**Advanced SQL – Post Assessment**

**Exercise 1**

Create a query with the following columns:

* FirstName and LastName, from the Person.Person table\*\*
* JobTitle, from the HumanResources.Employee table\*\*
* Rate, from the HumanResources.EmployeePayHistory table\*\*
* A derived column called "AverageRate" that returns the average of all values in the "Rate" column, in each row

*\*\*All the above tables can be joined on BusinessEntityID*. All the tables can be inner joined, and you do not need to apply any criteria.

1. Enhance your query by adding a derived column called "MaximumRate" that returns the largest of all values in the "Rate" column, in each row.
2. Add a new derived column called "DiffFromAvgRate" that returns the result of the following calculation: An employees's pay rate, MINUS the average of all values in the "Rate" column.
3. Enhance your query by adding a derived column called "PercentofMaxRate" that returns the result of the following calculation: An employees's pay rate, DIVIDED BY the maximum of all values in the "Rate" column, times 100.

**Exercise 2**

Create a query with the following columns:

* “Name” from the Production.Product table, which can be alised as “ProductName”
* “ListPrice” from the Production.Product table
* “Name” from the Production. ProductSubcategory table, which can be alised as “ProductSubcategory”\*
* “Name” from the Production.ProductCategory table, which can be alised as “ProductCategory”\*\*

*\*Join Production.ProductSubcategory to Production.Product on “ProductSubcategoryID”*

*\*\*Join Production.ProductCategory to ProductSubcategory on “ProductCategoryID”*

All the tables can be inner joined, and you do not need to apply any criteria.

1. Enhance your query from Exercise 1 by adding a derived column called "AvgPriceByCategory " that returns the average ListPrice *for the product category in each given row*.
2. Add a derived column called "AvgPriceByCategoryAndSubcategory" that returns the average ListPrice *for the product category AND subcategory in each given row.*
3. Add a derived column called "ProductVsCategoryDelta" that returns the result of the following calculation: A product's list price, MINUS the average ListPrice for that product’s category.

**Exercise 3**

Create a query with the following columns (feel free to borrow your code from Exercise 2):

* “Name” from the Production.Product table, which can be alised as “ProductName”
* “ListPrice” from the Production.Product table
* “Name” from the Production. ProductSubcategory table, which can be alised as “ProductSubcategory”\*
* “Name” from the Production.ProductCategory table, which can be alised as “ProductCategory”\*\*

\*Join Production.ProductSubcategory to Production.Product on “ProductSubcategoryID”

\*\*Join Production.ProductCategory to ProductSubcategory on “ProductCategoryID”

All the tables can be inner joined, and you do not need to apply any criteria.

1. Add a derived column called "Price Rank " that ranks all records in the dataset by ListPrice, in descending order. That is to say, the product with the most expensive price should have a rank of 1, and the product with the least expensive price should have a rank equal to the number of records in the dataset.
2. Add a derived column called"Category Price Rank" that ranks all products by ListPrice – within each category - in descending order. In other words, every product within a given category should be ranked relative to other products in the same category.
3. Add a derived column called "Top 5 Price In Category" that returns the string “Yes” if a product has one of the top 5 list prices in its product category, and “No” if it does not. You can use a CASE statement to make this work.

**Exercise 4**

1. Using the query created from the above exercise3, add a derived column called “Category Price Rank With Rank” that uses the RANK function to rank all products by ListPrice – within each category - in descending order. Observe the differences between the “Category Price Rank” and “Category Price Rank With Rank” fields.
2. Add a derived column called "Category Price Rank With Dense Rank" that that uses the DENSE\_RANK function to rank all products by ListPrice – within each category - in descending order. Observe the differences among the “Category Price Rank”, “Category Price Rank With Rank”, and “Category Price Rank With Dense Rank” fields.

**Exercise 5**

Create a query with the following columns:

* “PurchaseOrderID” from the Purchasing.PurchaseOrderHeader table
* “OrderDate” from the Purchasing.PurchaseOrderHeader table
* “TotalDue” from the Purchasing.PurchaseOrderHeader table
* “Name” from the Purchasing.Vendor table, which can be aliased as “VendorName”**\***

\*Join Purchasing.Vendor to Purchasing.PurchaseOrderHeader on BusinessEntityID = VendorID

i) Apply the following criteria to the query:Order must have taken place on or after 2013 and TotalDue must be greater than $500

ii) Add a derived column called "PrevOrderFromVendorAmt", that returns the “previous” TotalDue value (relative to the current row) within the group of all orders with the same vendor ID. We are defining “previous” based on order date.

iii)Add a derived column called "NextOrderByEmployeeVendor", that returns the “next” vendor name (the “name” field from Purchasing.Vendor) within the group of all orders that have the same EmployeeID value in Purchasing.PurchaseOrderHeader. Similar to the last exercise, we are defining “next” based on order date.

iv)Add a derived column called "Next2OrderByEmployeeVendor" that returns, within the group of all orders that have the same EmployeeID, the vendor name offset TWO orders into the “future” relative to the order in the current row. The code should be very similar to Exercise 3, but with an extra argument passed to the Window Function used.

**Exercise 6**

Create a query with the following columns:

* “OrderMonth”, a derived column (you’ll have to create this one yourself) featuring the month **number**corresponding with the Order Date in a given row
* “OrderYear”, a derived column featuring the **year**corresponding with the Order Date in a given row
* “SubTotal” from the **Purchasing.PurchaseOrderHeader** table

i)Your query should be an aggregate query – specifically, it should **sum**“SubTotal”, and **group by**the remaining fields.

ii)Add a derived column called "Rolling3MonthTotal", that displays  - for a given row - a running total of “SubTotal” for the prior three months (including the current row).

HINT: You will need to include multiple fields in your ORDER BY to get this to work!

iii)Add another derived column called "MovingAvg6Month", that calculates a rolling average of “SubTotal” for the previous 6 months, relative to the month in the “current” row. Note that this average should NOT include the current row.

iv)Add another derived column called “MovingAvgNext2Months” , that calculates a rolling average of “SubTotal” for the month in the current row **and** the next two months after that. This moving average will provide a kind of "forecast" for Subtotal by month.

**Exercise 7**

For this exercise, assume the CEO of our fictional company decided that the top 10 orders per month are actually outliers that need to be clipped out of our data before doing meaningful analysis.

Further, she would like the sum of sales AND purchases (minus these "outliers") listed side by side, by month.

We've got a query that already does this (see the file "CTEs - Exercise Starter Code.sql" but it's messy and hard to read. Re-write it using a CTE so other analysts can read and understand the code.

**CTEs - Exercise Starter Code.sql**

SELECT

A.OrderMonth,

A.TotalSales,

B.TotalPurchases

FROM (

SELECT

OrderMonth,

TotalSales = SUM(TotalDue)

FROM (

SELECT

OrderDate

,OrderMonth = DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1)

,TotalDue

,OrderRank = ROW\_NUMBER() OVER(PARTITION BY DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1) ORDER BY TotalDue DESC)

FROM AdventureWorks2019.Sales.SalesOrderHeader

) S

WHERE OrderRank > 10

GROUP BY OrderMonth

) A

JOIN (

SELECT

OrderMonth,

TotalPurchases = SUM(TotalDue)

FROM (

SELECT

OrderDate

,OrderMonth = DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1)

,TotalDue

,OrderRank = ROW\_NUMBER() OVER(PARTITION BY DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1) ORDER BY TotalDue DESC)

FROM AdventureWorks2019.Purchasing.PurchaseOrderHeader

) P

WHERE OrderRank > 10

GROUP BY OrderMonth

) B ON A.OrderMonth = B.OrderMonth

ORDER BY 1

**Exercise 8**

Refactor your solution from exercise2 using temp tables in place of CTEs.

**Exercise 9**

Use the starter code.sql and optimize your solution using Indexes

**Starter code.sql**

CREATE TABLE #PersonContactInfo

(

BusinessEntityID INT

,Title VARCHAR(8)

,FirstName VARCHAR(50)

,MiddleName VARCHAR(50)

,LastName VARCHAR(50)

,PhoneNumber VARCHAR(25)

,PhoneNumberTypeID VARCHAR(25)

,PhoneNumberType VARCHAR(25)

,EmailAddress VARCHAR(50)

)

INSERT INTO #PersonContactInfo

(

BusinessEntityID

,Title

,FirstName

,MiddleName

,LastName

)

SELECT

BusinessEntityID

,Title

,FirstName

,MiddleName

,LastName

FROM AdventureWorks2019.Person.Person

UPDATE A

SET

PhoneNumber = B.PhoneNumber,

PhoneNumberTypeID = B.PhoneNumberTypeID

FROM #PersonContactInfo A

JOIN AdventureWorks2019.Person.PersonPhone B

ON A.BusinessEntityID = B.BusinessEntityID

UPDATE A

SET PhoneNumberType = B.Name

FROM #PersonContactInfo A

JOIN AdventureWorks2019.Person.PhoneNumberType B

ON A.PhoneNumberTypeID = B.PhoneNumberTypeID

UPDATE A

SET EmailAddress = B.EmailAddress

FROM #PersonContactInfo A

JOIN AdventureWorks2019.Person.EmailAddress B

ON A.BusinessEntityID = B.BusinessEntityID

SELECT \* FROM #PersonContactInfo

**Exercise 10**

Create a view named **vw\_Top10MonthOverMonth** in your AdventureWorks database, based on the query below. Assign the view to the **Sales** schema.

HINT: You will need to make a slight tweak to the query code before it can be successfully converted to a view.

WITH Sales AS

(

SELECT

OrderDate

,OrderMonth = DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1)

,TotalDue

,OrderRank = ROW\_NUMBER() OVER(PARTITION BY DATEFROMPARTS(YEAR(OrderDate),MONTH(OrderDate),1) ORDER BY TotalDue DESC)

FROM AdventureWorks2019.Sales.SalesOrderHeader)

,Top10Sales AS(

SELECT

OrderMonth,

Top10Total = SUM(TotalDue)

FROM Sales

WHERE OrderRank <= 10

GROUP BY OrderMonth

)

SELECT

A.OrderMonth,

A.Top10Total,

PrevTop10Total = B.Top10Total

 FROM Top10Sales A

LEFT JOIN Top10Sales B

ON A.OrderMonth = DATEADD(MONTH,1,B.OrderMonth)

ORDER BY 1