = 1;
```

```sql
--  UPDATE → ERROR
-- UPDATE products SET pcode = 10 WHERE pcode = 2;

--  Allowed (non-key column)
UPDATE products SET pname = 'iPhone 15 Pro' WHERE pcode = 2;


/*============================================================
  CASE 2 : CASCADE
============================================================*/

DROP TABLE IF EXISTS orders;

CREATE TABLE orders
(
    oid INT PRIMARY KEY,
    ocname VARCHAR(50),
    odate DATETIME,
    pid INT,
    pcount INT,

    CONSTRAINT fk_orders_cascade
    FOREIGN KEY (pid) REFERENCES products(pcode)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);

INSERT INTO orders VALUES
(1,'Arun',GETDATE(),5,1);

--  UPDATE cascades
UPDATE products SET pcode = 50 WHERE pcode = 5;

-- OUTPUT:
-- orders.pid changes from 5 → 50

--  DELETE cascades
DELETE FROM products WHERE pcode = 50;

-- OUTPUT:
-- corresponding order row deleted


/*============================================================
  CASE 3 : SET NULL
============================================================*/

DROP TABLE IF EXISTS orders;

CREATE TABLE orders
(
    oid INT PRIMARY KEY,
```

```
    ocname VARCHAR(50),
    odate DATETIME,
    pid INT NULL,
    pcount INT,

    CONSTRAINT fk_orders_setnull
    FOREIGN KEY (pid) REFERENCES products(pcode)
    ON DELETE SET NULL
    ON UPDATE SET NULL
);

INSERT INTO orders VALUES
(1,'Tamil',GETDATE(),6,1);

--  UPDATE
UPDATE products SET pcode = 60 WHERE pcode = 6;

-- OUTPUT:
-- orders.pid = NULL

--  DELETE
DELETE FROM products WHERE pcode = 1;

-- OUTPUT:
-- orders.pid = NULL


/*=========================================================
  CASE 4 : SET DEFAULT
=========================================================*/

DROP TABLE IF EXISTS orders;

CREATE TABLE orders
(
    oid INT PRIMARY KEY,
    ocname VARCHAR(50),
    odate DATETIME,
    pid INT DEFAULT 0,
    pcount INT,

    CONSTRAINT fk_orders_setdefault
    FOREIGN KEY (pid) REFERENCES products(pcode)
    ON DELETE SET DEFAULT
    ON UPDATE SET DEFAULT
);

INSERT INTO orders VALUES
(1,'Kumar',GETDATE(),8,2);

--  UPDATE
UPDATE products SET pcode = 80 WHERE pcode = 8;
```

```
-- OUTPUT:
-- orders.pid = 0  (DEFAULT PRODUCT)

--  DELETE
DELETE FROM products WHERE pcode = 4;

-- OUTPUT:
-- orders.pid = 0


/*============================================================
  FINAL SUMMARY (INTERVIEW NOTE)
============================================================
• Cascading rules apply ONLY to FOREIGN KEY column
• Primary key non-referenced columns can be updated freely

RULE COMPARISON:
------------------------------------------------------------
NO ACTION   → Block update/delete
CASCADE     → Child follows parent
SET NULL    → FK becomes NULL
SET DEFAULT → FK becomes DEFAULT value
============================================================*/
```

TOPIC : 8 IDENTITY

```
/*============================================================
  FILE  : 07_identity_practice.sql
  TOPIC : SQL Server IDENTITY
  PURPOSE:
    - Understand IDENTITY
    - Auto increment behavior
    - Manual insert using IDENTITY_INSERT
    - Delete vs Truncate
    - Reset (RESEED)
    - Check identity values
============================================================*/


/*============================================================
  WHAT IS IDENTITY?
  ------------------------------------------------------------
  IDENTITY is used to auto-generate sequential numeric
  values for a column.
============================================================*/

/*
  SYNTAX:
  IDENTITY(seed, increment)

  seed      -> starting value
```

```
  increment -> step value

  Default: IDENTITY(1,1)
*/


/*=========================================================
  CREATE TABLE WITH IDENTITY
=========================================================*/

IF OBJECT_ID('employee') IS NOT NULL
    DROP TABLE employee;

CREATE TABLE employee
(
    empid   INT IDENTITY(1,1) PRIMARY KEY,
    empname VARCHAR(30)
);

--NOTE: IF ALREADY EXIST COLOUMN WAS NOT AN IDENTITY NOW WE WANT TO ADD IDENTITY
--THEN DROP THAT COLOUMN THEN ONLY ADD NEW COLOUMN WITH IDENTITY.

/*=========================================================
  INSERT DATA (AUTO GENERATED ID)
=========================================================*/

INSERT INTO employee VALUES
('Thillai'),
('Senthil'),
('Shanmugam'),
('Tamil');

SELECT * FROM employee;

/*
OUTPUT:
empid | empname
-----------------
1     | Thillai
2     | Senthil
3     | Shanmugam
4     | Tamil
*/


/*=========================================================
  MANUAL INSERT INTO IDENTITY (ERROR)
=========================================================*/

-- ERROR: Cannot insert explicit value for identity column
--INSERT INTO employee VALUES (5,'Dharshini');

--INSERT INTO employee(empid, empname)
```

```
--VALUES (5,'Dharshini');


/*=========================================================
  DELETE RECORD AND CHECK IDENTITY GAP
=========================================================*/

DELETE FROM employee WHERE empname = 'Senthil';

SELECT * FROM employee;

/*
OUTPUT:
empid | empname
-----------------
1     | Thillai
3     | Shanmugam
4     | Tamil
*/


/*=========================================================
  INSERT AFTER DELETE (IDENTITY NOT REUSED)
=========================================================*/

INSERT INTO employee VALUES ('Dharshini');

SELECT * FROM employee;

/*
OUTPUT:
empid | empname
-----------------
1     | Thillai
3     | Shanmugam
4     | Tamil
5     | Dharshini
*/


/*=========================================================
  ENABLE MANUAL IDENTITY INSERT
=========================================================*/

SET IDENTITY_INSERT employee ON;

INSERT INTO employee (empid, empname)
VALUES (2,'Senthil');

SELECT * FROM employee;

/*
OUTPUT:
```

```
empid | empname
-----------------
1     | Thillai
2     | Senthil
3     | Shanmugam
4     | Tamil
5     | Dharshini
*/


/*============================================================
  DUPLICATE IDENTITY VALUE (PRIMARY KEY ERROR)
============================================================*/

-- ERROR: Violation of PRIMARY KEY constraint
--INSERT INTO employee (empid, empname)
--VALUES (2,'Duplicate');

SET IDENTITY_INSERT employee OFF;


/*============================================================
  DELETE VS IDENTITY RESET
============================================================*/

DELETE FROM employee;

INSERT INTO employee VALUES ('ABI');

SELECT * FROM employee;

/*
OUTPUT:
empid | empname
-----------------
6     | ABI
NOTE:
DELETE does NOT reset IDENTITY
*/


/*============================================================
  RESET IDENTITY USING DBCC CHECKIDENT
============================================================*/

DBCC CHECKIDENT (employee, RESEED, 0);

INSERT INTO employee VALUES ('DHARSHI');

SELECT * FROM employee;

/*
OUTPUT:
```

```
empid | empname
-----------------
1     | DHARSHI
*/


/*============================================================
  TRUNCATE TABLE (BEST WAY TO RESET IDENTITY)
============================================================*/

TRUNCATE TABLE employee;

INSERT INTO employee VALUES ('ARUN');

SELECT * FROM employee;

/*
OUTPUT:
empid | empname
-----------------
1     | ARUN
*/


/*============================================================
  CHECK IDENTITY VALUES
============================================================*/

-- Last identity value in current scope (SAME TABLE, SAME PROCEDURE..) (SAFE)
SELECT SCOPE_IDENTITY() AS ScopeIdentity;

-- Last identity value in current session (NOT SAFE)
SELECT @@IDENTITY AS SessionIdentity;

-- Current identity value of table
-- DANGEOURS WHEN TWO PEPOPLE ARE INSERT ON SAME TIME
SELECT IDENT_CURRENT('employee') AS CurrentIdentity;


/*============================================================
  INTERVIEW NOTES
  ------------------------------------------------------------
  1) IDENTITY values are not reused
  2) DELETE does NOT reset identity
  3) TRUNCATE resets identity
  4) SCOPE_IDENTITY() is recommended
  5) IDENTITY_INSERT can be ON for only one table
============================================================*/
```

TOPIC : 9 SEQUENCE PRACTICE

```
/*============================================================
  FILE  : 08_sequence_practice.sql
  TOPIC : SQL Server SEQUENCE
============================================================*/


/*============================================================
  IDENTITY vs SEQUENCE (DETAILED COMPARISON)
============================================================*/

/*
------------------------------------------------------------
FEATURE                     IDENTITY                SEQUENCE
------------------------------------------------------------
Object Type                  Column Property         Database Object
------------------------------------------------------------
Table Dependency             Dependent on table      Independent object
------------------------------------------------------------
Value Generation             On INSERT only          On demand (NEXT VALUE)
------------------------------------------------------------
Multiple Table Usage          Not possible           Possible
------------------------------------------------------------
Manual Control                Limited                Full control
------------------------------------------------------------
Insert Explicit Value         (Default)              Always allowed
------------------------------------------------------------
Reset Value                  DBCC / TRUNCATE         ALTER SEQUENCE
------------------------------------------------------------
Delete Impact                Gaps remain             Gaps remain
------------------------------------------------------------
Cycle Support                 No                     Yes
------------------------------------------------------------
Cache Support                 No                     Yes
------------------------------------------------------------
Generate Before Insert        No                     Yes
------------------------------------------------------------
Performance                  Good                    Better (with CACHE)
------------------------------------------------------------
Use Case                     Simple PK               Complex numbering
------------------------------------------------------------
*/


/*============================================================
  REAL-TIME SCENARIO COMPARISON
============================================================*/

/*
CASE 1: SIMPLE EMPLOYEE TABLE
```

```
----------------------------
Requirement:
- Only one table
- Auto increment primary key
- No reuse needed

Best Choice: IDENTITY


CASE 2: MULTIPLE TABLES NEED SAME NUMBER SERIES
-----------------------------------------------
Tables:
- sales_order
- purchase_order
- return_order

Requirement:
- One common running number
- Generated before insert
- Full control

Best Choice: SEQUENCE


CASE 3: BUSINESS DOCUMENT NUMBER
--------------------------------
Requirement:
- Invoice number should restart every year
- Controlled reset
- Can be reused in many tables

Best Choice: SEQUENCE
*/


/*============================================================
  IDENTITY EXAMPLE (QUICK)
============================================================*/

IF OBJECT_ID('emp_identity') IS NOT NULL
    DROP TABLE emp_identity;

CREATE TABLE emp_identity
(
    empid INT IDENTITY(1,1) PRIMARY KEY,
    empname VARCHAR(30)
);

INSERT INTO emp_identity VALUES ('A'), ('B'), ('C');

SELECT * FROM emp_identity;

/*
```

```
OUTPUT:
empid | empname
----------------
1     | A
2     | B
3     | C
*/


/*=========================================================
  SEQUENCE EXAMPLE (QUICK)
=========================================================*/

IF OBJECT_ID('EMP_SEQ', 'SO') IS NOT NULL
    DROP SEQUENCE EMP_SEQ;

CREATE SEQUENCE EMP_SEQ
AS INT
START WITH 1
INCREMENT BY 1
NO CACHE;

IF OBJECT_ID('emp_sequence') IS NOT NULL
    DROP TABLE emp_sequence;

CREATE TABLE emp_sequence
(
    empid INT PRIMARY KEY,
    empname VARCHAR(30)
);

INSERT INTO emp_sequence
VALUES (NEXT VALUE FOR EMP_SEQ, 'A'),
       (NEXT VALUE FOR EMP_SEQ, 'B'),
       (NEXT VALUE FOR EMP_SEQ, 'C');

SELECT * FROM emp_sequence;

/*
OUTPUT:
empid | empname
----------------
1     | A
2     | B
3     | C
*/


/*=========================================================
  INTERVIEW ONE-LINERS
=========================================================*/

/*
```

```
1) IDENTITY is automatic but inflexible
2) SEQUENCE is manual but powerful
3) Use IDENTITY for simple tables
4) Use SEQUENCE for business numbering
5) SEQUENCE works outside INSERT also
6) SEQUENCE supports CACHE and CYCLE
*/


/*============================================================
  END OF FILE
============================================================*/
```

TOPIC : 10 HAVING

```
/*============================================================
  TOPIC NAME : HAVING CLAUSE
  FILE NAME  : HAVING_CONCEPT.sql
  DATABASE   : SQL SERVER
============================================================*/

/*============================================================
  1 WHAT IS HAVING?
------------------------------------------------------------
  HAVING clause is used to filter GROUPED data.
  It works AFTER GROUP BY.
  It is mainly used with aggregate functions.
============================================================*/

-- Simple definition:
-- WHERE  → filters rows
-- HAVING → filters groups (aggregate result)

/*============================================================
  2 FLOW OF QUERY CLAUSES (EXECUTION ORDER)
------------------------------------------------------------
  SQL does NOT execute in written order.
  Logical execution order is:

  1. FROM
  2. WHERE
  3. GROUP BY
  4. HAVING
  5. SELECT
  6. ORDER BY
============================================================*/

/*============================================================
  3 CREATE SAMPLE TABLE
============================================================*/
```

```sql
DROP TABLE IF EXISTS Employee;
GO

CREATE TABLE Employee
(
    EmpId   INT PRIMARY KEY,
    EmpName VARCHAR(50),
    City    VARCHAR(20),
    Salary  INT
);
GO

/*============================================================
  4 INSERT SAMPLE DATA
============================================================*/

INSERT INTO Employee VALUES
(1, 'Arun',   'MUMBAI',   20000),
(2, 'Bala',   'MUMBAI',   30000),
(3, 'Chandru','MUMBAI',   25000),
(4, 'David',  'DELHI',    15000),
(5, 'Ezhil',  'DELHI',    20000),
(6, 'Fazil',  'CHENNAI',  18000),
(7, 'Ganesh', 'CHENNAI',  12000);
GO

SELECT * FROM Employee;
GO

/*============================================================
  5 GROUP BY WITHOUT HAVING
============================================================*/

SELECT City, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY City;
GO

/*============================================================
  6 HAVING WITH AGGREGATE FUNCTIONS
============================================================*/

-- 6.1 SUM()
-- Cities where total salary > 60000
SELECT City, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY City
HAVING SUM(Salary) > 60000;
GO


-- 6.2 COUNT()
-- Cities having more than 2 employees
```

```sql
SELECT City, COUNT(*) AS EmpCount
FROM Employee
GROUP BY City
HAVING COUNT(*) > 2;
GO

-- 6.3 AVG()
-- Cities where average salary >= 20000
SELECT City, AVG(Salary) AS AvgSalary
FROM Employee
GROUP BY City
HAVING AVG(Salary) >= 20000;
GO

-- 6.4 MAX()
-- Cities where maximum salary >= 30000
SELECT City, MAX(Salary) AS MaxSalary
FROM Employee
GROUP BY City
HAVING MAX(Salary) >= 30000;
GO

-- 6.5 MIN()
-- Cities where minimum salary >= 15000
SELECT City, MIN(Salary) AS MinSalary
FROM Employee
GROUP BY City
HAVING MIN(Salary) >= 15000;
GO

/*============================================================
  7 WHY CANNOT USE WHERE WITH AGGREGATES?
============================================================*/

--  INVALID QUERY (ERROR)
-- WHERE executes BEFORE GROUP BY
-- Aggregate functions do NOT exist yet

/*
SELECT City, SUM(Salary)
FROM Employee
WHERE SUM(Salary) > 60000
GROUP BY City;
*/

--  CORRECT WAY USING HAVING

SELECT City, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY City
HAVING SUM(Salary) > 60000;
GO
```

```
/*=========================================================
  8 WHERE + HAVING TOGETHER (BEST PRACTICE)
=========================================================*/

-- Filter rows first using WHERE (FAST)
-- Then filter groups using HAVING

SELECT City, SUM(Salary) AS TotalSalary
FROM Employee
WHERE Salary >= 20000
GROUP BY City
HAVING SUM(Salary) > 50000;
GO

/*=========================================================
  9 IMPORTANT INTERVIEW POINTS
=========================================================*/

-- 1) WHERE cannot use aggregate functions
-- 2) HAVING must be used with GROUP BY (mostly)
-- 3) WHERE improves performance by filtering early
-- 4) HAVING is only for aggregated conditions

--WHERE condition use panninaa
--original (row-level) data-va filter pannalaam

--HAVING use panninaa
--aggregate result-a filter pannalaam
/*=========================================================
  END OF FILE
=========================================================*/
```

TOPIC : 11 DW WHERE HAVING

```
/*=========================================================
 FILE NAME  : Where_vs_Having.sql
 TOPIC      : Difference Between WHERE and HAVING Clause
 DATABASE   : SQL Server
 PURPOSE    :
   - Understand WHERE vs HAVING
   - Learn execution order
   - Learn performance impact
   - Practice interview-based examples
=========================================================*/


/*=========================================================
 STEP 1: CREATE SAMPLE TABLE
=========================================================*/

DROP TABLE IF EXISTS Sales;
```

```
GO

CREATE TABLE Sales
(
    Product     NVARCHAR(50),
    SaleAmount  INT
);
GO


/*============================================================
 STEP 2: INSERT SAMPLE DATA
============================================================*/

INSERT INTO Sales VALUES ('iPhone',   500);
INSERT INTO Sales VALUES ('Laptop',   800);
INSERT INTO Sales VALUES ('iPhone',  1000);
INSERT INTO Sales VALUES ('Speakers', 400);
INSERT INTO Sales VALUES ('Laptop',   600);
GO


/*============================================================
 VIEW DATA
============================================================*/
SELECT * FROM Sales;
GO


/*============================================================
 EXAMPLE 1: TOTAL SALES BY PRODUCT (GROUP BY)
============================================================*/

SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product;
GO

/*
 OUTPUT:
 --------------------------------
 Product     | TotalSales
 --------------------------------
 iPhone      | 1500
 Laptop      | 1400
 Speakers    | 400
*/


/*============================================================
 EXAMPLE 2: FILTER GROUPS USING HAVING
```

```
 Find products where total sales > 1000
============================================================*/

SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING SUM(SaleAmount) > 1000;
GO

/*
 OUTPUT:
 -------------------------
 Product  | TotalSales
 -------------------------
 iPhone   | 1500
 Laptop   | 1400
*/


/*============================================================
 EXAMPLE 3: INVALID USE OF WHERE WITH AGGREGATE
 (THIS WILL FAIL)
============================================================*/

--  WRONG QUERY
/*
SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
WHERE SUM(SaleAmount) > 1000;
*/

--  ERROR:
-- Incorrect syntax near the keyword 'WHERE'
-- Reason: WHERE cannot be used with aggregate functions


/*============================================================
 KEY RULE:
 WHERE  → Filters ROWS (before aggregation)
 HAVING → Filters GROUPS (after aggregation)
============================================================*/


/*============================================================
 EXAMPLE 4: USING WHERE (FILTER FIRST, THEN AGGREGATE)
 Calculate total sales of iPhone and Speakers
============================================================*/
```

```sql
SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE Product IN ('iPhone', 'Speakers')
GROUP BY Product;
GO

/*
 EXECUTION FLOW:
 1) WHERE filters rows
 2) GROUP BY groups rows
 3) SUM calculates totals

 PERFORMANCE: FAST
*/


/*============================================================
 EXAMPLE 5: USING HAVING (AGGREGATE FIRST, THEN FILTER)
 Calculate total sales of iPhone and Speakers
============================================================*/

SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING Product IN ('iPhone', 'Speakers');
GO

/*
 EXECUTION FLOW:
 1) All rows selected
 2) GROUP BY applied
 3) SUM calculated
 4) HAVING filters groups

 PERFORMANCE: SLOWER
*/


/*============================================================
 BEST PRACTICE:
 Always use WHERE instead of HAVING
 when aggregate condition is NOT required
============================================================*/


/*============================================================
 WHERE + HAVING TOGETHER (REAL INTERVIEW QUESTION)
============================================================*/
```

```sql
SELECT
    Product,
    SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE SaleAmount >= 500          -- row-level filter
GROUP BY Product
HAVING SUM(SaleAmount) > 1000;   -- group-level filter
GO

/*
 EXECUTION ORDER:
 1) FROM
 2) WHERE
 3) GROUP BY
 4) HAVING
 5) SELECT
*/


/*============================================================
 FINAL DIFFERENCE SUMMARY (INTERVIEW READY)
============================================================*/

-- 1) WHERE cannot be used with aggregate functions
--    HAVING can be used with aggregate functions

-- 2) WHERE filters individual rows
--    HAVING filters grouped data

-- 3) WHERE executes BEFORE GROUP BY
--    HAVING executes AFTER GROUP BY

-- 4) WHERE is faster than HAVING

-- 5) WHERE can be used with:
--    SELECT, INSERT, UPDATE, DELETE

-- 6) HAVING can be used ONLY with SELECT


/*============================================================
 END OF FILE
============================================================*/
```