**Question 1:**

Readers' association of a university is planning for a literary event. They plan to divide the books in their library, according to their number of copies sold, which signifies the popularity of the books. The data for these books is stored in the table BookDetails having the columns BookId, BookName and CopiesSold.

The books are to be divided into 5 groups.

Identify which of the following query is a valid syntax to achieve the above requirement.

**Assumption:**

1. BookDetails is a valid table that exists in the database.
2. SELECT NTILE(5, BookId), BookName, CopiesSold OVER (ORDER BY CopiesSold Desc) AS GroupNo FROM BookDetails
3. SELECT BookId, BookName, CopiesSold NTILE(5) OVER (ORDER BY CopiesSold Desc) AS GroupNo FROM BookDetails
4. SELECT NTILE(5) AS GroupNo, BookId, BookName, CopiesSold FROM BookDetails
5. SELECT BookId, BookName, CopiesSold NTILE(5) OVER CopiesSold AS GroupNo FROM BookDetails

**🧠 Understanding the Concept – NTILE() 📚📊**

NTILE(n) is a **ranking/window function** that splits the result set into n groups **based on ordering**.

✅ It’s typically used for:

* Dividing sales data into quartiles/deciles
* Grouping customers by order amount
* Bucketing records into percentile ranges

**✅ Syntax:**

NTILE(n) OVER (ORDER BY column)

**🎯 Step-by-Step Analysis 🧩✨**

**📌 Step 1: Identify what the query must do**

* Group books into **5 sets** based on descending popularity (i.e., CopiesSold DESC)
* Use a **valid window function syntax**

**📌 Step 2: Analyze Each Option**

🔹 **Option a – ❌ Invalid**

SELECT NTILE(5, BookId), BookName, CopiesSold OVER (ORDER BY CopiesSold DESC) AS GroupNo FROM BookDetails

* ❌ NTILE(5, BookId) is **invalid syntax** — NTILE only takes 1 argument (number of groups).
* ❌ Also, CopiesSold OVER (...) is not valid on its own.

🔹 **Option b – ✅ Correct**

SELECT BookId, BookName, CopiesSold, NTILE(5) OVER (ORDER BY CopiesSold DESC) AS GroupNo FROM BookDetails

* ✅ Proper syntax of NTILE: one argument + valid OVER clause
* ✅ Orders by CopiesSold DESC
* ✅ Returns group numbers as GroupNo

🔹 **Option c – ❌ Invalid**

SELECT NTILE(5) AS GroupNo, BookId, BookName, CopiesSold FROM BookDetails

* ❌ Missing OVER (ORDER BY ...) clause, which is **mandatory** with NTILE.

🔹 **Option d – ❌ Invalid**

SELECT BookId, BookName, CopiesSold, NTILE(5) OVER CopiesSold AS GroupNo FROM BookDetails

* ❌ OVER CopiesSold is **incorrect syntax**. Needs to be OVER (ORDER BY CopiesSold).

**📝 Final Output: 📢**

The only syntactically and logically correct query is:

**Option (b)**

SELECT BookId, BookName, CopiesSold, NTILE(5) OVER (ORDER BY CopiesSold DESC) AS GroupNo FROM BookDetails

**🕵️ Matching with Given Options: 🎲🔍**

|  |  |  |
| --- | --- | --- |
| **Option** | **Validity** | **Explanation** |
| a | ❌ | Invalid use of NTILE with 2 arguments |
| b | ✅ | Correct syntax and logic |
| c | ❌ | Missing OVER (ORDER BY ...) |
| d | ❌ | Invalid OVER clause syntax |

**✅ Correct Answer: (b)**

**📌 Exam Tip 💡:**

NTILE(n) OVER (ORDER BY column) is a powerful way to **bucket or group records**.  
Use it when you need **percentile rankings** or need to evenly **distribute rows into groups** for analytics!

**Question 2:**

You are developing a SQL Server 2012 database for a travel application. You need to design tables and other database objects. The requirement is to store media files in tables where each media file is less than 1 MB in size. The media files require fast access and will be retrieved frequently.

Which activity is most suitable for this requirement?

**Options:**

1. Using the VARBINARY datatype
2. Using the DATETIME2 datatype
3. Using the NVARCHAR datatype
4. Using the CAST function
5. Using the DATETIMEOFFSET datatype

**🧠 Understanding the Concepts – Storing Media Files 📸🎵🎬**

SQL Server supports **Binary Large Object (BLOB)** storage using the VARBINARY data type.

When media files (like images, videos, or PDFs) are relatively **small in size** and need **frequent access**, they can be stored **directly in the table** using VARBINARY(MAX).

**🎯 Step-by-Step Analysis 🧩✨**

**📌 Step 1: What do we need?**

* **Store media (binary) files** ✅
* **Each file is < 1 MB** ✅
* **Fast access (inline)** ✅

👉 The ideal data type in SQL Server for storing binary files up to 2 GB is VARBINARY(MAX)  
👉 For files less than 1 MB, storing them **in-row** provides **fast retrieval** performance

**📌 Step 2: Evaluate Each Option**

🔹 **Option a – ✅ Correct**

* VARBINARY is used to store **binary data**, including **images, documents, audio/video**, etc.
* Since files are <1 MB, they can be **stored inline (in-row)** in the table
* ✅ Suitable for **fast access**
* ✅ Perfect for **frequent retrieval**

🔹 **Option b – ❌ Incorrect**

* DATETIME2 is used for **storing date and time values** with precision.
* ❌ Not suitable for storing media files.

🔹 **Option c – ❌ Incorrect**

* NVARCHAR is used to store **Unicode character data (text)**.
* ❌ Not appropriate for binary files like images or videos.

🔹 **Option d – ❌ Incorrect**

* CAST() is a function, not a **data type or storage solution**.
* Used to convert data types (e.g., from INT to VARCHAR), not to store binary media.

🔹 **Option e – ❌ Incorrect**

* DATETIMEOFFSET is for storing **date-time with timezone info**.
* ❌ Not related to file storage.

**📝 Final Output: 📢**

The most suitable activity for storing small, frequently accessed media files is:

**Option (a) Using the VARBINARY datatype**

**🕵️ Matching with Given Options: 🎲🔍**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Usage** | **Suitable?** | **Reason** |
| a | ✅ Binary data (media, files) | ✅ Yes | Best for storing files <1MB |
| b | ❌ Date/Time values | ❌ No | Not binary storage |
| c | ❌ Unicode text | ❌ No | Not for binary data |
| d | ❌ Type conversion function | ❌ No | Not related to storage |
| e | ❌ DateTime with time zone | ❌ No | Not for files |

**✅ Correct Answer: (a) Using the VARBINARY datatype**

**📌 Exam Tip 💡:**

Use VARBINARY(MAX) to store **binary files up to 2 GB**.  
For files under **1 MB**, SQL Server stores them **in-row**, giving **fast access** and **performance** benefits.

**Question 3:**

Choose the most appropriate datatype for the following data stored in a Hospital database:

a. **PatientId** – stores a value to identify a patient uniquely, such as P1001, P1002, P1003 etc.  
b. **PatientName** – stores the name of the patient. It must allow storage of names in English, Hindi, Chinese and a few other languages.  
c. **PatientAge** – stores the age of the patient  
d. **PatientReports** – stores the complete reports and medical data associated with the patient  
e. **PatientBillAmount** – stores the bill amount of the patient, which includes the lab test charges.

**Datatypes:**

i. TINYINT  
ii. VARCHAR(MAX)  
iii. CHAR(5)  
iv. DECIMAL(15,2)  
v. NVARCHAR(50)

**Options:**

1. a-iii, b-ii, c-i, d-v, e-iv
2. a-v, b-iii, c-i, d-iv, e-ii
3. a-i, b-ii, c-iii, d-v, e-iv
4. a-i, b-v, c-iii, d-ii, e-iv
5. a-iii, b-v, c-i, d-i, e-iv

**🧠 Concept: Choosing Appropriate Data Types in SQL Server**

Choosing the right data types ensures efficient storage, accurate data representation, and performance optimization. Each column in a database table should use the data type that best fits the nature and size of the data it stores.

**🎯 Step-by-Step Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Data Requirement** | **Best Data Type** | **Reason** |
| **PatientId** | Stores values like P1001, P1002 (fixed-length alphanumeric) | CHAR(5) | Fixed-length storage, ideal for uniform ID formats |
| **PatientName** | Multilingual names (English, Hindi, Chinese, etc.) | NVARCHAR(50) | Unicode support for multilingual text |
| **PatientAge** | Age (small range of integers, 0–120) | TINYINT | Efficient for small numbers (range: 0–255) |
| **PatientReports** | Complete reports and medical data (large text) | VARCHAR(MAX) | Optimized for large and variable-length text data |
| **PatientBillAmount** | Currency values including decimals | DECIMAL(15,2) | Accurate monetary calculations with two decimal precision |

**🕵️ Option Mapping and Validation**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Option** | **a (PatientId)** | **b (PatientName)** | **c (PatientAge)** | **d (PatientReports)** | **e (PatientBillAmount)** | **Valid?** |
| a | CHAR(5) | VARCHAR(MAX) | TINYINT | NVARCHAR(50) | DECIMAL(15,2) | ❌ d should be VARCHAR(MAX) |
| b | NVARCHAR(50) | CHAR(5) | TINYINT | DECIMAL(15,2) | VARCHAR(MAX) | ❌ b should not use CHAR(5) for names |
| c | TINYINT | VARCHAR(MAX) | CHAR(5) | NVARCHAR(50) | DECIMAL(15,2) | ❌ a and d are mismatched |
| d | TINYINT | NVARCHAR(50) | CHAR(5) | VARCHAR(MAX) | DECIMAL(15,2) | ✅ All correct data type mappings |
| e | CHAR(5) | NVARCHAR(50) | TINYINT | TINYINT | DECIMAL(15,2) | ❌ d is incorrect (TINYINT not suitable for reports) |

**✅ Correct Answer: d) a-i, b-v, c-iii, d-ii, e-iv**

**📌 Key Points to Remember**

* Use CHAR(n) for fixed-format identifiers like codes or IDs.
* Use NVARCHAR(n) for multilingual text requiring Unicode support.
* Use TINYINT when storing values from 0 to 255.
* Use VARCHAR(MAX) for storing large or unbounded text like medical reports.
* Use DECIMAL(p,s) for precise financial data where p is total digits and s is digits after the decimal point.

**Question 4:**

Consider the following stored procedure:

CREATE PROCEDURE usp\_ConcatenateWords

(

@WordOne VARCHAR(10) = 'Hello ',

@WordTwo VARCHAR(10)

)

AS

BEGIN

SELECT @WordOne + @WordTwo

END

What will be the output if the following code is executed?

EXEC usp\_ConcatenateWords 7, 'stars'

**Options:**

1. 7 stars
2. Hello stars
3. Hello 7 stars
4. Error - cannot convert INT to VARCHAR

**🧠 Concept: SQL Server Stored Procedures and Implicit Data Type Conversion**

In SQL Server, when values are passed to a stored procedure, SQL attempts to **implicitly convert** them to the expected parameter data types. If the conversion is valid, execution continues; otherwise, an error is raised.

The procedure in this case has:

* @WordOne as VARCHAR(10) with a **default value** of 'Hello '
* @WordTwo as VARCHAR(10) (no default, must be supplied)

**🎯 Step-by-Step Analysis**

**Procedure Definition:**

CREATE PROCEDURE usp\_ConcatenateWords

(

@WordOne VARCHAR(10) = 'Hello ',

@WordTwo VARCHAR(10)

)

AS

BEGIN

SELECT @WordOne + @WordTwo

END

**Execution Statement:**

EXEC usp\_ConcatenateWords 7, 'stars'

**📌 Parameter Mapping:**

SQL Server matches parameters **by position**, not by name (unless specified explicitly).  
So:

|  |  |  |
| --- | --- | --- |
| **Position** | **Passed Value** | **Parameter** |
| 1 | 7 | @WordOne |
| 2 | 'stars' | @WordTwo |

* @WordOne gets the value 7, which is **implicitly converted to VARCHAR(10)** ➝ becomes '7'
* @WordTwo is assigned 'stars'

**Concatenation Result:**

'7' + 'stars' = '7stars'

This is valid in SQL Server and no error occurs.

**✅ Correct Answer: a) 7 stars**

**📌 Key Points to Remember**

* SQL Server allows **implicit conversion** from INT to VARCHAR when needed in string operations.
* When using EXEC without specifying parameter names, **positional mapping** is applied.
* Default parameter values are **used only when no argument is passed** for that parameter.
* String concatenation with + works when all operands are strings or convertible to strings.

**Question 5:**

Consider the following stored procedure:

CREATE PROCEDURE usp\_ConcatenateNum

(

@NumOne INT = 1,

@NumTwo INT,

@NumThree INT = NULL

)

AS

BEGIN

SET CONCAT\_NULL\_YIELDS\_NULL OFF

SELECT @NumOne + @NumTwo + @NumThree

END

What will be the output if the following code is executed?

EXEC usp\_ConcatenateNum 10, 2

**Options:**

1. 12
2. 13
3. 3
4. NULL

**🧠 Concept: CONCAT\_NULL\_YIELDS\_NULL Setting in SQL Server**

* This setting controls how **NULL values behave in string concatenation**.
* **Important:** It has **no effect on numeric addition** or arithmetic operations.
* In arithmetic, **any operation involving NULL returns NULL**, unless ISNULL() or COALESCE() is used.

**🎯 Step-by-Step Analysis**

**Stored Procedure:**

CREATE PROCEDURE usp\_ConcatenateNum

(

@NumOne INT = 1,

@NumTwo INT,

@NumThree INT = NULL

)

AS

BEGIN

SET CONCAT\_NULL\_YIELDS\_NULL OFF

SELECT @NumOne + @NumTwo + @NumThree

END

**Execution:**

EXEC usp\_ConcatenateNum 10, 2

**📌 Parameter Mapping:**

| **Parameter** | **Value Passed** | **Final Value** |
| --- | --- | --- |
| @NumOne | 10 | 10 |
| @NumTwo | 2 | 2 |
| @NumThree | not passed | NULL |

**Calculation:**

10 + 2 + NULL → NULL

* In **numeric expressions**, NULL **does not act like 0**.
* The presence of NULL in an arithmetic operation leads to a NULL result.
* The SET CONCAT\_NULL\_YIELDS\_NULL OFF has **no effect** here since it's **only relevant to string concatenation**, not numeric operations.

**✅ Correct Answer: d) NULL**

**📌 Key Points to Remember**

* Arithmetic with NULL returns NULL.
* Use ISNULL(value, 0) or COALESCE(value, 0) to treat NULL as 0 if needed.
* SET CONCAT\_NULL\_YIELDS\_NULL applies **only to string concatenation**.
* SQL parameters with default = NULL get NULL if no value is passed.

**Question 6:**

Which of the following system stored procedure reports information about database-level principals in the current database?

1. SP\_HELP
2. SP\_HELPDB
3. SP\_HELPTEXT
4. SP\_HELPUSER

**🧠 Concept: SQL Server System Stored Procedures for Metadata**

SQL Server provides several system stored procedures to retrieve **metadata** and **informational details** about databases, objects, and users.

**Database-level principals** refer to **users, roles, and permissions specific to a particular database**, not the server as a whole.

**🎯 Option Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Purpose** | **Validity** | **Notes** |
| **a) SP\_HELP** | Returns details about **any database object** (like a table, view, etc.) | ❌ Invalid | Not specific to users or principals |
| **b) SP\_HELPDB** | Returns information about **databases on the server** | ❌ Invalid | Works at the **server level**, not database-level principals |
| **c) SP\_HELPTEXT** | Displays **source code** of views, stored procedures, functions, etc. | ❌ Invalid | Not related to user information |
| **d) SP\_HELPUSER** | Returns information about **database-level principals** (users, roles, etc.) in the current database | ✅ Valid | Specifically designed for user-level data inside a database |

**✅ Correct Answer: d) SP\_HELPUSER**

**📌 Key Points to Remember**

* SP\_HELPUSER displays database-level **users, their roles, and login mappings**.
* It is run in the context of the current database.
* For server-level principals (logins, server roles), use catalog views like sys.server\_principals.
* SP\_HELP is for general object metadata, not user-related info.
* SP\_HELPDB shows database-level properties (name, size, status), not user info.

**Question 7:**

Consider the following code:

CREATE TABLE Employee

(

EmployeeNumber INT PRIMARY KEY,

EmployeeName VARCHAR(20),

Salary SMALLMONEY,

Stream VARCHAR(20)

)

GO

BEGIN TRAN

INSERT INTO Employee VALUES(1001, 'Sandra', 20000, 'MF')

BEGIN TRAN

INSERT INTO Employee VALUES(1002, 'Samantha', 20000, 'MS')

COMMIT

ROLLBACK

GO

How many record(s) will be inserted into the table Employee?

1. 0
2. 1
3. 2
4. Nested Transaction is not allowed in SQL Server

**🧠 Concept: Transactions and Nested Transactions in SQL Server**

SQL Server allows **nested transactions**, but they behave differently than expected:

* **BEGIN TRAN** increments a **transaction count**.
* Only the **outermost ROLLBACK** rolls back **all changes**, regardless of how many COMMITs are issued inside.
* **Inner COMMITs** don't persist data if a ROLLBACK occurs later in the outer transaction.

**🎯 Step-by-Step Analysis**

**Table Definition:**

CREATE TABLE Employee

(

EmployeeNumber INT PRIMARY KEY,

EmployeeName VARCHAR(20),

Salary SMALLMONEY,

Stream VARCHAR(20)

)

**Transaction Block:**

BEGIN TRAN

INSERT INTO Employee VALUES(1001, 'Sandra', 20000, 'MF')

BEGIN TRAN

INSERT INTO Employee VALUES(1002, 'Samantha', 20000, 'MS')

COMMIT

ROLLBACK

**Transaction Flow:**

1. Outer BEGIN TRAN → Start of main transaction
2. Insert Sandra → Temporarily staged
3. Inner BEGIN TRAN → Increases transaction count (nested transaction)
4. Insert Samantha → Temporarily staged
5. COMMIT → Only decreases transaction count, **no data is truly committed**
6. Outer ROLLBACK → **Rolls back entire transaction**, including both inserts

**✅ Correct Answer: a) 0**

**📌 Key Points to Remember**

* SQL Server **supports nested transactions** syntactically, but **they don't work as independent transactions**.
* Only the **outermost COMMIT** finalizes the changes.
* A ROLLBACK at any level **rolls back the entire transaction**, regardless of inner commits.
* Transaction nesting is primarily useful for **modular code**, but true nested commit/rollback behavior requires **savepoints** or error-handling logic.

**Question 8:**

What would be the output for the two SELECT statements when the following code is executed as a batch?

DECLARE @Num1 INT = 4.76

DECLARE @Num2 FLOAT = 5.03

SELECT CAST(@Num1 AS FLOAT) + CONVERT(FLOAT, @Num2) AS ResultOne

DECLARE @Num3 INT = '2.60'

SELECT CONVERT(FLOAT, @Num2) + CAST(@Num3 AS FLOAT) AS ResultTwo

**Options:**

1. ResultOne - Error - Conversion failed while converting INT to FLOAT  
   ResultTwo - Error - Conversion failed while converting VARCHAR to FLOAT
2. ResultOne - Error - Conversion failed while converting FLOAT to INT  
   ResultTwo - Error - Conversion failed while converting VARCHAR to FLOAT
3. ResultOne - 9.03  
   ResultTwo - Error - Conversion failed while converting VARCHAR to INT
4. ResultOne - 9  
   ResultTwo - 7
5. ResultOne - 10  
   ResultTwo - 7

**🧠 Concept: Data Type Conversion in SQL Server**

SQL Server performs **implicit and explicit data conversions** using:

* CAST() and CONVERT() functions
* Type compatibility rules apply
* Conversion from FLOAT to INT **truncates decimal**, not rounds
* Assigning a **string with a decimal** to an INT variable causes **conversion failure**

**🎯 Step-by-Step Analysis**

**Code Block:**

DECLARE @Num1 INT = 4.76

DECLARE @Num2 FLOAT = 5.03

SELECT CAST(@Num1 AS FLOAT) + CONVERT(FLOAT, @Num2) AS ResultOne

DECLARE @Num3 INT = '2.60'

SELECT CONVERT(FLOAT, @Num2) + CAST(@Num3 AS FLOAT) AS ResultTwo

**📌 Line-by-Line Breakdown:**

**🔹 Line 1:**

DECLARE @Num1 INT = 4.76

* **Result:** 4.76 is a decimal literal.
* When assigned to an INT, the value is **truncated**, not rounded.
* @Num1 = 4

**🔹 Line 2:**

DECLARE @Num2 FLOAT = 5.03

* Valid. @Num2 = 5.03

**🔹 Line 3:**

SELECT CAST(@Num1 AS FLOAT) + CONVERT(FLOAT, @Num2) AS ResultOne

* CAST(4 AS FLOAT) + CONVERT(FLOAT, 5.03) = 4.0 + 5.03 = 9.03
* ✅ No error

**🔹 Line 4:**

DECLARE @Num3 INT = '2.60'

* ❌ **Invalid**. '2.60' is a **string with a decimal**, not a valid integer.
* SQL Server tries to implicitly convert '2.60' to INT, which fails.

**✅ Correct Answer: c) ResultOne - 9.03; ResultTwo - Error - Conversion failed while converting VARCHAR to INT**

**📌 Key Points to Remember**

* Assigning a **decimal literal to an INT** truncates the value.
* CAST() and CONVERT() can change types between INT and FLOAT safely.
* Assigning a **string like '2.60' to an INT** variable causes a conversion error.
* SQL Server does **not** auto-round when assigning from float to int.

**Question 9:**

What will be the output of the following T-SQL statements?

DECLARE @VariableDate DATE

SELECT @VariableDate = '4-APR-2008'

SELECT @VariableDate = DATEADD(MM, -1, @VariableDate)

SELECT DATEPART(MM, @VariableDate)

**Options:**

1. March
2. 3
3. May
4. 5

**🧠 Concept: Date Functions in SQL Server**

* DATEADD(datepart, number, date) shifts the date by a specified interval.
* DATEPART(datepart, date) extracts a specific part of the date, like month or year.
* The MM datepart stands for **month**.
* Month values returned by DATEPART(MM, ...) are **integers from 1 to 12**.

**🎯 Step-by-Step Analysis**

DECLARE @VariableDate DATE

SELECT @VariableDate = '4-APR-2008'

* Assigns date value '2008-04-04' to @VariableDate.

SELECT @VariableDate = DATEADD(MM, -1, @VariableDate)

* Moves the date **1 month backward**.
* 4-Apr-2008 minus 1 month = 4-Mar-2008.

SELECT DATEPART(MM, @VariableDate)

* Extracts the **month number** from 4-Mar-2008.
* March is the **3rd month**.

**✅ Correct Answer: b) 3**

**📌 Key Points to Remember**

* DATEADD(MM, -1, date) subtracts one month from the date.
* DATEPART(MM, date) returns an **integer (1–12)** for the month.
* The output is **not** the month name like "March", but the month number.

**Question 10:**

Consider the following code:

CREATE TABLE Customers

(

CustomerNumber INT IDENTITY(\_\_\_\_, 1),

CustomerName VARCHAR(20)

)

INSERT INTO Customers VALUES('Tibco')

SELECT \* FROM Customers

On executing the above code, the following output is obtained:

|  |  |
| --- | --- |
| **CustomerNumber** | **CustomerName** |
| 1004 | Tibco |

Now, consider that the following code is executed:

SET IDENTITY\_INSERT Customers \_\_\_\_\_

INSERT INTO Customers (CustomerNumber, CustomerName)

VALUES (1005, 'Jack')

What should be entered in the 2 blanks respectively if the above statement has to execute successfully?

**Options:**

1. 1001 and ON
2. 1001 and OFF
3. 1004 and ON
4. 1004 and OFF

**🧠 Concept: IDENTITY and IDENTITY\_INSERT in SQL Server**

* IDENTITY(seed, increment) auto-generates values starting from seed, increasing by increment.
* When IDENTITY\_INSERT is set to ON, explicit values can be inserted into the identity column.
* Only **one table at a time** in a session can have IDENTITY\_INSERT set to ON.

**🎯 Step-by-Step Analysis**

**Code Snippet:**

CREATE TABLE Customers

(

CustomerNumber INT IDENTITY(\_\_\_\_, 1),

CustomerName VARCHAR(20)

)

* Identity column increments by 1.
* The **starting seed** must be set so that the **first inserted value is 1004**, as shown in the output.

✅ Required value for the first blank: 1004

**Next Snippet:**

SET IDENTITY\_INSERT Customers \_\_\_\_\_

* To insert a value **explicitly into an identity column**, IDENTITY\_INSERT must be set to **ON**.

✅ Required value for the second blank: ON

**✅ Correct Answer: c) 1004 and ON**

**📌 Key Points to Remember**

* IDENTITY(seed, increment) sets the starting and step value for auto-incremented identity columns.
* Use SET IDENTITY\_INSERT <table> ON to manually insert values into an identity column.
* After manual inserts, it's good practice to turn IDENTITY\_INSERT back OFF.
* Only one table in a session can have IDENTITY\_INSERT set to ON at any time.

**Question 11:**

Consider the following function:

CREATE FUNCTION ufn\_ConcatenateWords(@WordOne CHAR, @WordTwo CHAR, @WordThree CHAR)

RETURNS CHAR(3)

AS

BEGIN

RETURN @WordOne + @WordTwo + @WordThree

END

Which of the following function call(s) will give the resultant output as '345'?

**Options:**

a) DECLARE @Ret VARCHAR(10)

EXEC @Ret = ufn\_ConcatenateWords('3', '4', '5')

SELECT @Ret

b) SELECT dbo.ufn\_ConcatenateWords('3', '4', '5')

c) SELECT dbo.ufn\_ConcatenateWords(3, 4, 5)

d) DECLARE @Ret VARCHAR(10)

EXEC @Ret = ufn\_ConcatenateWords '3', '4', '5'

SELECT @Ret

e) DECLARE @Ret INT

EXEC @Ret = ufn\_ConcatenateWords 3, 4, 5

SELECT @Ret

f) DECLARE @Ret INT

EXEC @Ret = ufn\_ConcatenateWords(3, 4, 5)

SELECT @Ret

**📋 Function Definition Recap**

CREATE FUNCTION ufn\_ConcatenateWords(

@WordOne CHAR,

@WordTwo CHAR,

@WordThree CHAR

)

RETURNS CHAR(3)

AS

BEGIN

RETURN @WordOne + @WordTwo + @WordThree

END

This function:

* Accepts three CHAR inputs.
* Concatenates them into a CHAR(3) return value.

**🎯 Objective**

Determine which function calls will return the string '345' **without errors**.

**🧪 Option-by-Option Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Option** | **Code Snippet** | **Valid?** | **Output** | **Explanation** |
| **a** | DECLARE @Ret VARCHAR(10)EXEC @Ret = ufn\_ConcatenateWords('3', '4', '5')SELECT @Ret | ❌ Invalid | — | ❌ **Missing parentheses** in function call makes it syntactically ambiguous or invalid in many SQL Server versions |
| **b** | SELECT dbo.ufn\_ConcatenateWords('3', '4', '5') | ✅ Valid | '345' | ✅ Uses SELECT with valid CHAR arguments — perfect syntax |
| **c** | SELECT dbo.ufn\_ConcatenateWords(3, 4, 5) | ✅ Valid | '345' | ✅ Implicit conversion from INT to CHAR works correctly |
| **d** | DECLARE @Ret VARCHAR(10)EXEC @Ret = ufn\_ConcatenateWords '3', '4', '5'SELECT @Ret | ✅ Valid | '345' | ✅ EXEC used to assign function result to variable — **allowed**, and parentheses are optional here |
| **e** | DECLARE @Ret INTEXEC @Ret = ufn\_ConcatenateWords 3, 4, 5SELECT @Ret | ✅ Valid | 345 | ✅ Function returns '345' (CHAR) → implicitly cast to INT |
| **f** | DECLARE @Ret INTEXEC @Ret = ufn\_ConcatenateWords(3, 4, 5)SELECT @Ret | ❌ Invalid | — | ❌ EXEC with parentheses not supported for scalar function calls — causes an error |

**✅ Final Correct Options:**

**b), c), d), e)**

**❌ Incorrect Options:**

* **a)**: ❌ Invalid syntax due to missing parentheses
* **f)**: ❌ Using parentheses with EXEC causes an error when calling scalar functions

**📌 Key Points to Remember**

1. ✅ **Use SELECT to call scalar functions** when displaying results directly.
2. ✅ You can **assign function results using EXEC @var = FunctionName ...**, but:
   * ⚠ **Do not use parentheses** in the EXEC form.
   * EXEC @ret = FunctionName 'arg1', 'arg2' ✔ is allowed.
   * EXEC @ret = FunctionName('arg1', 'arg2') ❌ will fail.
3. ✅ SQL Server **supports implicit conversions** (e.g., 3 → '3') if the data types are compatible.
4. ❌ Scalar functions **do not support EXEC with parentheses** — that's reserved for stored procedures.
5. ❌ Function calls **must be syntactically complete** — parentheses are required when using SELECT, and optional (but sensitive) with EXEC.

**📝 Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Validity** | **Result** | **Reason** |
| **a** | ❌ Invalid | — | Missing parentheses in function call |
| **b** | ✅ Valid | '345' | Correct use of SELECT |
| **c** | ✅ Valid | '345' | Implicit conversion from INT to CHAR |
| **d** | ✅ Valid | '345' | Valid EXEC without parentheses |
| **e** | ✅ Valid | 345 | Implicit cast to INT |
| **f** | ❌ Invalid | — | Parentheses in EXEC not allowed for functions |

**Question 12:**

Which of the following query helps to award ranks for the salesmen of a company based on their total sales?

**Assumptions:**

1. Salesmen is a valid table that exists in the database.
2. SalesmanId, SalesmanName, TotalSales are a few columns in the Salesmen table.

**Options:**

1. SELECT SalesmanId, SalesmanName, TotalSales, RANK() OVER (ORDER BY SalesmanId DESC) AS RANK FROM Salesmen
2. SELECT RANK(SalesmanId), SalesmanName, TotalSales FROM Salesmen
3. SELECT SalesmanId, SalesmanName, RANK(TotalSales) OVER (ORDER BY TotalSales) AS RANK FROM Salesmen
4. SELECT SalesmanId, SalesmanName, TotalSales, RANK() OVER (ORDER BY TotalSales DESC) AS RANK FROM Salesmen

**🧠 Concept: Ranking Rows in SQL Server using RANK()**

* RANK() is a **window function** used to assign **ranks to rows** based on the values in a specific column.
* Syntax:
* RANK() OVER (ORDER BY column\_name [ASC|DESC])
* When multiple rows have the same value, they receive the **same rank**, and the next rank(s) are skipped.

**🎯 Option Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Valid Syntax?** | **Ranks by TotalSales?** | **Notes** |
| **a)** RANK() OVER (ORDER BY SalesmanId DESC) | ✅ Yes | ❌ No | Ranks based on **SalesmanId**, not TotalSales |
| **b)** RANK(SalesmanId) | ❌ No | ❌ No | RANK() is a **window function**, not used like this |
| **c)** RANK(TotalSales) OVER (ORDER BY TotalSales) | ❌ No | ❌ No | RANK() does **not take arguments** |
| **d)** RANK() OVER (ORDER BY TotalSales DESC) | ✅ Yes | ✅ Yes | Correct ranking based on **TotalSales (highest first)** |

**✅ Correct Answer: d) SELECT SalesmanId, SalesmanName, TotalSales, RANK() OVER (ORDER BY TotalSales DESC) AS RANK FROM Salesmen**

**📌 Key Points to Remember**

* RANK() assigns the same rank to rows with **equal values**.
* Use ORDER BY TotalSales DESC to rank from **highest to lowest sales**.
* RANK() must be used **without arguments** and always within an OVER() clause.
* For **sequential ranks** without gaps, consider using DENSE\_RANK().

**Question 13:**

Predict the output on executing the following create function statement.

**Assumption:**

1. Students is a valid table in the database with one column "StudentName"

CREATE FUNCTION ufn\_DeleteStudent(@StudentName VARCHAR(25))

RETURNS INT

AS

BEGIN

DELETE FROM Students WHERE StudentName = @StudentName

RETURN 1

END

What will happen?

**Options:**

1. Row gets added successfully
2. Compilation error
3. Runtime error
4. None of the above

**🧠 Concept: SQL Server Functions – Limitations on Side Effects**

* In SQL Server, **user-defined functions (UDFs)** must be **deterministic** and **side-effect-free**.
* This means:

❌ **Cannot perform INSERT, DELETE, or UPDATE operations** inside a scalar or table-valued function.

* SQL Server restricts these operations to ensure functions can be used in queries without causing data changes.

**🎯 Code Analysis**

CREATE FUNCTION ufn\_DeleteStudent(@StudentName VARCHAR(25))

RETURNS INT

AS

BEGIN

DELETE FROM Students WHERE StudentName = @StudentName

RETURN 1

END

* The function attempts to **delete data** from the Students table.
* This is **not allowed** in a SQL Server user-defined function.

**✅ Correct Answer: b) Compilation error**

**📌 Key Points to Remember**

* **User-defined functions cannot contain data modification statements** like:
  + INSERT, DELETE, UPDATE, MERGE
* Functions are meant for **read-only logic and value computations**.
* To modify data, use **stored procedures** instead of functions.
* SQL Server enforces this rule at **compile time**, resulting in a **compilation error**.

**Question 14:**

Salesperson A updates the number of rows in the table Products from 100 to 80, but does not commit the transaction. Meanwhile, Salesperson B observes that the table Products contains 80 rows.

Now Salesperson A rolls back the transaction and when Salesperson B checks the table data again, he observes that there are 100 rows in the table Products.

Identify the concurrency problem from the perspective of Salesperson B.

**Options:**

1. Dirty read
2. Phantom read
3. Lost update
4. Non-repeatable read

**🧠 Concept: Concurrency Problems in SQL Server**

SQL Server supports various **transaction isolation levels** to control how data is accessed and modified concurrently. When these controls are weak or relaxed, the following concurrency issues can arise:

| **Problem Type** | **Description** |
| --- | --- |
| **Dirty Read** | Reading data **modified by another transaction that hasn’t been committed yet** |
| **Non-repeatable Read** | Re-reading the same row returns **different data** due to another transaction's commit |
| **Phantom Read** | Repeating a query returns **additional rows** due to inserts by another transaction |
| **Lost Update** | Two transactions **overwrite each other’s changes** unintentionally |

**🎯 Situation Breakdown**

1. **Salesperson A** modifies the data (reduces rows from 100 to 80), but does **not commit**.
2. **Salesperson B** sees **80 rows** — **reads uncommitted data**.
3. Then, **Salesperson A rolls back** — data is **restored to 100 rows**.
4. Now **Salesperson B sees 100 rows** — meaning the earlier read was **invalid**.

This means:

* Salesperson B **read uncommitted data** (i.e., data that was later rolled back).

**✅ Correct Answer: a) Dirty read**

**📌 Key Points to Remember**

* **Dirty Read** happens when one transaction reads changes made by another transaction **before it's committed**.
* This issue is possible under **Read Uncommitted** isolation level or if NOLOCK hint is used.
* To prevent dirty reads, use **Read Committed** or higher isolation levels.
* Dirty reads can lead to viewing **temporary or invalid data**.

**Question 15:**

Consider the following stored procedure:

CREATE PROCEDURE usp\_EmployeeDetails

(

@EmployeeId INT

)

AS

BEGIN

DECLARE @RetValue INT

BEGIN TRAN

BEGIN TRY

IF EXISTS (SELECT \* FROM Employees WHERE EmployeeId = @EmployeeId)

BEGIN

DELETE FROM EmployeeSalaryDetails WHERE EmployeeId = @EmployeeId

SET @RetValue = 1

COMMIT

END

ELSE

BEGIN

ROLLBACK

SET @RetValue = -1

END

RETURN @RetValue

END TRY

BEGIN CATCH

ROLLBACK

RETURN -99

END CATCH

END

What will happen if the following code is executed after the stored procedure is created successfully?

DECLARE @ret INT

EXEC @ret = usp\_EmployeeDetails 1001

SELECT @ret

**Assumptions:**

1. EmployeeId ‘1001’ exists in the Employees table.
2. Employees is a valid table in the database.

**Options:**

1. The return value will be 1 and no row will be deleted from the EmployeeSalaryDetails table
2. The return value will be -1 and no row will be deleted from the EmployeeSalaryDetails table
3. The return value will be -99 and no row will be deleted from the EmployeeSalaryDetails table
4. The return value will be 1 and one row will be deleted from the EmployeeSalaryDetails table
5. The return value will be 1 and one row may or may not be deleted from the EmployeeSalaryDetails table

**🧠 Concept: Stored Procedures with Transactions, Return Values, and Error Handling in SQL Server**

This procedure uses:

* BEGIN TRAN / COMMIT / ROLLBACK to manage transaction scope
* TRY...CATCH block for exception handling
* Conditional logic with IF EXISTS
* A RETURN statement to send a status code

**🎯 Step-by-Step Analysis of the Procedure Execution**

**Assumptions:**

* EmployeeId = 1001 **exists** in the Employees table ✅
* Employees and EmployeeSalaryDetails are valid tables ✅

**Code Execution Flow:**

DECLARE @ret INT

EXEC @ret = usp\_EmployeeDetails 1001

SELECT @ret

**Inside the Procedure:**

IF EXISTS (SELECT \* FROM Employees WHERE EmployeeId = @EmployeeId)

* This condition is **true** for 1001.

Then:

DELETE FROM EmployeeSalaryDetails WHERE EmployeeId = @EmployeeId

SET @RetValue = 1

COMMIT

RETURN @RetValue

* SQL Server attempts to delete the matching row from EmployeeSalaryDetails.
* Regardless of whether a matching row is **found or not**, DELETE executes successfully (even if 0 rows are deleted).
* Return value is set to 1.
* Transaction is committed.
* The procedure returns 1 to the caller.

**Key Insight:**

* The presence of EmployeeId = 1001 in EmployeeSalaryDetails **is not verified or required**.
* Even if no row exists to delete, the DELETE command still executes successfully.
* So the stored procedure **always returns 1** when the employee exists, even if no row is actually deleted.

**✅ Correct Answer: e) The return value will be 1 and one row may or may not be deleted from the EmployeeSalaryDetails table**

**📌 Key Points to Remember**

* DELETE on a table that finds **no matching rows** does **not throw an error** — it just affects 0 rows.
* The return value of the procedure depends only on the **existence of the Employee in the Employees table**.
* RETURN values in stored procedures are often used for **status signaling** (e.g., success, failure, error codes).
* To check if a row was actually deleted, you would need to add logic using @@ROWCOUNT.

**Question 16:**

What will be the output of the following query?

SELECT CAST(DAY('2016-09-08 15:02:44.870') AS VARCHAR(6)) + ' Planets of the Solar system'

**Options:**

1. 8 Planets of the Solar system
2. 9 Planets of the Solar system
3. Eight Planets of the Solar system
4. Nine Planets of the Solar system
5. Error – Conversion failed while converting DATETIME to VARCHAR

**🧠 Concept: DAY() and CAST() Functions in SQL Server**

* DAY(datetime) extracts the **day part** (1 to 31) from a given date.
* CAST(expression AS datatype) explicitly converts a value to the specified type.
* + operator concatenates strings when both operands are strings (or converted to strings).

**🎯 Step-by-Step Evaluation**

**Query:**

SELECT CAST(DAY('2016-09-08 15:02:44.870') AS VARCHAR(6)) + ' Planets of the Solar system'

**1. DAY('2016-09-08 15:02:44.870')**

* Extracts **day** portion: **08**

**2. CAST(8 AS VARCHAR(6))**

* Converts integer 8 to string '8'

**3. '8' + ' Planets of the Solar system'**

* String concatenation results in:
* '8 Planets of the Solar system'

✅ This executes **without error**

**✅ Correct Answer: a) 8 Planets of the Solar system**

**📌 Key Points to Remember**

* DAY(datetime) returns an integer from 1 to 31
* CAST(... AS VARCHAR) allows integer-to-string conversion
* String concatenation using + works when both operands are strings
* SQL Server automatically handles basic type conversions in expressions

**Question 17:**

Consider the creation of two tables as follows:

a.

CREATE TABLE TableOne

(

ColumnOne DECIMAL(3,1) IDENTITY,

ColumnTwo DECIMAL(3,0)

)

b.

CREATE TABLE TableTwo

(

ColumnOne DECIMAL(3,1),

ColumnTwo DECIMAL(3,0) IDENTITY

)

Which of the above tables will be created successfully?

**Options:**

1. Only TableOne
2. Only TableTwo
3. Neither of the tables
4. Both the tables

**🧠 Concept: IDENTITY Column Rules in SQL Server**

* SQL Server allows **only one IDENTITY column per table**.
* The IDENTITY column must be of a **numeric type**: INT, BIGINT, SMALLINT, TINYINT, DECIMAL, or NUMERIC.
* However, for DECIMAL or NUMERIC, the **scale (number of digits after the decimal point) must be 0**.

✅ Allowed: DECIMAL(p, 0)  
❌ Not Allowed: DECIMAL(p, s) where **s > 0**

**🎯 Table-by-Table Validation**

**🔹 TableOne:**

CREATE TABLE TableOne

(

ColumnOne DECIMAL(3,1) IDENTITY, -- ❌ Not allowed: scale is 1

ColumnTwo DECIMAL(3,0)

)

* ❌ DECIMAL(3,1) has **scale > 0** → Invalid as an IDENTITY column

**🔹 TableTwo:**

CREATE TABLE TableTwo

(

ColumnOne DECIMAL(3,1),

ColumnTwo DECIMAL(3,0) IDENTITY -- ✅ Valid: scale = 0

)

* ✅ DECIMAL(3,0) is valid for IDENTITY

**✅ Correct Answer: b) Only TableTwo**

**📌 Key Points to Remember**

* An IDENTITY column must have a **scale of 0** when using DECIMAL or NUMERIC.
* Only **one identity column** is allowed per table.
* Common valid types: INT, BIGINT, SMALLINT, DECIMAL(p, 0), etc.
* Avoid specifying decimal places (scale > 0) for identity columns.

**Question 18:**

Consider the following stored procedure:

CREATE PROCEDURE usp\_UpdateCustomerDetails

(

@CustomerId INT,

@BillAmount DECIMAL(15,2)

)

AS

BEGIN

BEGIN TRAN

BEGIN TRY

IF EXISTS (SELECT \* FROM Customers WHERE CustomerId = @CustomerId)

BEGIN

INSERT INTO BillPayment VALUES (@CustomerId, @BillAmount)

COMMIT

END

ELSE

BEGIN

ROLLBACK

END

DELETE FROM PendingBillDetails WHERE CustomerId = @CustomerId

ROLLBACK

END TRY

BEGIN CATCH

ROLLBACK

END CATCH

END

What will be the result if the following statement is executed after the above stored procedure is created successfully?

EXEC usp\_UpdateCustomerDetails 1001, 1500

Assumptions:

1. Tables Customers, BillPayment, and PendingBillDetails are existing tables in the database.
2. The CustomerId 1001 exists in the table Customers but not in the table PendingBillDetails.
3. Table BillPayment consists of two columns: CustomerId and BillAmount.

**Options:**

1. One row will be added to table BillPayment and one row will be deleted from table PendingBillDetails
2. No row will be added to table BillPayment and one row will be deleted from table PendingBillDetails
3. One row will be added to table BillPayment and no rows will be deleted from table PendingBillDetails
4. No change in the table data of tables BillPayment and PendingBillDetails

**🧠 Concept: Transactions and Error Handling with TRY...CATCH in SQL Server**

* BEGIN TRAN starts a transaction block.
* COMMIT saves the changes; ROLLBACK undoes **all** operations from the last BEGIN TRAN.
* A ROLLBACK affects **the entire transaction**, even if called after a COMMIT.
* **After a COMMIT, if a ROLLBACK is issued within the same transaction block, it throws an error**:  
  **“The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.”**
* A DELETE with no matching rows does **not throw an error**—it silently affects 0 rows.

**🎯 Step-by-Step Execution of Procedure**

EXEC usp\_UpdateCustomerDetails 1001, 1500

**Assumptions:**

* CustomerId = 1001 **exists** in Customers
* CustomerId = 1001 **does not exist** in PendingBillDetails
* All three tables are valid

**Code Execution Breakdown:**

BEGIN TRAN

BEGIN TRY

IF EXISTS (SELECT \* FROM Customers WHERE CustomerId = @CustomerId)

BEGIN

INSERT INTO BillPayment VALUES (@CustomerId, @BillAmount)

COMMIT

END

ELSE

BEGIN

ROLLBACK

END

DELETE FROM PendingBillDetails WHERE CustomerId = @CustomerId

ROLLBACK

END TRY

BEGIN CATCH

ROLLBACK

END CATCH

**➤ What happens?**

1. CustomerId = 1001 **exists** → IF condition is true
2. **Inserts** (1001, 1500) into BillPayment
3. **COMMIT** is executed → ✅ Transaction is closed
4. Then it attempts:
5. DELETE FROM PendingBillDetails WHERE CustomerId = 1001
   * No matching row → 0 rows deleted, **no error**
6. **Next line is ROLLBACK** → ❌ This causes an **error**:
   * No open transaction left to roll back
   * This triggers the CATCH block
7. Inside CATCH, another ROLLBACK is called → Also fails silently

**Final Effect:**

* Even though the ROLLBACK after COMMIT is **invalid**, the **INSERT was already committed**.
* No rows were deleted (because no match), and no rollback undid the insert.

**✅ Correct Answer: c) One row will be added to table BillPayment and no rows will be deleted from table PendingBillDetails**

**📌 Key Points to Remember**

* A DELETE without matching rows does **not throw an error**.
* Once a COMMIT occurs, the transaction is **closed**—any subsequent ROLLBACK is invalid.
* Invalid ROLLBACK after COMMIT throws an error, triggering the CATCH block, but **already committed data remains** unchanged.
* The stored procedure has a **logical flaw**: post-COMMIT actions are still under assumption of an active transaction.

**Question 19:**

Consider the table Employee with the following data:

|  |  |  |
| --- | --- | --- |
| EmployeeId | EmployeeName | Salary |
| 1001 | James | 25000 |
| 1002 | John | 10000 |
| 1003 | Jack | 20000 |
| 1004 | Jill | 25000 |

Assume that the following query is executed:

SELECT EmployeeId, DENSE\_RANK() OVER (ORDER BY Salary DESC) FROM Employee ORDER BY Salary

What will be the dense rank of EmployeeId '1003' according to the above query?

**Options:**

1. 4
2. 2
3. 3
4. 0

**🧠 Concept: DENSE\_RANK() in SQL Server**

* DENSE\_RANK() assigns ranks to rows based on the **ordering of a specified column**.
* **Duplicate values get the same rank**, and the **next rank is not skipped** (unlike RANK()).
* Syntax:
* DENSE\_RANK() OVER (ORDER BY column [ASC|DESC])

**🎯 Step-by-Step Analysis**

**Employee Table:**

| **EmployeeId** | **EmployeeName** | **Salary** |
| --- | --- | --- |
| 1001 | James | 25000 |
| 1002 | John | 10000 |
| 1003 | Jack | 20000 |
| 1004 | Jill | 25000 |

**Query:**

SELECT EmployeeId, DENSE\_RANK() OVER (ORDER BY Salary DESC)

FROM Employee

ORDER BY Salary

**🔹 Step 1: Ranking (DENSE\_RANK OVER ORDER BY Salary DESC)**

Ranks are based on descending salary:

| **Salary** | **Rank** |
| --- | --- |
| 25000 | 1 |
| 25000 | 1 |
| 20000 | 2 |
| 10000 | 3 |

So:

* 1001 → Rank 1
* 1004 → Rank 1
* 1003 → Rank 2 ✅
* 1002 → Rank 3

**🔹 Step 2: Final Output (Sorted by Salary ASC)**

The query **orders the final result by Salary ASC**, but that does **not affect** the computed rank.

**✅ Correct Answer: b) 2**

**📌 Key Points to Remember**

* DENSE\_RANK() does **not skip** ranks for duplicates.
* ORDER BY inside the OVER() clause controls how ranks are calculated.
* ORDER BY outside the query controls **final row order**, not rank logic.
* Ranking functions are calculated **before the final sort**.

**Question 20:**

Consider the statements given below and choose the correct option, with respect to **IDENTITY** in SQL Server:

i. An identity column must not allow NULL values  
ii. IDENT\_CURRENT returns the last inserted identity value for a table in any session  
iii. @@IDENTITY returns the first inserted value among a group of INSERT statements  
iv. The identity column value can be explicitly inserted using the statement SET IDENTITY\_INSERT <table\_name> OFF

**Options:**

1. Statements iii and iv are correct and statements i and ii are incorrect
2. Statements i and ii are correct and statements iii and iv are incorrect
3. Statements i, ii and iii are correct and statement iv is incorrect
4. Statements i, ii and iv are correct and statement iii is incorrect

**🧠 Concept: Identity Columns and Related Functions in SQL Server**

**Let’s evaluate each statement:**

**i. *An identity column must not allow NULL values***

✅ **True**

* Identity columns **automatically generate values**; hence, NULL values are **not allowed**.

**ii. *IDENT\_CURRENT returns the last inserted identity value for a table in any session***

✅ **True**

* IDENT\_CURRENT('table\_name') returns the **last identity value** inserted **into a specified table**, **regardless of session or scope**.

**iii. *@@IDENTITY returns the first inserted value among a group of INSERT statements***

❌ **False**

* @@IDENTITY returns the **last identity value** generated **in the current session**, **regardless of table or scope**.
* It doesn't return the **first**, but the **last**.

**iv. *The identity column value can be explicitly inserted using the statement SET IDENTITY\_INSERT <table\_name> OFF***

❌ **False**

* To explicitly insert into an identity column, you must use:  
  SET IDENTITY\_INSERT <table\_name> \*\*ON\*\*
* OFF disables manual insertion.

**✅ Correct Answer: b) Statements i and ii are correct and statements iii and iv are incorrect**

**📌 Key Points to Remember**

* IDENTITY columns do **not allow NULLs**
* IDENT\_CURRENT is **global to all sessions**
* @@IDENTITY is **session-scoped** and returns the **last** inserted identity value
* Use SET IDENTITY\_INSERT <table\_name> ON to insert values manually into an identity column

**Question 21:**

Jack wants to fetch the names of the 5 least performing students of his department, who have scored minimum marks in a test.

Which of the following queries will help him to do so?

Assumptions:

1. The data for StudentName and MarksScored is stored in the table StudentsMarksDetails.
2. StudentsMarksDetails is a valid table in the database.

**Options:**

1. SELECT TOP \* StudentName FROM StudentsMarksDetails ORDER BY MarksScored
2. SELECT TOP 5 StudentName FROM StudentsMarksDetails ORDER BY MarksScored ASC
3. SELECT TOP 5 \* FROM StudentsMarksDetails ORDER BY MarksScored DESC
4. SELECT TOP FIVE StudentName FROM StudentsMarksDetails ORDER BY MarksScored
5. SELECT TOP 5 StudentName FROM StudentsMarksDetails ORDER BY MarksScored DESC

**🧠 Concept: SELECT TOP n with ORDER BY in SQL Server**

* SELECT TOP n is used to retrieve the **first n rows** from a query result.
* ORDER BY determines the **sorting order**.
* To get the **lowest values**, use ORDER BY column ASC.
* To get the **highest values**, use ORDER BY column DESC.

**🎯 Requirement:**

* Fetch names of the **5 least performing students** based on **minimum marks**.

**🕵️ Option Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Option** | **Query** | **Valid Syntax?** | **Logic Correct?** | **Notes** |
| **a)** SELECT TOP \* StudentName | ❌ No | ❌ No | Invalid syntax – TOP requires a number |  |
| **b)** SELECT TOP 5 StudentName ... ORDER BY MarksScored ASC | ✅ Yes | ✅ Yes | Correctly fetches lowest 5 marks |  |
| **c)** SELECT TOP 5 \* ... ORDER BY MarksScored DESC | ✅ Yes | ❌ No | Gets top 5 scorers, not lowest |  |
| **d)** SELECT TOP FIVE ... | ❌ No | ❌ No | TOP requires a **number**, not a word |  |
| **e)** SELECT TOP 5 StudentName ... ORDER BY MarksScored DESC | ✅ Yes | ❌ No | Gets highest 5 marks, not lowest |  |

**✅ Correct Answer: b) SELECT TOP 5 StudentName FROM StudentsMarksDetails ORDER BY MarksScored ASC**

**📌 Key Points to Remember**

* Use SELECT TOP n ... ORDER BY column ASC to fetch **lowest n values**.
* Use DESC for **highest n values**.
* TOP must be followed by an integer (e.g., TOP 5), not a string (e.g., TOP FIVE).
* Always verify both **syntax** and **logic** in selection queries.

**Question 22:**

Consider the following table:

CREATE TABLE TableOne

(

ColumnOne INT IDENTITY(100, -1),

ColumnTwo INT

)

Consider the following INSERT statements:

a.

SET IDENTITY\_INSERT TableOne OFF

INSERT INTO TableOne VALUES (1)

b.

SET IDENTITY\_INSERT TableOne ON

INSERT INTO TableOne (ColumnOne, ColumnTwo) VALUES (1, 2)

Which of the above INSERT statements will execute successfully?

**Options:**

1. Only a
2. Only b
3. Neither a nor b
4. Both a and b

**🧠 Concept: Identity Columns and IDENTITY\_INSERT in SQL Server**

* IDENTITY(seed, increment) auto-generates values for a column.
* When IDENTITY\_INSERT is **OFF**, you can **only omit the identity column** and let SQL Server generate its value.
* When IDENTITY\_INSERT is **ON**, you **must explicitly include** the identity column in the column list and provide a value for it.

**🎯 Step-by-Step Analysis**

**Table Definition:**

CREATE TABLE TableOne

(

ColumnOne INT IDENTITY(100, -1),

ColumnTwo INT

)

**Statement a:**

SET IDENTITY\_INSERT TableOne OFF

INSERT INTO TableOne VALUES (1)

* This inserts a value into ColumnTwo only.
* ColumnOne gets auto-generated (first value = 100).
* ✅ Valid. Executes successfully.

**Statement b:**

SET IDENTITY\_INSERT TableOne ON

INSERT INTO TableOne (ColumnOne, ColumnTwo) VALUES (1, 2)

* Explicitly inserts a value into the identity column.
* IDENTITY\_INSERT is ON, and the identity column is included in the column list.
* ✅ Valid. Executes successfully.

**✅ Correct Answer: d) Both a and b**

**📌 Key Points to Remember**

* Use IDENTITY\_INSERT OFF when letting SQL Server auto-generate the identity column value.
* Use IDENTITY\_INSERT ON to manually insert a value into the identity column.
* When inserting into a table with an identity column and IDENTITY\_INSERT is OFF, do **not** specify the identity column in the INSERT statement.
* Only one table can have IDENTITY\_INSERT ON per session.

**Question 23:**

Consider the following code:

CREATE FUNCTION ufn\_FunctionOne(@NumOne VARCHAR(10), @NumTwo FLOAT)

RETURNS INT

AS

BEGIN

DECLARE @Sum INT

SET @Sum = CAST(@NumOne AS INT) - CONVERT(INT, @NumTwo)

RETURN @Sum

END

What would be the output if the following code is executed after the function is created successfully?

SELECT dbo.ufn\_FunctionOne('100', 10.50)

**Options:**

1. 91
2. 89.50
3. 90
4. Error – Conversion failed while converting INT to FLOAT
5. Error – Conversion failed while converting FLOAT to INT

**🧠 Concept: Data Type Conversion in SQL Server Functions**

SQL Server allows **explicit data type conversions** using:

* CAST(expression AS datatype)
* CONVERT(datatype, expression)

**Key behavior:**

* When converting from FLOAT to INT, the **decimal part is truncated**, not rounded.
* A VARCHAR containing numeric characters (e.g., '100') can be successfully cast to INT.

**🎯 Step-by-Step Analysis**

**Function Definition:**

CREATE FUNCTION ufn\_FunctionOne(@NumOne VARCHAR(10), @NumTwo FLOAT)

RETURNS INT

AS

BEGIN

DECLARE @Sum INT

SET @Sum = CAST(@NumOne AS INT) - CONVERT(INT, @NumTwo)

RETURN @Sum

END

**Function Call:**

SELECT dbo.ufn\_FunctionOne('100', 10.50)

**Evaluation:**

1. @NumOne = '100' → CAST('100' AS INT) = 100 ✅
2. @NumTwo = 10.50 → CONVERT(INT, 10.50) = 10 ✅
3. Result: 100 - 10 = 90

**✅ Correct Answer: c) 90**

**📌 Key Points to Remember**

* CAST('100' AS INT) = 100 (valid string-to-int conversion)
* CONVERT(INT, 10.50) = 10 (decimal part truncated)
* SQL Server truncates, not rounds, when converting from FLOAT to INT
* No error occurs in this case because all conversions are valid

**Question 24:**

What will be the output when the following code is executed?

CREATE TABLE Employees

(

EmployeeNumber TINYINT IDENTITY PRIMARY KEY,

EmployeeName VARCHAR(20)

)

INSERT INTO Employees VALUES('Prem'), (NULL), ('John')

SET ANSI\_NULLS ON

SELECT COUNT(\*) FROM Employees WHERE EmployeeName IS NULL

SELECT COUNT(\*) FROM Employees WHERE EmployeeName = NULL

SET ANSI\_NULLS OFF

**Options:**

1. 0 0
2. 1 0
3. 0 1
4. 1 1

**🧠 Concept: NULL Comparisons and ANSI\_NULLS in SQL Server**

* IS NULL is the **correct** way to check for NULL values.
* Using = NULL **never works** as expected; it always evaluates to **UNKNOWN**, regardless of ANSI\_NULLS setting.
* ANSI\_NULLS ON means: comparisons using = NULL return **UNKNOWN**, which behaves like **FALSE** in filters.
* Even with ANSI\_NULLS OFF, = NULL **still doesn't behave reliably** and should **never** be used.

**🎯 Step-by-Step Analysis**

**Table Creation:**

CREATE TABLE Employees

(

EmployeeNumber TINYINT IDENTITY PRIMARY KEY,

EmployeeName VARCHAR(20)

)

**Data Insertion:**

INSERT INTO Employees VALUES('Prem'), (NULL), ('John')

* Inserts 3 rows:
  + EmployeeName = 'Prem'
  + EmployeeName = NULL
  + EmployeeName = 'John'

**Query 1:**

SET ANSI\_NULLS ON

SELECT COUNT(\*) FROM Employees WHERE EmployeeName IS NULL

* ✅ Correct way to filter NULL
* One record matches → **Result: 1**

**Query 2:**

SELECT COUNT(\*) FROM Employees WHERE EmployeeName = NULL

* ❌ Invalid comparison
* = NULL always returns **UNKNOWN**
* No rows match → **Result: 0**

**✅ Correct Answer: b) 1 0**

**📌 Key Points to Remember**

* Always use IS NULL or IS NOT NULL for checking NULL values.
* = NULL and <> NULL always return **UNKNOWN**, not TRUE or FALSE.
* ANSI\_NULLS affects some behaviors, but = NULL never works for filtering.
* Avoid relying on SET ANSI\_NULLS OFF; it is deprecated and can produce inconsistent results.

**Question 25:**

Consider the following code:

CREATE TABLE Stream

(

StreamId SMALLINT IDENTITY(10, -2),

StreamName VARCHAR(20)

)

INSERT INTO Stream VALUES('MS')

SET IDENTITY\_INSERT Stream ON

INSERT INTO Stream(StreamId, StreamName) VALUES(0, 'MF')

SET IDENTITY\_INSERT Stream OFF

INSERT INTO Stream VALUES('BIM')

GO

What is the last generated value of StreamId?

**Options:**

1. 2
2. 8
3. -2
4. Error - Negative value is not allowed in identity

**🧠 Concept: IDENTITY(seed, increment) and IDENTITY\_INSERT in SQL Server**

* IDENTITY(seed, increment) defines how values are **automatically generated** in a column.
* **Seed** is the starting value; **increment** determines how values increase (or decrease if negative).
* SQL Server **supports negative identity increments**.
* When using SET IDENTITY\_INSERT ON, you manually insert values into the identity column.
* After manual insertions, auto-generated values **continue from the last identity value inserted by the system**, not manually inserted ones.

**🎯 Step-by-Step Analysis**

**Table Definition:**

CREATE TABLE Stream

(

StreamId SMALLINT IDENTITY(10, -2),

StreamName VARCHAR(20)

)

* Identity starts at 10 and **decreases by 2** each time: 10, 8, 6, 4, ...

**First Insert:**

INSERT INTO Stream VALUES('MS')

* Auto-generates StreamId = 10

**Manual Insert (with IDENTITY\_INSERT ON):**

SET IDENTITY\_INSERT Stream ON

INSERT INTO Stream(StreamId, StreamName) VALUES(0, 'MF')

SET IDENTITY\_INSERT Stream OFF

* Manually inserts StreamId = 0
* SQL Server does **not** change the identity counter for manual insert

**Next Auto-Insert:**

INSERT INTO Stream VALUES('BIM')

* Identity values continue from last auto value: 10
* Next value = 10 - 2 = 8

✅ So the **last generated identity value** is **8**

**✅ Correct Answer: b) 8**

**📌 Key Points to Remember**

* SQL Server **allows negative identity increments**.
* IDENTITY\_INSERT lets you manually insert into an identity column.
* Manual identity inserts **do not affect** the identity counter.
* After resuming automatic inserts, SQL Server continues from the **last auto-generated** identity value using the defined increment.