Intermediate T-SQL

# Summarizing Data

|  |
| --- |
| Creating aggregationsThis chapter uses data gathered by the National UFO Reporting Center. The data is contained in the Incidents table and in this lesson, you will be aggregating values in the DurationSeconds column. |
| -- Calculate the average, minimum and maximum  SELECT AVG(DurationSeconds) AS Average,  MIN(DurationSeconds) AS Minimum,  MAX(DurationSeconds) AS Maximum  FROM Incidents |
| 1.2 Creating grouped aggregations You can calculate statistics for each group using GROUP BY. For example, you can calculate the maximum value for each state using the following query:  SELECT State, MAX(DurationSeconds)  FROM Incidents  GROUP BY State  To filter even further, for example, to find the values for states where the maximum value is greater than 10, you can use the HAVING clause. |
| -- Calculate the aggregations by Shape  SELECT Shape,  AVG(DurationSeconds) AS Average,  MIN(DurationSeconds) AS Minimum,  MAX(DurationSeconds) AS Maximum  FROM Incidents  GROUP BY Shape  -- Return records where minimum of DurationSeconds is greater than 1  HAVING MIN(DurationSeconds) > 1 |
| 1.3 Removing missing values There are a number of different techniques you can use to fix missing data in T-SQL and in this exercise, you will focus on returning rows with non-missing values. For example, to return all rows with non-missing SHAPE values, you can use:  SELECT \*  FROM Incidents  WHERE Shape IS NOT NULL |
| -- Return the specified columnsSELECT IncidentDateTime, IncidentStateFROM Incidents-- Exclude all the missing values from IncidentStateWHERE IncidentState IS NOT NULL |
| 1.4 Imputing missing values (I)In the previous exercise, you looked at the non-missing values in the IncidentState column. But what if you want to replace the missing values with another value instead of omitting them? You can do this using the ISNULL() function. Here we replace all the missing values in the Shape column using the word 'Saucer':SELECT Shape, ISNULL(Shape, 'Saucer') AS Shape2FROM IncidentsYou can also use ISNULL() to replace values from a different column instead of a specified word. |
| -- Check the IncidentState column for missing values and replace them with the City columnSELECT IncidentState, ISNULL(IncidentState, City) AS LocationFROM Incidents-- Filter to only return missing values from IncidentStateWHERE IncidentState IS NULL |
| 1.5 Imputing missing values (II) What if you want to replace missing values in one column with another and want to check the replacement column to make sure it doesn't have any missing values? To do that you need to use the COALESCE statement.  SELECT Shape, City, COALESCE(Shape, City, 'Unknown') as NewShape  FROM Incidents  +----------------+-----------+-------------+  | Shape | City | NewShape |  +----------------+-----------+-------------+  | NULL | Orb | Orb |  | Triangle | Toledo | Triangle |  | NULL | NULL | Unknown |  +----------------+-----------+-------------+ |
| -- Replace missing valuesSELECT Country, COALESCE(Country, IncidentState, City) AS LocationFROM IncidentsWHERE Country IS NULL |
| 1.6 Using CASE statements In this exercise, you will use a CASE statement to create a new column which specifies whether the Country is USA or International. |
| SELECT Country,CASE WHEN Country = 'us' THEN 'USA'ELSE 'International'END AS SourceCountryFROM Incidents |
| 1.7 Creating several groups with CASE In this exercise, you will write a CASE statement to group the values in the DurationSeconds into 5 groups based on the following ranges:   | **DurationSeconds** | **SecondGroup** | | --- | --- | | <= 120 | 1 | | > 120 and <= 600 | 2 | | > 600 and <= 1200 | 3 | | > 1201 and <= 5000 | 4 | | For all other values | 5 | |
| SELECT DurationSeconds,-- Start by looking at the values in the column less than or equal to 120CASE WHEN (DurationSeconds <= 120) THEN 1-- Continue by looking at the column again for values between 121 and 600WHEN (DurationSeconds > 120 AND DurationSeconds <= 600) THEN 2-- Continue by looking at the column again for values between 601 and 1200WHEN (DurationSeconds > 601 AND DurationSeconds <= 1200) THEN 3-- Continue by looking at the column again for values between 1201 and 5000WHEN (DurationSeconds > 1201 AND DurationSeconds <= 5000) THEN 4-- For all other values in DurationSecondsELSE 5END AS SecondGroupFROM Incidents |

## Math Functions

This chapter explores essential math operations such as rounding numbers, calculating squares and square roots, and counting records. You will also work with dates in this chapter!

|  |
| --- |
| 2.1 Calculating the total In this chapter, you will use the shipments data. The Shipments table has several columns such as:   * MixDesc: the concrete type * Quantity: the amount of concrete shipped   In this exercise, your objective is to calculate the total quantity for each type of concrete used. |
| -- Write a query that returns an aggregation  SELECT MixDesc, SUM(Quantity) AS Total  FROM Shipments  -- Group by the relevant column  GROUP BY MixDesc |
| 2.2 Counting the number of rows In this exercise, you will calculate the number of orders for each concrete type. Since each row represents one order, all you need to is count the number of rows for each type of MixDesc.  -- Count the number of rows by MixDesc  SELECT MixDesc, COUNT(\*)  FROM Shipments  GROUP BY MixDesc |
| 2.3 Counting the number of days between dates In this exercise, you will calculate the difference between the order date and ship date. |
| -- Return the difference in OrderDate and ShipDateSELECT OrderDate, ShipDate,DATEDIFF(DD, OrderDate, ShipDate) AS DurationFROM Shipments |
| 2.4 Adding days to a date In this exercise, you will calculate the approximate delivery date of an order based on ShipDate. -- Return the DeliveryDate as 5 days after the ShipDateSELECT OrderDate,DATEADD(DD, 5, ShipDate) AS DeliveryDateFROM Shipments |
| 2.5 Rounding numbers Sometimes, you only care about the whole dollar amount and want to ignore the decimal values of the cost. In this exercise, you will round the cost to the nearest dollar. -- Round Cost to the nearest dollarSELECT Cost,ROUND(Cost, 0) AS RoundedCostFROM Shipments |
| 2.6 Truncating numbers Since rounding can sometimes be misleading, i.e., **$16.8** becomes **$17** while **$16.4** remains **$16**, you may want to truncate the values after the decimal instead of rounding them. When you truncate the numbers, both **$16.8** and **$16.4** remain **$16**. In this exercise, you will do just that, truncate the Cost column to a whole number.  -- Truncate cost to whole number  SELECT Cost,  ROUND(Cost, 0, 1) AS TruncateCost  FROM Shipments |
| 2.7 Calculating the absolute value The Shipments table contains some bad data. There was a problem with the scales, and the weights show up as negative numbers. In this exercise, you will write a query to convert all negative weights to positive weights. -- Return the absolute value of DeliveryWeightSELECT DeliveryWeight,ABS(DeliveryWeight) AS AbsoluteValueFROM Shipments |
| 2.8 Calculating squares and square roots It's time for you to practice calculating squares and square roots of columns. -- Return the square and square root of WeightValueSELECT WeightValue,SQUARE(WeightValue) AS WeightSquare,SQRT(WeightValue) AS WeightSqrtFROM Shipments |

1. Processing Data in SQL Server

In this chapter, you will create variables and write while loops to process data. You will also write complex queries by using derived tables and common table expressions.

|  |
| --- |
| 3.1 Creating and using variables In T-SQL, to create a variable you use the DECLARE statement. The variables must have an at sign (@) as their first character. Like most things in T-SQL, variables are not case sensitive. To assign a value to a variable, you can either use the keyword SET or a SELECT statement followed by an equal sign and a value. |
| -- Create the variable  DECLARE @counter INT  -- Assign a value to the variable  SET @counter = 20  -- Increment the variable by one  SET @counter = @counter + 1  -- Print the variable  SELECT @counter |
| 3.2 Creating a WHILE loop  In this exercise, you will use the variable you created in the previous exercise you write a WHILE loop. Recall that structure:  WHILE some\_condition  BEGIN  -- Perform some operation here  END |
| DECLARE @counter INT  SET @counter = 20  -- Create a loop  WHILE @counter < 30  -- Loop code starting point  BEGIN  SELECT @counter = @counter + 1  -- Loop finish  END  -- Check the value of the variable  SELECT @counter |
| **What are Derived tables?**   * Query which is treated like a temporary table * Always contained within the main query * Thera r specified in the FROM clause * \* Can contain intermediate calculations to be used the main query or different joins than in the main query   SELECT a.\* FROM Kidney a  --This derived table computes the Average age jointed to the actual table  JOIN (SELECT AVG(Age) AS AverageAge  FROM Kidney) b  ON a.Age = b.AverageAge |
| **3.3** Queries with derived tables (I) The focus of this lesson is derived tables. You can use derived tables when you want to break down a complex query into smaller steps. Derived tables are a great solution if you want to create intermediate calculations that need to be used in a larger query.  In this exercise, you will calculate the maximum value of the blood glucose level for each record by age.  SELECT a.RecordId, a.Age, a.BloodGlucoseRandom,  -- Maximum Glucose value from the derived table  b.MaxGlucose  FROM Kidney a  -- Derived table  JOIN (SELECT Age, MAX(BloodGlucoseRandom) AS MaxGlucose FROM Kidney GROUP BY Age) b  -- Join on Age  ON a.Age = b.Age |
| **3.4** Queries with derived tables (II) In this exercise, you will create a derived table to return all patient records with the highest BloodPressure at their Age level.  SELECT \*  FROM Kidney a  -- JOIN and create the derived table  JOIN (SELECT Age, MAX(BloodPressure) AS MaxValue  FROM Kidney  GROUP BY Age) b  -- JOIN on BloodPressure and MaxValue  ON a.BloodPressure = b.MaxValue  -- Join on Age  AND a.Age = b.Age |
| **Common Table Expressions** CTE syntax  --CTE definition start with the keyword WITH  -- Followed by the CTE names and the columns it contains  WITH CTEName (Col1, Col2)  AS  --- Define the CTE query  (  --The two columns from the definition above  SELECT Col1, Col2  FROM TableName  )  Example:  --Create a CTL to get the Maximum BloodPressure by Age  WITH BlookPressureAge(Age, MaxBloodPressure)  As  (SELECT Age, MAX(BloodPressure) AS MaxBloodPressure  FROM Kidney  GROUP BY Age)  --Create a query to use the CTE as a table  SELECT a.Age, Min(a.BloodPressure), b.MaxBloodPressure  FROM Kidney a  --Join the CTE with the table  JOIN BloodpressureAge b  ON a.Age = b.Age  GROUP BY a.Age, b.MaxBloodPressure |
| **3.6** Creating CTEs (I) A Common table expression or CTE is used to create a table that can later be used with a query. To create a CTE, you will always use the WITH keyword followed by the CTE name and the name of the columns the CTE contains. The CTE will also include the definition of the table enclosed within the AS().  In this exercise, you will use a CTE to return all the ages with the maximum BloodGlucoseRandom in the table. **-- Create the CTE****WITH BloodGlucoseRandom (MaxGlucose)** **AS (SELECT MAX(BloodGlucoseRandom) AS MaxGlucose FROM Kidney)****SELECT a.Age, b.MaxGlucose****FROM Kidney a****-- Join the CTE** **JOIN BloodGlucoseRandom b****ON a.BloodGlucoseRandom = b.MaxGlucose** |
| **3.7** Creating CTEs (II) In this exercise, you will use a CTE to return all the information regarding the patient(s) with the maximum BloodPressure. **-- Create the CTE****WITH BloodPressure (MaxBloodPressure)** **AS (SELECT MAX(BloodPressure) AS MaxBloodPressure FROM Kidney)****SELECT \*****FROM Kidney a****-- Join the CTE** **JOIN BloodPressure b****ON a.BloodPressure = b.MaxBloodPressure** |

1. Window Functions

|  |
| --- |
| Grouping data in T-SQL  SELECT SalesPerson, SalesYear, CurrentQuota, ModifiedData  FROM SaleGoal  WHERE SalesYear = 2011  Window Syntax in T-SQL   * Create the window with OVER clause * PARTITION BY creates the frame * If you do not include PARTITION BY the frames is the entire table * To arrange the results, use ORDER BY * Allows aggregations to be created at the same time as the window.   SELECT SalesPerson, SalesYear, CurrentQuota,  SUM(CurrentQuote)  OVER (PARTITION BY SalesYear) As YearlyTotal,  ModifiedDate AS ModDate  FROM SaleGoal |
| 4.1 Window functions with aggregations (I) To familiarize yourself with the window functions, you will work with the Orders table in this chapter. Recall that using OVER(), you can create a window for the entire table. To create partitions using a specific column, you need to use OVER() along with PARTITION BY.  SELECT OrderID, TerritoryName,  -- Total price for each partition  SUM(OrderPrice)  -- Create the window and partitions  OVER(PARTITION BY TerritoryName) AS TotalPrice  FROM Orders |
| 4.2 Window functions with aggregations (II) In the last exercise, you calculated the sum of all orders for each territory. In this exercise, you will calculate the number of orders in each territory.  SELECT OrderID, TerritoryName,  -- Number of rows per partition  COUNT(\*)  -- Create the window and partitions  OVER(PARTITION BY TerritoryName) AS TotalOrders  FROM Orders |
| 4.3 First value in a window Suppose you want to figure out the first OrderDate in each territory or the last one? How would you do that? You can use the window functions FIRST\_VALUE() and LAST\_VALUE(), respectively! Here are the steps:   * First, create partitions for each territory * Then, order by OrderDate * Finally, use the FIRST\_VALUE() and/or LAST\_VALUE() functions as per your requirement  SELECT TerritoryName, OrderDate,-- Select the first value in each partitionFIRST\_VALUE(OrderDate)-- Create the partitions and arrange the rowsOVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS FirstOrderFROM Orders |
| 4.3 Previous and next values What if you want to shift the values in a column by one row up or down? You can use the exact same steps as in the previous exercise but with two new functions, LEAD(), for the next value, and LAG(), for the previous value. So you follow these steps:   * First, create partitions * Then, order by a certain column * Finally, use the LEAD() and/or LAG() functions as per your requirement  SELECT TerritoryName, OrderDate,-- Previous OrderDate in the windowLAG(OrderDate)-- Create the partitions and arrange the rowsOVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS PreviousOrder,-- Next OrderDate in the windowLEAD(OrderDate)-- Create the partitions and arrange the rowsOVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS NextOrderFROM Orders |
| 4.4 Creating running totals You usually don't have to use ORDER BY when using aggregations, but if you want to create running totals, you should arrange your rows! In this exercise, you will create a running total of OrderPrice. SELECT TerritoryName, OrderDate,-- Create a running totalSUM(OrderPrice)-- Create the partitions and arrange the rowsOVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS TerritoryTotalFROM Orders |
| 4.5 Assigning row numbers Records in T-SQL are inherently unordered. Although in certain situations, you may want to assign row numbers for reference. In this exercise, you will do just that. SELECT TerritoryName, OrderDate,-- Assign a row numberRow\_Number()-- Create the partitions and arrange the rowsOVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS OrderCountFROM Orders |
| 4.6 Calculating standard deviation Calculating the standard deviation is quite common when dealing with numeric columns. In this exercise, you will calculate the running standard deviation, similar to the running total you calculated in the previous lesson.  SELECT OrderDate, TerritoryName,  -- Calculate the standard deviation  STDEV(OrderPrice)  OVER(PARTITION BY TerritoryName ORDER BY OrderDate) AS StdDevPrice  FROM Orders |
| 4.7 Calculating mode (I) Unfortunately, there is no function to calculate the mode, the most recurring value in a column. To calculate the mode:   * First, create a CTE containing an ordered count of values using ROW\_NUMBER() * Write a query using the CTE to pick the value with the highest row number   In this exercise, you will write the CTE needed to calculate the mode of OrderPrice. -- Create a CTE Called ModePrice which contains two columnsWITH ModePrice(OrderPrice, UnitPriceFrequency)AS(SELECT OrderPrice,ROW\_NUMBER()OVER(PARTITION BY OrderPrice ORDER BY OrderPrice) AS UnitPriceFrequencyFROM Orders)-- Return all of the rows in the CTESELECT \* FROM ModePrice |
| 4.8 Calculating mode (II) In the last exercise, you created a CTE which assigned row numbers to each unique value in OrderPrice. All you need to do now is to find the OrderPrice with the highest row number. -- CTE from the previous exerciseWITH ModePrice (OrderPrice, UnitPriceFrequency)AS(SELECT OrderPrice,ROW\_NUMBER()OVER (PARTITION BY OrderPrice ORDER BY OrderPrice) AS UnitPriceFrequencyFROM Orders)-- Calculate the modeSELECT OrderPrice AS ModeFROM ModePrice-- The WHERE clause should only return the maximum value of UnitPriceFrequencyWHERE UnitPriceFrequency IN (SELECT MAX(UnitPriceFrequency) From ModePrice) |

## 