QUERYCRAFT

HANDBOOK

A Beginner's Handbook to
SQL Server Mastery

By
Vidya Niwas Pandey

QueryCraft

A Hands-On Beginner's Tutorial

By Vidya Niwas Pandey

Brief Author Bio:

An Application Developer proficient in SQL, Vidya Niwas Pandey holds a Master's in Computer Application and a Bachelor's in Science. With a rich background spanning two years of computer teaching, his expertise is geared towards making SQL accessible to beginners.

Introduction/Foreword:

Welcome to QueryCraft, where the art of crafting SQL queries comes to life! In this hands-on tutorial, we embark on a journey from the fundamentals to mastery. Whether you're a novice or just seeking a refresher, this book is your guide to becoming proficient in SQL Server queries. Let's explore the world of databases together!

Acknowledgments:

I would like to express my gratitude to [Acknowledged Person/Organization] for their valuable insights and support during the creation of this book.

Copyright Information:

Contact Information:

For inquiries, feedback, or further assistance, please contact Vidya Niwas Pandey at vidyaniwas2@gmail.com.

Table of Contents

GETTING STARTED

DATA MANIPULATION

31. INSERT Multiple Rows
32. INSERT INTO SELECT
33. UPDATE
34. UPDATE JOIN
35. DELETE
36. MERGE
37. PIVOT
38. Transaction

DATABASE NORMALIZATION

1. First Normal Form (1 NF)
2. Second Normal Form (2 NF)
3. Third Normal Form (3 NF)
4. Boyce Codd Normal Form or Fourth Normal Form (BCNF or 4 NF)
5. Fifth Normal Form (5 NF)
6. Sixth Normal Form (6 NF)

DATA DEFINITION

1. Create New Database
2. Drop Database
3. Create Schema
4. Alter Schema
5. Drop Schema
6. Create New Table
    - Identity Column
    - Sequence
7. Add Column
8. Modify Column
9. Drop Column
10. Computed Columns
11. Rename Table
12. Drop Table
13. Truncate Table
14. Temporary Tables
15. Synonym
16. SELECT INTO
17. PRIMARY KEY
18. FOREIGN KEY
19. CHECK Constraint
20. UNIQUE Constraint
21. NOT NULL Constraint

CREATING INDEXES IN SQL SERVER

1. Clustered Index
2. Non-Clustered Index
3. Unique Index
4. Filtered Index
5. Column Store Index
6. Hash Index

EXPRESSIONS

1. CASE
2. COALESCE
3. NULLIF

AGGREGATE FUNCTION

1. AVG
2. CHECKSUM_AGG
3. COUNT
4. COUNT_BIG
5. MAX
6. MIN
7. STDEV
8. STDEVP
9. SUM
10. VAR
11. VARP

STRING FUNCTIONS

1. ASCII
2. CHAR
3. CHARINDEX
4. CONCAT
5. CONCAT_WS
6. DIFFERENCE
7. FORMAT
8. LEFT
9. LEN
10. LOWER
11. LTRIM
12. NCHAR
13. PATINDEX
14. QUOTENAME
15. REPLACE
16. REPLICATE
17. REVERSE
18. RIGHT

19. RTRIM
20. SOUNDEX
21. SPACE
22. STR
23. STRING_AGG
24. STRING_ESCAPE
25. STRING_SPLIT
26. STUFF
27. SUBSTRING
28. TRANSLATE
29. TRIM
30. UNICODE
31. UPPER

SYSTEM FUNCTIONS

1. CAST
2. CONVERT
3. CHOOSE
4. ISNULL
5. ISNUMERIC
6. IIF
7. TRY_CAST
8. TRY_CONVERT
9. TRY_PARSE
10. Convert date time to string
11. Convert string to date time
12. Convert date time to date

WINDOW FUNCTIONS

1. CUME_DIST
2. DENSE_RANK
3. FIRST_VALUE
4. LAG
5. LAST_VALUE
6. LEAD
7. NTILE
8. PERCENT_RANK
9. RANK
10. ROW_NUMBER

DATE FUNCTIONS

1. CURRENT_TIMESTAMP
2. GETUTCDATE
3. GETDATE
4. SYSDATETIME
5. SYSUTCDATETIME
6. SYSDATETIMEOFFSET

RETURNING THE DATE AND TIME PARTS

1. DATENAME
2. DATEPART
3. DAY
4. MONTH
5. YEAR

RETURNING A DIFFERENCE BETWEEN TWO DATES

1. DATEDIFF

MODIFYING DATES

1. DATEADD
2. EOMONTH
3. SWITCHOFFSET
4. TODATETIMEOFFSET

CONSTRUCTING DATE AND TIME FROM THEIR PARTS

1. DATEFROMPARTS
2. DATETIME2FROMPARTS
3. DATETIMEOFFSETFROMPARTS
4. TIMEFROMPARTS

VALIDATING DATE AND TIME VALUES

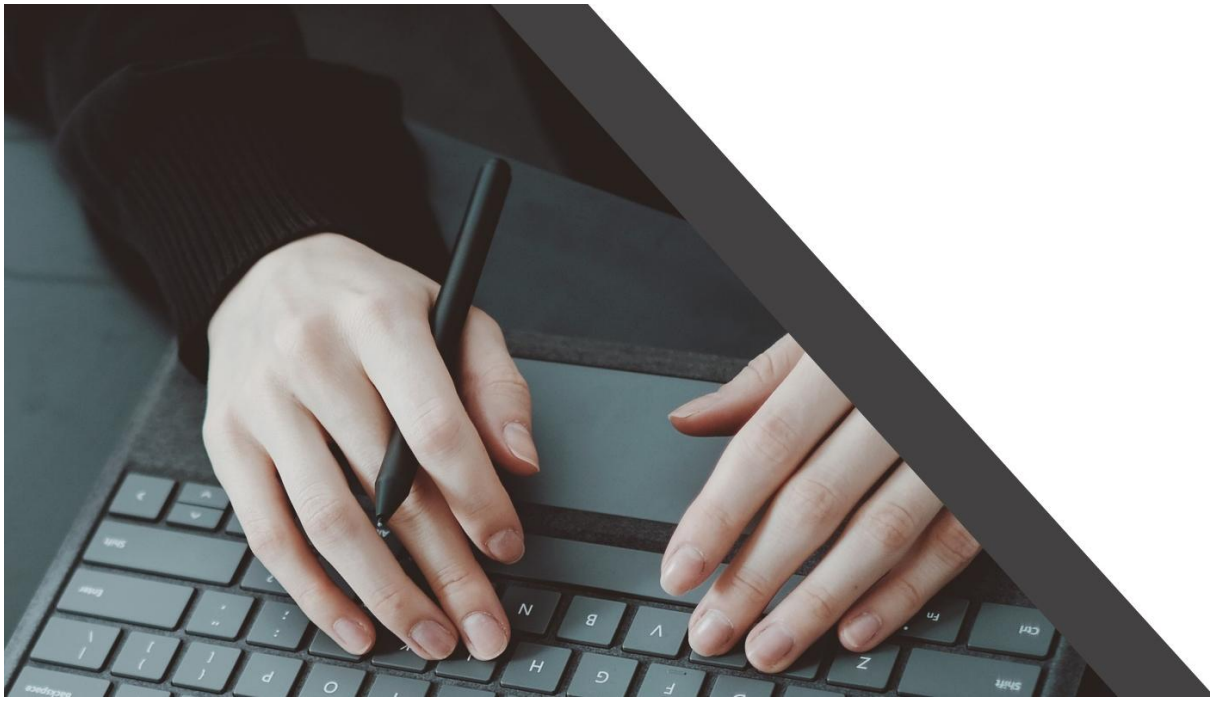
1. Function
2. ISDA

SQL SERVER STORED PROCEDURE

1. Create a Stored Procedure in SQL Server with input parameters
2. Stored Procedure to insert data
3. Stored Procedure to update table
4. Stored Procedure to select data from table
5. Stored Procedure to delete data from table
6. Stored Procedure to validate username and password
7. Stored Procedure in SQL to add two numbers
8. Stored Procedure in SQL with multiple queries
9. Stored Procedure for insert and update in SQL Server with output parameter
10. SQL Server Stored Procedure to list columns
11. Dynamic where clause in SQL Server Stored Procedure
12. SQL Server Stored Procedure return select result concatenate

Tables in SQL Server

1. Creating a System-Versioned Temporal Table
2. Modifying Data in a System-Versioned Temporal Table
3. Views
4. Loops
5. #Tables



GETTING STARTED

1. What is SQL SERVER?

SQL Server is a powerful relational database management system (RDBMS) developed by Microsoft. It provides a secure and scalable platform for managing and storing data. Here are key points to understand about SQL Server:

- **Definition:** SQL Server is a software product that manages the storage, organization, retrieval, and security of data in a relational database.
- **Key Components:**
    - **Database Engine:** The core service for storing, processing, and securing data.
    - **SQL Server Management Studio (SSMS):** A tool for managing SQL Server databases.
    - **Integration Services (SSIS), Analysis Services (SSAS), Reporting Services (SSRS):** Additional services for ETL (Extract, Transform, and Load), data analysis, and reporting.
- **Versions:** SQL Server has various editions, including express (free), Standard, and Enterprise, each with different features and capabilities.
- **Use Cases:** SQL Server is used in various industries for applications ranging from small-scale projects to enterprise-level systems.
- **SQL Language:** SQL Server uses the SQL (Structured Query Language) for querying and manipulating data. Understanding SQL is crucial for effective interaction with the database.

2. Install the SQL Server

Before you can start working with SQL Server, you need to install it on your machine. Here's a step-by-step guide on installing SQL Server:

- **System Requirements:** Check the system requirements for the version of SQL Server you plan to install.
- **Download and Run Installer:** Visit the official Microsoft website to download the SQL Server installer. Run the installer to begin the installation process.
- **Setup Wizard:** Follow the setup wizard, providing necessary information such as the edition you want to install, licensing terms, and installation type (standalone or custom).
- **Feature Selection:** Choose the SQL Server features you want to install. Common features include the Database Engine and SQL Server Management Tools.
- **Instance Configuration:** Configure the SQL Server instance by specifying a unique instance name, and choose between default or named instances.
- **Server Configuration:** Set up authentication mode (Windows Authentication or Mixed Mode) and provide credentials for the SQL Server.

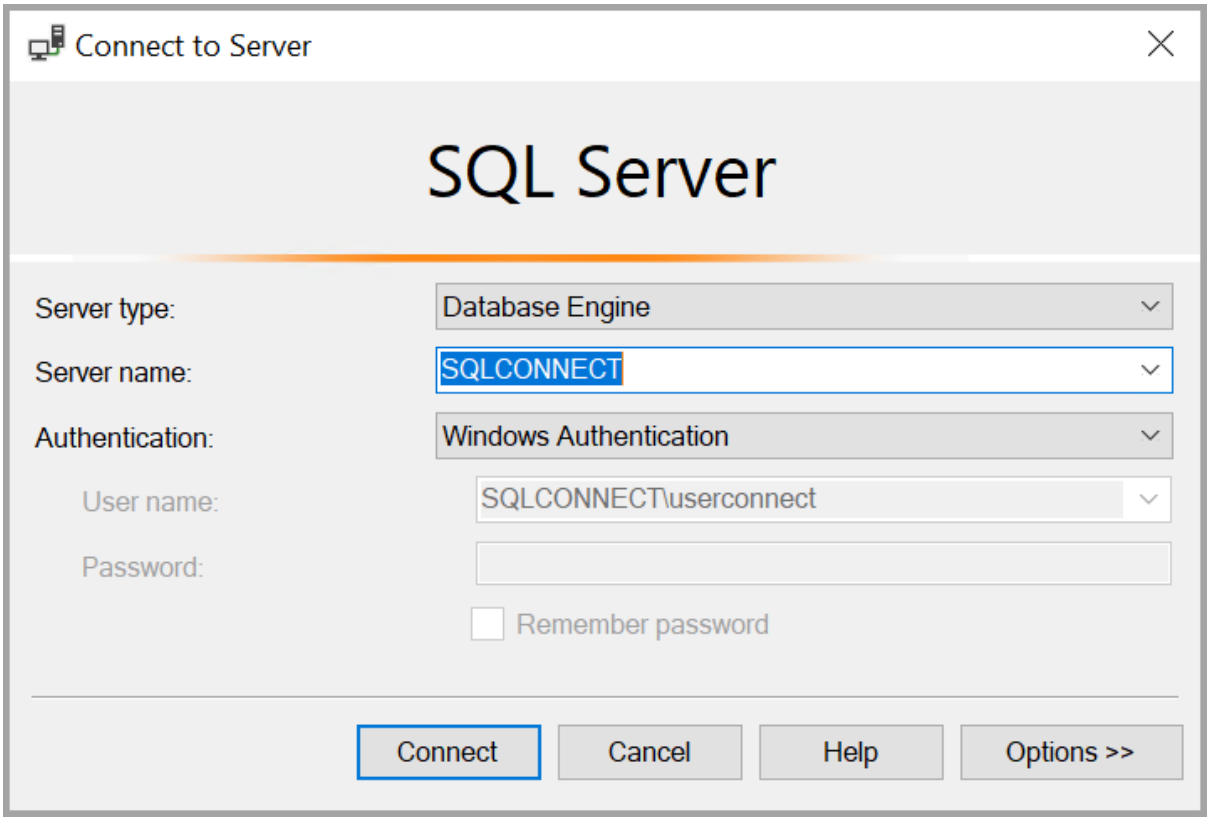
- **Database Engine Configuration:** Configure server authentication mode and add SQL Server administrators.
- **Installation:** Begin the installation process. Once completed, you'll have a fully functional SQL Server instance on your machine.
- **Verification:** Verify the successful installation by connecting to the SQL Server instance using SQL Server Management Studio.

This initial setup lays the foundation for your journey with SQL Server, enabling you to start creating databases, tables, and executing queries.

3. Connect to the SQL Server

Now that you've set up your digital file cabinet (SQL Server), let's talk about how to open it and start using it.

- **Open the Cabinet:** Imagine you have the digital file cabinet in your home office. To use it, you need to open it, right?
- **Connect to SQL Server:** In the computer world, this is called "connecting to SQL Server." It's like opening the door to your digital file cabinet. You do this using a tool called SQL Server Management Studio (SSMS).
- **Enter the Room (SSMS):** When you open SSMS, it's like entering the room where your digital file cabinet is. SSMS is your way of interacting with SQL Server.
- **Provide Some Details:** Just like you need a key to enter a room, SSMS needs some information to connect to your SQL Server. This includes the name of your server (where the digital file cabinet is located) and your login details.

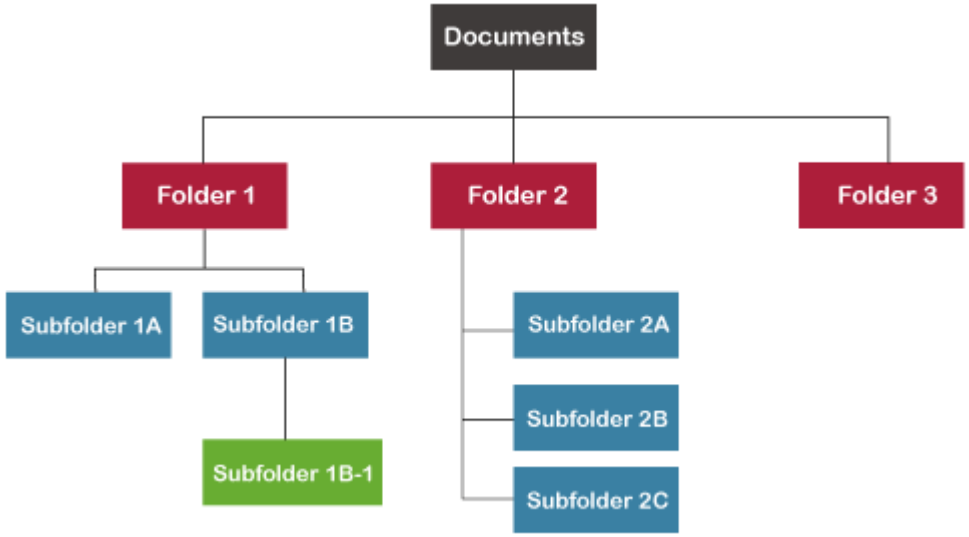


- **Success! You're In:** Once you click "Connect," it's like turning the key and walking into the room. Now, you can see all the drawers and sections inside your digital file cabinet.

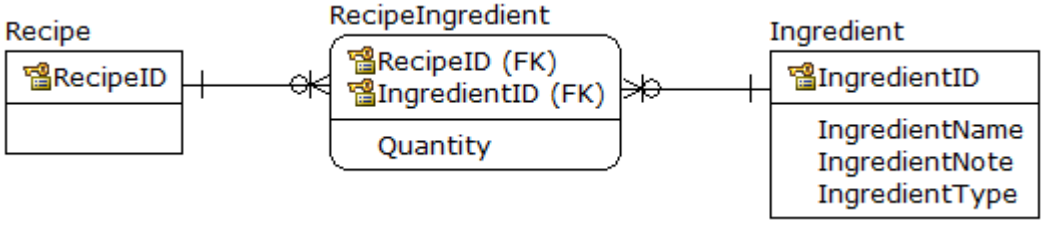
4. SQL Server Sample Database

Now that you're inside the room with your digital file cabinet (SQL Server), you might be wondering what kind of information you can store and organize. Let's talk about having a "Sample Database."

- **Empty Drawers:** Imagine your digital file cabinet is empty right now. A sample database is like a set of folders and documents that someone already put in the drawers. It's there to help you understand how to organize and manage information.



- **Learn by Example:** Just like learning to cook by following a recipe, a sample database lets you see how things are organized and how different types of information are stored.



- **Practice Zone:** You can play around with this sample database, try out different SQL queries (remember, it's like giving instructions to the cabinet), and see how the information responds.

- **No Fear of Messing Up:** Since it's just a sample, you don't have to worry about making mistakes. It's a safe space to practice and get comfortable with your digital file cabinet.

- **Ready to Explore:** Once you've connected to SQL Server and have a sample database, you're ready to explore and start using your digital file cabinet to organize and manage your data.

5. Load Sample Database

Now that you're inside your digital file cabinet (SQL Server) and have seen the sample folders and documents, let's talk about how to "load" or put in a sample database.

- **Empty Shelves:** Right now, your digital file cabinet might have empty shelves. Loading a sample database is like bringing in a set of folders and documents to fill those shelves.
- **Purpose of a Sample Database:** Think of it as a practice set. It contains fictional data that you can use to understand how to work with real information later.
- **Finding the Right Sample:** There are various sample databases available online. Some are designed for beginners, while others simulate real-world scenarios. Choose one that fits your learning goals.
- **Downloading the Sample:** It's like getting a digital box of folders and documents. Download the sample database from a reliable source, and it usually comes in a compressed file.
- **Unpacking the Box:** Once downloaded, you need to "unzip" or extract the files. It's like taking items out of a physical box and placing them on your desk.
- **Knowing Your Cabinet's Address:** Before loading the sample, make sure you know where your SQL Server is located (its address). This is crucial for placing the sample in the right spot.
- **Using SQL Server Management Studio (SSMS):** Open SSMS, the tool you use to interact with your digital file cabinet. It's like having your work desk ready.
- **Running a Script:** The sample usually comes with a script – a set of instructions in SQL language. Running this script is like following a recipe to organize the folders and documents in your digital file cabinet.
- **Reviewing the Contents:** After running the script, explore the contents of your sample database. It's like opening the folders and seeing what's inside.
- **Practice, Explore, Learn:** Now, you can practice writing queries (instructions) to retrieve information from your sample database. It's your safe space to explore and learn without worrying about making real mistakes.

Create a simple table named "Products" and insert some sample data for the examples. We'll assume a basic structure with columns like **ProductName**, **Price**, **Category**, and **Description**.

Create the Products table

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(50),
    Price DECIMAL(10, 2),
    Category VARCHAR(50),
    Description VARCHAR(100),
    ManufactureDate DATE
);
```

Insert some sample data

```
INSERT INTO Products (ProductID, ProductName, Price, Category, Description, ManufactureDate)
VALUES
    (1, 'Laptop', 800.00, 'Electronics', 'High-performance laptop', '2022-01-10'),
    (2, 'Smartphone', 500.00, 'Electronics', 'Latest smartphone model', '2022-02-15'),
    (3, 'T-shirt', 20.00, 'Clothing', 'Comfortable cotton t-shirt', '2022-03-05'),
    (4, 'Headphones', 150.00, 'Electronics', 'Noise-canceling headphones', '2022-04-20'),
    (5, 'Jeans', 40.00, 'Clothing', 'Classic blue jeans', '2022-05-12'),
    (6, 'Tablet', 300.00, 'Electronics', 'Compact tablet device', '2022-06-30'),
     (7, 'Sneakers', 60.00, 'Footwear', 'Running sneakers', '2022-07-18');
```

Create the Order table

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    ProductID INT,
    Quantity INT,
    OrderDate DATE,
    CONSTRAINT FK_ProductID FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);

INSERT INTO Orders (OrderID, ProductID, Quantity, OrderDate)
VALUES
    (1, 1, 2, '2023-01-15'),
    (2, 2, 1, '2023-01-16'),
    (3, 1, 1, '2023-01-17'),
     Add more rows as needed
    (4, 3, 3, '2023-01-18');
```

DATA MANIPULATION

1. SELECT

This query retrieves the names and prices of all products from the Products table.

```
SELECT ProductName, Price FROM Products;
```

The **SELECT** statement is the foundation of SQL queries. It specifies the columns you want to retrieve from a table. In this example, we're retrieving the **ProductName** and **Price** columns from the "Products" table.

2. ORDER BY

This query retrieves product names and prices from the Products table, ordered by price in descending order.

```
SELECT ProductName, Price FROM Products ORDER BY Price DESC
```

The **ORDER BY** clause is used to sort the result set in ascending (**ASC**) or descending (**DESC**) order based on one or more columns. In this example, we're ordering the products by price in descending order.

3. OFFSET FETCH

This query skips the first 5 rows and fetches the next 10 rows of products from the Products table, ordered by price.

```
SELECT ProductName, Price FROM Products ORDER BY Price OFFSET 5 ROWS FETCH NEXT 10 ROWS ONLY;
```

The **OFFSET** and **FETCH** clauses are used for pagination. **OFFSET** skips a specified number of rows, and **FETCH** retrieves a specified number of rows. This is useful for implementing page-wise data retrieval.

4. SELECT TOP

This query retrieves the top 5 product names and prices from the Products table, ordered by price.

```
SELECT TOP 5 ProductName, Price FROM Products ORDER BY Price;
```

The **SELECT TOP** clause limits the number of rows returned by a query. In this example, we're retrieving the top 5 product names and prices from the "Products" table, ordered by price.

5. SELECT DISTINCT

This query retrieves unique product categories from the Products table.

```
SELECT DISTINCT Category FROM Products;
```

The **SELECT DISTINCT** statement is used to retrieve unique values from a column. In this example, we're retrieving unique product categories from the "Products" table.

6. WHERE

This query retrieves product names and prices from the Products table where the category is 'Electronics'.

```
SELECT ProductName, Price FROM Products WHERE Category = 'Electronics';
```

The **WHERE** clause is used to filter rows based on a specified condition. In this example, we're filtering products where the category is 'Electronics'.

7. NULL

This query retrieves product names from the Products table where the description is missing (NULL).

```
SELECT ProductName FROM Products WHERE Description IS NULL;
```

The **NULL** keyword represents missing or unknown data. In this example, we're retrieving product names where the description is not specified (NULL).

8. AND

This query retrieves product names and prices from the Products table where the category is 'Electronics' and the price is greater than 500.

```
SELECT ProductName, Price FROM Products WHERE Category = 'Electronics' AND Price > 500;
```

The **AND** operator combines multiple conditions in a **WHERE** clause. In this example, we're retrieving products where the category is 'Electronics' and the price is greater than 500.

9. OR

This query retrieves product names and prices from the Products table where the category is either 'Electronics' or 'Clothing'.

```
SELECT ProductName, Price FROM Products WHERE Category = 'Electronics' OR Category = 'Clothing';
```

The **OR** operator combines multiple conditions in a **WHERE** clause. In this example, we're retrieving products where the category is either 'Electronics' or 'Clothing'.

10. IN

This query retrieves product names and prices from the Products table where the category is either 'Electronics' or 'Clothing'.

```
SELECT ProductName, Price FROM Products WHERE Category IN ('Electronics', 'Clothing');
```

The **IN** operator is used to specify multiple values in a **WHERE** clause. In this example, we're retrieving products where the category is either 'Electronics' or 'Clothing'.

11. BETWEEN

This query retrieves product names and prices from the Products table where the price is between $50 and $100.

```
SELECT ProductName, Price FROM Products WHERE Price BETWEEN 50 AND 100;
```

The BETWEEN operator is used to filter rows based on a range of values. In this example, we're retrieving products where the price is between $50 and $100.

12. LIKE

Contains the word 'phone'.

```
SELECT ProductName FROM Products WHERE ProductName LIKE '%phone%';
```

The LIKE operator is used for pattern matching in a WHERE clause. In this example, we're retrieving products where the product name contains the word 'phone'.

13. Column & Table Aliases

This query retrieves product names and their prices, using aliases for column names.

```
SELECT PName = ProductName, Cost = Price FROM Products;
```

Aliases are used to give a table or column a temporary name in a query. In this example, we're assigning aliases PName and Cost to the ProductName and Price columns, respectively.

Create a second table named "Categories" to use in the examples involving joins. This table will store information about different product categories.

Create the Categories table

```
CREATE TABLE Categories ( CategoryID INT PRIMARY KEY, CategoryName VARCHAR(50) );
```

Insert some sample data into the Categories table

```
INSERT INTO Categories (CategoryID, CategoryName) VALUES (1, 'Electronics'), (2, 'Clothing'), (3, 'Footwear'), (4, 'Accessories');
```

In this example:

- The **Categories** table has columns for **CategoryID** and **CategoryName**.
- The **CategoryID** is the primary key, ensuring each category has a unique identifier.
- We've inserted some sample data with different category IDs and names.

14. Joins

Joins are used to combine rows from two or more tables based on a related column.

15. INNER JOIN

This query retrieves product names and their categories from the Products and Categories tables using INNER JOIN.

```
SELECT ProductName, CategoryName FROM Products INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

An INNER JOIN returns only the rows where there is a match in both tables. In this example, we're retrieving product names and their categories.

16. LEFT JOIN

This query retrieves all product names and their categories from the Products and Categories tables using LEFT JOIN.

```
SELECT ProductName, CategoryName FROM Products LEFT JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

A LEFT JOIN returns all rows from the left table and matching rows from the right table. In this example, we're retrieving all product names and their categories, even if there is no matching category.

17. RIGHT JOIN

This query retrieves all categories and their associated product names from the Products and Categories tables using RIGHT JOIN.

```
SELECT ProductName, CategoryName FROM Products RIGHT JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

A RIGHT JOIN returns all rows from the right table and matching rows from the left table. In this example, we're retrieving all categories and their associated product names, even if there are no matching products.

18. FULL OUTER JOIN

This query retrieves all product names and their categories from the Products and Categories tables using FULL OUTER JOIN.

```
SELECT ProductName, CategoryName FROM Products FULL OUTER JOIN Categories
ON Products.CategoryID = Categories.CategoryID;
```

A FULL OUTER JOIN returns all rows when there is a match in either the left or right table. In this example, we're retrieving all product names and their categories, including those without matching products or categories.

19. Self-Join

This query retrieves employees and their managers from an Employees table using a self-join.

```
SELECT e.EmployeeName, m.EmployeeName AS ManagerName FROM Employees e
JOIN Employees m ON e.ManagerID = m.EmployeeID;
```

A self-join is a regular join, but the table is joined with itself. In this example, we're retrieving employees and their respective managers.

20. CROSS JOIN

This query retrieves all possible combinations of product names and categories from the Products and Categories tables using CROSS JOIN.

```
SELECT ProductName, CategoryName FROM Products CROSS JOIN Categories;
```

A CROSS JOIN returns the Cartesian product of the two tables, i.e., all possible combinations of rows. In this example, we're retrieving all combinations of product names and categories. Top of Form

21. GROUP BY

This query retrieves the average price for each product category from the Products table.

```
SELECT Category, AVG(Price) AS AvgPrice
FROM Products
GROUP BY Category;
```

The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows. In this example, we're grouping products by category and calculating the average price for each category.

22. HAVING

This query retrieves product categories and their average prices, but only for categories with an average price greater than $50.

```
SELECT Category, AVG(Price) AS AvgPrice
FROM Products
GROUP BY Category
HAVING AVG(Price) > 50;
```

The HAVING clause is used in combination with GROUP BY to filter the results based on a condition applied to the grouped data. In this example, we're retrieving product categories and their average prices, but only for categories with an average price greater than $50.

23. GROUPING SETS

This query retrieves the total count of products and the average price, grouped by both category and manufacturer.

```
SELECT Category, Manufacturer, COUNT(*) AS ProductCount, AVG(Price) AS AvgPrice
FROM Products
GROUP BY GROUPING SETS (Category, Manufacturer);
```

The GROUPING SETS clause allows you to specify multiple groupings in a single query. In this example, we're retrieving the total count of products and the average price, grouped by both category and manufacturer.

24. CUBE

This query retrieves the total count of products and the average price, considering all possible combinations of category, manufacturer, and year.

```
SELECT Category, Manufacturer, YEAR(ManufactureDate) AS ProductionYear,
COUNT(*) AS ProductCount, AVG(Price) AS AvgPrice

FROM Products

GROUP BY CUBE (Category, Manufacturer, YEAR(ManufactureDate));
```

The CUBE operation generates all possible combinations of grouping sets. In this example, we're retrieving the total count of products and the average price, considering all possible combinations of category, manufacturer, and year of manufacture.

25. ROLLUP

This query retrieves the total count of products and the average price, providing subtotals for each category and an overall total.

```
SELECT Category, Manufacturer, COUNT(*) AS ProductCount, AVG(Price) AS AvgPrice

FROM Products

GROUP BY ROLLUP (Category, Manufacturer);
```

The ROLLUP operation provides subtotals and grand totals for a set of columns. In this example, we're retrieving the total count of products and the average price, providing subtotals for each category and an overall total.

26. Subquery

This query retrieves products with prices greater than the average price for their respective categories.

```
SELECT ProductName, Price, Category

FROM Products

WHERE Price > (SELECT AVG(Price) FROM Products AS Subquery WHERE
Subquery.Category = Products.Category);
```

A subquery is a query embedded within another query. In this example, we're retrieving products with prices greater than the average price for their respective categories using a subquery.

27. Correlated Subquery

This query retrieves products with prices greater than the average price for their respective categories, using a correlated subquery.

```
SELECT ProductName, Price, Category

FROM Products AS p

WHERE Price > (SELECT AVG(Price) FROM Products AS Subquery WHERE Subquery.Category = p.Category);
```

A correlated subquery is a subquery that refers to columns of the outer query. In this example, we're using a correlated subquery to retrieve products with prices greater than the average price for their respective categories.

28. EXISTS

This query retrieves product categories that have at least one product with a price greater than $100.

```
SELECT DISTINCT Category

FROM Products AS p

WHERE EXISTS (SELECT 1 FROM Products WHERE Category = p.Category AND Price > 100);
```

The EXISTS keyword is used to check the existence of rows in a subquery. In this example, we're retrieving product categories that have at least one product with a price greater than $100.

29. ANY

This query retrieves product names with prices greater than any price in the 'Electronics' category.

```
SELECT ProductName, Price

FROM Products
```

```
WHERE Price > ANY (SELECT Price FROM Products WHERE Category = 'Electronics');
```

The ANY keyword is used to compare a value to a set of values. In this example, we're retrieving product names with prices greater than any price in the 'Electronics' category.

30. ALL

This query retrieves product names with prices greater than all prices in the 'Clothing' category.

```
SELECT ProductName, Price
FROM Products
WHERE Price > ALL (SELECT Price FROM Products WHERE Category = 'Clothing');
```

The ALL keyword is used to compare a value to all values in a set. In this example, we're retrieving product names with prices greater than all prices in the 'Clothing' category.

31. UNION

This query retrieves distinct product names from both the 'Electronics' and 'Clothing' categories.

```
SELECT ProductName
FROM Products
WHERE Category = 'Electronics'
UNION
SELECT ProductName
FROM Products
WHERE Category = 'Clothing';
```

The UNION operator is used to combine the results of two or more SELECT statements, eliminating duplicates. In this example, we're retrieving distinct product names from both the 'Electronics' and 'Clothing' categories.

32. INTERSECT

This query retrieves product names that exist in both the 'Electronics' and 'Clothing' categories.

```
SELECT ProductName
FROM Products
WHERE Category = 'Electronics'
INTERSECT
SELECT ProductName
FROM Products
WHERE Category = 'Clothing';
```

The INTERSECT operator is used to retrieve common rows between two SELECT statements. In this example, we're retrieving product names that exist in both the 'Electronics' and 'Clothing' categories.

33. EXCEPT

This query retrieves product names from the 'Electronics' category that do not exist in the 'Clothing' category.

```
SELECT ProductName
FROM Products
WHERE Category = 'Electronics'
EXCEPT
SELECT ProductName
FROM Products
WHERE Category = 'Clothing';
```

The EXCEPT operator is used to retrieve rows from the first SELECT statement that do not exist in the second SELECT statement. In this example, we're retrieving product names from the 'Electronics' category that do not exist in the 'Clothing' category.

34. Common Table Expression (CTE)

This query uses a Common Table Expression (CTE) to retrieve products with prices greater than $200.

```
WITH ExpensiveProducts AS (
    SELECT ProductName, Price
    FROM Products
    WHERE Price > 200
)
SELECT * FROM ExpensiveProducts;
```

A Common Table Expression (CTE) is a named temporary result set defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement. In this example, we're using a CTE to retrieve products with prices greater than $200.

35. Recursive CTE

This query uses a Recursive Common Table Expression (CTE) to generate a sequence of numbers from 1 to 5.

```
WITH RecursiveSequence AS (
    SELECT 1 AS Number
    UNION ALL
    SELECT Number + 1
    FROM RecursiveSequence
    WHERE Number < 5
)
SELECT * FROM RecursiveSequence;
```

A Recursive Common Table Expression (CTE) is used to perform recursive queries, such as traversing hierarchical data. In this example, we're using a recursive CTE to generate a sequence of numbers from 1 to 5.

36. INSERT

This query inserts a new product into the Products table.

```
INSERT INTO Products (ProductName, Price, Category)
VALUES ('Smartwatch', 150.00, 'Electronics');
```

The INSERT statement is used to add new rows to a table. In this example, we're inserting a new product named 'Smartwatch' into the "Products" table.

37. INSERT Multiple Rows

This query inserts multiple new products into the Products table.

```
INSERT INTO Products (ProductName, Price, Category)
VALUES
    ('Backpack', 30.00, 'Accessories'),
    ('Sunglasses', 25.00, 'Accessories'),
    ('Fitness Tracker', 80.00, 'Electronics');
```

The INSERT statement can be used to insert multiple rows in a single query. In this example, we're inserting multiple new products into the "Products" table.

38. INSERT INTO SELECT

This query inserts products from the 'Electronics' category into a new table named 'ElectronicsProducts'.

```
INSERT INTO ElectronicsProducts (ProductName, Price)
SELECT ProductName, Price
FROM Products
WHERE Category = 'Electronics';
```

The INSERT INTO SELECT statement is used to insert rows into a table from the result of a SELECT statement. In this example, we're inserting products from the 'Electronics' category into a new table named 'ElectronicsProducts'.

39. UPDATE

This query updates the price of all products in the 'Clothing' category to $50.

```
UPDATE Products
SET Price = 50
WHERE Category = 'Clothing';
```

The UPDATE statement is used to modify existing records in a table. In this example, we're updating the price of all products in the 'Clothing' category to $50.

40. UPDATE JOIN

This query updates the price of products in the 'Electronics' category by increasing it by 10%, using a JOIN.

```
UPDATE Products
SET Price = Price * 1.10
FROM Products
JOIN Categories ON Products.CategoryID = Categories.CategoryID
WHERE Categories.Category = 'Electronics';
```

The UPDATE JOIN statement is used to update data in one table based on data from another table. In this example, we're updating the price of products in the 'Electronics' category by increasing it by 10%.

41. DELETE

This query deletes all products with a price less than $20.

```
DELETE FROM Products
WHERE Price < 20;
```

The DELETE statement is used to remove rows from a table based on a specified condition. In this example, we're deleting all products with a price less than $20.

42. MERGE

This query uses the MERGE statement to synchronize data between a source and target table.

```
MERGE INTO TargetTable AS Target
USING SourceTable AS Source
ON Target.ID = Source.ID
WHEN MATCHED THEN
    UPDATE SET Target.ColumnName = Source.ColumnName
WHEN NOT MATCHED THEN
    INSERT (ID, ColumnName1, ColumnName2)
    VALUES (Source.ID, Source.ColumnName1, Source.ColumnName2);
```

The MERGE statement is used to perform an upsert operation, combining the operations of INSERT, UPDATE, and DELETE based on a specified condition. In this example, we're synchronizing data between a source and target table.

43. PIVOT

This query uses the PIVOT operator to transform rows into columns, displaying total sales by category.

```
SELECT *
FROM (
    SELECT Category, Price
    FROM Products
) AS SourceTable
PIVOT (
    SUM(Price)
    FOR Category IN ([Electronics], [Clothing], [Accessories])
) AS PivotTable;
```

The PIVOT operator is used to transform rows into columns, aggregating data in the process. In this example, we're displaying total sales by category using the PIVOT operator.

44. Transaction

This query demonstrates the use of a transaction to ensure the atomicity of a series of SQL statements.

```
BEGIN TRANSACTION;


 SQL statements go here
UPDATE Products
SET StockQuantity = StockQuantity - 1
WHERE ProductID = 101;


INSERT INTO OrderHistory (ProductID, OrderDate)
VALUES (101, GETDATE());
```

Commit the transaction

```
COMMIT;
```

A transaction is a sequence of one or more SQL statements that are executed as a single unit of work. The BEGIN TRANSACTION starts a new transaction, and the COMMIT statement ends the transaction, making all changes permanent. If an error occurs, the ROLLBACK statement can be used to undo the changes made during the transaction, ensuring atomicity. In this example, we're updating the stock quantity of a product and recording the order in the order history table as part of a transaction.

DATABASE NORMALIZATION

1. First Normal Form (1NF)

First Normal Form (1NF) is a database normalization step that ensures all values in a table's columns are atomic and cannot be divided further. It eliminates repeating groups of data and ensures each column contains only one piece of information.

Example:

Before 1NF

ProductID | ProductName | Colors

-

1 | Laptop | Red, Blue

2 | Smartphone | Black

3 | Camera | Red, Green

After 1NF

ProductID | ProductName

-

1 | Laptop

2 | Smartphone

3 | Camera

ProductID | Color

-

1 | Red

1 | Blue

2 | Black

3 | Red

3 | Green

In the "Before 1NF" example, the "Colors" column violates 1NF as it contains multiple values. After applying 1NF, the data is split into two tables, separating the repeating values.

2. Second Normal Form (2NF)

Second Normal Form (2NF) builds on 1NF and addresses partial dependencies. A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.

Example:

Before 2NF

OrderID | ProductID | ProductName | Category

1 | 101 | Laptop | Electronics

2 | 102 | Shirt | Clothing

3 | 101 | Laptop | Electronics

After 2NF

Products

ProductID | ProductName | Category

-

101 | Laptop | Electronics

102 | Shirt | Clothing

Orders

OrderID | ProductID

-

1 | 101

2 | 102

3 | 101

In the "Before 2NF" example, "ProductName" and "Category" are dependent on "ProductID" but not the entire primary key. After applying 2NF, we separate the tables to eliminate partial dependencies.

3. Third Normal Form (3NF)

Third Normal Form (3NF) builds on 2NF and addresses transitive dependencies. A table is in 3NF if it is in 2NF, and no transitive dependencies exist—non-key attributes are not dependent on other non-key attributes.

Example:

Before 3NF

EmployeeID | ProjectID | EmployeeName | Department

1 | 101 | John Doe | IT

2 | 102 | Jane Smith | HR

3 | 103 | Bob Johnson | IT

After 3NF

Employees

EmployeeID | EmployeeName | Department

-

1 | John Doe | IT

2 | Jane Smith | HR

3 | Bob Johnson | IT

Projects

ProjectID | ProjectName

-

101 | Project A

102 | Project B

103 | Project C

EmployeeProjects

EmployeeID | ProjectID

-

1 | 101

2 | 102

3 | 103

In the "Before 3NF" example, "Department" is dependent on "EmployeeName" but not the primary key. After applying 3NF, we create separate tables for employees, projects, and the relationship between them.

4. Boyce-Codd Normal Form (BCNF or 4NF)

Boyce-Codd Normal Form (BCNF) is a stricter form of 3NF that deals with situations where there are overlapping candidate keys. A table is in BCNF if, for every non-trivial functional dependency, the determinant is a superkey.

Example:

Before BCNF

StudentID | CourseID | Instructor | CourseName

1 | 101 | Dr. Smith | Math

2 | 101 | Prof. Johnson| Math

3 | 102 | Dr. Brown | History

After BCNF

Students

StudentID | Instructor

-

1 | Dr. Smith

2 | Prof. Johnson

3 | Dr. Brown

Courses

CourseID | CourseName

101 | Math

102 | History

StudentCourses

StudentID | CourseID

-

1 | 101

2 | 101

3 | 102

In the "Before BCNF" example, the combination of "StudentID" and "CourseID" is a candidate key, and "Instructor" is dependent on "CourseID" but not the entire key. After applying BCNF, we separate the tables to ensure that dependencies are on superkeys.

5. Fifth Normal Form (5NF)

Fifth Normal Form (5NF) deals with cases where a table contains join dependencies, and it aims to minimize redundancy and dependency.

6. Sixth Normal Form (6NF)

Sixth Normal Form (6NF) deals with the elimination of redundancy involving multivalued dependencies in a table. It aims to further reduce data redundancy and improve data integrity.

DATA DEFINITION

1. Create New Database

This query creates a new database named '**SampleDB**'.

```
CREATE DATABASE SampleDB;
```

The CREATE DATABASE statement is used to create a new database in SQL Server. In this example, we create a database named 'SampleDB'.

2. Drop Database

This query drops the 'SampleDB' database.

```
DROP DATABASE SampleDB;
```

The DROP DATABASE statement is used to delete an existing database in SQL Server. In this example, we drop the 'SampleDB' database.

3. Create Schema

This query creates a new schema named 'Sales' in the 'SampleDB' database.

```
CREATE SCHEMA Sales;
```

The CREATE SCHEMA statement is used to create a new schema in SQL Server. In this example, we create a schema named 'Sales' within the 'SampleDB' database.

4. Alter Schema

This query alters the 'Sales' schema, changing the owner to a different user.

```
ALTER SCHEMA Sales TRANSFER TO NewOwner;
```

The ALTER SCHEMA statement is used to modify an existing schema in SQL Server. In this example, we alter the 'Sales' schema, transferring ownership to a new user named 'NewOwner'.

5. Drop Schema

This query drops the 'Sales' schema.

```
DROP SCHEMA Sales;
```

The DROP SCHEMA statement is used to delete an existing schema in SQL Server. In this example, we drop the 'Sales' schema.

6. Create New Table

This query creates a new table named 'Customers' in the 'Sales' schema.

```
CREATE TABLE Sales.Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100)
);
```

The CREATE TABLE statement is used to create a new table in SQL Server. In this example, we create a table named 'Customers' in the 'Sales' schema with columns for customer information.

7. Identity Column

This query creates a new table with an identity column.

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY IDENTITY(1,1),
    ProductName VARCHAR(100),
    Price DECIMAL(10, 2)
);
```

The IDENTITY property is used to automatically generate unique values for a column. In this example, the 'ProductID' column is an identity column, starting at 1 and incrementing by 1.

8. Sequence

Sequences are not directly supported in SQL Server like some other database systems. Instead, you can use the SEQUENCE object, introduced in SQL Server 2012.

This query creates a sequence named 'ProductSeq'.

```
CREATE SEQUENCE ProductSeq
  START WITH 1
  INCREMENT BY 1;
```

9. Add Column

This query adds a new column named 'PhoneNumber' to the 'Customers' table.

```
ALTER TABLE Sales.Customers
ADD PhoneNumber VARCHAR(20);
```

The ALTER TABLE ADD COLUMN statement is used to add a new column to an existing table. In this example, we add a 'PhoneNumber' column to the 'Customers' table.

10. Modify Column

This query modifies the data type of the 'Price' column in the 'Products' table.

```
ALTER TABLE Products
ALTER COLUMN Price DECIMAL(12, 2);
```

The ALTER TABLE ALTER COLUMN statement is used to modify the data type of an existing column. In this example, we change the data type of the 'Price' column in the 'Products' table.

11. Drop Column

This query drops the 'PhoneNumber' column from the 'Customers' table.

```
ALTER TABLE Sales.Customers
DROP COLUMN PhoneNumber;
```

The ALTER TABLE DROP COLUMN statement is used to remove an existing column from a table. In this example, we drop the 'PhoneNumber' column from the 'Customers' table.

12. Computed Columns

This query adds a computed column 'FullName' to the 'Customers' table.

```
ALTER TABLE Sales.Customers
ADD FullName AS (FirstName + ' ' + LastName);
```

Computed columns are calculated based on other columns in the table. In this example, we add a computed column 'FullName' concatenating 'FirstName' and 'LastName' in the 'Customers' table.

13. Rename Table

This query renames the 'Customers' table to 'Clients'.

```
EXEC sp_rename 'Sales.Customers', 'Clients';
```

The sp_rename system stored procedure is used to rename an existing table in SQL Server. In this example, we rename the 'Customers' table to 'Clients' in the 'Sales' schema.

14. Drop Table

This query drops the 'Clients' table.

```
DROP TABLE Sales.Clients;
```

The DROP TABLE statement is used to delete an existing table in SQL Server. In this example, we drop the 'Clients' table in the 'Sales' schema.

15. Truncate Table

This query removes all rows from the 'Products' table without logging individual row deletions.

```
TRUNCATE TABLE Products;
```

The TRUNCATE TABLE statement is used to remove all rows from a table without logging individual row deletions. In this example, we truncate the 'Products' table.

16. Temporary Tables

This query creates a temporary table '#TempOrders' to store temporary data.

```
CREATE TABLE #TempOrders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT
);
```

Insert data into the temporary table.

```
INSERT INTO #TempOrders (OrderID, OrderDate, CustomerID)
VALUES (1, '2023-01-01', 101),
       (2, '2023-01-02', 102);
```

Temporary tables are used to store temporary data within a session. In this example, we create a temporary table '#TempOrders' and insert some sample data.

17. Synonym

This query creates a synonym 'ProductSyn' for the 'Products' table.

```
CREATE SYNONYM ProductSyn
FOR Sales.Products;
```

Now, we can query 'ProductSyn' instead of 'Sales.Products'.

```
SELECT * FROM ProductSyn;
```

A synonym is an alias for a database object. In this example, we create a synonym 'ProductSyn' for the 'Products' table, allowing us to use the synonym in queries.

18. SELECT INTO

This query selects data from the 'Orders' table into a new table 'OrdersArchive'.

```
SELECT *
INTO Sales.OrdersArchive
FROM Sales.Orders
WHERE OrderDate < '2023-01-01';
```

The SELECT INTO statement is used to create a new table and insert data into it from an existing table. In this example, we create a table 'OrdersArchive' and copy data from the 'Orders' table based on a condition.

19. PRIMARY KEY

This query adds a primary key constraint to the 'EmployeeID' column in the 'Employees' table.

```
ALTER TABLE Employees
ADD CONSTRAINT PK_Employees PRIMARY KEY (EmployeeID);
```

A primary key constraint uniquely identifies each record in a table. In this example, we add a primary key constraint to the 'EmployeeID' column in the 'Employees' table.

20. FOREIGN KEY

This query adds a foreign key constraint to the 'ProductID' column in the 'OrderDetails' table.

```
ALTER TABLE OrderDetails
ADD CONSTRAINT FK_OrderDetails_Products
FOREIGN KEY (ProductID)
REFERENCES Products(ProductID);
```

A foreign key constraint establishes a link between two tables by referencing the primary key of one table in another. In this example, we add a foreign key constraint to the 'ProductID' column in the 'OrderDetails' table, referencing the 'ProductID' column in the 'Products' table.

21. CHECK Constraint

This query adds a check constraint to ensure that the 'Quantity' column is greater than 0.

```
ALTER TABLE OrderDetails
ADD CONSTRAINT CHK_QuantityGreaterThanZero
CHECK (Quantity > 0);
```

A check constraint ensures that values in a column meet specific conditions. In this example, we add a check constraint to the 'Quantity' column in the 'OrderDetails' table to ensure that it is greater than 0.

22. UNIQUE Constraint

This query adds a unique constraint to the 'Email' column in the 'Customers' table.

```
ALTER TABLE Customers
ADD CONSTRAINT UQ_Customers_Email
UNIQUE (Email);
```

A unique constraint ensures that values in a column are unique across the table. In this example, we add a unique constraint to the 'Email' column in the 'Customers' table.

23. NOT NULL Constraint

This query adds a not null constraint to the 'ProductName' column in the 'Products' table.

```
ALTER TABLE Products
ALTER COLUMN ProductName VARCHAR(100) NOT NULL;
```

A not null constraint ensures that a column cannot contain null values. In this example, we add a not null constraint to the 'ProductName' column in the 'Products' table.

CREATING INDEXES IN SQL SERVER

1. Clustered Index

This query creates a clustered index on the 'EmployeeID' column in the 'Employees' table.

```
CREATE CLUSTERED INDEX CI_Employees_EmployeeID
ON Employees(EmployeeID);
```

A clustered index determines the physical order of data in a table and is created on the actual table rows. In this example, we create a clustered index on the 'EmployeeID' column in the 'Employees' table.

2. Non-Clustered Index

This query creates a non-clustered index on the 'LastName' column in the 'Employees' table.

```
CREATE NONCLUSTERED INDEX NCI_Employees_LastName
ON Employees(LastName);
```

A non-clustered index does not affect the physical order of data in a table and is stored separately. In this example, we create a non-clustered index on the 'LastName' column in the 'Employees' table.

3. Unique Index

This query creates a unique index on the 'Email' column in the 'Customers' table.

```
CREATE UNIQUE INDEX UI_Customers_Email
ON Customers(Email);
```

A unique index ensures that no duplicate values are allowed in the indexed column(s). In this example, we create a unique index on the 'Email' column in the 'Customers' table.

4. Filtered Index

This query creates a filtered index on the 'OrderDate' column in the 'Orders' table for orders in 2023.

```
CREATE INDEX FI_Orders_OrderDate_2023
ON Orders(OrderDate)
WHERE YEAR(OrderDate) = 2023;
```

A filtered index includes only a subset of data that meets a specific condition. In this example, we create a filtered index on the 'OrderDate' column in the 'Orders' table for orders in the year 2023.

5. Column Store Index

This query creates a column store index on the 'ProductDescription' column in the 'Products' table.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX CSI_Products_ProductDescription
ON Products(ProductDescription);
```

A column store index stores and processes columnar data rather than row-based data, optimizing data compression and query performance. In this example, we create a column store index on the 'ProductDescription' column in the 'Products' table.

6. Hash Index

Hash indexes are not directly supported in SQL Server as a separate index type. However, you can use hash functions within computed columns and then create indexes on those computed columns to achieve similar functionality.

Example:

This query adds a computed column 'HashedProductID' using a hash function.

```
ALTER TABLE Products
ADD HashedProductID AS HASHBYTES('SHA1', CAST(ProductID AS
VARBINARY(4)));
```

This query creates a non-clustered index on the computed column 'HashedProductID'.

```
CREATE NONCLUSTERED INDEX NCI_Products_HashedProductID
ON Products(HashedProductID);
```

In this example, we add a computed column 'HashedProductID' using the SHA-1 hash function and create a non-clustered index on that computed column to simulate a hash index.

EXPRESSIONS

1. CASE

The CASE statement is used for conditional logic in SQL queries. It allows you to perform different actions based on different conditions.

This query uses CASE to categorize employees based on their salary.

```
SELECT
    EmployeeName,
    Salary,
    CASE
        WHEN Salary > 50000 THEN 'High Salary'
        WHEN Salary > 30000 THEN 'Moderate Salary'
        ELSE 'Low Salary'
    END AS SalaryCategory
FROM Employees;
```

In this example, the CASE statement categorizes employees into salary categories based on different conditions.

2. COALESCE

The COALESCE function is used to return the first non-null expression among its arguments.

This query uses COALESCE to handle null values in the 'MiddleName' column.

```
SELECT
    EmployeeID,
    FirstName,
    COALESCE(MiddleName, 'N/A') AS MiddleName,
```

```
    LastName
FROM Employees;
```

In this example, the COALESCE function is used to replace null values in the 'MiddleName' column with 'N/A'.

3. NULLIF

The NULLIF function is used to return null if the two specified expressions are equal; otherwise, it returns the first expression.

This query uses NULLIF to return null if the 'EndDate' and 'StartDate' columns are equal.

```
SELECT
    ProjectID,
    ProjectName,
    StartDate,
    EndDate,
    NULLIF(EndDate, StartDate) AS Duration
FROM Projects;
```

In this example, the NULLIF function is used to calculate the duration of a project, returning null if the 'EndDate' and 'StartDate' columns are equal.

AGGREGATE FUNCTION

1. AVG

The AVG function calculates the average value of a numeric column.

This query calculates the average salary of employees.

```
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

In this example, the AVG function is used to calculate the average salary of employees.

2. CHECKSUM_AGG

The CHECKSUM_AGG function computes a hash value over a set of values.

This query uses CHECKSUM_AGG to calculate a hash value for the 'ProductID' column.

```
SELECT CHECKSUM_AGG(ProductID) AS ProductChecksum
FROM Products;
```

In this example, the CHECKSUM_AGG function is used to calculate a hash value for the 'ProductID' column.

3. COUNT

The COUNT function counts the number of rows in a result set.

This query counts the number of orders in the 'Orders' table.

```
SELECT COUNT(*) AS NumberOfOrders
FROM Orders;
```

In this example, the COUNT function is used to count the number of orders in the 'Orders' table.

4. COUNT_BIG

The COUNT_BIG function is similar to COUNT but returns a bigint data type.

This query uses COUNT_BIG to count the number of employees.

```
SELECT COUNT_BIG(*) AS NumberOfEmployees
FROM Employees;
```

In this example, the COUNT_BIG function is used to count the number of employees, and the result is of type bigint.

5. MAX

The MAX function returns the maximum value in a set of values.

This query retrieves the maximum salary among employees.

```
SELECT MAX(Salary) AS MaximumSalary
FROM Employees;
```

In this example, the MAX function is used to find the maximum salary among employees.

6. MIN

The MIN function returns the minimum value in a set of values.

This query retrieves the minimum salary among employees.

```
SELECT MIN(Salary) AS MinimumSalary
FROM Employees;
```

In this example, the MIN function is used to find the minimum salary among employees.

7. STDEV

The STDEV function calculates the standard deviation of a set of numeric values.

This query calculates the standard deviation of order amounts.

```
SELECT STDEV(OrderAmount) AS OrderAmountStdDev
FROM OrderDetails;
```

In this example, the STDEV function is used to calculate the standard deviation of order amounts.

8. STDEVP

The STDEVP function calculates the population standard deviation of a set of numeric values.

This query calculates the population standard deviation of product prices.

```
SELECT STDEVP(Price) AS PriceStdDevPop
FROM Products;
```

In this example, the STDEVP function is used to calculate the population standard deviation of product prices.

9. SUM

The SUM function calculates the sum of a set of numeric values.

This query calculates the total sales amount.

```
SELECT SUM(TotalAmount) AS TotalSales
FROM Sales;
```

In this example, the SUM function is used to calculate the total sales amount.

10. VAR

The VAR function calculates the variance of a set of numeric values.

This query calculates the variance of product prices.

```
SELECT VAR(Price) AS PriceVariance
FROM Products;
```

In this example, the VAR function is used to calculate the variance of product prices.

11. VARP

The VARP function calculates the population variance of a set of numeric values.

This query calculates the population variance of employee ages.

```
SELECT VARP(Age) AS AgeVariancePop
FROM Employees;
```

In this example, the VARP function is used to calculate the population variance of employee ages.

STRING FUNCTIONS

1. ASCII

The ASCII function returns the ASCII code value of a character.

This query returns the ASCII code for the first character in the 'ProductName' column.

```
SELECT ASCII(SUBSTRING(ProductName, 1, 1)) AS FirstCharacterASCII
FROM Products;
```

In this example, the ASCII function is used to retrieve the ASCII code for the first character in the 'ProductName' column.

2. CHAR

The CHAR function returns a character based on its ASCII code value.

This query returns a character based on the ASCII code value 65.

```
SELECT CHAR(65) AS CharacterA;
```

In this example, the CHAR function is used to return the character corresponding to the ASCII code value 65, which is 'A'.

3. CHARINDEX

The CHARINDEX function returns the starting position of a substring within a string.

This query returns the position of 'Tech' within the 'CompanyName' column.

```
SELECT CHARINDEX('Tech', CompanyName) AS TechPosition
FROM Suppliers;
```

In this example, the CHARINDEX function is used to find the position of the substring 'Tech' within the 'CompanyName' column.

4. CONCAT

The CONCAT function concatenates two or more strings.

This query concatenates the 'FirstName' and 'LastName' columns with a space in between.

```
SELECT CONCAT(FirstName, ' ', LastName) AS FullName
FROM Employees;
```

In this example, the CONCAT function is used to concatenate the 'FirstName' and 'LastName' columns with a space in between.

5. CONCAT_WS

The CONCAT_WS function concatenates strings with a specified separator.

This query concatenates the 'FirstName', 'LastName', and 'City' columns with a comma as a separator.

```
SELECT CONCAT_WS(',', FirstName, LastName, City) AS FullNameCity
FROM Customers;
```

In this example, the CONCAT_WS function is used to concatenate the 'FirstName', 'LastName', and 'City' columns with a comma as a separator.

6. DIFFERENCE

The DIFFERENCE function returns an integer that represents the difference between the SOUNDEX values of two strings.

This query returns the difference between the SOUNDEX values of 'Apple' and 'Appel'.

```
SELECT DIFFERENCE('Apple', 'Appel') AS SoundexDifference;
```

In this example, the DIFFERENCE function is used to calculate the difference between the SOUNDEX values of two strings.

7. FORMAT

The FORMAT function formats a value with the specified format.

This query formats the 'OrderDate' column as 'YYYY-MM-DD'.

```
SELECT FORMAT(OrderDate, 'yyyy-MM-dd') AS FormattedOrderDate
FROM Orders;
```

In this example, the FORMAT function is used to format the 'OrderDate' column in a specific date format.

8. LEFT

The LEFT function returns a specified number of characters from the left side of a string.

This query returns the first three characters of the 'ProductName' column.

```
SELECT LEFT(ProductName, 3) AS FirstThreeCharacters
FROM Products;
```

In this example, the LEFT function is used to retrieve the first three characters of the 'ProductName' column.

9. LEN

The LEN function returns the number of characters in a string.

This query returns the length of the 'CompanyName' column.

```
SELECT LEN(CompanyName) AS CompanyNameLength
FROM Suppliers;
```

In this example, the LEN function is used to determine the length of the 'CompanyName' column.

10. LOWER

The LOWER function converts all characters in a string to lowercase.

This query converts the 'City' column values to lowercase.

```
SELECT LOWER(City) AS LowercaseCity
FROM Customers;
```

In this example, the LOWER function is used to convert the values in the 'City' column to lowercase.

11. LTRIM

The LTRIM function removes leading spaces from a string.

This query removes leading spaces from the 'ProductName' column.

```
SELECT LTRIM(ProductName) AS TrimmedProductName
FROM Products;
```

In this example, the LTRIM function is used to remove leading spaces from the 'ProductName' column.

12. NCHAR

The NCHAR function returns the Unicode character based on the specified integer code.

This query returns the Unicode character for the code 65.

```
SELECT NCHAR(65) AS UnicodeCharacterA;
```

In this example, the NCHAR function is used to return the Unicode character for the code 65, which is 'A'.

13. PATINDEX

The PATINDEX function returns the starting position of a pattern in a string.

This query returns the position of the pattern 'Tech' within the 'CompanyName' column.

```
SELECT PATINDEX('%Tech%', CompanyName) AS TechPosition
FROM Suppliers;
```

In this example, the PATINDEX function is used to find the position of the pattern 'Tech' within the 'CompanyName' column.

14. QUOTENAME

The QUOTENAME function returns a Unicode string with delimiters added to make the input string a valid SQL Server delimited identifier.

This query adds square brackets to the 'CategoryName' values.

```
SELECT QUOTENAME(CategoryName, '[') AS QuotedCategoryName
FROM Categories;
```

In this example, the QUOTENAME function is used to add square brackets to the 'CategoryName' values.

15. REPLACE

The REPLACE function replaces occurrences of a specified string with another string.

This query replaces 'Mr.' with 'Ms.' in the 'Title' column.

```
SELECT REPLACE(Title, 'Mr.', 'Ms.') AS UpdatedTitle
FROM Employees;
```

In this example, the REPLACE function is used to replace occurrences of 'Mr.' with 'Ms.' in the 'Title' column.

16. REPLICATE

The REPLICATE function repeats a string a specified number of times.

This query repeats the '*' character five times.

```
SELECT REPLICATE('*', 5) AS StarPattern;
```

In this example, the REPLICATE function is used to repeat the '*' character five times.

17. REVERSE

The REVERSE function reverses a string.

This query reverses the characters in the 'ProductName' column.

```
SELECT REVERSE(ProductName) AS ReversedProductName
FROM Products;
```

In this example, the REVERSE function is used to reverse the characters in the 'ProductName' column.

18. RIGHT

The RIGHT function returns a specified number of characters from the right side of a string.

This query returns the last three characters of the 'ProductName' column.

```
SELECT RIGHT(ProductName, 3) AS LastThreeCharacters
FROM Products;
```

In this example, the RIGHT function is used to retrieve the last three characters of the 'ProductName' column.

19. RTRIM

The RTRIM function removes trailing spaces from a string.

This query removes trailing spaces from the 'ProductName' column.

```
SELECT RTRIM(ProductName) AS TrimmedProductName
FROM Products;
```

In this example, the RTRIM function is used to remove trailing spaces from the 'ProductName' column.

20. SOUNDEX

The SOUNDEX function returns a four-character code to compare the similarity of two strings.

This query returns the SOUNDEX value for the 'CompanyName' column.

```
SELECT SOUNDEX(CompanyName) AS CompanyNameSoundex
FROM Customers;
```

In this example, the SOUNDEX function is used to calculate the SOUNDEX value for the 'CompanyName' column.

21. SPACE

The SPACE function returns a string consisting of a specified number of space characters.

This query returns a string with ten space characters.

```
SELECT SPACE(10) AS TenSpaces;
```

In this example, the SPACE function is used to generate a string with ten space characters.

22. STR

The STR function converts a numeric value to a string with specified precision and optional format.

This query converts the numeric value 123.456 to a string with two decimal places.

```
SELECT STR(123.456, 6, 2) AS FormattedNumber;
```

In this example, the STR function is used to convert the numeric value 123.456 to a string with two decimal places.

23. STRING_AGG

The STRING_AGG function concatenates values from multiple rows into a single string using a specified separator.

This query concatenates 'ProductName' values with a comma separator.

```
SELECT STRING_AGG(ProductName, ', ') AS ConcatenatedProducts
FROM Products;
```

In this example, the STRING_AGG function is used to concatenate 'ProductName' values with a comma separator.

24. STRING_ESCAPE

The STRING_ESCAPE function escapes special characters in a string.

This query escapes special characters in the 'Description' column.

```
SELECT STRING_ESCAPE(Description, 'json') AS EscapedDescription
FROM Products;
```

In this example, the STRING_ESCAPE function is used to escape special characters in the 'Description' column for JSON format.

25. STRING_SPLIT

The STRING_SPLIT function splits a string into rows based on a specified separator.

This query splits the 'Tags' column into rows using a comma as a separator.

```
SELECT value AS Tag
FROM Products
CROSS APPLY STRING_SPLIT(Tags, ',');
```

In this example, the STRING_SPLIT function is used to split the 'Tags' column into rows using a comma as a separator.

26. STUFF

The STUFF function deletes a specified length of characters from a string and then inserts another string at a specified starting point.

This query replaces characters in the 'ProductName' column starting from the third position.

```
SELECT STUFF(ProductName, 3, 5, 'New') AS UpdatedProductName
FROM Products;
```

In this example, the STUFF function is used to replace characters in the 'ProductName' column starting from the third position with the string 'New'.

27. SUBSTRING

The SUBSTRING function returns a portion of a string, starting at a specified position with a specified length.

This query returns the first five characters of the 'ProductName' column.

```
SELECT SUBSTRING(ProductName, 1, 5) AS FirstFiveCharacters
FROM Products;
```

In this example, the SUBSTRING function is used to retrieve the first five characters of the 'ProductName' column.

28. TRANSLATE

The TRANSLATE function replaces multiple characters in a string with characters in another string.

This query replaces vowels with '*' in the 'ProductName' column.

```
SELECT TRANSLATE(ProductName, 'aeiou', '*') AS TranslatedProductName
FROM Products;
```

In this example, the TRANSLATE function is used to replace vowels with '*' in the 'ProductName' column.

29. TRIM

The TRIM function removes leading and trailing spaces from a string.

This query removes leading and trailing spaces from the 'City' column.

```
SELECT TRIM(City) AS TrimmedCity
FROM Customers;
```

In this example, the TRIM function is used to remove leading and trailing spaces from the 'City' column.

30. UNICODE

The UNICODE function returns the Unicode code point of the first character of a string.

This query returns the Unicode code point for the first character in the 'ProductName' column.

```
SELECT UNICODE(SUBSTRING(ProductName, 1, 1)) AS FirstCharacterUnicode
FROM Products;
```

In this example, the UNICODE function is used to retrieve the Unicode code point for the first character in the 'ProductName' column.

31. UPPER

The UPPER function converts all characters in a string to uppercase.

This query converts the 'ProductName' values to uppercase.

```
SELECT UPPER(ProductName) AS UppercaseProductName
FROM Products;
```

In this example, the UPPER function is used to convert the 'ProductName' values to uppercase.

SYSTEM FUNCTIONS

1. CAST

The CAST function is used to explicitly convert an expression or value to a specified data type.

This query casts the 'UnitPrice' column as an integer.

```
SELECT ProductName, CAST(UnitPrice AS INT) AS RoundedUnitPrice
FROM Products;
```

In this example, the CAST function is used to cast the 'UnitPrice' column as an integer, rounding the values.

2. CONVERT

The CONVERT function is similar to CAST and is used to explicitly convert an expression or value to a specified data type.

This query converts the 'OrderDate' column to a string in the 'YYYY-MM-DD' format.

```
SELECT CONVERT(VARCHAR, OrderDate, 23) AS FormattedOrderDate
FROM Orders;
```

In this example, the CONVERT function is used to convert the 'OrderDate' column to a string in the 'YYYY-MM-DD' format.

3. CHOOSE

The CHOOSE function returns the item at the specified index from a list of values.

This query returns the day of the week based on the 'OrderDay' column value.

```
SELECT OrderDay, CHOOSE(OrderDay, 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat') AS DayOfWeek
FROM OrderSummary;
```

In this example, the CHOOSE function is used to return the day of the week based on the 'OrderDay' column value.

4. ISNULL

The ISNULL function replaces NULL with a specified replacement value.

This query replaces NULL values in the 'City' column with 'Unknown'.

```
SELECT CustomerID, ISNULL(City, 'Unknown') AS CustomerCity
FROM Customers;
```

In this example, the ISNULL function is used to replace NULL values in the 'City' column with 'Unknown'.

5. ISNUMERIC

The ISNUMERIC function checks whether an expression is numeric.

This query checks if the 'UnitPrice' column values are numeric.

```
SELECT ProductName, UnitPrice, ISNUMERIC(UnitPrice) AS IsNumeric
FROM Products;
```

In this example, the ISNUMERIC function is used to check if the 'UnitPrice' column values are numeric.

6. IIF

The IIF function returns one of two values based on a specified condition.

This query returns 'High' if the 'UnitPrice' is greater than 50, otherwise 'Low'.

```
SELECT ProductName, UnitPrice,
    IIF(UnitPrice > 50, 'High', 'Low') AS PriceCategory
FROM Products;
```

In this example, the IIF function is used to categorize products as 'High' or 'Low' based on their 'UnitPrice'.

7. TRY_CAST

The TRY_CAST function tries to cast an expression to a specified data type. If the cast fails, it returns NULL.

This query tries to cast 'InvalidDate' values to a date format.

```
SELECT OrderID, TRY_CAST(InvalidDate AS DATE) AS ValidOrderDate
FROM Orders;
```

In this example, the TRY_CAST function is used to try to cast 'InvalidDate' values to a date format, returning NULL for invalid dates.

8. TRY_CONVERT

The TRY_CONVERT function is similar to TRY_CAST and tries to convert an expression to a specified data type.

This query tries to convert 'InvalidAmount' values to a decimal format.

```
SELECT InvoiceID, TRY_CONVERT(DECIMAL(10, 2), InvalidAmount) AS
ValidInvoiceAmount
FROM Invoices;
```

In this example, the TRY_CONVERT function is used to try to convert 'InvalidAmount' values to a decimal format, returning NULL for invalid amounts.

9. TRY_PARSE

The TRY_PARSE function tries to parse a string to a date or time data type. If the parse fails, it returns NULL.

This query tries to parse 'InvalidDate' values to a date format.

```
SELECT ShipmentID, TRY_PARSE(InvalidDate AS DATE) AS ValidShipmentDate
FROM Shipments;
```

In this example, the TRY_PARSE function is used to try to parse 'InvalidDate' values to a date format, returning NULL for invalid dates.

10. Convert date time to string

This query converts the 'OrderDate' column to a string in the 'YYYY-MM-DD' format.

```
SELECT OrderID, CONVERT(VARCHAR, OrderDate, 23) AS FormattedOrderDate
FROM Orders;
```

This example demonstrates how to convert the 'OrderDate' column to a string in the 'YYYY-MM-DD' format using the CONVERT function.

11. Convert string to date time

This query converts the 'DateString' column to a datetime format.

```
SELECT EventID, CONVERT(DATETIME, DateString, 101) AS EventDate
FROM Events;
```

This example shows how to convert the 'DateString' column, representing dates in a string format, to a datetime format using the CONVERT function.

12. Convert date time to date

This query converts the 'BirthDate' column to a date format.

```
SELECT EmployeeID, CONVERT(DATE, BirthDate) AS BirthDay
FROM Employees;
```

In this example, the CONVERT function is used to convert the 'BirthDate' column to a date format, extracting only the date portion.

WINDOW FUNCTIONS

1. CUME_DIST

The CUME_DIST function calculates the cumulative distribution of a value within a group of values.

Example:

```
SELECT ProductName, UnitPrice, CUME_DIST() OVER (ORDER BY UnitPrice) AS CumulativeDistribution
FROM Products;
```

This example calculates the cumulative distribution of product prices using the CUME_DIST function.

2. DENSE_RANK

The DENSE_RANK function assigns a rank to each distinct row within a result set, with no gaps in ranking values.

Example:

```
SELECT ProductName, UnitPrice, DENSE_RANK() OVER (ORDER BY UnitPrice) AS DenseRank
FROM Products;
```

This example assigns a dense rank to product prices using the DENSE_RANK function.

3. FIRST_VALUE

The FIRST_VALUE function returns the first value in an ordered set of values within a specified window frame.

Example:

```
SELECT ProductName, UnitPrice, FIRST_VALUE(ProductName) OVER (ORDER BY UnitPrice) AS FirstProduct
FROM Products;
```

This example retrieves the first product name in the ordered set of product prices using the FIRST_VALUE function.

4. LAG

The LAG function retrieves data from a previous row within the result set.

Example:

```
SELECT ProductName, UnitPrice, LAG(UnitPrice) OVER (ORDER BY UnitPrice) AS PreviousPrice
FROM Products;
```

This example retrieves the previous product price for each row using the LAG function.

5. LAST_VALUE

The LAST_VALUE function returns the last value in an ordered set of values within a specified window frame.

Example:

```
SELECT ProductName, UnitPrice, LAST_VALUE(ProductName) OVER (ORDER BY UnitPrice) AS LastProduct
FROM Products;
```

This example retrieves the last product name in the ordered set of product prices using the LAST_VALUE function.

6. LEAD

The LEAD function retrieves data from a subsequent row within the result set.

Example:

```
SELECT ProductName, UnitPrice, LEAD(UnitPrice) OVER (ORDER BY UnitPrice) AS NextPrice
FROM Products;
```

This example retrieves the next product price for each row using the LEAD function.

7. NTILE

The NTILE function divides the result set into a specified number of roughly equal groups, or "tiles."

Example:

```
SELECT ProductName, UnitPrice, NTILE(4) OVER (ORDER BY UnitPrice) AS PriceGroup
FROM Products;
```

This example divides products into four price groups using the NTILE function.

8. PERCENT_RANK

The PERCENT_RANK function calculates the relative rank of a row within a result set as a percentage.

Example:

```
SELECT ProductName, UnitPrice, PERCENT_RANK() OVER (ORDER BY UnitPrice) AS PercentRank
FROM Products;
```

This example calculates the percent rank of product prices using the PERCENT_RANK function.

9. RANK

The RANK function assigns a rank to each distinct row within a result set.

Example:

```
SELECT ProductName, UnitPrice, RANK() OVER (ORDER BY UnitPrice) AS Rank
FROM Products;
```

This example assigns a rank to product prices using the RANK function.

10. ROW_NUMBER

The ROW_NUMBER function assigns a unique number to each row within a result set.

Example:

```
SELECT ProductName, UnitPrice, ROW_NUMBER() OVER (ORDER BY UnitPrice) AS RowNum
FROM Products;
```

This example assigns a row number to each row of products using the ROW_NUMBER function.

These examples assume the existence of the "Products" table with the specified columns. Adjust the table and column names based on your actual database schema.

Test & Evaluation

Section 1: SQL Basics and Data Manipulation

1. **SQL Basics:**
   - What is the purpose of SQL Server?
   - Briefly explain the process of installing SQL Server.
2. **Data Manipulation:**
   - Write a SQL query to retrieve all columns from the "Products" table.
   - Provide the SQL code to order the products by their unit prices in descending order.
   - Explain the difference between the `SELECT TOP` and `SELECT DISTINCT` clauses.
   - Demonstrate the use of the `WHERE` clause with an example.

Section 2: Joins and Subqueries

3. **Joins:**
   - Write a query to perform an inner join between the "Products" and "Orders" tables based on the common "ProductID" column.
   - Explain the purpose of a self-join with a practical example.
4. **Subqueries:**
   - Provide an example of a subquery in SQL.
   - Explain the concept of a correlated subquery and when it might be useful.

Section 3: Window Functions and Indexing

5. **Window Functions:**
   - Write a query using the `ROW_NUMBER()` function to assign a unique number to each row in the "Orders" table.
   - Explain the purpose of the `CUME_DIST` function and how it differs from `PERCENT_RANK`.
6. **Indexes:**
   - Define the term "Clustered Index" and provide an example.
   - Explain the use of a filtered index and when it might be beneficial.

Section 4: Functions in SQL Server

7. **String Functions:**
   - Demonstrate the use of the `CONCAT` function with an example.
   - Write a query using the `CHARINDEX` function to find the position of a specific character in a string.
8. **System Functions:**
   - Explain the purpose of the `TRY_CAST` function and when it should be used.

Section 5: Advanced SQL Concepts

9. **Database Normalization:**
   - Define the First Normal Form (1 NF) and provide an example.

- Explain the concept of Boyce Codd Normal Form (BCNF) in the context of database normalization.

10. **Stored Procedures:**
- Write a simple stored procedure to retrieve data from a table.
- Explain the use of output parameters in stored procedures.

Section 6: Additional Topics

11. **Creating and Modifying Tables:**
- Write a query to create a new table with appropriate columns.
- Demonstrate how to modify an existing column in a table.

12. **Advanced Functions:**
- Explain the purpose of the `LEAD` function with an example.

Section 7: Practice Scenarios

13. **Scenario:**
- Given a scenario where you need to find the average unit price of products in each category, write the SQL query to achieve this.

14. **Scenario:**
- In a scenario where you have to create a temporary table to store intermediate results, provide the SQL code to create and use the temporary table.

15. **Scenario:**
- Explain the steps involved in creating a system-versioned temporal table.

Evaluation

- Evaluate your understanding of the covered topics on a scale from 1 to 10, with 10 being the highest.

SQL SERVER DATE FUNCTIONS

RETURNING THE CURRENT DATE AND TIME

1. CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns the current date and time in the session time zone.

Example:

```
SELECT CURRENT_TIMESTAMP AS CurrentDateTime;
```

2. GETUTCDATE

The GETUTCDATE function returns the current UTC date and time.

Example:

```
SELECT GETUTCDATE() AS CurrentUTCDateTime;
```

3. GETDATE

The GETDATE function returns the current date and time in the session time zone.

Example:

```
SELECT GETDATE() AS CurrentDateTime;
```

4. SYSDATETIME

The SYSDATETIME function returns the current date and time, including fractional seconds and time zone offset.

Example:

```
SELECT SYSDATETIME() AS CurrentSysDateTime;
```

5. SYSUTCDATETIME

The SYSUTCDATETIME function returns the current UTC date and time, including fractional seconds.

Example:

```
SELECT SYSUTCDATETIME() AS CurrentSysUTCDateTime;
```

6. SYSDATETIMEOFFSET

The SYSDATETIMEOFFSET function returns the current date and time, including fractional seconds and the time zone offset.

Example:

```
SELECT SYSDATETIMEOFFSET() AS CurrentSysDateTimeOffset;
```

These functions can be useful in various scenarios, such as logging timestamps, capturing the time of data modifications, or handling time-sensitive calculations. Adjust the examples based on your specific use case and time zone considerations.

RETURNING THE DATE AND TIME PARTS

1. DATENAME

The DATENAME function returns a character string representing the specified datepart of a date.

Example:

```
SELECT DATENAME(MONTH, GETDATE()) AS CurrentMonth;
```

2. DATEPART

The DATEPART function returns an integer representing the specified datepart of a date.

Example:

```
SELECT DATEPART(YEAR, GETDATE()) AS CurrentYear;
```

3. DAY

The DAY function returns an integer representing the day of the month for a specified date.

Example:

```
SELECT DAY(GETDATE()) AS CurrentDayOfMonth;
```

4. MONTH

The MONTH function returns an integer representing the month of a specified date.

Example:

```
SELECT MONTH(GETDATE()) AS CurrentMonth;
```

5. YEAR

The YEAR function returns an integer representing the year of a specified date.

Example:

```
SELECT YEAR(GETDATE()) AS CurrentYear;
```

These functions are useful when you need to perform date-based calculations or when you want to extract specific information from date and time values in your SQL queries. Adjust the examples based on your specific needs and the columns you're working with in your database.

RETURNING A DIFFERENCE BETWEEN TWO DATES

The DATEDIFF function in SQL Server calculates the difference between two dates, returning an integer that represents the number of date or time units (such as days, months, or seconds) between them. The syntax for the DATEDIFF function is as follows:

```
DATEDIFF(datepart, startdate, enddate);
```

datepart: The part of the date to compare, such as day, month, year, hour, etc.

startdate: The starting date.

enddate: The ending date.

Example:

Calculate the difference in days between two dates

```
SELECT DATEDIFF(DAY, '2023-01-01', '2023-02-15') AS DaysDifference;
```

Calculate the difference in months between two dates

```
SELECT DATEDIFF(MONTH, '2023-01-01', '2023-02-15') AS MonthsDifference;
```

Calculate the difference in hours between two datetime values

```
SELECT DATEDIFF(HOUR, '2023-01-01 12:00:00', '2023-01-01 18:30:00') AS
HoursDifference;
```

In these examples:

The first query calculates the difference in days between January 1, 2023, and February 15, 2023.

The second query calculates the difference in months between January 1, 2023, and February 15, 2023.

The third query calculates the difference in hours between 12:00 PM and 6:30 PM on January 1, 2023.

You can customize the datepart, startdate, and enddate values based on your specific use case. The result will be an integer representing the difference in the specified units.

MODIFYING DATES

1. DATEADD

The DATEADD function adds a specified number of date or time units to a specified date or time.

Example:

Add 7 days to the current date

```
SELECT DATEADD(DAY, 7, GETDATE()) AS NewDate;
```

2. EOMONTH

The EOMONTH function returns the last day of the month containing a specified date.

Example:

Get the last day of the month for the current date

```
SELECT EOMONTH(GETDATE()) AS LastDayOfMonth;
```

3. SWITCHOFFSET

The SWITCHOFFSET function changes the time zone offset of a datetimeoffset value without changing its UTC value.

Example:

Switch the time zone offset to a different value

```
DECLARE @dt DATETIMEOFFSET = '2023-01-15 12:00:00 -05:00';

SELECT SWITCHOFFSET(@dt, '+02:00') AS UpdatedDateTimeOffset;
```

4. TODATETIMEOFFSET

The TODATETIMEOFFSET function converts a datetime or smalldatetime value to datetimeoffset by adding a specified time zone offset.

Example:

Convert a datetime value to datetimeoffset with a specific time zone offset

```
DECLARE @dt DATETIME = '2023-01-15 12:00:00';

SELECT TODATETIMEOFFSET(@dt, '+03:00') AS DateTimeOffsetValue;
```

These functions are useful for tasks like adding or subtracting time intervals, finding the end of a month, and adjusting time zone offsets. Customize the examples based on your specific requirements and the datetime values you are working with.

CONSTRUCTING DATE AND TIME FROM THEIR PARTS

1. DATEFROMPARTS

The DATEFROMPARTS function creates a date value from the specified year, month, and day.

Example:

Create a date value from year, month, and day

```
SELECT DATEFROMPARTS(2023, 3, 1) AS ResultDate;
```

2. DATETIME2FROMPARTS

The DATETIME2FROMPARTS function constructs a datetime2 value from the specified year, month, day, hour, minute, second, and fractional seconds.

Example:

Create a datetime2 value from individual parts

```
SELECT DATETIME2FROMPARTS(2023, 3, 1, 12, 30, 45, 500) AS ResultDateTime2;
```

3. DATETIMEOFFSETFROMPARTS

The DATETIMEOFFSETFROMPARTS function constructs a datetimeoffset value from the specified year, month, day, hour, minute, second, fractional seconds, time zone offset hours, and time zone offset minutes.

Example:

Create a datetimeoffset value from individual parts

```
SELECT DATETIMEOFFSETFROMPARTS(2023, 3, 1, 12, 30, 45, 500, 3, 0) AS ResultDateTimeOffset;
```

4. TIMEFROMPARTS

The TIMEFROMPARTS function creates a time value from the specified hour, minute, second, and fractional seconds.

Example:

Create a time value from individual parts

```
SELECT TIMEFROMPARTS(12, 30, 45, 500) AS ResultTime;
```

These functions are useful when you have date and time information in separate columns and need to construct a complete datetime value. Adjust the examples based on the specific date, time, and fractional seconds values you have in your scenario.

SQL SERVER STORED PROCEDURE

1. Create a Stored Procedure with Input Parameters:

```
CREATE PROCEDURE sp_InsertData
    @FirstName NVARCHAR(50),
    @LastName NVARCHAR(50)
AS
BEGIN
    INSERT INTO YourTable (FirstName, LastName)
    VALUES (@FirstName, @LastName);
END;
```

2. Stored Procedure to Insert Data:

```
CREATE PROCEDURE sp_InsertData
    @FirstName NVARCHAR(50),
    @LastName NVARCHAR(50)
AS
BEGIN
    INSERT INTO YourTable (FirstName, LastName)
    VALUES (@FirstName, @LastName);
END;
```

3. Stored Procedure to Update Table:

```
CREATE PROCEDURE sp_UpdateData
    @RecordID INT,
    @NewValue NVARCHAR(50)
AS
BEGIN
    UPDATE YourTable
    SET ColumnToUpdate = @NewValue
    WHERE ID = @RecordID;
END;
```

4. Stored Procedure to Select Data from Table:

```
CREATE PROCEDURE sp_SelectData
    @ID INT
AS
BEGIN
    SELECT *
    FROM YourTable
    WHERE ID = @ID;
END;
```

5. Stored Procedure to Delete Data from Table:

```
CREATE PROCEDURE sp_DeleteData
    @RecordID INT
AS
BEGIN
    DELETE FROM YourTable
    WHERE ID = @RecordID;
END;
```

Replace YourTable with the actual name of your table and adjust column names and types accordingly. These are basic examples, and you can customize them based on your specific needs.

6. Stored Procedure to Validate Username and Password:

```
CREATE PROCEDURE sp_ValidateUser
    @Username NVARCHAR(50),
    @Password NVARCHAR(50)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Users WHERE Username = @Username AND Password = @Password)
        SELECT 'Valid' AS Status;
    ELSE
        SELECT 'Invalid' AS Status;
END;
```

7. Stored Procedure in SQL to Add Two Numbers:

```
CREATE PROCEDURE sp_AddNumbers
   @Number1 INT,
   @Number2 INT
AS
BEGIN
   SELECT @Number1 + @Number2 AS SumResult;
END;
```

8. Stored Procedure in SQL with Multiple Queries:

```
CREATE PROCEDURE sp_MultipleQueries
AS
BEGIN
    Query 1
   SELECT * FROM Table1;

    Query 2
   SELECT * FROM Table2;
END;
```

9. Stored Procedure for Insert and Update with Output Parameter:

```
CREATE PROCEDURE sp_InsertAndUpdate
    @ID INT,
    @Value NVARCHAR(50),
    @UpdatedValue NVARCHAR(50) OUTPUT
AS
BEGIN
     Insert
    INSERT INTO YourTable (ID, Column1)
    VALUES (@ID, @Value);

     Update
    UPDATE YourTable
    SET Column1 = @UpdatedValue
    WHERE ID = @ID;

     Set output parameter
    SET @UpdatedValue = @Value;
END;
```

10. SQL Server Stored Procedure to List Columns:

```
CREATE PROCEDURE sp_ListColumns
    @TableName NVARCHAR(50)
AS
BEGIN
    SELECT COLUMN_NAME
    FROM INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_NAME = @TableName;
END;
```

Replace YourTable with the actual name of your table. Customize these examples based on your specific requirements and database schema.

11. Dynamic WHERE Clause in SQL Server Stored Procedure:

```
CREATE PROCEDURE sp_DynamicWhereClause
    @ColumnName NVARCHAR(50),
    @ColumnValue NVARCHAR(50)
AS
BEGIN
    DECLARE @SqlQuery NVARCHAR(MAX);

    SET @SqlQuery = 'SELECT * FROM YourTable WHERE ' +
QUOTENAME(@ColumnName) + ' = @ColumnValue';

    EXEC sp_executesql @SqlQuery, N'@ColumnValue NVARCHAR(50)', @ColumnValue;
END;
```

This example demonstrates a stored procedure that dynamically builds a SELECT query with a WHERE clause based on the specified column name and value. Replace YourTable with your actual table name.

12. SQL Server Stored Procedure to Return Select Result Concatenated:

```
CREATE PROCEDURE sp_ConcatenateResults
    @Parameter NVARCHAR(50)
AS
BEGIN
    DECLARE @ConcatenatedResult NVARCHAR(MAX);

    SELECT @ConcatenatedResult = COALESCE(@ConcatenatedResult + ', ', '') +
ColumnToConcatenate
    FROM YourTable
    WHERE SomeColumn = @Parameter;

    SELECT @ConcatenatedResult AS ConcatenatedResult;
END;
```

In this stored procedure, it concatenates values from a specific column (ColumnToConcatenate) based on a condition and returns the concatenated result. Adjust the column names and conditions based on your actual schema and requirements.

Tables in SQL Server

1. Creating a System-Versioned Temporal Table:

```
Creating a System-Versioned Temporal Table
CREATE TABLE YourTemporalTable
(
    ID INT PRIMARY KEY,
    DataColumn NVARCHAR(50),
    ValidFrom datetime2 GENERATED ALWAYS AS ROW START HIDDEN,
    ValidTo datetime2 GENERATED ALWAYS AS ROW END HIDDEN,
    PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
);
```

This example creates a temporal table with a system-versioned period. The ValidFrom and ValidTo columns are automatically managed by the system.

2. Modifying Data in a System-Versioned Temporal Table:

```
Modifying Data in a System-Versioned Temporal Table
UPDATE YourTemporalTable
SET DataColumn = 'NewValue'
WHERE ID = 1;
```

You can modify data in a temporal table as you would in a regular table. The system will handle the versioning of data.

3. Views:

```
Creating a View
CREATE VIEW YourView AS
SELECT Column1, Column2
FROM YourTable
WHERE SomeCondition;
```

Views allow you to encapsulate complex queries and present them as a virtual table. Replace YourTable with the actual table and customize the query.

4. Loops:

SQL Server doesn't have traditional loop constructs like some other programming languages. Instead, you use set-based operations. However, you can achieve looping behavior using cursors or recursive queries. Here's a basic example:

```
Using a WHILE loop
DECLARE @Counter INT = 1;

WHILE @Counter <= 10
BEGIN
    PRINT 'Loop iteration: ' + CAST(@Counter AS NVARCHAR(10));
    SET @Counter = @Counter + 1;
END;
```

5. Temporary Tables:

Creating a Temporary Table

```
CREATE TABLE #YourTempTable
(
    ID INT PRIMARY KEY,
    DataColumn NVARCHAR(50)
);
```

Inserting Data into Temporary Table

```
INSERT INTO #YourTempTable (ID, DataColumn)
VALUES (1, 'DataValue');
```

Querying Temporary Table

```
SELECT * FROM #YourTempTable;
```

Dropping Temporary Table

```
DROP TABLE #YourTempTable;
```

Temporary tables are used to store temporary data within a session or a query. The table is automatically dropped when the session or query is completed. Replace #YourTempTable with a unique name for your temporary table.

QueryCraft: A Hands-On Beginner's Tutorial

Congratulations! You've reached the end of **QueryCraft: A Hands-On Beginner's Tutorial**. We hope this journey through the world of SQL Server and database management has equipped you with the skills and confidence to navigate the data realm with ease.

Closing Thoughts

As you close this book, remember that your journey in mastering SQL is just beginning. Continue to explore, practice, and apply your newfound knowledge to real-world scenarios. The ability to query databases is a powerful skill that will open doors to exciting opportunities in your career and projects.

Thank You

A heartfelt thank you for choosing QueryCraft as your guide. Writing this book has been a labor of love, and we sincerely hope it has made the complex world of SQL Server more accessible and enjoyable for you.

We extend our gratitude to all the readers who embarked on this learning adventure. Your curiosity and dedication inspire us to continue creating valuable content for learners like you.

Stay Connected

We'd love to hear about your experiences with QueryCraft and answer any questions you may have. Feel free to reach out to the author:

Author: Vidya Niwas Pandey

- Email: vidyaniwas2@gmail.com

Stay connected for updates, additional resources, and more SQL insights:

Keep Querying, Keep Crafting!

As you step into the world of SQL, remember that every query is an opportunity to craft solutions, unravel insights, and shape data to your advantage. May your databases be optimized, your queries be swift, and your data always tell the story you seek.

Happy querying!

Thank you once again for being a part of QueryCraft. Wishing you a bright and data-filled future!