Romi Dhillon

sql mANUAL

SQL Server Management Studio

Contents

[1.1. Query to create database 3](#_Toc153025679)

[1.2. Query to create table 3](#_Toc153025680)

[1.3. Making comments 3](#_Toc153025681)

[1.4. Inserting data into a table 3](#_Toc153025682)

[1.5. Selecting columns 3](#_Toc153025683)

[1.6. Using an alias when selecting columns 3](#_Toc153025684)

[1.7. Deleting rows from a table 3](#_Toc153025685)

[1.8. Deleting tables 3](#_Toc153025686)

[1.9. Data Types 4](#_Toc153025687)

[1.10. Precision 4](#_Toc153025688)

[1.11. Storing the data considering UNICODE 5](#_Toc153025689)

[1.12. Identity 5](#_Toc153025690)

[1.13. Referential Identity concept 5](#_Toc153025691)

[1.14. Unique, not null, and Default Constraints 5](#_Toc153025692)

[1.15. Identity constraint 6](#_Toc153025693)

[1.16. Difference between primary key and unique 7](#_Toc153025694)

[1.17. Foreign Key Constraint 7](#_Toc153025695)

[1.18. Check Constraint 7](#_Toc153025696)

[1.19. Foreign Key Constraint with the correct constraints 7](#_Toc153025697)

[1.20. Creating a Schema 8](#_Toc153025698)

[1.21. GetDate Function and smalldatetime data type. 9](#_Toc153025699)

[1.22. Seeing the distinct of unique entries in a specific column 10](#_Toc153025700)

[1.23. Selecting specific columns in a table 11](#_Toc153025701)

[1.24. Altering a table to add a column 11](#_Toc153025702)

[1.25. Altering a table to drop a column 11](#_Toc153025703)

[1.26. Altering a table to change a datatype for a column 12](#_Toc153025704)

[1.27. Ordering values 12](#_Toc153025705)

[1.28. Where clause for conditional statements. 13](#_Toc153025706)

[1.29. Where clause combining conditional statements. 14](#_Toc153025707)

[1.30. Like statement 14](#_Toc153025708)

[1.31. Not like statement 15](#_Toc153025709)

[1.32. IN statement to check for multiple values in a column 16](#_Toc153025710)

[1.33. COUNT statement to count the number of items 16](#_Toc153025711)

[1.34. COUNT statement to count the number of distinct items 16](#_Toc153025712)

[1.35. COUNT statement when having NULL values 17](#_Toc153025713)

[1.36. Min and Max values. 17](#_Toc153025714)

[1.37. Average and Standard Deviation 17](#_Toc153025715)

[1.38. Groupby function 18](#_Toc153025716)

[1.39. Having Clause 19](#_Toc153025717)

[1.40. Inner Join 20](#_Toc153025718)

[1.41. Left Join 21](#_Toc153025719)

[1.42. Right Join 22](#_Toc153025720)

[1.43. Full outer Join 23](#_Toc153025721)

[1.44. Declaring Variables 24](#_Toc153025722)

[1.45. IF Statement 24](#_Toc153025723)

[1.46. IF ELSE statement 24](#_Toc153025724)

[1.47. Variable setting 25](#_Toc153025725)

[1.48. Case When 25](#_Toc153025726)

[1.49. LEN Function 26](#_Toc153025727)

[1.50. Lower function 27](#_Toc153025728)

[1.51. Trim function 27](#_Toc153025729)

[1.52. REVERSE Function 28](#_Toc153025730)

[1.53. Substring function to extract letters from within a string. 28](#_Toc153025731)

[1.54. Ceiling and Floor functions 28](#_Toc153025732)

[1.55. Round functions 29](#_Toc153025733)

[1.56. Absolute function 29](#_Toc153025734)

[1.57. Current timestamp function 29](#_Toc153025735)

[1.58. DATEADD Function 30](#_Toc153025736)

[1.59. DATEDIFF function 31](#_Toc153025737)

[1.60. DATEPART function 32](#_Toc153025738)

[1.61. ISNUMERIC function to check if values are numeric or not. 33](#_Toc153025739)

[1.62. COALESCE Function 33](#_Toc153025740)

[1.63. Stored Procedures 34](#_Toc153025741)

[1.64. Why do we need to create a stored procedure? 35](#_Toc153025742)

[1.65. Altering a stored procedure 35](#_Toc153025743)

[1.66. Stored procedure with schema 35](#_Toc153025744)

[1.67. Parameterised Stored Procedure 36](#_Toc153025745)

[1.68. Parameterised Stored Procedure for multiple parameters 36](#_Toc153025746)

[1.69. Functions 37](#_Toc153025747)

[1.70. Views 39](#_Toc153025748)

[1.71. Cursors 41](#_Toc153025749)

[1.72. Indexing 41](#_Toc153025750)

**SQL Manual**

You can have a scenario where you have multiple databases each having their own tables. You may have a database to manage customer tables, and another for product, etc.

You can create a table using two ways. One is a query and one is a table.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Click save, and then give the table a name. Click refresh, and then you will see that dbo.Employee has been created as a table. Right click on the table and you can add values to the table by going to edit top 200 rows.

A screenshot of a computer program

Description automatically generated

After adding the values:

A screenshot of a computer

Description automatically generated

Right click the table and select top 1000 rows. You will now see the rows:

A screenshot of a computer

Description automatically generated

In the same way, you can right click and create a database as well.

# Query to create database



# Query to create table



Make sure that all queries or scripts are run in the database that you want to execute them in. You can select the database in the dropdown.

# Making comments

You can add two dashes to make anything into a comment.

# Inserting data into a table

A screenshot of a computer

Description automatically generated

You can insert rows or multiple rows of data as above.

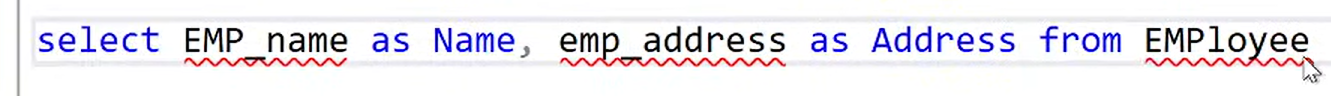
# Selecting columns

A close-up of words

Description automatically generated

You can select all columns, or you can choose to select only some columns.

# Using an alias when selecting columns



This only changes the name for the selection purpose, but not the name of the column in the DB.

# Deleting rows from a table

A blue text on a white background

Description automatically generated

Above we are saying to delete all rows from the Employe table.

# Deleting tables

A close up of a sign

Description automatically generated

You can delete tables with the query above.

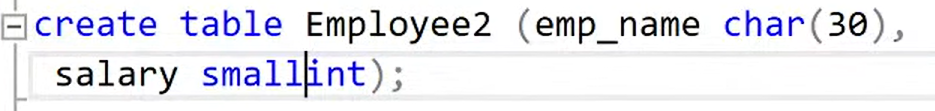
# Data Types

You must design your database with the correct data types

**A screenshot of a computer

Description automatically generated**

So based on the type of data, you can can state the data type.

****

# Precision

When defining the data type, you can state the precision. So if you want to store 123, you want to store it as five digits and 2 decimal places.

**A screenshot of a computer

Description automatically generated**

So a practical example of this is:

**A screenshot of a computer

Description automatically generated**

# Storing the data considering UNICODE

A close up of a computer screen

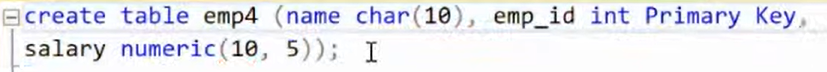
Description automatically generated

Sometimes people have letters which don’t fall in ASCII, and may be foreign names. You can use nchar data type and you must use the N keyword when entering the value into the table.

# Identity

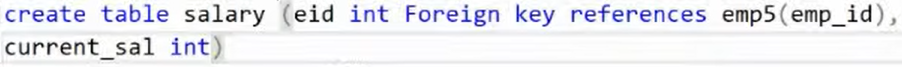
We all have some sort of a unique identity, for example employee id’s or passport numbers. There could be many davids, or johns in this world. When you store information in a table, you have to be able to uniquely identify each and every row. A unique ID is essentially a primary key constraint. You cant have two rows with the same ID in a table. Also, a null ID is not allowed.

Using the primary key constraint.



# Referential Identity concept

When creating a table which holds a foreign key, it is used to reference a different table.



So we create a new table named salary and give it a column named eid which is a foreign key to the emp5 table which column emp\_id. You can only insert values into the eid column in the foreign key table if the emp\_id in the emp5 table exists, otherwise it will not be allowed.

# Unique, not null, and Default Constraints

Lets create a new table in a database.

CREATE TABLE Citizens (

Citizen\_ID INT PRIMARY KEY,

Citizen\_Name VARCHAR(100) NOT NULL,

NI\_Number VARCHAR(20) UNIQUE,

Country VARCHAR(50) DEFAULT 'United Kingdom');

Notice how the national insurance number has a UNIQUE constraint and how the country has a default constraint. That is because we want the NI number to be unique (without being the primary key) and we want the country to have a default value of united kingdom. We have stated that the name cannot be a NULL value, so the user is forced to insert a value for it. We then insert values into the table:

INSERT INTO Citizens (Citizen\_ID, Citizen\_Name, NI\_Number, Country)

VALUES

(1, 'John Doe', 'AB123456C', 'United Kingdom'),

(2, 'Jane Smith', 'CD789012D', 'United Kingdom'),

(3, 'Alice Johnson', 'EF345678E', 'United Kingdom'),

(4, 'Bob Wilson', 'GH901234F', 'United Kingdom'),

(5, 'Eva Davis', 'IJ567890G', 'United Kingdom');

# Identity constraint

Notice how you have been putting in the values 1,2,3,4,5 for the primary key. However, you do not have to do that. You can apply the identity constraint to the primary key to force it to auto increment.

CREATE TABLE Citizens (

Citizen\_ID INT PRIMARY KEY IDENTITY(1,1),

Citizen\_Name VARCHAR(100),

NI\_Number VARCHAR(20) UNIQUE,

Country VARCHAR(50) DEFAULT 'United Kingdom'

);

You can now insert into without putting the primary key in there:

INSERT INTO Citizens (Citizen\_Name, NI\_Number, Country)

VALUES

('Mary Johnson', 'KL123456H', 'United Kingdom'),

('Tom Williams', 'MN789012I', 'United Kingdom'),

('Grace Brown', 'OP345678J', 'United Kingdom');

Lets say that you wanted to alter the table that was created before and you wanted to add the identity constraint to it. You can do this with the use of the ALTER key word.

ALTER TABLE Citizens

ALTER COLUMN Citizen\_ID INT PRIMARY KEY IDENTITY(1,1);

# Difference between primary key and unique

The unique constraint allows a NULL value, whereas the primary key does not.

# Foreign Key Constraint

Lets assume that we are creating a voters table that needs to be linked to the citizen table through the citizen\_id. We state the foreign key references keywords and connect the citizen\_id.

A screenshot of a computer code

Description automatically generated

# Check Constraint

We can check if certain columns have a condition. For the example below we have added a check if the age is greater than or equal to 18. We have also added a check to state that the height must be greater or equal to 120. Also, the mobile number must have a length which is equal to 11. If these conditions are not met, then a value cannot be added into the respective columns. Finally, we have declared the Citizen\_ID column so that it exists in both tables. We then add a foreign key constraint linking the two columns which have the same name.

CREATE TABLE Citizen\_info (

Info\_ID smallint PRIMARY KEY,

Age smallint CHECK (Age >= 18),

Height smallint CHECK (Height >= 120),

Mobile\_no INT UNIQUE CHECK (LEN(Mobile\_no) = 11),

Citizen\_ID INT,

FOREIGN KEY (Citizen\_ID) REFERENCES Citizens(Citizen\_ID)

);

# Foreign Key Constraint with the correct constraints

CREATE TABLE Citizen\_info (

Info\_ID smallint PRIMARY KEY IDENTITY (1,1),

Age smallint CHECK (Age >= 18),

Height smallint CHECK (Height >= 120),

Mobile\_no bigint UNIQUE CHECK (LEN(Mobile\_no) = 11),

Citizen\_ID int,

FOREIGN KEY (Citizen\_ID) REFERENCES Citizens(Citizen\_ID)

);

INSERT INTO Citizen\_info (Age, Height, Mobile\_no, Citizen\_ID)

VALUES

(25, 160, 12345678901, 1),

(30, 175, 23456789012, 2),

(22, 150, 34567890123, 3);

select \* from Citizens

select \* from Citizen\_info

In the code above, I have created a table named Citizen info which will be connected to the citizens table. The info id column is the primary key and the age, height, and mobile number are all checked for specific inputs. The citizen ID has a foreign key constraint. The values are inserted into the table, and notice how I do not need to add the info id as it is auto incremented. I add the citizen id which exists in the citizens table. If I try to add a citizen Id in the citizens info table that does not exist in the citizens info table, an error will be received. The results look like this:

A screenshot of a computer

Description automatically generated

# Creating a Schema

You can create tables and put them inside schemas. Schemas are just like folders or containers. Inside a database, you can have different schemas, and inside that schema, you will have tables. Lets now create a schema named bus drivers and then put two tables inside the bus\_drivers schema named employees and salary.

CREATE SCHEMA BusDrivers

CREATE TABLE BusDrivers.Employees (

Employee\_id SMALLINT PRIMARY KEY IDENTITY (1,1),

Age SMALLINT CHECK (Age >= 18),

Height SMALLINT CHECK (Height >= 100)

);

CREATE TABLE BusDrivers.Salary (

Salary INT,

CURRENCY NVARCHAR(10) DEFAULT N'£',

Employee\_id SMALLINT,

FOREIGN KEY (Employee\_id) REFERENCES BusDrivers.Employees(Employee\_id)

);

-- Inserting three rows into the BusDrivers.Employees table

INSERT INTO BusDrivers.Employees (Age, Height)

VALUES

(25, 175),

(30, 160),

(22, 180);

-- Inserting corresponding rows into the BusDrivers.Salary table

INSERT INTO BusDrivers.Salary (Salary, Employee\_id)

VALUES

(50000, 1),

(60000, 2),

(45000, 3);

SELECT \* FROM BusDrivers.Employees

SELECT \* FROM BusDrivers.Salary

A screenshot of a computer

Description automatically generated

A schema is a great way of putting your tables into categories.

# GetDate Function and smalldatetime data type.

If we want to get the current date or todays date, then we can do that by using the GetDate function. Lets say we want to insert values into a table, but we want want to insert the current date, we can do that. Lets create a table named prisoners in the MinistryOfJustice schema.

CREATE SCHEMA MinistryOfJustice

Create TABLE MinistryOfJustice.Prisoners (

Pristoner\_id SMALLINT PRIMARY KEY IDENTITY (1,1),

Prisoner\_name varchar (20),

Date\_of\_entry smalldatetime

);

-- Inserting three rows into the MinistryOfJustice.Prisoners table

INSERT INTO MinistryOfJustice.Prisoners (Prisoner\_name, Date\_of\_entry)

VALUES

('John Doe', '2023-12-01 10:30:00'),

('Jane Smith', '2023-12-01 11:45:00'),

('Bob Wilson', '2023-12-02 09:15:00');

select \* from MinistryOfJustice.Prisoners

INSERT INTO MinistryOfJustice.Prisoners (Prisoner\_name, Date\_of\_entry)

VALUES

('Jeremy Salsa', GETDATE()),

('Laura Bailey', GETDATE()),

('Hannah Winter', GETDATE());

SELECT \* FROM MinistryOfJustice.Prisoners

I first create the schema and then create a table in that schema. I add values to the table by specifying the date. However, the second time I add rows to the table, I use the getdate() function to add todays date for the entry. We could even add a check to ensure that todays date is used. We could also put a default value of todays date as well if we wish.

This is the result:

A screenshot of a computer

Description automatically generated

The TOP function

If we want to see the top number of rows in a table we can use:

select TOP (3) \* FROM MinistryOfJustice.Prisoners

The result is as below:

A screenshot of a computer

Description automatically generated

# Seeing the distinct of unique entries in a specific column

You can use the DISTINCT keyword to see the distinct values in a column.

select DISTINCT(CompanyName) FROM [SalesLT].[Customer]

A screenshot of a computer

Description automatically generated

This aggregation will give you all of the distinct values in a column. If you choose more than one column, It will give you the distinct combination of both rows.

**Selecting all columns**

You use the asterisk to select all columns

SELECT \* FROM dbo.Employee

# Selecting specific columns in a table

SELECT Citizen\_ID, Citizen\_Name FROM [dbo].[Citizens]

The second query is much better performance wise. When you have large tables, this will be better when running this query.

# Altering a table to add a column

ALTER TABLE dbo.Citizens ADD SALARY INT

When altering any table, you have to use the alter table command and then you can add a column by using the add command.

A screenshot of a computer

Description automatically generated

# Altering a table to drop a column

As you want to drop a column, you are essentially altering the table, so you have to state the alter table command first, followed by drop column.

ALTER TABLE dbo.Citizens DROP COLUMN SALARY

A screenshot of a computer

Description automatically generated

The table has been altered, and the column named salary has been dropped.

# Altering a table to change a datatype for a column

Lets say that we first alter a table and add a column named salary with a data type of smallint.

ALTER TABLE dbo.Citizens ADD SALARY SMALLINT

We then realise that the data type is wrong, and we actually needed the data type INT. We can alter the table and the column by doing the following.

ALTER TABLE dbo.Citizens ALTER COLUMN SALARY INT

# Ordering values

Lets say we first select three columns from a table:

select ProductID, Name, StandardCost from [SalesLT].[Product]

A screenshot of a table

Description automatically generated

We then want to order by the standard cost. We can do this with:

select ProductID, Name, StandardCost from [SalesLT].[Product] ORDER BY StandardCost

A screenshot of a product list

Description automatically generated

If you want to change the order by high to low you just use the DESC command:

select ProductID, Name, StandardCost from [SalesLT].[Product] ORDER BY StandardCost DESC

# Where clause for conditional statements.

If I want to see all the rows where the standard cost is above 2000, then you can use the where clause:

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE StandardCost >= 2000

A screenshot of a computer

Description automatically generated

You can also use the between clause, which allows you to see the number of rows that fall between certain amounts:

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE StandardCost between 1500 and 2500

A screenshot of a table

Description automatically generated

# Where clause combining conditional statements.

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE StandardCost >= 2000 and ProductID > 752

A screenshot of a computer

Description automatically generated

We can also use the OR operator.

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE StandardCost >= 2000 or ProductID > 752

A screenshot of a table

Description automatically generated

# Like statement

If you want the query to return values where the Name column starts with letters that match with mountain, then you would write this query:

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE Name LIKE 'Mountain%'

A screenshot of a table

Description automatically generated

However, if you want to look for the letter’s mountain inside any part of a record, then you would use two percentages.

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE Name LIKE '%Mountain%'

A screenshot of a data

Description automatically generated

# Not like statement

Lets now say that we want to find records where the letters for mountain as one word do not occur anywhere in the Name column, we can use the NOT LIKE statement:

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE Name NOT LIKE '%Mountain%'

**A screenshot of a table

Description automatically generated**

# IN statement to check for multiple values in a column

Lets say that I want to search for two specific product id’s in a column, I can use the IN clause to do this. I can also use the same concept in any other column when searching for things, as long as I am looking for a specific value.

select ProductID, Name, StandardCost from [SalesLT].[Product] WHERE ProductID IN (680, 706)

**A screenshot of a computer

Description automatically generated**

# COUNT statement to count the number of items

Lets say I wanted to count the number of products in the table. I can do this by using the count function.

select COUNT (ProductID) AS Product\_Count from [SalesLT].[Product]

The count function counts the number of rows in a column. If I was to put in

select COUNT (\*) AS Product\_Count from [SalesLT].[Product]

we would still have the same count, as we are now counting the number of rows for every column, which is still the same count as the number of rows in the ProductID column.

# COUNT statement to count the number of distinct items

Lets say that I want to count the distinct number of colours in the table. If I was to count all of the rows, the answer would be 225. However, if was to count the number of distinct items in the colour column, the answer is 9.

select COUNT (DISTINCT Color) AS Color\_count from [SalesLT].[Product]

# COUNT statement when having NULL values

Remember that the count function will not count NULL values. This means that if you have a column with loads of null values, the count function will only count the rows that do not have these null values.

The statements below can be used to count the number of null values and not null values respectively.

select COUNT (DISTINCT Color) AS Color\_count from [SalesLT].[Product] WHERE Color IS NULL

select COUNT (DISTINCT Color) AS Color\_count from [SalesLT].[Product] WHERE Color IS NOT NULL

# Min and Max values.

**You can select the minimum and maximum values in a column with:**

select min(StandardCost) from [SalesLT].[Product]

select max(StandardCost) from [SalesLT].[Product]

You can use the above in more creative ways as shown below.

SELECT \* FROM [SalesLT].[Product] WHERE StandardCost IN (SELECT MIN(StandardCost) FROM [SalesLT].[Product])

The query above first executes the inner query and then executes the outer. We are first saying to select the minimum value in the standard cost column. We are then saying to select all columns where standard cost contains that minimum value. In essence, the above is like saying the following.

SELECT \* FROM [SalesLT].[Product] WHERE StandardCost IN (0.85)

The approach above is called subqueries, as shown below.

A white board with red text and black text

Description automatically generated

# Average and Standard Deviation

SELECT AVG(StandardCost) as Average\_Cost FROM [SalesLT].[Product]

SELECT STDEV(StandardCost) as Average\_Cost FROM [SalesLT].[Product]

**Mathematical Operations**

You can use SQL to perform maths as well as shown below. You can also use mathematical functions.

SELECT 10\*2 +5

SELECT SQRT(100)

So if you want the sqrt of each row in the standard cost column, you can do this with:

SELECT sqrt(StandardCost) FROM [SalesLT].[Product]

# Groupby function

Let’s say that we want to find out the number of units sold for each of the different products in the example below. We have to group each of the products in order to find this out.

A screen shot of a computer

Description automatically generated

Lets now do the same example for a real example in my database.

In the statement below, we are only grouping the colours by their name.

SELECT Color FROM [SalesLT].[Product] group by Color

In the query below, we are grouping the colours by their name and we are getting the count of each of the colours that have been grouped.

SELECT Color, COUNT(Color) AS color\_count FROM [SalesLT].[Product] group by Color

A screenshot of a computer

Description automatically generated

You must made sure that the columns that you want to group by must be the only columns that are selected.

You can now get additional information with additional aggregate functions:

SELECT Color, COUNT(Color) AS color\_count, SUM (StandardCost) AS sum\_of\_standard\_cost FROM [SalesLT].[Product] group by Color

A table with numbers and text

Description automatically generated

We can group the colours, and then perform additional aggregate functions to get the colour count and then sum of the standard cost for each colour.

# Having Clause

When you perform a group by statement, you MUST use the having clause if you want to filter or perform any conditional statements on the table. In normal cases where there is no group by function applied, you would use the WHERE clause, however, in this case you have to use the having clause.

Let’s say that we want to only get the records where the sum of the standard cost is less than 200 from the example above. We can do this as:

SELECT Color, COUNT(Color) AS color\_count, SUM (StandardCost) AS sum\_of\_standard\_cost

FROM [SalesLT].[Product] group by Color HAVING SUM(StandardCost) <200

Notice how you cannot use the ALIAS when using the having clause. For example, I cannot write HAVING sum\_of\_standard\_cost < 200.

# Inner Join

When understanding joins, you must understand the concept of left and right tables. Without understanding this concept, it has hard to understand joins.

Right table.

Left table.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

You have been assigned a task to get the product categories for each of the product id’s and product names in the right table, as the product category name does not exist in the right table. In order to do this, you have to join the tables. You can join the tables in different ways, but we will first look at the inner join.

In the inner join, you start by selecting the columns that you want to see in both tables, and you use an alias so that you know which column you are seeing from which table (left and right table). You then write ‘INNER JOIN’ and then write the name of the right table. You then join them on their primary/foreign keys.

SELECT S.ProductCategoryID, S.Name, P.Name, P.ProductID FROM [SalesLT].[ProductCategory] AS S

INNER JOIN [SalesLT].[Product] AS P

ON S.ProductCategoryID = P.ProductCategoryID

You get the following result:

A screenshot of a table

Description automatically generated

Essentially, the inner join matches the rows where the product category id’s exist in both tables. As the product category ID of 5 (any many others that are not shown above) exist in the left table, they will be joined to the right table. However, notice how product categories 1,2,3, and 4 do not exist in the joint table as these product category ids’ do not exist in the right table. Lets say we want to see product categories in the joint table, even though their corresponding right table values are NULL. We can do this with a left join. A left join includes all primary key values in the joint table, even if they don’t exist in the right table.

# Left Join

SELECT S.ProductCategoryID, S.Name, P.Name, P.ProductID FROM [SalesLT].[ProductCategory] AS S

LEFT JOIN [SalesLT].[Product] AS P

ON S.ProductCategoryID = P.ProductCategoryID

You can now see from the results that Product Categories 1,2,3, and 4 have been included, even though the columns Name and Product ID in the right table have values of NULL (as they don’t exist in the right table).

A screenshot of a computer

Description automatically generated

# Right Join

Lets now look at different tables to depict the concept of a right join. The

Right table.

Left table.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

You can already see that the CustomerID column in the right table has values that are not present in the left table. Lets join the tables so that we get the CustomerID, AddressID from the left table, and the CustomerID, and first Name on the right table. However, lets perform a right join so that all CustomerID’s that exist in the right table end up in the joint table.

SELECT L.CustomerID, L.AddressID, R.CustomerID, R.FirstName FROM [SalesLT].[CustomerAddress] AS L

RIGHT JOIN [SalesLT].[Customer] AS R

ON L.CustomerID = R.CustomerID

By performing the query above, you can see the join below.

A screenshot of a table

Description automatically generated

If you go down, you can see that the values in the left table are also included, as they exist in the right table as well.

A screenshot of a table

Description automatically generated

At this point it is important to clarify that the left and right joins also match the primary/foreign key items between both tables, the only difference is that the left table also includes the primary/foreign key items in the left table that don’t exist in the right, and the right join does vice versa.

# Full outer Join

The full outer join matches the primary/foreign key between two tables, but also includes the primary/foreign key records that don’t match and exist in the left table and those that don’t match and exist in the right table. Therefore, this shows a full join of both tables whether or not the values match or not.

SELECT L.CustomerID, L.AddressID, R.CustomerID, R.FirstName FROM [SalesLT].[CustomerAddress] AS L

FULL OUTER JOIN [SalesLT].[Customer] AS R

ON L.CustomerID = R.CustomerID

# Declaring Variables

In SQL, we can declare variables by using the declare statement. In the example below, I am declaring a variable named max\_list\_price. As part of the syntax, I have to use to @ symbol, and then after that, you have to declare the data type. I then make the query equal to the variable and it gets stored In the variable. I then print the variable. The answer will be a stored number rather than a table result.

declare @max\_list\_price INT

select @max\_list\_price = max(ListPrice) from [SalesLT].[Product]

print @max\_list\_price

I can also choose to display the answer as a string, but for this i need to use the CAST function which converts the integer input into the stated output (which in this case is a varchar). The result is shown below.

declare @max\_list\_price INT

select @max\_list\_price = max(ListPrice) from [SalesLT].[Product]

print 'The maximum list price for a product is ' + CAST(@max\_list\_price AS VARCHAR)

A screenshot of a computer

Description automatically generated

# IF Statement

For the same example above, we can write if statements. We see in the example below that if the max list price variable is greater than 3000, then we print the statement. Then in SQL is begin.

IF @max\_list\_price < 3000

BEGIN

print 'the maximum price of ' + CAST (@max\_list\_price AS VARCHAR) + ' is too high'

END



# IF ELSE statement

In the statement below, we also add an else clause. If the max price is less than 3000, we state that the price is acceptable, else we state that the price is too high. The result shows that the else statement was executed.

IF @max\_list\_price < 3000

BEGIN

print 'the maximum price of ' + CAST (@max\_list\_price AS VARCHAR) + ' is acceptable'

END

ELSE

BEGIN

print 'the maximum price of ' + CAST (@max\_list\_price AS VARCHAR) + ' is too high'

END



# Variable setting

Above we declared a variable, but we can set a variable after declaring it too. The example above has been modified to declare a second variable and then set it. We now declare two variables and set the acceptable list price. We then add a value into the max list price and then use logic to compare.

declare @max\_list\_price INT

declare @acceptable\_list\_price INT

SET @acceptable\_list\_price = 3000

select @max\_list\_price = max(ListPrice) from [SalesLT].[Product]

IF @max\_list\_price < @acceptable\_list\_price

BEGIN

print 'the maximum price of ' + CAST (@max\_list\_price AS VARCHAR) + ' is acceptable'

END

ELSE

BEGIN

print 'the maximum price of ' + CAST (@max\_list\_price AS VARCHAR) + ' is too high'

END

# Case When

Let’s take a look at the following table.

select\* FROM [SalesLT].[SalesOrderDetail] order by UnitPrice

**A screenshot of a computer

Description automatically generated**

I want to be able to score the unit price for each sales order based on a condition. I can do this with a CASE WHEN clause. We start by selecting the salesorderid column, and then we write a case for the second column (which is going to hold the grade for each salesorderid). We write the difference when clauses and then end the statement and also assign an alias for the column. We then continue with the from clause to state which table this is for.

SELECT SalesOrderID,

CASE

WHEN UnitPrice >=0 and UnitPrice <5 THEN 'Extremely Low'

WHEN UnitPrice >=5 and UnitPrice <200 THEN 'Low'

WHEN UnitPrice >=200 and UnitPrice <500 THEN 'Moderate'

WHEN UnitPrice >=500 and UnitPrice <1000 THEN 'High'

WHEN UnitPrice >=1000 and UnitPrice <1500 THEN 'Very High'

ELSE 'Out of Range'

END as 'Grade'

FROM [SalesLT].[SalesOrderDetail]

The result looks like the following:

A table of numbers and letters

Description automatically generated

# LEN Function

The length function can be used to find out the length of the string in terms of letters.

Select \*, LEN(Name) as letters\_in\_name, LEFT(Name,1) 'first\_initial' from [dbo].[Employee]

In the query above, I select all columns, and then I select another column which holds the length of the name column with an alias of letters\_in\_name. I also have another column which uses the LEFT function to select the letters in the name column, starting from the left, and then selecting 1 letter from the left. These columns are all selected from the employee table. The result is shown below.

In the same way, I could also use the RIGHT function to select letters from the right hand side.

A screenshot of a white paper with black text

Description automatically generated

# Lower function

Lets now expand the example above, by using the lower function. Lets say that we want another column that holds the name column values but all as lower characters. In the same way, I could also use the UPPER function.

Select \*, LEN(Name) as letters\_in\_name, LEFT(Name,1) 'first\_initial', LOWER(Name) as lower\_characters from [dbo].[Employee]

A screenshot of a computer

Description automatically generated

We now get the result above.

# Trim function

If there are spaces after or before a word, the trim function makes sure that such spaces are removed.

select ModifiedDate, FORMAT(ModifiedDate,'d','en-us') AS ModifiedDate\_US from [SalesLT].[SalesOrderDetail]

In the example above, I have selected the origin ModifiedDate column from the SalesOrderDetail column, and I have selected a new column to create which changes the format of the ModifiedDate column. It sets the format to just the date using ‘d’ and then uses the English-us format. This changes the format of the date as seen below. Both columns can be seen side to side.

A screenshot of a data

Description automatically generated

# REVERSE Function

This is used to reverse the letters or numbers in a column. An example of this is given below.

select ModifiedDate, REVERSE(ModifiedDate) as Reversed\_column FROM [SalesLT].[SalesOrderDetail]

A screenshot of a data

Description automatically generated

# Substring function to extract letters from within a string.

Let’s say that I want to extract the fourth letter from each name in the Name column, and only want to extract one letter.

SELECT Name, SUBSTRING(Name, 4, 1) as fourth\_letter from [dbo].[Employee]

I use the substring function, starting with the column I want to extract the substring from, then the letter number, and then how many letters beyond that point that I want to extract.

# Ceiling and Floor functions

You can round values up or down, and this is done with the ceiling and floor functions.

Select ListPrice, CEILING(ListPrice) AS Ceiling\_value, FLOOR (ListPrice) AS Floor\_value FROM [SalesLT].[Product]

The results are shown below:

A screenshot of a table

Description automatically generated

# Round functions

This function can be used to round numbers up.

# Absolute function

Sometimes you have to clean your data, as negative values may be put into the fields. In order to change negative values to positive, you have to use the absolute function. We have the situation below, where Cathy has an age of -21, which is not possible.

A screenshot of a computer

Description automatically generated

SELECT \*, ABS(Age) as adjusted\_age FROM dbo.employee

The function above results to the following result

A screenshot of a computer

Description automatically generated

The -21 has now changed to 21.

# Current timestamp function

I can add the current timestamp as a new column

select \*, CURRENT\_TIMESTAMP as timestamp\_now from [dbo].[Employee]

A screenshot of a computer

Description automatically generated

# DATEADD Function

**This function is used to add days, months, or years to the current date**

select SellEndDate, DATEADD(YEAR, 5, SellEndDate) AS added\_five\_years from [SalesLT].[Product]

I use the dateadd function, I state that I want to add years to the date, then state that I want to add 5 years, and on which column I want to add this to). I then create an alias for the new column. The result is as follows:

A screenshot of a computer

Description automatically generated

I can also do this for months as below:

select SellEndDate, DATEADD(MONTH, 5, SellEndDate) AS added\_five\_months from [SalesLT].[Product]

A screenshot of a table

Description automatically generated

I can also do this for days instead:

select SellEndDate, DATEADD(DAY, 5, SellEndDate) AS added\_five\_days from [SalesLT].[Product]

A screenshot of a computer

Description automatically generated

# DATEDIFF function

I want to know how many days, months, or years of difference there is between two dates. I can do this as:

select SellStartDate, SellEndDate, DATEDIFF(DAY, SellStartDate, SellEndDate) as difference\_between\_start\_and\_end from [SalesLT].[Product]

The result shows that there is a difference of 364 days difference.

A screenshot of a table

Description automatically generated

# DATEPART function

You can get part of a date by using the DATEPART function. Lets say we just want the year from the sellEndDate. We can do this as:

select SellStartDate, DATEPART(YEAR,SellStartDate) as SellYear from [SalesLT].[Product]

A table of numbers with numbers on it

Description automatically generated

Can so the same for month:

select SellStartDate, DATEPART(MONTH,SellStartDate) as SellMonth from [SalesLT].[Product]

A screenshot of a table

Description automatically generated

# ISNUMERIC function to check if values are numeric or not.

select Name, ISNUMERIC(Name) as Is\_numeric from [SalesLT].[Product]

In the example above, we are checking if the name column is numeric or not.

A screenshot of a computer

Description automatically generated

# COALESCE Function

This function returns the first NOT NULL value in a column. This is useful if a column has a lot of NULL values.

SELECT Color, Size, COALESCE(Color, Size, 'No Color or Size') FROM [SalesLT].[Product]

If the color exists, then the coalesce function will give the name of the color. If the color does not exist and the size does, then the coalesce column will give the name of the size. If a null value exists for both, then the coalesce column will fice a value stated as the third argument in the query, which is ‘No Color or Size’.

A table with numbers and text

Description automatically generated

# Stored Procedures

To write a stored procedure, you have to use something called CREATE. To create anything in SQL, you have to use CREATE. The template for creating a stored procedure is:

A whiteboard with green writing

Description automatically generated

So lets write a stored procedure to calculate the total amount of money due to be paid by all customers from the SalesOrderHeader table. The query simply sums the TotalDue column . The syntax for the stored procedure is the same as above, but to execute the query in the stored procedure, you have to use EXEC followed by the name of the stored procedure.

CREATE PROCEDURE get\_total\_amount\_due

AS

BEGIN

SELECT SUM(TotalDue) AS TOTAL\_AMOUNT\_DUE FROM [SalesLT].[SalesOrderHeader]

END

EXEC get\_total\_amount\_due

Once you execute the stored procedure, the result is as follows:

A close-up of a number

Description automatically generated

# Why do we need to create a stored procedure?

In the left navigation under the database you are working in, you will see a folder name programmability. In this folder there is a folder named stored procedure. The beauty of stored procedures is that you don’t need to write the same queries over and over again. For example, if I want to find out the total amount due next time, I just have to execute the stored procedure as the following:

EXEC [dbo].[get\_total\_amount\_due]

If you don’t create a stored procedure, you will have to save the queries you have written on your machine, which is fiddly.

# Altering a stored procedure

Lets say we want to alter the stored procedure above to sum the total tax amount rather than the total amount due, I can do this as:

ALTER PROCEDURE get\_total\_amount\_due

AS

BEGIN

SELECT SUM(TaxAmt) AS TOTAL\_AMOUNT\_DUE FROM [SalesLT].[SalesOrderHeader]

END

EXEC get\_total\_amount\_due

# Stored procedure with schema

If you don’t assign a schema to a stored procedure, it will get stored in the schema named dbo. However, lets create a schema and then put the stored procedure in there.

CREATE SCHEMA Operations\_procedures;

CREATE PROCEDURE Operations\_procedures.get\_total\_amount\_due

AS

BEGIN

SELECT SUM(TaxAmt) AS TOTAL\_AMOUNT\_DUE FROM [SalesLT].[SalesOrderHeader]

END

EXEC Operations\_procedures.get\_total\_amount\_due

# Parameterised Stored Procedure

You can set a required parameter which you can specify when you execute the stored procedure. The @customerID is the parameter that you are passing in, and you specify its data type. You then pass it into the query where the CustomerID column is equal to that parameter that I have named @customerID. I could have called it anything. I then execute the stored procedure and pass in the value of 3 for the @customerID. I have a returned result for the first name, middle name, and last name for customer with ID 3.

CREATE PROCEDURE Operations\_procedures.get\_customer\_name @customerID INT

AS

BEGIN

SELECT FirstName, MiddleName, LastName FROM [SalesLT].[Customer]

WHERE CustomerID = @customerID

END

EXEC Operations\_procedures.get\_customer\_name 3

A screenshot of a computer

Description automatically generated

# Parameterised Stored Procedure for multiple parameters

ALTER PROCEDURE Operations\_procedures.check\_name

@firstName varchar,

@lastName varchar

AS

BEGIN

SELECT FirstName, MiddleName, LastName

FROM [SalesLT].[Customer]

WHERE FirstName LIKE @firstName + '%' and LastName LIKE @lastName + '%'

END

EXEC Operations\_procedures.check\_name 'kath', 'Gar'

Two instances of Kathleen Garza appeared when searching for kath and gar.

**A screenshot of a computer

Description automatically generated**

# Functions

You can write functions in SQL with the following syntax.

**A white board with red writing

Description automatically generated**

A different variable of the syntax is given below.

**A white board with red writing

Description automatically generated**

Lets now write a function to get the first letter from the color column.

CREATE FUNCTION get\_first\_letter\_of\_color(@color NVARCHAR(100))

RETURNS CHAR(1)

AS

BEGIN

DECLARE @C CHAR(1)

SELECT @C = SUBSTRING(@Color, 1, 1)

RETURN @C

END

This line declares a function named get\_first\_letter\_of\_color that takes a single parameter @color of type NVARCHAR(100) and returns a single character (CHAR(1)). In SQL, the CHAR(n) data type is used to define a fixed-length character string with a specified length n. In the case of CHAR(1), it declares a character variable that can hold exactly one character. The keywords AS and BEGIN mark the beginning of the function body. The next line declares a local variable @C of type CHAR(1) to store the first letter of the color. The following line assigns the value of the first character of the @color parameter to the local variable @C. The SUBSTRING function is used to extract a substring from the @color parameter, starting from the first character (position 1) and with a length of 1.

So to summarise, we are saying that the function returns a fixed length character of length one, and then we ensure that the variable C holds the fixed length character of variable 1 which is calculated using the substring. We then return C which is a fixed length of 1 character, and hence it is returned by the function (which stipulates that CHAR(1) needs to be returned. The result is shown below:

A screenshot of a computer

Description automatically generated

Lets now create a function which gets the length of each word in the colour column. This time, we state that we want to return an INT instead, and we declare @C to be an INT. We then use the Length function to get the colour length, and then return C, which then returns the data type of INT within the function. The outer function always returns a data type, whereas the inner function actually does the calculation and passes the variable to the outer return function. The outer return function checks if the inner return functions output and datatype match that stated in the outer return.

CREATE FUNCTION get\_color\_length(@color NVARCHAR(100))

RETURNS INT

AS

BEGIN

DECLARE @C INT

SELECT @C = LEN(@Color)

RETURN @C

END

The result is shown below:

A table of numbers with black text

Description automatically generated

All functions are stored in the programmability folder as shown below:

A screenshot of a computer

Description automatically generated

# Views

A stored procedure is good as you can store your queries in there. For example, a left join on two columns can be stored as a stored procedure. However, a stored procedure only contains the query, it does not show the table. The stored procedure needs to be executed in order to get the resulting table. A view allows you to store the tables as virtual tables, and this virtual table can be referred to just as the way you refer to as normal table. You can do queries on these views.

Lets now create a view to join two tables.

CREATE VIEW Product\_Category AS

SELECT S.ProductCategoryID, S.Name, P.ProductID, P.Color FROM [SalesLT].[ProductCategory] AS S

INNER JOIN [SalesLT].[Product] AS P

ON S.ProductCategoryID = P.ProductCategoryID

The syntax to create a view is CREATE VIEW [view name] AS.

After executing, you can see the view in the folder directory of the left. It is named dbo.Product\_Category.

A screenshot of a computer

Description automatically generated

We can now query the view directly as the following:

SELECT \* FROM Product\_Category

The result is:

A screenshot of a table

Description automatically generated

This is very useful when joining two tables together.You can drop views as:

DROP VIEW dbo.Product\_Category

# Cursors

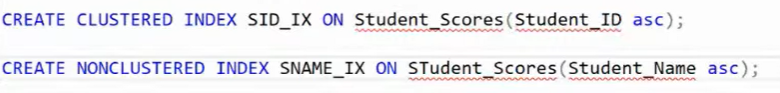
A white sheet with black text

Description automatically generated

This is a way to loop over rows and get info. This is not recommended due to performance issues. You can now do these transformations in another technology.

# Indexing

You can create clustered or non-clustered indexes with the syntax below. You have to get rid of the primary key constraints first if you want to create these.



In SQL, an index is a structure that enhances the speed of data retrieval operations on a database table. Two common types of indexes are clustered and non-clustered indexes.

A clustered index determines the physical order of the data rows in the table based on the indexed columns. There can be only one clustered index per table, and it directly affects the order in which the rows are stored on disk. When a table has a clustered index, the rows are organized in the same order as the index, providing faster retrieval of ranges of data. However, because the physical order of the rows is determined by the clustered index, any updates to the indexed columns may result in the rearrangement of the entire table, potentially impacting performance.

On the other hand, a non-clustered index does not alter the physical order of the data rows in the table. Instead, it creates a separate structure that includes a sorted list of references to the actual data rows. This allows for efficient retrieval of specific rows based on the indexed columns without changing the physical order of the table. A table can have multiple non-clustered indexes, providing flexibility in optimizing queries that involve different columns. While non-clustered indexes generally do not offer the same level of performance for range queries as clustered indexes, they are more suitable for scenarios where frequent data modifications occur, as they don't require reorganizing the entire table upon updates.