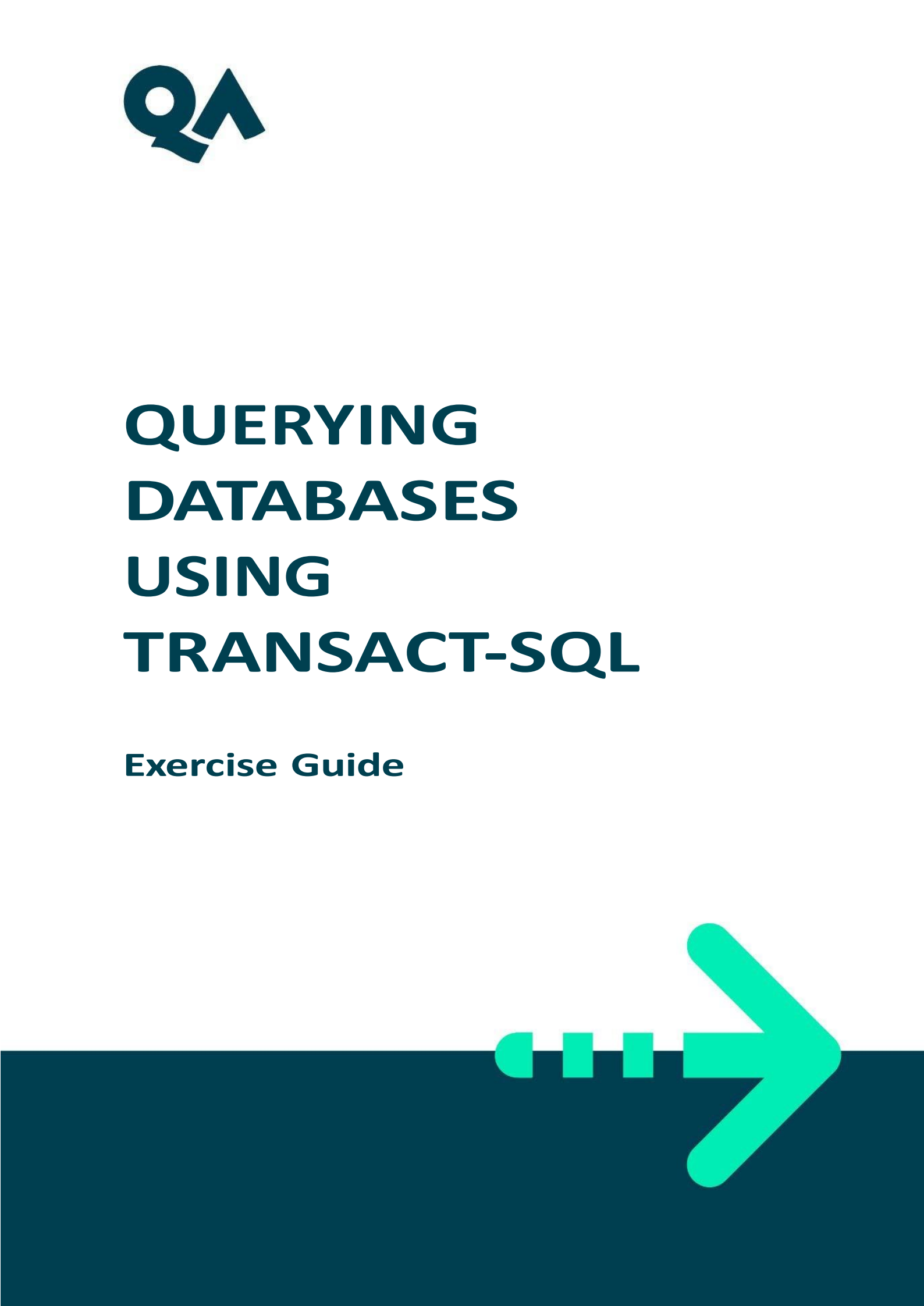

QUERYING DATABASES USING TRANSACT-SQL

Exercise Guide

Contents

Module 2 – retrieving data

Overview

In this lab, you will be working with the dbo.Products table in the Northwind database to produce a list of products and their stock levels.

If you're feeling confident, there is a second exercise which uses some string concatenation on the dbo.Employees table.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- write a simple select statement.
- write a select statement that picks out only certain columns from the source table.
- write a select statement that uses calculated values.

Setup: launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Administrator.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: retrieve data by using the SELECT statement

Northwind Traders are attempting to learn more about their stock levels. You have been asked to produce a list of products.

In this exercise, you will create simple SELECT statement queries.

The main tasks for this exercise are as follows:

1. Write a query using the SELECT * statement against the Products table.
2. Write a query that selects the ProductID, ProductName, UnitPrice and UnitsInStock columns from the Products table.
3. Add columns to the second query so that it also calculates the current value of stock and the future value of stock.

Task 1: write a query using SELECT *

1. Create a new query and save it with a filename of 'SelectAllProductDetails.sql'
2. Create a query that uses the Northwind database and then displays all columns and all rows from the dbo.Products table.
3. Execute the query by hitting the <F5> key.
4. Browse the result set in the Results pane. You should see 77 rows returned.

Task 2: write a query that selects individual columns from a table

1. Create another new query and call it 'StockList.sql'.
2. It should access the dbo.Products table in the Northwind database.
3. It should only include the ProductID, ProductName, UnitPrice and UnitsInStock columns.
4. Execute the query by hitting the <F5> key.
5. Browse the result set in the Results pane and confirm that only the specified columns appear.

Task 3: write a query that selects individual columns from a table

1. Edit the existing 'StockList.sql' query.
2. Add the UnitsOnOrder column to the select list.
3. Add a calculated column that multiplies UnitPrice by UnitsInStock and alias it as CurrentStockValue.
4. Add a second calculated column that adds UnitsOnOrder to UnitsInStock and multiplies the result by UnitPrice. Alias the new column as FutureStockValue.
5. Execute the query by hitting the <F5> key.
6. Browse the result set in the Results pane and confirm that the current stock value of ProductID 2, Chang, is 323 and its future value is 1083.

(Optional) Exercise 2: concatenate strings in a select list

Write a query that selects the FullName and telephone Extension from the dbo.Employees table in the Northwind database.

You'll need to concatenate the FirstName and LastName columns and you'll need to put a space between them. There are 9 rows in the Employees table.

Completed scripts

Exercise 1 task 1:

```
USE Northwind

SELECT *
FROM dbo.Products
```

Exercise 1 task 2:

```
USE Northwind

SELECT ProductID, ProductName, UnitPrice, UnitsInStock
FROM dbo.Products
```

Exercise 1 task 3:

```
SELECT
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
```

Exercise 2:

```
SELECT
  FirstName + ' ' + LastName AS FullName,
  Extension FROM
  dbo.Employees
```

Module 3 – Filtering Rows

Overview

In this lab, you will work with the dbo.Products table to produce a list of products in a certain category, adding various different filters to the query.

Then you will work with the dbo.Orders table to find orders that were placed between certain dates for certain customers.

The third exercise involves working with string comparisons on the dbo.Customers table.

The last exercise involves finding rows containing NULL values in the dbo.Customers table.

The 'optional extras' are some final suggestions for additional tasks that you may choose to attempt if you are feeling confident. They are intended to inspire you to play around with the various filter operations once you've completed the core tasks and are entirely optional.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- write a query that uses an equality filter.
- write a query that uses a comparison filter.
- write a query that uses the IN and BETWEEN operators.
- write a query statement that looks for text within a column.
- write a query that correctly identifies NULL values.

Setup: Launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Administrator.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: Basic WHERE clauses

Northwind Traders are attempting to learn more about their currently stocked beverages. You have been asked to produce a list of current products that are in that category and that have a high value.

In this exercise, you will use basic WHERE clauses.

The main tasks for this exercise are as follows:

1. Write a query that selects only the rows in the dbo.Products table which have a value of 0 (zero) in the Discontinued column.
2. Modify the query so that it only returns non-discontinued Products with a CategoryID of 1 (beverages).
3. Modify the query to include an additional filter that removes Products with a unit price less than 40.

Task 1: write a query that only returns current products

1. Create a new query and save it with a name of 'CurrentBeverageProducts.sql'.
2. Write a query that uses the Northwind database and displays the ProductID, ProductName, CategoryID, Discontinued and UnitPrice columns from the dbo.Products table.
3. Execute the query and verify that it returns 77 rows.
4. Add a WHERE clause to the query that asks for only those products with a Discontinued value of 0 (zero).
5. Execute the query and verify that it returns only 69 rows.

Task 2: write a query that only returns current products in category 1

1. Modify your existing query.
2. Add an additional statement to the WHERE clause that further limits the results to products with a CategoryID of 1.
3. Execute the query and verify that it now returns only 11 rows.

Task 3: write a query that only returns expensive, current products in category 1

1. Modify your existing query.
2. Add an additional statement to the WHERE that further limits the results to products with a unit price greater than or equal to 40.
3. Execute the query and verify that it now retrieves just 2 products – Cote de Blaye and Ipoh Coffee.

Exercise 2: using IN and BETWEEN

Northwind Traders have had some complaints about orders that were placed in a certain month for certain customers. You have been asked to produce a query of all the orders that were placed by those customers in that month.

In this exercise, you will use the IN and BETWEEN operators.

The main tasks for this exercise are as follows:

1. Write a query that selects only the rows in the dbo.Orders table which were placed by either Alfreds Futterkiste (CustomerID 'ALFKI'), Ernst Handle ('ERNSH') or Simon's bistro ('SIMOB').
2. Modify the query so that it only returns orders for those customers placed between the 1st August and the 31st August 1997.
3. Ensure that the query uses IN and BETWEEN operators.

Task 1: write a query that only returns orders for some customers

1. Create a new query and save it with a name of 'CustomerComplaints.sql'.
2. Write a query that uses the Northwind database and displays the OrderID, CustomerID and OrderDate columns from the dbo.Orders table.
3. Limit the results to only those orders with a CustomerID of either 'ALFKI', 'ERNSH', or 'SIMOB'.
4. Remember that string literals need to be enclosed in 'single quotes'.
5. Execute the query and verify that it returns 43 rows.

Task 2: write a query that returns orders for some customers in August 1997

1. Modify your existing query.
2. Further limit the results of the query so that it only returns orders with an OrderDate greater than or equal to the 1st August 1997 and an OrderDate of less than or equal to the 31st August 1997.
3. Remember that date literals, like string literals, need to be enclosed in 'single quotes' ideally using the format 'YYYYMMDD'.
4. Execute the query and verify that there are now just three rows – OrderIDs 10633, 10642, and 10643.
5. Make sure the query returns the three rows we want. 37 rows and 73 rows are wrong for different reasons.

Task 3: ensure that the query uses IN and BETWEEN

1. Modify your existing query.
2. If it doesn't already, ensure that the query uses IN for the CustomerIDs and BETWEEN for the OrderDates.
3. Notice that the query is much easier to read and doesn't require so many parentheses.
4. Execute the query and verify that it still returns three rows.

Exercise 3: String comparisons

Northwind Traders Marketing department has asked you to produce a list of all your customer contacts who are involved in the sales function.

In this exercise, you will use the LIKE operator and wildcard characters.

The main tasks for this exercise are as follows:

1. Write a query that selects rows from the dbo.Customers table where the Contact's job description starts with the word 'Sales'.
2. Modify the query so that it only returns contact details for anyone with the word 'sales' anywhere in their job description.
3. Experiment with case sensitivity.

Task 1: write a query that only returns contact details for some customers

1. Create a new query and save it with a name of 'SalesContacts.sql'.
2. Write a query that uses the Northwind database and selects the CustomerID, CompanyName, ContactName, ContactTitle and Phone columns from the dbo.Customers table.
3. Execute the query and verify that it returns 91 rows.
4. Now add a WHERE clause which looks for a ContactTitle that begins with the word 'sales'.
5. You should retrieve 40 rows.

Task 2: use leading and trailing wildcards

1. Modify the existing query.
2. Modify the LIKE comparison so that it looks for Contacts where the word 'sales' appears anywhere in their ContactTitle column.
3. You're expecting 43 rows now – there are 3 customer contacts whose job title is 'Assistant Sales Agent'.

Task 3: investigate case sensitivity in SQL Server

1. Modify the existing query.

2. Try looking for SALES, or Sales, or sales, or even SaLeS. You should still see the same 43 rows – by default, SQL Server is configured to be case insensitive.

Exercise 4: NULL comparisons

Northwind Traders Marketing department wants to update all their contact details to ensure that they have fax numbers for everyone. You have been asked to produce a list of all your customer contacts that don't have a Fax number listed.

In this exercise, you will use the IS NULL operator.

The main tasks for this exercise are as follows:

1. Write a query that selects rows from the dbo.Customers table where the Fax column has a NULL value.

Task 1: write a query that only lists customers with no fax number listed

1. Create a new query and save it with a name of 'FaxlessContacts.sql'.
2. Write a query that uses the Northwind database and selects the CustomerID, CompanyName, ContactName, ContactTitle, Phone and Fax columns from the dbo.Customers table.
3. Execute the query and verify that it returns 91 rows.
4. Now add a WHERE clause which looks for null values in the Fax column.
5. You should retrieve 22 rows.

(Optional) Exercise 5: filtering on expressions

You have been asked to produce a query on products whose future stock value is greater than 2000. Re-use the final 'StockList.sql' query from Lab 2 Exercise 1 Task 3 but add an appropriate WHERE clause. You should get 14 rows.

Optional extras:

1. Modify the 'CurrentBeverageProducts.sql' expensive products query from Exercise 1 so that it looks for current products in both category 1 and category 8.
2. Modify the 'CustomerComplaints.sql' problem orders query from Exercise 2 so that it looks at a bigger range of dates but also limits the results to only certain EmployeeIDs (say 1,3,5,7 and 9).
3. Try using an '= NULL' comparison instead of an IS NULL in the query from exercise 4. This should return no rows.

Completed scripts

Exercise 1 task 1:

```
USE Northwind

SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products WHERE
  Discontinued = 0
```

Exercise 1 task 2:

```
SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice
FROM
  dbo.Products
WHERE
  Discontinued = 0 AND CategoryID = 1
```

Exercise 1 task 3:

```
SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products
WHERE
  Discontinued = 0 AND CategoryID = 1 AND UnitPrice >= 40
```

Exercise 2 task 1:

```
USE Northwind

SELECT
  OrderID, CustomerID, OrderDate FROM
  dbo.Orders
WHERE
  CustomerID = 'ALFKI' OR CustomerID = 'ERNSH'  OR
CustomerID = 'SIMOB'
```

Exercise 2 task 2:

```
SELECT
  OrderID, CustomerID, OrderDate FROM
  dbo.Orders
WHERE
  (CustomerID = 'ALFKI' OR CustomerID = 'ERNSH'
   OR CustomerID = 'SIMOB')
AND
  (OrderDate >= '19970801' AND OrderDate <= '19970831')
```

Exercise 2 task 3:

```
SELECT
  OrderID, CustomerID, OrderDate FROM
  dbo.Orders
WHERE
  CustomerID IN ('ALFKI', 'ERNSH', 'SIMOB') AND
  OrderDate BETWEEN '19970801'  AND '19970831'
```

Exercise 3 task 1:

```
USE Northwind

SELECT
  CustomerID, CompanyName, ContactName, ContactTitle, Phone
FROM   dbo.Customers
WHERE
  ContactTitle LIKE 'sales%'
```

Exercise 3 task 2:

```
SELECT
  CustomerID,  CompanyName,  ContactName,  ContactTitle,  Phone  FROM
dbo.Customers
WHERE
  ContactTitle LIKE '%sales%'
```

Exercise 4 task 1:

```
SELECT
  CustomerID, CompanyName, ContactName, ContactTitle, Phone, Fax FROM
  dbo.Customers WHERE
  Fax IS NULL
```

Exercise 5:

```
SELECT
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
WHERE
  (UnitsInStock + UnitsOnOrder) * UnitPrice > 2000
```

Optional extras 1

```
USE Northwind

SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products
WHERE
  Discontinued = 0 AND CategoryID IN (1,8) AND UnitPrice >= 40
```

Optional extras 2

```
USE Northwind

SELECT
  OrderID, CustomerID, OrderDate FROM
  dbo.Orders
WHERE
  CustomerID IN ('ALFKI', 'ERNSH', 'SIMOB') AND
  OrderDate BETWEEN '19960928' AND '19980228'
AND
     EmployeeID IN (1,3,5,7,9)
```

Optional extras 3

```
USE Northwind

SELECT
  CustomerID, CompanyName, ContactName, ContactTitle, Phone, Fax FROM
  dbo.Customers
WHERE
  Fax = NULL
```

Module 4 – Sorting Rows

Overview

In this lab, you will work with the dbo.Products table to produce a list of products sorted by category and by unit price within those categories.

In the second exercise you will revisit an earlier product query to sort the output.

The third exercise asks you to produce a query of unique countries from the dbo.Customers table.

The fourth exercise involves listing the ten most expensive products.

The final, optional, exercise involves listing the ten products with the highest value of current stock.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- write a query that sorts on a single column.
- write a query that sorts on multiple columns.
- write a query that sorts on a calculated column.
- write a query that selects unique values.
- write a query that returns only a specified number of rows.

Setup: Launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Administrator.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: Basic ORDER BY clauses

Northwind Traders are attempting to learn more about their product pricing. You have been asked to produce a list of current products, sorted by category and then by unit price in reverse order.

In this exercise, you will use basic ORDER BY clauses.

The main tasks for this exercise are as follows:

1. Write a query that selects only the rows in the dbo.Products table which have a value of 0 (zero) in the Discontinued column.
2. Modify the query so that it only returns non-discontinued Products sorted by their CategoryIDs.
3. Modify the query to include an additional sort that sorts on UnitPrice from highest to lowest.

Task 1: write a query that only returns current products

1. Create a new query and save it with a name of 'CurrentProducts.sql'.
2. Write a query that uses the Northwind database and displays the ProductID, ProductName, CategoryID, Discontinued and UnitPrice columns from the dbo.Products table and only selects rows with a Discontinued value of 0.
3. Execute the query and verify that it returns 69 rows.

Task 2: write a query that sorts current products by category

1. Modify your existing query.
2. Add an ORDER BY clause to the query that sorts the results based on the CategoryID column.
3. Execute the query and verify that it returns the 69 rows but that all the products with a CategoryID of 1 are together, all those with CategoryID 2 are together, etc.

Task 3: write a query that sorts current products by category and unit price

1. Modify your existing query.
2. Add an additional statement to the ORDER BY clause that sorts the results in reverse order on UnitPrice.
3. Execute the query and verify that your 69 rows are still sorted by CategoryID but that within those categories the most expensive products are displayed first.

Exercise 2: sorting on calculated columns

Northwind Traders are still trying to get a handle on their stock levels. They are really pleased with a stock query you wrote earlier but now that they know you can sort data, they'd like you to make it easier to see which products have the highest value of future stock.

In this exercise, you will work on an earlier query and add an ORDER BY clause to it.

The main tasks for this exercise are as follows:

1. Re-use, or copy and paste, or re-type the StockList.sql query from Lab 2 Exercise 1 Task 3.
2. Modify the query so that it sorts the results in reverse order of future stock value.

Task 1: re-use an existing query

1. Create a copy of the script file 'StockList.sql'.
2. Rename the new file 'SortedStockList.sql'.

   Note: If you can't find the query file, the SQL is reproduced below:

```
SELECT
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
```

3. Open the query in SSMS.
4. Execute the query and verify that it returns 77 rows.

Task 2: write a query that sorts on a calculated column

1. Modify the existing query.
2. Add an order by clause that sorts on the value of the calculated 'FutureStockValue' column, in reverse order.

   NOTE: There are three different ways of achieving this.
3. Execute the query and verify that the first product returned is Cote de Blaye, product id 38.

Exercise 3: SELECT DISTINCT

You have been asked to produce a list of the unique countries in which Northwind Traders has customers.

In this exercise, you will use the DISTINCT modifier on a SELECT statement.

The main tasks for this exercise are as follows:

1. Write a query that selects the Country column from the dbo.Customers table.
2. Modify the query so that it only returns a single row for duplicate country names.

Task 1: write a query that selects the customer's countries

1. Create a new query and save it with a name of 'CustomerCountries.sql'.
2. Write a query that uses the Northwind database and selects the Country column from the dbo.Customers table.
3. Execute the query and verify that it returns 91 rows.

Task 2: select distinct rows

1. Modify the existing query.
2. Modify the SELECT statement so that it looks for distinct values only.
3. You're expecting 21 rows now. Notice that the countries are sorted alphabetically as well.

Exercise 4: SELECT TOP

You have been asked to produce a list of all the top ten most expensive products based on unit price.

In this exercise, you will use the TOP modifier.

The main task for this exercise is as follows:

1. Write a query that selects the top ten rows from the dbo.Products table sorted on reverse value of unit price.

Task 1: write a query of the top ten most expensive products

1. Create a new query and save it with a name of 'MostExpensiveProducts.sql'.
2. Write a query that uses the Northwind database and selects the ProductID, ProductName and UnitPrice columns from the dbo.Products table.
3. Order the results in descending order of UnitPrice.
4. Restrict the results to the first ten results
5. You should retrieve 10 rows. Cote de Blaye, unsurprisingly, should be the first.

(Optional) Exercise 5: more TOPping

You have been asked to modify a copy of the SortedStockList.sql query from Exercise 2 Task 2 of this lab, and select only the top ten products based on their current stock value.

Completed scripts

Exercise 1 task 1:

```
USE Northwind

SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products
WHERE
  Discontinued = 0
```

Exercise 1 task 2:

```
SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products
WHERE
  Discontinued = 0
ORDER BY
  CategoryID
```

Exercise 1 task 3:

```
SELECT
  ProductID, ProductName, CategoryID, Discontinued, UnitPrice FROM
dbo.Products
WHERE
  Discontinued = 0 ORDER BY
  CategoryID,
  UnitPrice DESC
```

Exercise 2 task 1:

```
USE Northwind

SELECT
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
```

Exercise 2 task 2:

```
SELECT
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
ORDER BY
  FutureStockValue DESC
```

Exercise 3 task 1:

```
USE Northwind

SELECT
  Country FROM
  dbo.Customers
```

Exercise 3 task 2:

```
SELECT DISTINCT
  Country FROM
  dbo.Customers
```

Exercise 4:

```
USE Northwind

SELECT TOP 10
  ProductID, ProductName, UnitPrice FROM
  dbo.Products ORDER
BY
  UnitPrice DESC
```

Exercise 5:

```
SELECT TOP 10
  ProductID, ProductName, UnitPrice, UnitsInStock, UnitsOnOrder,
  UnitPrice * UnitsInStock AS CurrentStockValue,
  (UnitsInStock + UnitsOnOrder) * UnitPrice AS FutureStockValue FROM
dbo.Products
ORDER BY
  CurrentStockValue DESC
```

Module 5 – Grouping and Aggregating

Overview

In this lab, you will work with the dbo.Orders table to find out general information about the orders on the system.

In the second exercise, you will start to investigate individual customers' ordering habits.

In the third exercise you will work with the dbo.[Order Details] table to start drilling down into exactly what products were ordered by customers and filtering the results of aggregated queries.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- write a query that calculates single aggregates.
- write a query that calculates aggregates for groups of information.
- write a query that aggregates, groups and filters on calculated values.

Setup: Launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Administrator.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: Basic aggregates

You want to learn more about the data stored in the dbo.Orders table, specifically those placed by Nancy Davolio.

In this exercise, you will use basic aggregates.

The main tasks for this exercise are as follows:

1. Write a query that selects a count of the orders.
2. Modify the query so that it also calculates the earliest and latest dates on which orders were placed.
3. Modify the query to only calculate the values for Nancy's orders (employee id 1).

Task 1: write a query that selects a count of rows

1. Create a new query and save it with a name of 'OrderAnalysis.sql'.
2. Write a query that uses the Northwind database and displays a count of all the rows in the dbo.Orders table. Alias the count to call it 'NumberOfOrders'.
3. Execute the query and verify that the result set contains a single value of 830.

Task 2: write a query that selects maximums and minimums

1. Modify your existing query.
2. Add an aggregate to the select list that calculates the minimum OrderDate value. Alias it as 'EarliestOrder'.
3. Add another aggregate that calculates the maximum OrderDate value. Alias it as 'LatestOrder'.
4. Execute the query and verify that the answers are 830, 4th July 1996 and 6th May 1998.

Task 3: write a query that selects aggregates for only one employee

1. Modify your existing query.
2. Add a WHERE clause to your query so that it only includes rows with an EmployeeID equal to 1.
3. Execute the query and verify that the answers are now 123, 17th July 1996 and 6th May 1998.

Exercise 2: grouping aggregates

You have been asked to Write a query on the number of orders placed by each customer. The query is to be sorted by the number of orders placed, from highest to lowest.

In this exercise, you will use the GROUP BY clause and an ORDER BY.

The main tasks for this exercise are as follows:

1. Write a query that counts the number of OrderIDs in the dbo.Orders table.
2. Modify the query so that it includes the CustomerID and groups the results on that column.
3. Modify the query to sort in reverse order on the number of orders placed.

Task 1: create a query that counts orders

1. Create a new query and save it with a name of 'CustomerOrders.sql'.
2. Write a query that uses the Northwind database and displays a count of all the OrderIDs in the dbo.Orders table. Alias the aggregate as 'NumberOfOrders'.
3. Execute the query and verify that it returns a value of 830.

Task 2: write a query that groups orders based on the customer's ID

1. Modify the existing query.
2. Add the CustomerID column to the query's select list.

   NOTE: At this point, the query won't work.
3. Add a GROUP BY clause to the query that groups the results based on the CustomerID column.
4. Execute the query and verify that it returns 89 rows, the top one being ALFKI with a NumberOfOrders of 6.

Task 3: write a query that sorts order counts in descending order

1. Modify the existing query.
2. Add an ORDER BY clause to the query that sorts on the NumberOfOrders column in descending order.
3. Execute the query and verify that it returns 89 rows, the top one now being SAVEA with a count of 31.

Exercise 3: aggregating calculated values and filtering aggregates

Northwind Traders are trying to see exactly how much customers are spending on products.

In this exercise, you will use aggregate functions on calculated columns.

The main tasks for this exercise are as follows:

1. Write a query that sums the Quantity column of the dbo.[Order Details] table, grouped by product ids.
2. Modify the query so that it sums the Quantity times the UnitPrice and sorts the results in descending order of those values.
3. Modify the query to filter the results to only include those products with a total value of less than or equal to 5000.

Task 1: write a query that sums quantities of products sold

1. Create a new query and save it with a name of 'ProductSales.sql'.
2. Write a query that uses the Northwind database and selects the ProductID and the sum of the Quantity columns from the dbo.[Order Details] table.
   Alias the aggregate column as 'TotalSold'
3. Group the results on the ProductID column.
4. Execute the query and verify that it returns 77 rows, the first row being product id 23, with a TotalSold of 580.

Task 2: write a query that sums a calculation

1. Modify the existing query.
2. Modify the SUM aggregate so that it adds up the value of the Quantity column multiplied by the UnitPrice column for each product. Change the alias name to 'TotalValue'.
3. Sort the results on the TotalValue column, in descending order.
4. Execute the query and verify that the top-selling product is productid 38, with a total sales value of 149984.20.

Task 3: write a query that filters aggregate values

1. Modify the existing query.
2. Add a HAVING clause to the query to only return the rows with a TotalValue of less than or equal to 5000.
3. REMEMBER: just like with a WHERE clause, the actual column named TotalValue doesn't exist yet in your HAVING, so you'll need to re-use the calculation.
4. Execute the query and verify that you now see only 16 rows, the first of which is product 23 with a value of 4840.20.

Completed scripts

Exercise 1 task 1:

```
USE Northwind

SELECT COUNT(*) as NumberOfOrders
FROM dbo.Orders
```

Exercise 1 task 2:

```
SELECT
  COUNT(*) as NumberOfOrders,
  MIN(OrderDate) as EarliestOrder,
  MAX(OrderDate) as LatestOrder
FROM dbo.Orders
```

Exercise 1 task 3:

```
SELECT COUNT(*) as NumberOfOrders,
  MIN(OrderDate) as EarliestOrder, MAX(OrderDate) as LatestOrder
FROM dbo.Orders
WHERE EmployeeID = 1
```

Exercise 2 task 1:

```
USE Northwind

SELECT COUNT(OrderID) AS NumberOfOrders
FROM dbo.Orders
```

Exercise 2 task 2:

```
SELECT CustomerID, COUNT(OrderID) AS NumberOfOrders
FROM dbo.Orders
GROUP BY CustomerID
```

Exercise 2 task 3:

```
SELECT CustomerID, COUNT(OrderID) AS NumberOfOrders
FROM dbo.Orders
GROUP BY CustomerID
ORDER BY NumberOfOrders DESC
```

Exercise 3 task 1:

```
USE Northwind

SELECT ProductID, SUM(Quantity) AS TotalSold
FROM dbo.[Order Details]
GROUP BY ProductID
```

Exercise 3 task 2:

```
SELECT ProductID, SUM(Quantity * UnitPrice) AS TotalValue
FROM dbo.[Order Details]
GROUP BY ProductID
ORDER BY TotalValue DESC
```

Exercise 3 task 3:

```
SELECT ProductID, SUM(Quantity * UnitPrice) AS TotalValue
FROM dbo.[Order Details]
GROUP BY ProductID
HAVING SUM(Quantity * UnitPrice) <= 5000
ORDER BY TotalValue DESC
```

Module 6 – Using Multiple Tables

Overview

In this lab, you will work with the dbo.Orders, dbo.Customers, dbo.[Order Details] and dbo.Products tables to find out detailed information about the orders on the system.

In the second exercise, you will investigate individual customers' ordering patterns.

In the third exercise you will work with the dbo.Suppliers, dbo.Customers and dbo.Employees table to produce a company-wide telephone directory.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- write a query that uses an inner join to select rows from two tables.
- write a query that filters joined data.
- write a query that uses an inner join to select rows from more than two tables.
- write a query that investigates the difference between left and right outer joins.
- write a query that combines the results of multiple other queries.

Setup: launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Admin.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: join fundamentals

You want to write a query about UK-based customers and the orders that they have placed.

In this exercise, you will use inner joins, the fundamental type of join in SQL.

The main tasks for this exercise are as follows:

1. Write a query that selects the customer id, city and company and contact names from the Customers table and the order number and order date from the Orders table.
2. Modify the query so that it only picks up orders placed by UK Customers and sorts the results by customer and order date.
3. Modify the query so that it also includes details about the name of the product that was ordered, and how many units of each product were ordered.

Task 1: write a query that selects rows from the Customers and Orders tables

1. Create a new query and save it with a name of 'UKCustomerOrders.sql'.
2. Write a query that uses the Northwind database and selects the CustomerID, CompanyName, ContactName and City columns from the dbo.Customers table. Alias the table as 'c'.
3. Execute the query and verify that 91 rows are returned.
4. Modify the query to join rows from the dbo.Orders table that have the same value in the CustomerID column. Alias Orders as 'o'.
5. Add the OrderID and OrderDate columns from the Orders table to the select list.
6. Execute the query and verify that 830 rows are returned.

Task 2: write a query that filters and sorts the results

1. Modify your existing query.
2. Add a where clause to limit the rows returned to those Customers whose Country column is equal to UK.
3. Execute the query and verify that 56 rows are returned, all of whose City values are British cities.
4. Add an order by clause to ensure that results are sorted on CompanyName then OrderDate
5. Execute the query and verify that 56 rows are returned, the first one is for AROUT from November 1996 and the last one is for SEVES from February 1998.

Task 3: write a query that includes rows from more than two tables

1. Make a copy of your existing query.
2. Join the dbo.[Order Details] table aliased as 'od'. Use the OrderID column in Orders and Order Details to find matching rows.
3. Add the ProductID and Quantity columns to the select list. You can also remove the City column from the earlier queries – that was just to add a visual verification that you were only getting UK customers.
4. Execute the query and verify that 135 rows are returned – we're getting more rows back because we're selecting every row down at the Order Detail level now.
5. Join yet another table – the Products table. Alias it as 'p' and use the ProductID column from Order Details to find the matching rows. Include the ProductID and ProductName columns in your select list.
6. Execute the query and verify that the first order is for 25 lots of Guarana Fantastica and the last one is for 20 Scottish Longbreads.
7. (Extra credit) Limit the results to only include products in the Seafood category and display the category name as well. (18 rows)

   Hint: You will need to add the Categories table to your existing query to do this.

Exercise 2: investigate outer joins

In this exercise you will investigate left and right outer joins. You will also revisit aggregation from an earlier exercise.

The main tasks for this exercise are as follows:

1. Create a query that gets a count of Customers.
2. Create a query that selects the Company Name from Customers and the number of orders for those customers from the Orders table.
3. Modify the query to retrieve all customers, regardless of whether they have placed orders and add a column showing the minimum order date to the select list.

Task 1: create a query that counts customers

1. Create a new query and save it with a name of 'CustomerOrderAnalysis.sql'.
2. Write a query that uses the Northwind database and displays a count of all the rows in the dbo.Customers table.
3. Execute the query and make a note of the result.

Task 2: write a query that groups orders based on the customer's name

(Note: This is similar to a task in an earlier lab.)

1. Comment out or delete the Customer count query.
2. Write a new query that selects the CompanyName column from the Customers table and a count of OrderID columns from the Orders. Alias the count column as 'NumOrders'. You will need a group by clause.
3. Add an order by clause to sort the records in number of orders placed.
4. Execute the query and make a note of the number of rows returned. You will see that it differs from the number of customers. Take a few moments to think about why that might be.

   Hint: what is the lowest count of orders?

Task 3: write a query that uses left and right outer joins

1. Modify the existing query.
2. Change the query so that it selects all of the rows from the Customers table and those rows from the Orders table where there is a match.
3. Execute the query and verify that it now returns 91 rows, the top two now being FISSA and Paris Specialités, both with a count of 0.
4. Change the left outer join to a right outer join and confirm that you're back to only seeing 89 rows. In this case, we get the same results as for an inner join.
5. Change the order in which the tables are listed so that the right outer join returns 91 rows.
6. Finally, add another column to the query that selects the minimum order date from the Orders table and alias it as 'MinDate'.
7. Execute the query and note that we have a couple of null values for the inactive customers. The COUNT aggregate was able to come up with a total of zero for those customers but the MIN has no values to work on so it must return null. In an actual query, you might want to tidy that up so that it shows some text instead.

Exercise 3: create a telephone directory

Northwind Traders want to create a centralised telephone directory for all suppliers, customers, and employees.

In this exercise, you will use the UNION operator to combine results.

The main tasks for this exercise are as follows:

1. Write a query that retrieves the company name, contact name and telephone number of all customers.
2. In the same query, write a query that retrieves the same columns for all suppliers.
3. In the same query, write a query that retrieves the full name and extension number for all employees.
4. Combine the results using a UNION operator.

Task 1: write a query that retrieves customer contact details

1. Create a new query and save it with a name of 'ContactDirectory.sql'.
2. Write a query that uses the Northwind database and selects the CompanyName, ContactName and Phone columns from the dbo.Customers table.
3. Execute the query and verify that it returns 91 rows.

Task 2: Write a query that retrieves supplier contact details

1. In the same script file, add another query that selects the same three columns from the Suppliers table.
2. Select only that part of the script and execute the query. Verify that 29 rows are returned.

Task 3: write a query that retrieves employee contact details

1. In the same script file, add another query that selects rows from the Employees table.
2. In the select list for the query, add a column that concatenates the FirstName column to a space character and the LastName column (building up a string containing the employee's full name).
3. Add the Extension column as well.
4. Execute the query and verify that you retrieve nine rows of two columns – one containing the full name and one containing the extension.

Task 4: combine the results using the UNION operator

1. In the same script file, between the customers query and the suppliers query, add the UNION keyword.
2. Select both parts of the script and execute. Verify that you get 120 rows back. Scroll through the results and note that they are sorted alphabetically by company name.
3. Now add the UNION keyword between the suppliers query and the employees query and attempt to run the whole script.
4. You get an error, because the employees query only has two columns in it whilst the other two have three columns.
5. You need to add a company name column to the employee's part of the query. Now, the Employees table doesn't have a company name column but that's not a problem because we actually know what company all of the employees work for – Northwind!
6. At the start of the select list for employees, add the string 'Northwind Traders'.
7. Run the whole query again and verify that 129 rows are returned. Scroll through the list and note that the Northwind Traders employees are all floating around in the middle of the results.
8. Change the UNIONs to UNION ALL and rerun the query. This time all the employees are right at the very end. Also, although you probably won't have noticed, the query has run considerably quicker than before as it hasn't tried to remove duplicates by sorting first.
9. Feel free to add an order by clause if you want to.

Completed scripts

Exercise 1 task 1:

```
SELECT c.CustomerID, c.CompanyName, c.ContactName,
       c.City, o.OrderID, o.OrderDate
FROM   dbo.Customers AS c
JOIN   dbo.orders AS o
ON     c.CustomerID = o.CustomerID
```

Exercise 1 task 2:

```
SELECT   c.CustomerID, c.CompanyName, c.ContactName,
         c.City, o.OrderID, o.OrderDate
FROM     dbo.Customers AS c
JOIN     dbo.orders AS o
ON       c.CustomerID = o.CustomerID
WHERE    c.Country = 'UK'
ORDER BY c.CompanyName, o.OrderDate
```

Exercise 1 task 3:

```
SELECT   c.CustomerID, c.CompanyName, c.ContactName,
         o.OrderID, o.OrderDate, od.ProductID,
         p.ProductName, od.Quantity
FROM     dbo.Customers AS c
JOIN     dbo.orders AS o
ON       c.CustomerID = o.CustomerID
JOIN     dbo.[Order Details] AS od
ON       o.OrderID = od.OrderID
JOIN     dbo.Products AS p
ON       od.ProductID = p.ProductID WHERE
c.Country = 'UK'
ORDER BY c.CompanyName, o.OrderDate
```

Exercise 2 task 1:

```
USE Northwind

SELECT COUNT(*) FROM dbo.Customers
```

Exercise 2 task 2:

```
SELECT   c.CompanyName, COUNT(o.OrderID) AS NumOrders
FROM     dbo.Customers AS c
JOIN     dbo.Orders AS o
ON     o.CustomerID = c.CustomerID
GROUP BY c.CompanyName
ORDER BY NumOrders
```

Exercise 2 task 3:

```
 SELECT
      c.CompanyName, COUNT(o.OrderID) AS NumOrders,
      MIN(o.OrderDate) AS MinDate FROM
      dbo.Orders AS o
RIGHT OUTER JOIN
      dbo.Customers AS c
ON
      o.CustomerID = c.CustomerID
GROUP BY
      c.CompanyName
ORDER BY
      NumOrders
```

Exercise 3 task 1:

```
USE Northwind

SELECT CompanyName,
FROM dbo.Customers
                      ContactName, Phone
```

Exercise 3 task 2:

```
                      ContactName, Phone
SELECT CompanyName,
FROM dbo.Suppliers
```

Exercise 3 task 3:

```
                                       Extension
SELECT FirstName + ' ' + LastName,
FROM   dbo.Employees
```

Exercise 3 task 4:

```
SELECT CompanyName, ContactName, Phone
FROM   dbo.Customers

UNION ALL

SELECT CompanyName, ContactName, Phone
FROM   dbo.Suppliers

UNION ALL

SELECT 'Northwind Traders',
       FirstName + ' ' + LastName,    Extension
FROM   dbo.Employees
```

Module 7 – Common Functions

Overview

In this lab, you will work with existing queries that might be familiar from earlier labs and enhance them with the use of some common functions.

In the first exercise, you will include some string manipulation in the Contact Directory to enable you to sort it by the contact's last name.

In the second exercise you will use some date functions to work with orders from a given year.

In the third exercise you will add some default text to an outer-joined query when there are no matches on the right-hand side.

Finally, you will add some date formatting to the same query.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- use string functions.
- use date functions.
- use null functions.
- format dates.

Setup: launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Admin.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: string functions

You want to modify the ContactDirectory query so that you can sort it based on the contact's last name. For employees, this is not a problem as their last names and first names are stored in two separate columns, but for customers and suppliers, you need to start manipulating strings.

The main tasks for this exercise are as follows:

1. Add a column to an existing query that calculates the number of characters in the contact name's first name (by finding the first space).
2. Use that calculation (twice) to get the first name and the last name out of the query. Repeat for the suppliers table and modify the employees query so that it uses separate first and last names.
3. Sort the union query by last name.

Task 1: add a column to an existing query

1. Open the script file called Ex01.sql from the C:\Coursefiles\QATSQL\CommonFunctions\Begin folder. Save it as 'ImprovedDirectory.sql'.
2. Just focusing on the Customers part of the query for now (in fact you might want to comment the rest out), add another column to the select list that uses the CHARINDEX function to find the first occurrence of a space character (' ') in the ContactName column.

   Hint: Remember the syntax of the CHARINDEX function is as follows:
   CHARINDEX('what you are searching for', string you are looking for it in).
3. Run that part of the query and confirm that that calculation is returning the right numbers (it will be one more than the length of the contact's first name).

Task 2: calculate the first and last names based on the length of the first name.

1. Add a column that uses the LEFT function, passing it the contactname and (a copy of) the calculation from task 1 to retrieve the contact's first name only.

   HINT: Remember the syntax of the LEFT function is as follows: LEFT(string, how many characters you wish to return from the left of the string).

   Hint: In the previous task you used the CHARINDEX command to locate the position of the space in each ContactName.

2. Alias the column as 'FirstName'.

3. Add another column that uses the SUBSTRING function, passing it the contactname, (a copy of) the calculation from task 1, and the value 50, to retrieve the contact's last name only. The value 50 just says 'take all of the rest of the string'. You might want to add 1 to the value returned by the CHARINDEX function to remove a leading space, but it's not going to stop things from working.

   HINT: Remember the syntax of the SUBSTRING function is as follows: SUBSTRING(string, position to start at, number of characters to return).

   HINT: In Task 1 you used the CHARINDEX command to locate the position of the space in each ContactName.

4. Alias the new column as 'LastName'.

5. Ensure that the Customers query only has four columns – CompanyName, FirstName, LastName and Phone and duplicate it for the Suppliers table.

6. Finally modify the Employees part of the query so that it selects first and last names as two separate columns.

7. Run the whole query and confirm that it returns 129 rows, with 4 columns.

Task 3: Sort the results on last name

1. Add an order by clause that sorts the results of the union on last name.

2. Execute and confirm that Mr. Acconti is top of list and Mr. Yorres is last.

Exercise 2: work with date functions

In an earlier lab, you may have created a query to get the count of orders for customers using outer joins. The rest of this lab will tweak the output from that lab.

The main tasks for this exercise are as follows:

1. Modify an existing query so that it only retrieves information about orders placed in a specified year.

Task 1: create a query that filters by year

1. Open the script file called Ex02.sql from the C:\Coursefiles\QATSQL\CommonFunctions\Begin folder and save it as 'ImprovedOrderAnalysis.sql'.
2. Add a where clause that uses the YEAR function to select only those rows whose order date was placed in 1996.
3. Execute the query and verify that it returns 67 rows. You might be expecting it to return 91 rows because of the left join but the WHERE clause is saying 'if they've never placed orders, I'm not interested!' **Exercise 3: work with ISNULL and CONVERT**

The main tasks for this exercise are as follows:

1. Modify an existing query so that it outputs an informational message rather than the word 'null'.
2. Modify the same query so that it formats the dates.

Task 1: create a query that outputs an informational message

1. Open the script file called Ex03.sql from the C:\Coursefiles\QATSQL\CommonFunctions\Begin folder and save it as 'FormattedOrderAnalysis.sql'.
2. Wrap the MIN aggregate call in an ISNULL function call that returns the text 'None placed' when the value is null.
3. Attempt to execute the query and note that it fails – you are currently trying to output text in a datetime column.

Task 2: use the convert function to format a datetime column

1. Work with the existing query.
2. Wrap the MIN aggregate call in a CONVERT function call that converts the OrderDate column into a VARCHAR(20) using format code 106 (or pick another style). So, the outer-most function should be ISNULL, the
   CONVERT, then MIN right in the middle). It might take some fiddling to get the brackets all to match up.
3. Execute the query and bask in the beauty of your formatted query that returns 91 rows.

Completed scripts

Exercise 1 task 1:

```
SELECT
      CompanyName,
      CHARINDEX(' ', ContactName),
      ContactName,
      Phone
FROM
      dbo.Customers
```

Exercise 1 task 2:

```
SELECT CompanyName,
       LEFT(ContactName, CHARINDEX(' ', ContactName))
         AS FirstName,
       SUBSTRING(ContactName,
         CHARINDEX(' ', ContactName) + 1, 50)
         AS LastName,
        Phone
FROM   dbo.Customers
UNION ALL
SELECT CompanyName,
       LEFT(ContactName, CHARINDEX(' ', ContactName))
         AS FirstName,
       SUBSTRING(ContactName,
         CHARINDEX(' ', ContactName) + 1, 50)
         AS LastName,
        Phone
FROM   dbo.Suppliers
UNION ALL
SELECT 'Northwind Traders', FirstName,
       LastName, Extension
FROM   dbo.Employees
```

Exercise 1 task 3:

```
…
ORDER BY LastName
```

Exercise 2 task 1:

```
SELECT  c.CompanyName, COUNT(o.OrderID) AS NumOrders,
        MIN(o.OrderDate) AS MinDate
FROM      dbo.Customers AS c
LEFT JOIN dbo.Orders AS o
ON        o.CustomerID = c.CustomerID
WHERE     YEAR(o.OrderDate) = 1996
GROUP BY  c.CompanyName
ORDER BY  NumOrders
```

Exercise 3 task 1:

```
SELECT  c.CompanyName, COUNT(o.OrderID) AS NumOrders,
       ISNULL(MIN(o.OrderDate), 'None placed') AS MinDate
FROM      dbo.Customers AS c
LEFT JOIN dbo.Orders AS o
ON        o.CustomerID = c.CustomerID
GROUP BY  c.CompanyName
ORDER BY  NumOrders
```

Exercise 3 task 2:

```
SELECT  c.CompanyName, COUNT(o.OrderID) AS NumOrders,
       ISNULL(
         CONVERT(
           VARCHAR(20),
           MIN(o.OrderDate),
           106),
         'None placed') AS MinDate
FROM      dbo.Customers AS c
LEFT JOIN dbo.Orders AS o
ON        o.CustomerID = c.CustomerID
GROUP BY  c.CompanyName
ORDER BY  NumOrders
```

Module 8 – Views and Stored Procedures

Overview

In this lab, you will use an existing view to see just how much it simplifies a complex query.

In the second exercise, you will create your own view.

In the third exercise you will use existing stored procedures to retrieve data.

At the end of this chapter is a 'completed scripts' section which tells you what to type at each stage if you are really stuck.

Objectives

At the end of this lab, you will be able to:

- use a predefined view.
- write and use a simple view.
- execute predefined stored procedures.

Setup: launch SQL Server Management Studio (if necessary)

1. Launch the virtual machine.
2. Log on as Student.
3. Launch SQL Server Management Studio.
4. Connect to the server.

Exercise 1: use a predefined view

You want to write a query that retrieves full information about orders, including the name of the employee who placed the order, the name of the product that was ordered, details about the company name the order was for, which address the order was to be shipped to, calculations about the order total after applying discounts and much more. You will use a view to do this.

The main tasks for this exercise are as follows:

1. Examine a prewritten query and note its complexity.
2. Use a view that retrieves the same data.

Task 1: examine a complex query

1. Open the script file called 'Ex01Task01.sql' from the C:\Coursefiles\QATSQL\ViewsAndStoredProcedures\End folder.
2. Execute the query and verify that 2155 rows are returned.
3. Add a where clause to filter the results only for UK customers (it's the Country column within the Customers – it's in there somewhere!). You'll need to ensure that you use the customers table's alias because there are at least two Country columns in the tables the query uses.
4. Confirm that 135 rows are returned.
5. Now imagine having to rewrite that query every time you wanted full details about an order or filter or sort it in a different way!

Task 2: use a view

1. Create a new query called 'UseView.sql'.
2. Write a select statement that retrieves every row and column from the dbo.Invoices table.
3. Execute the query and verify that 2155 rows are returned – the same as the previous complex query. The Invoices 'table' is actually a view, which is a stored version of the query you just looked at.
4. Add a where clause to filter the results only for UK customers. You don't need to worry about which table it's coming from this time, because as far as the view is concerned, there's only the one Country column.
5. Experiment with sorting and filtering options and only selecting some of the columns.

Exercise 2: Create a telephone directory view

In an earlier lab, you may have created a query to generate a combined contact directory for Northwind Traders. In this exercise, rather than having it stored in a script file, you will store it in the database as a view.

The main tasks for this exercise are as follows:

1. Create (or reuse) a query that retrieves the company name, contact name and telephone number of all customers, suppliers, and employees.
2. Store that query as a view in the database.
3. Use the view in another query.

Task 1: write a query that retrieves Customer contact details

1. Create a new query and save it with a name of 'DirectoryView.sql'.
2. If you completed an earlier lab exercise on UNIONs, copy the directory query into your new script. If not, copy the content of Ex02Starter.txt from the C:\Coursefiles\QATSQL\ViewsAndStoredProcedures\Begin folder.
3. Execute the query and verify that it returns 129 rows.

Task 2: create a new view in the database

1. Immediately before the first SELECT statement of the contact directory query, add the appropriate commands to create a new view called 'dbo.ContactDirectory'. If you have a 'USE Northwind' at the start of your script, you might need to put a 'GO' statement between that and your create statement.
2. Run the script and confirm that you get a message saying 'command(s) completed successfully'.

Task 3: write a query that uses your new view

1. Create a new script file called 'UseMyView.sql'.
2. Write a query that selects all columns from the ContactDirectory view. Don't worry about any red-squiggly lines – that just means that SSMS hasn't caught up with the fact that there's a new view. Make sure it retrieves 129 rows. If you get an error message saying 'invalid object name' then you might have a problem…
3. Add a where clause to only select those contacts with a contact name that starts with the letter 'a'. You'll need to use a like operator.
4. Execute the query and verify that 14 contacts are returned, two of which are Northwind employees.

Exercise 3: use existing stored procedures to retrieve rows

Northwind Traders already have a number of stored procedures defined. In this exercise you will get some practice executing them

The main tasks for this exercise are as follows:

1. Execute a stored procedure.
2. Execute a stored procedure using named parameters.

Task 1: execute the CustOrderHist stored procedure

1. Create a new query and save it with a name of 'OrdHist.sql'.
2. Execute the dbo.CustOrderHist stored procedure, passing it a parameter of 'ALFKI'
3. Verify that 11 rows are returned.
4. Try it with a couple of different CustomerID parameters to make sure that it returns different values for different customers.

Task 2: execute the SalesByCategory stored procedure

1. Create a new query and save it with a name of 'CatSales.sql'.
2. Execute the dbo.SalesByCategory stored procedure 3 times, passing it a category name of 'Seafood' each time and a year of 1996, 1997 then 1998.
3. Verify that 12 rows are returned each time, but the Total Purchase values change. This procedure calculates total sales by category by year (so perhaps it's badly named…)
4. Remove the second parameter and execute the proc again. It still works and retrieves the sales from 1998 as a default.
5. Remove the category name parameter and execute. It fails.
6. Let's try it another way – year first then category. It doesn't fail but doesn't give us back any meaningful results. It is actually looking for a category called '199x', which doesn't exist.
7. One final experiment. Use named parameters. Leave them the 'wrong way round' but before the year, type '@OrdYear = '; and before the category, type '@CategoryName = '. It should work correctly again.

Completed scripts

Exercise 1 task 1:

```
… WHERE c.Country = 'UK'
```

Exercise 1 task 2:

```
SELECT * FROM dbo.Invoices
WHERE Country = 'UK'
```

Exercise 2 task 2:

```
CREATE VIEW dbo.ContactDirectory
AS

SELECT CompanyName, ContactName, Phone
FROM   dbo.Customers
UNION ALL
SELECT CompanyName, ContactName, Phone
FROM   dbo.Suppliers
UNION ALL
SELECT 'Northwind Traders',
       FirstName + ' ' + LastName,     Extension
FROM   dbo.Employees
```

Exercise 2 task 3:

```
SELECT * FROM dbo.ContactDirectory
WHERE ContactName LIKE 'A%'
```

Exercise 3 task 1:

```
EXEC dbo.CustOrderHist ‘ALFKI’
```

Exercise 3 task 2:

```
EXEC dbo.SalesByCategory
     @OrdYear = 1997, @CategoryName = 'Seafood'
```

Module 9 – Table Expressions

Overview

The main purpose of this lab is to familiarise yourself with key features of table expressions with TSQL:

- Views.
- In-line table valued functions.
- Derived tables.
- Common table expressions.
- Temporary tables.

Objectives

At the end of this lab, you will be able to:

- create a new view in SQL.
- create a new in-line table valued function with and without parameters in SQL.
- use a derived table in SQL.
- create a common table expression.
- create temporary tables and check their scope.

Setup: launch SQL Server Management Studio

1. On the Start menu, select All Programs, and then select SQL Server Management Studio.
2. When Microsoft SQL Server Management Studio window opens, the Connect to Server dialog box will appear.
3. In the Connect to Server dialog box, select Connect to accept the default settings.
4. On the toolbar, select New Query, and either select the QATSQLPLUS database in the Available Databases box, or type USE QATSQLPLUS in the query window.
5. If you wish, you may save your queries to My Documents or the desktop. All modules are separate, and you will not require any queries from this module in any later module.

Exercise 1: views

In this exercise, you will create a view to return all the courses available with the vendor name.

The main tasks for this exercise are as follows:

1. Create a query that returns the correct information.
2. Convert the query to a view called CourseList.
3. Use the CourseList view.

Task 1: create the query

1. Write a query to join the Course and Vendor tables. The column shared by the tables is VendorID. The columns returned should be:
    - CourseName
    - CourseID ▪ VendorName
2. Test the query.
3. Keep the query window open for the following tasks.

Task 2: create a view

1. Using the query from Task 1, create a view called dbo.CourseList.
2. Test the view dbo.CourseList using a SELECT query.
3. Keep the query window open for the following tasks.

Exercise 2: in-line table-valued function (TVF)

In this exercise, you will create two inline TVF to return the number of courses attended and number of days training received per delegate. The first should return the information for all delegates, and the second should return the information for a specific DelegateID.

The main tasks for this exercise are as follows:

1. Create a query that returns the correct information.
2. Convert the query to a TVF called udf_DelegateDays. Test the function udf_DelegateDays.
3. Create a second TVF called udf_IndividualDelegateDays to use the same basic query but only return the result for a specific DelegateID.

Task 1: create the query

1. Write a query to join the Delegate, DelegateAttendance, and CourseRun tables. The columns returned should be:
    - DelegateID
    - Sum(DurationDays) → DelegateDays
    - Count(*) → DelegateCourses
2. Test the query.
3. Keep the query window open for the following tasks.

Task 2: create a TVF

1. Using the query from task 1, create a TVF called udf_DelegateDays. As this will be an in-line table valued function the return will be type TABLE. No parameters are to be passed to this function.
2. Test the TVF using a SELECT query.
3. Keep the query window open for the following tasks.

Task 3: create a TVF

1. Using the query from task 1, create a TVF called udf_IndividualDelegateDays. As this will be an in-line table valued function the return will be type TABLE. A parameter called @DelegateID (data type INT) should be passed to this function.
2. Test the TVF using a SELECT query passing the DelegateID 1.
3. Keep the query window open for the following tasks.

Exercise 3: derived tables

In this exercise, you will write a query that returns all delegates that have been taught by Jason Bourne.

The main tasks for this exercise are as follows:

1. Create a query that returns the CourseRunID and StartDate for each course taught by Jason Bourne.
2. Incorporate the query from task 1 as a derived table and return the delegate information.

Task 1: create the derived table query

1. Write a query to join the Trainer and CourseRun tables. The only columns returned should be the CourseRunID and StartDate where the trainer name was Jason Bourne.
2. Test the query and ensure 2 rows are returned.
3. Keep the query window open for the following tasks.

Task 2: create the full query

1. Write a query to join the DelegateAttendance, Delegate and derived table from task 1. The columns returned from the query should be:
    - DelegateID
    - DelegateName
    - CompanyName
    - StartDate
2. Test the query and ensure 19 rows are returned.
3. Keep the query window open for the following tasks.

Exercise 4: temporary tables

In this exercise, you will write a query that returns and stores all Microsoft courses into temporary tables and view the scopes of the temporary table types.

The main tasks for this exercise are as follows:

1. Create a query that returns all Microsoft courses.
2. Create and insert the data into both a local and global temporary table.
3. Check the availability of the temporary tables in a different session.

Task 1: create the Microsoft course query

1. Write a query to select all columns from the dbo.Course table where the VendorID = 2.
2. Run the query and ensure 6 rows are returned.
3. Keep the query window open for the following tasks.

Task 2: create and use temporary tables

1. Using the query from task 1, select the query result into #MicrosoftLocal and ##MicrosoftGlobal. The SELECT INTO code takes the form:

    - SELECT <columns>
    - INTO <destination table name>
    - FROM <source table(s)>

2. In the same query window, write a query to view the content of both #MicrosoftLocal and ##MicrosoftGlobal.
3. Run the queries and ensure six rows are returned from each.
4. Keep the query window open for the following tasks.

Task 3: review availability of the temporary table to other sessions

1. On the toolbar, select New Query.
2. In the new query window, write a query to view the content of both #MicrosoftLocal and ##MicrosoftGlobal.
3. Run the queries separately. The global temporary table (##MicrosoftGlobal) should return six rows. The local temporary table will return an error.
4. Keep the query window open for the following tasks.

Answers

The answers below are for example only. Coding style and order of columns should not matter.

Task	Code

Ex 1 Task 1	`SELECT V.VendorName, C.CourseName, C.CourseID`
	` FROM dbo.Vendor AS V`
	`  INNER JOIN dbo.Course AS C`
	`  ON V.VendorID = C.VendorID`

Ex 1 Task 2	`CREATE VIEW dbo.CourseList AS`
	` SELECT V.VendorName, C.CourseName, C.CourseID`
	`  FROM dbo.Vendor AS V`
	`  INNER JOIN dbo.Course AS C`
	`  ON V.VendorID = C.VendorID`
	`GO`
	`SELECT * FROM dbo.CourseList`

Ex 2 Task 1	`SELECT D.DelegateID,`
	` SUM(CR.DurationDays) AS DelegateDays,`
	` COUNT(*) AS DelegateCourses`
	` FROM dbo.Delegate AS D`
	`  INNER JOIN dbo.DelegateAttendance AS DA`
	`  ON D.DelegateID = DA.DelegateID`
	`  INNER JOIN dbo.CourseRun AS CR`
	`  ON CR.CourseRunID = DA.CourseRunID`
	` GROUP BY D.DelegateID`

Ex 2 Task 2	`CREATE FUNCTION udf_DelegateDays()`
	` RETURNS TABLE`
	` AS`
	` RETURN (`
	`  SELECT D.DelegateID, SUM(CR.DurationDays) AS DelegateDays,`
	`  COUNT(*) AS DelegateCourses`
	`  FROM dbo.Delegate AS D`
	`  INNER JOIN dbo.DelegateAttendance AS DA`
	`  ON D.DelegateID = DA.DelegateID`
	`  INNER JOIN dbo.CourseRun AS CR`
	`  ON CR.CourseRunID = DA.CourseRunID`
	`  GROUP BY D.DelegateID`
	` )`
	`GO`
	`SELECT * FROM dbo.udf_DelegateDays()`

Ex 2 Task 3	

```
CREATE FUNCTION dbo.udf_IndividualDelegateDays(@DelegateID INT)
RETURNS TABLE
 AS
 RETURN(
  SELECT @DelegateID AS DelegateID,
    SUM(CR.DurationDays) AS DelegateDays,
    COUNT(*) AS DelegateCourses
   FROM dbo.Delegate AS D
    INNER JOIN dbo.DelegateAttendance AS DA
     ON D.DelegateID = DA.DelegateID     INNER
JOIN dbo.CourseRun AS CR
     ON CR.CourseRunID = DA.CourseRunID
   WHERE D.DelegateID = @DelegateID
 )
GO
SELECT * FROM dbo.udf_IndividualDelegateDays(1)
```

Task	Code
Ex 3 Task 1	

```
SELECT CourseRunID, StartDate
 FROM dbo.Trainer AS T
  INNER JOIN dbo.CourseRun AS CR
   ON T.TrainerID = CR.TrainerID
 WHERE TrainerName = 'Jason Bourne'
```

Ex 3 Task 2	

```
SELECT D.DelegateID, D.DelegateName, D.CompanyName, JB.StartDate
 FROM dbo.Delegate AS D
  INNER JOIN dbo.DelegateAttendance AS DA
   ON D.DelegateID = DA.DelegateID
  INNER JOIN (
   SELECT CourseRunID, StartDate
    FROM dbo.Trainer AS T
     INNER JOIN dbo.CourseRun AS CR
      ON T.TrainerID = CR.TrainerID
    WHERE TrainerName = 'Jason Bourne'
   ) AS JB
   ON DA.CourseRunID = JB.CourseRunID
```

Task	Code

Ex 4 Task 1	`SELECT *`
	`  FROM dbo.Course`
	`  WHERE VendorID = 2`

Ex 4 Task 2	`SELECT * INTO #MicrosoftLocal`
	`  FROM dbo.Course WHERE VendorID = 2`
	`SELECT * INTO ##MicrosoftGlobal`
	`  FROM dbo.Course WHERE VendorID = 2`
	`GO`
	`SELECT *`
	`  FROM #MicrosoftLocal`
	
	`SELECT *`
	`  FROM ##MicrosoftGlobal`

Module 10 – Advanced grouping

Overview

The main purpose of this lab is to familiarise yourself with how to use pivot and the advanced grouping clauses within TSQL:

- Aggregations using ROLLUP and GROUPING SETS.

Objectives

At the end of this lab, you will be able to:

- use advanced grouping techniques like ROLLUP

Setup: launch SQL Server Management Studio

1. On the Start menu, select All Programs, and then select SQL Server Management Studio.
2. The Microsoft SQL Server Management Studio window opens, and then the Connect to Server dialog box will appear.
3. In the Connect to Server dialog box, select Connect to accept the default settings.
4. On the toolbar, select New Query, and either select the QATSQLPLUS database in the Available Databases box, or type USE QATSQLPLUS in the query window.
5. If you wish, you may save your queries to My Documents or the desktop. All modules are separate, and you will not require any queries from this module in any later module.

Exercise 1: attendance by date and vendor

In this exercise, you will create a query to return a matrix-like result using the VendorCourseDateDelegateCount view.

The main tasks for this exercise are as follows:

1. Create a query to review the content of the dbo.VendorCourseDateDelegateCount view.

Task 1: review table content

1. Write a query to return the dbo.VendorCourseDateDelegateCount view.
   The columns returned should be:
   - VendorName
   - CourseName
   - StartDate
   - NumberDelegates
2. Test the query. The query should return 9 rows.
3. Keep the query window open for the following tasks.

Exercise 2: custom subtotals

In this exercise, you find produced a summary of number of delegates with custom subtotals.

The main tasks for this exercise are as follows:

1. Create a query that returns list of events with a subtotal for VendorName, CourseName and StartDate.
2. Create a query that returns list of events with a subtotal for only VendorName, and CourseName.

Task 1: using ROLLUP

1. Write a query to return all columns from the dbo.VendorCourseDateDelegateCount view. The columns returned should be:
    - VendorName
    - CourseName
    - StartDate
    - NumberDelegates
2. Test the query. The query should return nine rows.
3. Change the query so that the NumberDelegates is aggregated using the SUM function, keeping the other columns in the SELECT list. Add a GROUP BY clause as appropriate.
4. Test the query. The query should still return 9 rows.
5. Change the query to add the WITH ROLLUP clause. Unlike most clauses in TSQL, the order of columns listed in the GROUP BY does matter when using ROLLUP.
6. Test the query. The query should return 20 rows. If the number of rows is not 20 then recheck the order of the columns in the GROUP BY clause. Extra rows will be added to summarise the combinations of:
    - VendorName, CourseName, StartDate
    - VendorName, CourseName
    - VendorName • Total

	Vendorname	CourseName	StartDate	TotalDelegates
1	Microsoft	Administering a SQL Database Infrastructure (2016)	2016-10-24	17
2	Microsoft	Administering a SQL Database Infrastructure (2016)	NULL	17
3	Microsoft	Developing SQL Data Models (2016)	2016-10-31	7
4	Microsoft	Developing SQL Data Models (2016)	NULL	7
5	Microsoft	Querying data with Transact-SQL (2016)	2016-10-10	2
6	Microsoft	Querying data with Transact-SQL (2016)	NULL	2
7	Microsoft	NULL	NULL	26
8	Oracle	mySQL for Developers	2016-10-03	2
9	Oracle	mySQL for Developers	NULL	2
10	Oracle	NULL	NULL	2
11	QA	QASQLRB08	2016-10-26	3
12	QA	QASQLRB08	NULL	3
13	QA	QATSQL	2016-10-03	8
14	QA	QATSQL	2016-10-13	3
15	QA	QATSQL	NULL	11
16	QA	QATSQLPLUS	2016-10-17	5
17	QA	QATSQLPLUS	2016-10-24	2
18	QA	QATSQLPLUS	NULL	7
19	QA	NULL	NULL	21
20	NULL	NULL	NULL	49

7. In the picture above the rows 2, 4, 6, 9, 12, 15, and 18 show the total for VendorName and CourseName regardless of the StartDate. Row 20 shows the grand total.
8. Keep the query window open for the following tasks.

Answers

The answers below are for example only. Coding style and order of columns should not matter.

Task	Code

Ex 1 Task 1	`--Task 1:`
	`SELECT *`
	`FROM dbo.VendorCourseDateDelegateCount`

Ex 2 Task 1	`--Task 1a:`
	`SELECT Vendorname, CourseName, StartDate, NumberDelegates`
	`FROM dbo.VendorCourseDateDelegateCount`
	
	`--Task 1b:`
	`SELECT Vendorname, CourseName, StartDate,`
	`SUM(NumberDelegates) AS TotalDelegates`
	`FROM dbo.VendorCourseDateDelegateCount`
	`GROUP BY Vendorname, CourseName, StartDate`
	`GO`
	
	`--Task 1c:`
	`SELECT Vendorname, CourseName, StartDate,`
	`SUM(NumberDelegates) AS TotalDelegates`
	`FROM dbo.VendorCourseDateDelegateCount`
	`GROUP BY Vendorname, CourseName, StartDate`
	`WITH ROLLUP`
	`GO`

Module 11 – Transactions

Overview

The main purpose of this lab is to familiarise yourself with transactions within Microsoft SQL Server:

- Review the original code and identify possible issues.
- Throw errors instead of printing an error message.
- Catch an error from within code.

Objectives

At the end of this lab, you will be able to:

- declare an error within the sys.messages system table.
- raise a previously declared error.
- throw an error.
- produce code that catches errors and deals with the error efficiently.

Setup: launch SQL Server Management Studio

1. On the Start menu, select All Programs, and then select SQL Server Management Studio.
2. The Microsoft SQL Server Management Studio window opens, and then the Connect to Server dialog box will appear.
3. In the Connect to Server dialog box, select Connect to accept the default settings.
4. On the toolbar, select New Query, and either select the QATSQLPLUS database in the Available Databases box, or type USE QATSQLPLUS in the query window.
5. If you wish, you may save your queries to My Documents or the desktop. All modules are separate, and you will not require any queries from this module in any later module.

Exercise 1: identifying possible issues

In this exercise, you will review the code and table constraints to determine possible issues that may arise.

The main tasks for this exercise are as follows:

1. Review the code and data constraints.
2. Identify possible issues.

Task 1: design and code

1. The two tables used in this exercise are:
   - dbo.BookTransfers
     - ProductID int not null
     - TransferDate datetime not null
     - TransferReason varchar(30) not null
     - TransferAmount int not null
   - dbo.BookStock
     - ProductID not null
     - StockAmount int not null must be >= 0
2. The original code:

```
INSERT INTO dbo.BookTransfers VALUES (@ProductID,getdate(),'Transfer Out',@Amount)
UPDATE dbo.BookStock
    SET StockAmount = StockAmount - @Amount
    WHERE ProductID = @ProductID
```

3. What issues may arise from the code given the table designs?

Task 2: alter the code to stop issues

1. Open the start code from 'C:\Coursefiles\QATSQLPLUS\M07 E01 Transactions.sql'.
2. Update the code to:
   - Check and reject NULL variable values
   - Start a TRY block
   - Start a transaction within the TRY block
   - Perform the original insert and update
   - Commit the transaction
   - End the TRY block
   - Start a CATCH block
   - Rollback the transaction
   - THROW an error
   - End the CATCH block
3. Test the query with different parameters:
   - @ProductID = 1, @Amount = 3000
     - The result should be that an error will be thrown
   - @ProductID = 1, @Amount = 1
     - The result should be that an error will be thrown
   - @ProductID = NULL, @Amount = 3
     - The result should be that an error will be thrown
   - @ProductID = 4, @Amount = 1
     - 2 '(1 row(s) affected)' messages
4. The query can be saved if you want.

Answers

The answers below are for example only. Coding style and order of columns should not matter.

Task	Code
Ex 1 Task 1	Possible issues: • All columns do not allow NULLS, so any variables should be checked for NULL • The StockAmount value must be greater or equal to zero, so during the UPDATE the calculation StockAmount - @Amount may cause a negative • Insert may succeed and the Update fail, leading to unbalanced stock figures

Ex 1 Task 2	`-- TASK 2:`
	`DECLARE @ProductID INT = 1`
	`DECLARE @Amount INT = 20`
	
	`IF (@ProductID IS NULL OR @Amount IS NULL)`
	`BEGIN;`
	`THROW 59999, 'Neither variable is allowed to be NULL',1`
	`RETURN`
	`END`
	
	`BEGIN TRY`
	`BEGIN TRAN`
	`INSERT INTO dbo.BookTransfers VALUES`
	`(@ProductID,getdate(),'Transfer Out',-@Amount)`
	`UPDATE dbo.BookStock`
	`SET StockAmount = StockAmount - @Amount`
	`WHERE ProductID = @ProductID`
	`COMMIT TRAN`
	`END TRY`
	`BEGIN CATCH`
	`ROLLBACK TRAN;`
	`THROW 59999,'An error occurred in the transaction. Everything rolled back',1`
	`END CATCH`
	`GO`