/\* Orysya Stus - Assignment 3 \*/

USE JProCo

/\* Lab 7.1: Rank() & Dense\_Rank() Skill Checks 1-3 \*/

/\*

Skill Check 1: In the JProCo database, use the correct ranking function to assign ranked values for each record in the Employee table based on the HireDate. The most recent HireDate should have a ranked value of 1 and each distinct date older than the first date should add 1 to the ranked value without any gaps. The newest hires will appear at the top of the list and the oldest hire dates will be at the bottom of the list. The ranked field should be aliased as GreenRank.

\*/

SELECT

\*,

DENSE\_RANK() OVER(ORDER BY HireDate DESC) AS GreenRank

FROM Employee

/\*

Skill Check 2: In the JProCo database, join the Employee and PayRates tables together to display the FirstName, LastName, YearlySalary and a ranking expression aliased as yrRank. Order the ranked values based on the highest to lowest YearlySalary values. The result set should only contain the fields for the first two ranked values provided by the RANK() function. Hint: It is possible to complete this exercise without implementing a derived table when using the TOP( n) clause. When done, the results of the query should resemble those shown in the figure below.

\*/

SELECT TOP 2

E.FirstName,

E.LastName,

P.YearlySalary,

RANK() OVER(ORDER BY P.YearlySalary DESC) AS yrRank

FROM Employee AS E

JOIN PayRates AS P

ON E.EmpID = P.EmpID

/\*

Skill Check 3: In the JProCo database, write a query that will display all the fields and records of the Grant table, plus a field to display the results of a ranking function that is aliased as AmountRank. The ranking function should assign the ranked values based on sorting the Amount field from the highest to the lowest value. When encountering a tie, assign the same ranked value to each of the tied rows, allowing for gaps in the ranked values to occur when finding the next non-tied row. When done, the query results should resemble those shown in the figure below.

\*/

SELECT

\*,

RANK() OVER(ORDER BY Amount DESC) AS AmountRank

FROM [Grant]

/\*

Skill Check 4: In the JProCo database context, write a query that uses the DENSE\_RANK() function to find only the 5th highest RetailPrice value in the CurrentProducts table. The expression field should be aliased as PriceRank. The field selection list should display only these fields in the following order: PriceRank, ProductID, ProductName, RetailPrice and Origination Date. Hint: Use a derived table to materialize the expression field, so it can be used as criteria to find the fifth record. When done the query results should resemble those shown in the figure below.

\*/

SELECT \* FROM

(SELECT \*, DENSE\_RANK() OVER(ORDER BY RetailPrice DESC) AS PriceRank

FROM CurrentProducts

ORDER BY RetailPrice DESC) AS Ranked

WHERE PriceRank <= 5

/\* Lab 7.2: ROW\_NUMBER \*/

/\*

Skill Check 1: In the JProCo database, locate the StateList table to write a query including all fields and records ranked by the smallest to largest value in the LandMass field. In the SELECT list, add three columns named RankNo, DRankNo and RowNo (representing the RANK(), DENSE\_RANK() and ROW\_NUMBER() functions) after the four columns already included in the StateList table. When done, the results of the query should resemble those shown below.

\*/

SELECT

\*,

RANK() OVER(ORDER BY LandMass) AS RankNo,

DENSE\_RANK() OVER(ORDER BY LandMass) AS DRankNo,

ROW\_NUMBER() OVER(ORDER BY LandMass) AS RowNo

FROM StateList

/\* Lab 8.1: NTILE \*/

/\*

Skill Check 1: In the JProCo database, locate the StateList table to write a query including all fields and records ranked by the largest to smallest value in the LandMass field. Group the rankings in 5% increments for all 63 records. In the selection list, add a column named StateGroup (representing the NTILE() function) after the four columns already included in the StateList table. When done, the query results should resemble those shown below.

\*/

SELECT

\*,

NTILE(16) OVER(ORDER BY LandMass DESC) AS StateGroup

FROM StateList

/\* Lab 8.2: Predicating Row Functions \*/

/\*

Our ranking and tiling examples have all used integers. For example, we can come in 4th place but we can’t come in 4 ½ th place. In this section, we will continue dealing with arithmetic integers. By definition , an even number is a number evenly divided by 2. While this is a simple concept, let us take a moment to review a couple of examples, as it is essential to understand this as we explore the arithmetic operator, modulo (commonly abbreviated as mod). o 30 divided by 2 has a quotient value of 15 with a remainder value of 0, so the number 30 is an even number. o 31 divided by 2 has a quotient value of 15 with a remainder value of 1, so the number 31 is an odd number. In other words, any number divided by two with a remainder of zero is an even number and conversely, any number divided by two with a remainder other than zero is an odd number. Understanding the quotient and remainder properties of division is important before we move into the basic concept behind the arithmetic operator modulo (“ mod” for short). When you apply modulo to any odd number, you always get the same result:

The answer is 1 (one). When you ‘mod’ any even number by 2, the answer will always be 0. So what do the following numbers have in common? 3, 6, 9, 12, 15… The answer is they all have a remainder of zero if you mod them by 3. So what do the following numbers have in common? 1, 4, 7, 10, 13… The answer is they all have a remainder of 1 if you mod them by 3. Modulo Examples Let’s begin by writing a very simple query to display the numbers 7 and 6 as shown below.

\*/

SELECT 7, 6

/\*

What results should we expect when dividing these two integers by the number 2? We learned at an early age that 7 / 2 = 3.5 (3 with a remainder of 1) and 6 / 2 = 3 (3 with a remainder of 0). What could we expect if we query six and seven divided by two? They should both be cut in half. 6/ 2 would be threeand7/ 2 would be three and a half or three with a remainder of one. Notice in the figure below the query returned both as three in the result set. Since we are working with integers there is no half and the remainder is discarded.

\*/

SELECT (7/2) AS [7/2], (6/2) AS [6/2]

/\*

So why is 3 shown as the result set for each of these calculations in the figure above? This is because integer data types are whole numbers (no fractions). We need to be careful when dividing integer data types, as the results returned may not always be what we intended. What results should we see when the dividing operand is replaced with an operand for modulo? The operand sign for modulo used by SQL (and many other languages) is the percent sign ‘%’. We can expect that 7 mod 2 will have a remainder of 1 and that 6 mod 2 will have a remainder of 0, as shown in the figure below.

\*/

SELECT (7 % 2) AS [7 % 2], (6 % 2) AS [6 % 2]

/\*

Finding Even-numbered Records

Oftentimes, we are challenged with tasks having more than one solution and it takes a bit more thought to find the most efficient approach. For example; The department heads of JProCo have asked us to create two reports, one with all the even-numbered CustomerID values from the Customer table and the other report should have all the odd numbered CustomerID values. If we write a simple SELECT query on the Customer table, we find that there are 775 records. By filtering on CustomerID mod two equal to zero, we will limit the records to only those that have a CustomerID with an even number shown below.

\*/

SELECT \*

FROM Customer

WHERE CustomerID % 2 = 0

/\*

Finding Odd-numbered Records

The risk of dividing customers on an identity field is that deletions can cause gaps. With deletions, even-numbered CustomerID values may not be every other row. A much better process to return every other customer would be to use the row numbers. With a simple modification to the WHERE clause, we can produce a report that finds only the records with odd numbered CustomerID values.

\*/

SELECT \*

FROM Customer

WHERE CustomerID % 2 = 1

/\*

Finding Every Fifth Record

What if we were assigned a more complicated task, such as finding the first record and then every fifth record afterwards in a result set? In other words, we must filter the results from a query to display only the 1st, 5th, 11th, 16th, 21st, etc... records for the required report. The figure below provides an example, using the ROW\_NUMBER() function with the OVER() clause.

\*/

SELECT \*

FROM (SELECT ROW\_NUMBER() OVER(ORDER BY RetailPrice DESC) AS RowID, \*

FROM CurrentProducts) AS DrvTbl WHERE RowID % 5 = 1

/\* Lab 8.2: Predicating Row Functions \*/

/\*

Skill Check 1: Write the query to find the even-numbered employees from the Employee table. When done your result should resemble the figure below.

\*/

SELECT \*

FROM Employee

WHERE EmpID % 2 = 0

/\*

Skill Check 2: From the Employee table of JProCo, show the even-numbered employee rows by HireDate. Alias the ranked row as SeniorRow. When done your result should resemble

\*/

SELECT \*

FROM (SELECT

DENSE\_RANK() OVER(ORDER BY HireDate ASC) AS SeniorRow,

\*

FROM Employee) AS A

WHERE A.SeniorRow % 2 = 0

/\*

Skill Check 3: From the CurrentProducts table of JProCo find the most recent product in each category. The expression field that finds the most recent origination date by category should be called MaxCatDate. When done your result should resemble the figure below.

\*/

SELECT TOP (5)

\*,

RANK() OVER(ORDER BY OriginationDate DESC) AS MaxCatDate

FROM CurrentProducts

GROUP BY OriginationDate

/\* Lab 10.2 Pivot \*/

SELECT \* FROM

(SELECT ProductID, OrderYear, RetailPrice

FROM vSales) AS Sales

PIVOT

(

SUM(RetailPrice)

FOR OrderYear in

([2009], [2010], [2011], [2012], [2013])

) as pvt

/\*

Skill Check 2: 1627 unit quanties sold in 2009. 136 of those were purchased in a quantity of 1. 308 products were purchased in a quantity of 2 as you see in the first record. Knowing the quanity of sales per invoice from vSales, write a query using PIVOT and the dbo.vSales view to achieve the result set you see in below. Be sure to Alias the fields as seen here.

\*/

SELECT

OrderYear,

Quantity,

COUNT(ProductID) AS OrderCount

FROM vSales

GROUP BY OrderYear, Quantity

ORDER BY OrderYear, Quantity

SELECT

OrderYear,

[1] AS Qty1,

[2] AS Qty2,

[3] AS Qty3,

[4] AS Qty4,

[5] AS Qty5,

[6] AS Qty6

FROM (SELECT

OrderYear,

Quantity,

COUNT(ProductID) AS OrderCount

FROM vSales

GROUP BY OrderYear, Quantity

ORDER BY OrderYear, Quantity

PIVOT

(

COUNT(ProductID)

For Quantity in

([1], [2], [3], [4], [5], [6])

) AS pvt

/\* Lab 10.3: UNPIVOT \*/

/\*

Please run the lab setup script downloaded from the Joes 2 Pros website and reset the database using the (SQLQueries2012Vol2Chapter10SpecialSetup.sql), please make sure to close all query windows within SSMS. After you run the SQLQueries2012Vol2Chapter10SpecialSetup.sql script then JProCo database will have the SalesGrid table as shown below.

\*/

SELECT \*

FROM SalesGrid

/\*

Skill Check 1: Write a query using UNPIVOT which will achieve the result set you see in the figure below.

\*/

SELECT \* FROM

(SELECT \* FROM SalesGrid) AS SG

UNPIVOT

(TotalSales

for CalYear in (

[2009],

[2010],

[2011],

[2012],

[2013])

) AS UNPVT