Contents

1 INF99X: Sample Course **3**
- 1.1 What are we doing? . . . 3
- 1.2 How should I use these files relative to the released MOC files? . . . 3
- 1.3 What about changes to the student handbook? . . . 3
- 1.4 How do I contribute? . . . 3
- 1.5 Notes . . . 4
  - 1.5.1 Classroom Materials . . . 4
- 1.6 It is strongly recommended that MCTs and Partners access these materials and in turn, provide them separately to students. Pointing students directly to GitHub to access Lab steps as part of an ongoing class will require them to access yet another UI as part of the course, contributing to a confusing experience for the student. An explanation to the student regarding why they are receiving separate Lab instructions can highlight the nature of an always-changing cloud-based interface and platform. Microsoft Learning support for accessing files on GitHub and support for navigation of the GitHub site is limited to MCTs teaching this course only. . . . 4
- 1.7 title: Online Hosted Instructions permalink: index.html layout: home . . . 4

2 Content Directory **4**
- 2.1 Labs . . . 4
- 2.2 Demos . . . 4
- 2.3 {% assign demos = site.pages | where_exp:"page", "page.url contains '/Instructions/Demos'" %} | Module | Demo | | --- | --- | {% for activity in demos %}| {{ activity.demo.module }} | [{{ activity.demo.title }}](/home/ll/Azure_clone/Azure_new/dp-080-Transact-SQL/{{ site.github.url }}{{ activity.url }}) | {% endfor %} . . . 4
- 2.4 demo: title: 'Module 1 Demonstrations' module: 'Module 1: Getting Started with Transact-SQL' 4

3 Module 1 Demonstrations **4**
- 3.1 Explore the lab environment . . . 4
- 3.2 Run basic SELECT queries . . . 5
- 3.3 demo: title: 'Module 2 Demonstrations' module: 'Module 2: Sorting and Filtering Query Results' 5

4 Module 2 Demonstrations **5**
- 4.1 demo: title: 'Module 3 Demonstrations' module: 'Module 3: Using Joins and Subqueries' . . . 5

5 Module 3 Demonstrations **5**
- 5.1 demo: title: 'Module 4 Demonstrations' module: 'Module 4: Using Built-in Functions' . . . 6

6 Module 4 Demonstrations **6**
- 6.1 demo: title: 'Module 5 Demonstrations' module: 'Module 5: Modifying Data' . . . 6

7 Module 5 Demonstrations **6**
- 7.1 lab: title: 'Lab Environment Setup' module: 'Setup' . . . 6

8 Lab Environment Setup **6**
- 8.1 Base Operating System . . . 6
- 8.2 Microsoft SQL Server Express 2019 . . . 6
- 8.3 Microsoft Azure Data Studio . . . 7
- 8.4 AdventureWorks LT Database . . . 7
- 8.5 lab: title: 'Get Started with Transact-SQL' module: 'Module 1: Getting Started with Transact-SQL' . . . 7

9 Get Started with Transact-SQL **7**
- 9.1 Explore the *AdventureWorks* database . . . 7
- 9.2 Use SELECT queries to retrieve data . . . 8
- 9.3 Work with data types . . . 9
- 9.4 Handle NULL values . . . 10
- 9.5 Challenges . . . 11
  - 9.5.1 Challenge 1: Retrieve customer data . . . 11
  - 9.5.2 Challenge 2: Retrieve customer order data . . . 11
  - 9.5.3 Challenge 3: Retrieve customer contact details . . . 12
- 9.6 Challenge Solutions . . . 12

9.6.1 Challenge 1 . . . 12
9.6.2 Challenge 2 . . . 12
9.6.3 Challenge 3: . . . 13
9.7 lab: title: 'Sort and Filter Query Results' module: 'Module 2: Sorting and Filtering Query Results' 13

10 Sort and Filter Query Results **13**
10.1 Sort results using the ORDER BY clause . . . 14
10.2 Restrict results using TOP . . . 14
10.3 Retrieve pages of results with OFFSET and FETCH . . . 15
10.4 Use the ALL and DISTINCT options . . . 15
10.5 Filter results with the WHERE clause . . . 16
10.6 Challenges . . . 17
10.6.1 Challenge 1: Retrieve data for transportation reports . . . 17
10.6.2 Challenge 2: Retrieve product data . . . 17
10.7 Challenge Solutions . . . 18
10.7.1 Challenge 1 . . . 18
10.7.2 Challenge 2 . . . 18
10.8 lab: title: 'Query Multiple Tables with Joins' module: 'Module 3: Using Joins and Subqueries' . 18

11 Query Multiple Tables with Joins **18**
11.1 Use inner joins . . . 19
11.2 Use outer joins . . . 20
11.3 Use a cross join . . . 21
11.4 Use a self join . . . 21
11.5 Challenges . . . 22
11.5.1 Challenge 1: Generate invoice reports . . . 22
11.5.2 Challenge 2: Retrieve customer data . . . 22
11.5.3 Challenge 3: Create a product catalog . . . 22
11.6 Challenge Solutions . . . 22
11.6.1 Challenge 1 . . . 22
11.6.2 Challenge 2 . . . 23
11.6.3 Challenge 3 . . . 23
11.7 lab: title: 'Use Subqueries' module: 'Module 3: Using Joins and Subqueries' . . . 23

12 Use Subqueries **23**
12.1 Use simple subqueries . . . 24
12.2 Use correlated subqueries . . . 25
12.3 Challenges . . . 26
12.3.1 Challenge 1: Retrieve product price information . . . 26
12.3.2 Challenge 2: Analyze profitability . . . 26
12.4 Challenge Solutions . . . 26
12.4.1 Challenge 1 . . . 26
12.4.2 Challenge 2 . . . 27
12.5 lab: title: 'Use Built-in Functions' module: 'Module 4: Using Built-in Functions' . . . 27

13 Use Built-in Functions **27**
13.1 Scalar functions . . . 28
13.2 Use logical functions . . . 29
13.3 Use aggregate functions . . . 30
13.4 Group aggregated results with the GROUP BY clause . . . 30
13.5 Filter groups with the HAVING clause . . . 31
13.6 Challenges . . . 31
13.6.1 Challenge 1: Retrieve order shipping information . . . 31
13.6.2 Challenge 2: Aggregate product sales . . . 31
13.7 Challenge Solutions . . . 32
13.7.1 Challenge 1 . . . 32
13.7.2 Challenge 2 . . . 32
13.8 lab: title: 'Modify Data' module: 'Module 5: Modifying Data' . . . 33

14 Modify Data **33**
14.1 Insert data . . . 33

14.2 Update data . . . 35
14.3 Delete data . . . 36
14.4 Challenges . . . 36
14.4.1 Challenge 1: Insert products . . . 36
14.4.2 Challenge 2: Update products . . . 37
14.4.3 Challenge 3: Delete products . . . 37
14.5 Challenge Solutions . . . 37
14.5.1 Challenge 1 . . . 37
14.5.2 Challenge 2 . . . 38
14.5.3 Challenge 3 . . . 38

1 INF99X: Sample Course

- **Download Latest Student Handbook and AllFiles Content**
- **Are you a MCT?** - Have a look at our GitHub User Guide for MCTs
- **Need to manually build the lab instructions?** - Instructions are available in the MicrosoftLearning/Docker-Build repository

1.1 What are we doing?

- To support this course, we will need to make frequent updates to the course content to keep it current with the Azure services used in the course. We are publishing the lab instructions and lab files on GitHub to allow for open contributions between the course authors and MCTs to keep the content current with changes in the Azure platform.
- We hope that this brings a sense of collaboration to the labs like we've never had before - when Azure changes and you find it first during a live delivery, go ahead and make an enhancement right in the lab source. Help your fellow MCTs.

1.2 How should I use these files relative to the released MOC files?

- The instructor handbook and PowerPoints are still going to be your primary source for teaching the course content.
- These files on GitHub are designed to be used in conjunction with the student handbook, but are in GitHub as a central repository so MCTs and course authors can have a shared source for the latest lab files.
- It will be recommended that for every delivery, trainers check GitHub for any changes that may have been made to support the latest Azure services, and get the latest files for their delivery.

1.3 What about changes to the student handbook?

- We will review the student handbook on a quarterly basis and update through the normal MOC release channels as needed.

1.4 How do I contribute?

- Any MCT can submit a pull request to the code or content in the GitHub repro, Microsoft and the course author will triage and include content and lab code changes as needed.
- You can submit bugs, changes, improvement and ideas. Find a new Azure feature before we have? Submit a new demo!

1.5 Notes

1.5.1 Classroom Materials

1.6 It is strongly recommended that MCTs and Partners access these materials and in turn, provide them separately to students. Pointing students directly to GitHub to access Lab steps as part of an ongoing class will require them to access yet another UI as part of the course, contributing to a confusing experience for the student. An explanation to the student regarding why they are receiving separate Lab instructions can highlight the nature of an always-changing cloud-based interface and platform. Microsoft Learning support for accessing files on GitHub and support for navigation of the GitHub site is limited to MCTs teaching this course only.

1.7 title: Online Hosted Instructions permalink: index.html layout: home

2 Content Directory

Hyperlinks to each of the lab exercises and demos are listed below.

2.1 Labs

{% assign labs = site.pages | where_exp:"page", "page.url contains '/Instructions/Labs'" %} | Module | Lab | | --- | --- | {% for activity in labs %}| {{ activity.lab.module }} | [{{ activity.lab.title }}{% if activity.lab.type %} - {{ activity.lab.type }}{% endif %}](/home/ll/Azure_clone/Azure_new/dp-080-Transact-SQL/{{ site.github.url }}{{ activity.url }}) | {% endfor %}

2.2 Demos

2.3 {% assign demos = site.pages | where_exp:"page", "page.url contains '/Instructions/Demos'" %} | Module | Demo | | --- | --- | {% for activity in demos %}| {{ activity.demo.module }} | [{{ activity.demo.title }}](/home/ll/Azure_clone/Azure_new/dp-080-Transact-SQL/{{ site.github.url }}{{ activity.url }}) | {% endfor %}

2.4 demo: title: 'Module 1 Demonstrations' module: 'Module 1: Getting Started with Transact-SQL'

3 Module 1 Demonstrations

This file contains guidance for demonstrations you can use to help students understand key concepts taught in the module.

3.1 Explore the lab environment

Throughout the course, students use a hosted environment that includes **Azure Data Studio** and a local instance of SQL Server Express containing a simplified version of the **adventureworks** sample database.

1. Start the hosted lab environment (for any lab in this course), and log in if necessary.
2. Start Azure Data Studio, and in the **Connections** tab, select the **AdventureWorks** connection. This will connect to the SQL Server instance and show the objects in the **adventureworks** database.
3. Expand the **Tables** folder to see the tables that are defined in the database. Note that there are a few tables in the **dbo** schema, but most of the tables are defined in a schema named **SalesLT**.
4. Expand the **SalesLT.Product** table and then expand its **Columns** folder to see the columns in this table. Each column has a name, a data type, an indication of whether it can contain *null* values, and in some cases an indication that the columns is used as a primary key (PK) or foreign key (FK).
5. Right-click the **SalesLT.Product** table and use the **Select Top 1000** option to create and run a new query script that retrieves the first 1000 rows from the table.

6. Review the query results, which consist of 1000 rows - each row representing a product that is sold by the fictitious *Adventure Works Cycles* company.
7. Close the **SQLQuery_1** pane that contains the query and its results.
8. Explore the other tables in the database, which contain information about product details, customers, and sales orders.
9. In Azure Data Studio, create a new query (you can do this from the **File** menu or on the *welcome* page).
10. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection (do this even if the query was already connected by clicking **Disconnect** first - it's useful for students to see how to connect to the saved connection!).
11. In the query editor, enter the following code:

    ```
    SELECT * FROM SalesLT.Product;
    ```

12. Use the **Run** button to run the query, and and after a few seconds, review the results, which includes all fields for all products.
13. in the results pane, select the **Messages** tab. This tab provides output messages from the query, and is a useful way to check the number of rows returned by the query.

3.2 Run basic SELECT queries

Use these example queries at appropriate points during the module presentation.

1. In Azure Data Studio, open the file at **https://raw.githubusercontent.com/MicrosoftLearning/dp-080-Transact-SQL/master/Scripts/module01-demos.sql**
2. Connect the script to the saved **AdventureWorks** connection.
3. Select and run each query when relevant (when text is selected in the script editor, the **Run** button runs only the selected text).

3.3 demo: title: 'Module 2 Demonstrations' module: 'Module 2: Sorting and Filtering Query Results'

4 Module 2 Demonstrations

This file contains guidance for demonstrations you can use to help students understand key concepts taught in the module.

> **Tip**: You can use the hosted lab environment for any lab in this course to perform these demo's.

1. In Azure Data Studio, open the file at **https://raw.githubusercontent.com/MicrosoftLearning/dp-080-Transact-SQL/master/Scripts/module02-demos.sql**
2. Connect the script to the saved **AdventureWorks** connection.
3. Select and run each query when relevant (when text is selected in the script editor, the **Run** button runs only the selected text).

4.1 demo: title: 'Module 3 Demonstrations' module: 'Module 3: Using Joins and Subqueries'

5 Module 3 Demonstrations

This file contains guidance for demonstrations you can use to help students understand key concepts taught in the module.

> **Tip**: You can use the hosted lab environment for any lab in this course to perform these demo's.

1. In Azure Data Studio, open the file at **https://raw.githubusercontent.com/MicrosoftLearning/dp-080-Transact-SQL/master/Scripts/module03-demos.sql**
2. Connect the script to the saved **AdventureWorks** connection.

3. Select and run each query when relevant (when text is selected in the script editor, the **Run** button runs only the selected text).

5.1 demo: title: 'Module 4 Demonstrations' module: 'Module 4: Using Built-in Functions'

6 Module 4 Demonstrations

This file contains guidance for demonstrations you can use to help students understand key concepts taught in the module.

> **Tip**: You can use the hosted lab environment for any lab in this course to perform these demo's.

1. In Azure Data Studio, open the file at **https://raw.githubusercontent.com/MicrosoftLearning/dp-080-Transact-SQL/master/Scripts/module04-demos.sql**
2. Connect the script to the saved **AdventureWorks** connection.
3. Select and run each query when relevant (when text is selected in the script editor, the **Run** button runs only the selected text).

6.1 demo: title: 'Module 5 Demonstrations' module: 'Module 5: Modifying Data'

7 Module 5 Demonstrations

This file contains guidance for demonstrations you can use to help students understand key concepts taught in the module.

> **Tip**: You can use the hosted lab environment for any lab in this course to perform these demo's.

1. In Azure Data Studio, open the file at **https://raw.githubusercontent.com/MicrosoftLearning/dp-080-Transact-SQL/master/Scripts/module05-demos.sql**
2. Connect the script to the saved **AdventureWorks** connection.
3. Select and run each query when relevant (when text is selected in the script editor, the **Run** button runs only the selected text).

7.1 lab: title: 'Lab Environment Setup' module: 'Setup'

8 Lab Environment Setup

The labs in this repo are designed to be performed in a hosted environment, provided on Microsoft Learn or in official course deliveries by Microsoft and partners.

> **Note**: The following information is provided to help you understand what's installed in the hosted environment, and to provide a guide for what you need to install if you want to try the labs on your own computer. However, please note that these guidelines are provided as-is with no warranty. Microsoft cannot provide support for your own lab environment.

8.1 Base Operating System

The hosted lab environment provided for this course is based on Microsoft Windows 10 with the latest updates applied as of March 12th 2021.

The required software for the labs can also be installed on Linux and Apple Mac computers, but this configuration has not been tested.

8.2 Microsoft SQL Server Express 2019

1. Download Microsoft SQL Server Express 2019 from the Microsoft download center.
2. Run the downloaded installer and select the **Basic** installation option.

8.3 Microsoft Azure Data Studio

1. Download and install Azure Data Studio from the Azure Data Studio documentation, following the appropriate instructions for your operating system.

8.4 AdventureWorks LT Database

The labs use a lightweight version of the AdventureWorks sample database. Note that this is not the same as the official sample database, so use the following instructions to create it.

1. Download the **adventureworkslt.sql** script, and save it on your local computer.
2. Start Azure Data Studio, and open the **adventureworkslt.sql** script file you downloaded.
3. In the script pane, connect to your SQL Server Express server server using the following information:
   - **Connection type**: SQL Server
   - **Server**: localhost\SQLExpress
   - **Authentication Type**: Windows Authentication
   - **Database**: master
   - **Server group**: <Default>
   - **Name**: *leave blank*
4. Ensure the **master** database is selected, and then run the script to create the **adventureworks** database. This will take a few minutes.
5. After the database has been created, on the **Connections** pane, in the **Servers** section, create a new connection with the following settings:
   - **Connection type**: SQL Server
   - **Server**: localhost\SQLExpress
   - **Authentication Type**: Windows Authentication
   - **Database**: adventureworks
   - **Server group**: <Default>
   - **Name**: AdventureWorks

8.5 lab: title: 'Get Started with Transact-SQL' module: 'Module 1: Getting Started with Transact-SQL'

9 Get Started with Transact-SQL

In this lab, you will use some basic SELECT queries to retrieve data from the **AdventureWorks** database.

9.1 Explore the *AdventureWorks* database

We'll use the **AdventureWorks** database in this lab, so let's start by exploring it in Azure Data Studio.

1. Start Azure Data Studio, and in the **Connections** tab, select the **AdventureWorks** connection by clicking on the arrow just to the left of the name. This will connect to the SQL Server instance and show the objects in the **AdventureWorks** database.

2. Expand the **Tables** folder to see the tables that are defined in the database. Note that there are a few tables in the **dbo** schema, but most of the tables are defined in a schema named **SalesLT**.

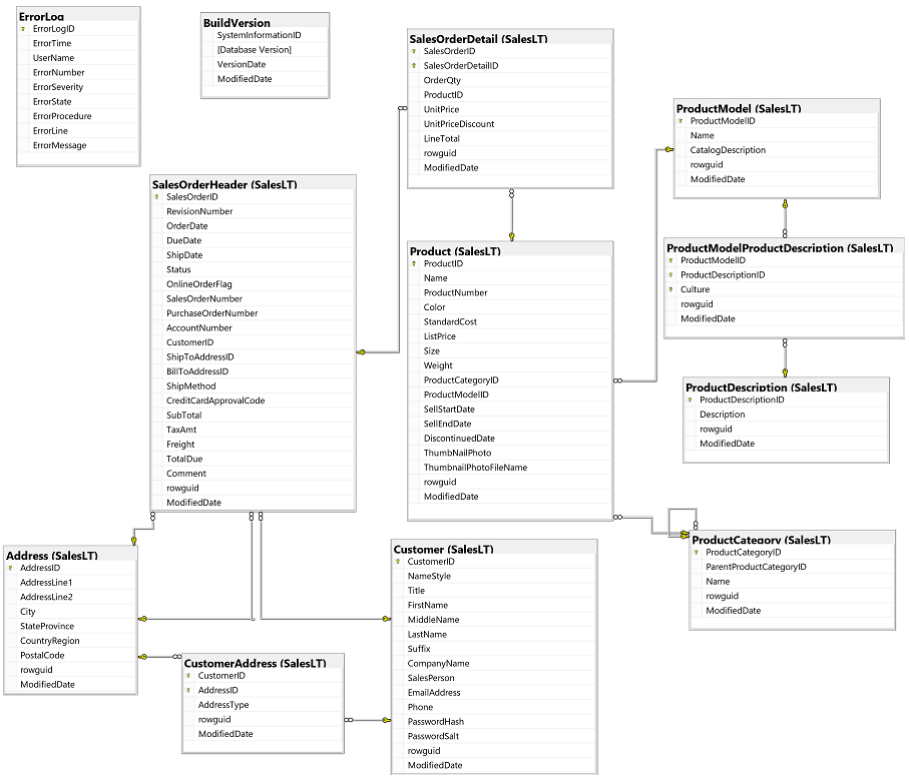
3. Expand the **SalesLT.Product** table and then expand its **Columns** folder to see the columns in this table. Each column has a name, a data type, an indication of whether it can contain *null* values, and in some cases an indication that the columns is used as a primary key (PK) or foreign key (FK).

4. Right-click the **SalesLT.Product** table and use the **Select Top 1000** option to create and run a new query script that retrieves the first 1000 rows from the table.

5. Review the query results, which consist of 1000 rows - each row representing a product that is sold by the fictitious *Adventure Works Cycles* company.

6. Close the **SQLQuery_1** pane that contains the query and its results.

7. Explore the other tables in the database, which contain information about product details, customers, and sales orders. The tables are related through primary and foreign keys, as shown here (you may need to resize the pane to see them clearly):



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

9.2 Use SELECT queries to retrieve data

Now that you've had a chance to explore the **AdventureWorks** database, it's time to dig a little deeper into the product data it contains by querying the **Product** table.

1. In Azure Data Studio, create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, ensure that the **AdventureWorks** database is selected at the top of the query pane. If not, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

   ```
   SELECT * FROM SalesLT.Product;
   ```

4. Use the **Run** button to run the query, and and after a few seconds, review the results, which includes all columns for all products.
5. In the query editor, modify the query as follows:

   ```
   SELECT Name, StandardCost, ListPrice
   FROM SalesLT.Product;
   ```

6. Use the **Run** button to re-run the query, and and after a few seconds, review the results, which this time include only the **Name**, **StandardCost**, and **ListPrice** columns for all products.
7. Modify the query as shown below to include an expression that results in a calculated column, and then re-run the query:

   ```
   SELECT Name, ListPrice - StandardCost
   FROM SalesLT.Product;
   ```

8. Note that the results this time include the **Name** column and an unnamed column containing the result of subtracting the **StandardCost** from the **ListPrice**.
9. Modify the query as shown below to assign names to the columns in the results, and then re-run the query.

   ```
   SELECT Name AS ProductName, ListPrice - StandardCost AS Markup
   FROM SalesLT.Product;
   ```

10. Note that the results now include columns named **ProductName** and **Markup**. The **AS** keyword has been used to assign an *alias* for each column in the results.

11. Replace the existing query with the following code, which also includes an expression that produces a calculated column in the results:

```
SELECT ProductNumber, Color, Size, Color + ', ' + Size AS ProductDetails
FROM SalesLT.Product;
```

12. Run the query, and note that the + operator in the calculated **ProductDetails** column is used to *concatenate* the **Color** and **Size** column values (with a literal comma between them). The behavior of this operator is determined by the data types of the columns - had they been numeric values, the + operator would have *added* them. Note also that some results are *NULL* - we'll explore NULL values later in this lab.

9.3 Work with data types

As you just saw, columns in a table are defined as specific data types, which affects the operations you can perform on them.

1. Replace the existing query with the following code, and run it:

```
SELECT ProductID + ': ' + Name AS ProductName
FROM SalesLT.Product;
```

2. Note that this query returns an error. The + operator can be used to *concatenate* text-based values, or *add* numeric values; but in this case there's one numeric value (**ProductID**) and one text-based value (**Name**), so it's unclear how the operator should be applied.

3. Modify the query as follows, and re-run it:

```
SELECT CAST(ProductID AS varchar(5)) + ': ' + Name AS ProductName
FROM SalesLT.Product;
```

4. Note that the effect of the **CAST** function is to change the numeric **ProductID** column into a **varchar** (variable-length character data) value that can be concatenated with other text-based values.

5. Modify the query to replace the **CAST** function with a **CONVERT** function as shown below, and then re-run it:

```
SELECT CONVERT(varchar(5), ProductID) + ': ' + Name AS ProductName
FROM SalesLT.Product;
```

6. Note that the results of using **CONVERT** are the same as for **CAST**. The **CAST** function is an ANSI standard part of the SQL language that is available in most database systems, while **CONVERT** is a SQL Server specific function.

7. Another key difference between the two functions is that **CONVERT** includes an additional parameter that can be useful for formatting date and time values when converting them to text-based data. For example, replace the existing query with the following code and run it.

```
SELECT SellStartDate,
   CONVERT(nvarchar(30), SellStartDate) AS ConvertedDate,
   CONVERT(nvarchar(30), SellStartDate, 126) AS ISO8601FormatDate
FROM SalesLT.Product;
```

8. Replace the existing query with the following code, and run it.

```
SELECT Name, CAST(Size AS Integer) AS NumericSize
FROM SalesLT.Product;
```

9. Note that an error is returned because some **Size** values are not numeric (for example, some item sizes are indicated as *S*, *M*, or *L*).

10. Modify the query to use a **TRY_CAST** function, as shown here.

```
SELECT Name, TRY_CAST(Size AS Integer) AS NumericSize
FROM SalesLT.Product;
```

11. Run the query and note that the numeric **Size** values are converted successfully to integers, but that non-numeric sizes are returned as *NULL*.

9.4 Handle NULL values

We've seen some examples of queries that return *NULL* values. *NULL* is generally used to denote a value that is *unknown*. Note that this is not the same as saying the value is *none* - that would imply that you *know* that the value is zero or an empty string!

1. Modify the existing query as shown here:

   ```
   SELECT Name, ISNULL(TRY_CAST(Size AS Integer),0) AS NumericSize
   FROM SalesLT.Product;
   ```

2. Run the query and view the results. Note that the **ISNULL** function replaces *NULL* values with the specified value, so in this case, sizes that are not numeric (and therefore can't be converted to integers) are returned as **0**.

   In this example, the **ISNULL** function is applied to the output of the inner **TRY_CAST** function, but you can also use it to deal with *NULL* values in the source table.

3. Replace the query with the following code to handle *NULL* values for **Color** and **Size** values in the source table:

   ```
   SELECT ProductNumber, ISNULL(Color, '') + ', ' + ISNULL(Size, '') AS ProductDetails
   FROM SalesLT.Product;
   ```

   The **ISNULL** function replaces *NULL* values with a specified literal value. Sometimes, you may want to achieve the opposite result by replacing an explicit value with *NULL*. To do this, you can use the **NULLLIF** function.

4. Try the following query, which replaces the **Color** value "Multi" to *NULL*.

   ```
   SELECT Name, NULLIF(Color, 'Multi') AS SingleColor
   FROM SalesLT.Product;
   ```

   In some scenarios, you might want to compare multiple columns and find the first one that isn't *NULL*. For example, suppose you want to track the status of a product's availability based on the dates recorded when it was first offered for sale or removed from sale. A product that is currently available will have a **SellStartDate**, but the **SellEndDate** value will be *NULL*. When a product is no longer sold, a date is entered in its **SellEndDate** column. To find the first non-*NULL* column, you can use the **COALESCE** function.

5. Use the following query to find the first non-*NULL* date for product selling status.

   ```
   SELECT Name, COALESCE(SellEndDate, SellStartDate) AS StatusLastUpdated
   FROM SalesLT.Product;
   ```

   The previous query returns the last date on which the product selling status was updated, but doesn't actually tell us the sales status itself. To determine that, we'll need to check the dates to see if the **SellEndDate** is *NULL*. To do this, you can use a **CASE** expression in the **SELECT** clause to check for *NULL* **SellEndDate** values. The **CASE** expression has two variants: a *simple* **CASE** what evaluates a specific column or value, or a *searched* **CASE** that evaluates one or more expressions.

   In this example, or **CASE** experssion must determine if the **SellEndDate** column is *NULL*. Typically, when you are trying to check the value of a column you can use the = operator; for example the predicate **SellEndDate = '01/01/2005'** returns **True** if the **SellEndDate** value is *01/01/2005*, and **False** otherwise. However, when dealing with *NULL* values, the default behavior may not be what you expect. Remember that *NULL* actually means *unknown*, so using the = operator to compare two unknown values always results in a value of *NULL* - semantically, it's impossible to know if one unknown value is the same as another. To check to see if a value is *NULL*, you must use the **IS NULL** predicate; and conversely to check that a value is not *NULL* you can use the **IS NOT NULL** predicate.

6. Run the following query, which includes *searched* **CASE** that uses an **IS NULL** expression to check for *NULL* **SellEndDate** values.

   ```
   SELECT Name,
       CASE
           WHEN SellEndDate IS NULL THEN 'Currently for sale'
           ELSE 'No longer available'
       END AS SalesStatus
   FROM SalesLT.Product;
   ```

The previous query used a *searched* **CASE** expression, which begins with a **CASE** keyword, and includes one or more **WHEN...THEN** expressions with the values and predicates to be checked. An **ELSE** expression provides a value to use if none of the **WHEN** conditions are matched, and the **END** keyword denotes the end of the **CASE** expression, which is aliased to a column name for the result using an **AS** expression.

In some queries, it's more appropriate to use a *simple* **CASE** expression that applies multiple **WHERE...THEN** predictes to the same value.

7. Run the following query to see an example of a *simple* **CASE** expression that produced different results depending on the **Size** column value.

```
SELECT Name,
    CASE Size
        WHEN 'S' THEN 'Small'
        WHEN 'M' THEN 'Medium'
        WHEN 'L' THEN 'Large'
        WHEN 'XL' THEN 'Extra-Large'
        ELSE ISNULL(Size, 'n/a')
    END AS ProductSize
FROM SalesLT.Product;
```

8. Review the query results and note that the **ProductSize** column contains the text-based description of the size for *S*, *M*, *L*, and *XL* sizes; the measurement value for numeric sizes, and *n/a* for any other sizes values.

9.5 Challenges

Now that you've seen some examples of **SELECT** statements that retrieve data from a table, it's time to try to compose some queries of your own.

> **Tip**: Try to determine the appropriate queries for yourself. If you get stuck, suggested answers are provided at the end of this lab.

9.5.1 Challenge 1: Retrieve customer data

Adventure Works Cycles sells directly to retailers, who then sell products to consumers. Each retailer that is an Adventure Works customer has provided a named contact for all communication from Adventure Works. The sales manager at Adventure Works has asked you to generate some reports containing details of the company's customers to support a direct sales campaign.

1. Retrieve customer details
   - Familiarize yourself with the **SalesLT.Customer** table by writing a Transact-SQL query that retrieves all columns for all customers.
2. Retrieve customer name data
   - Create a list of all customer contact names that includes the title, first name, middle name (if any), last name, and suffix (if any) of all customers.
3. Retrieve customer names and phone numbers
   - Each customer has an assigned salesperson. You must write a query to create a call sheet that lists:
     - The salesperson
     - A column named **CustomerName** that displays how the customer contact should be greeted (for example, *Mr Smith*)
     - The customer's phone number.

9.5.2 Challenge 2: Retrieve customer order data

As you continue to work with the Adventure Works customer data, you must create queries for reports that have been requested by the sales team.

1. Retrieve a list of customer companies
   - You have been asked to provide a list of all customer companies in the format *Customer ID* : *Company Name* - for example, *78: Preferred Bikes*.
2. Retrieve a list of sales order revisions
   - The **SalesLT.SalesOrderHeader** table contains records of sales orders. You have been asked to retrieve data for a report that shows:

- The sales order number and revision number in the format () – for example SO71774 (2).
- The order date converted to ANSI standard *102* format (*yyyy.mm.dd* – for example *2015.01.31*).

9.5.3 Challenge 3: Retrieve customer contact details

Some records in the database include missing or unknown values that are returned as NULL. You must create some queries that handle these NULL values appropriately.

1. Retrieve customer contact names with middle names if known

   - You have been asked to write a query that returns a list of customer names. The list must consist of a single column in the format *first last* (for example *Keith Harris*) if the middle name is unknown, or *first middle last* (for example *Jane M. Gates*) if a middle name is known.

2. Retrieve primary contact details

   - Customers may provide Adventure Works with an email address, a phone number, or both. If an email address is available, then it should be used as the primary contact method; if not, then the phone number should be used. You must write a query that returns a list of customer IDs in one column, and a second column named **PrimaryContact** that contains the email address if known, and otherwise the phone number.

     IMPORTANT: In the sample data provided, there are no customer records without an email address. Therefore, to verify that your query works as expected, run the following **UPDATE** statement to remove some existing email addresses before creating your query:

     ```
     UPDATE SalesLT.Customer
     SET EmailAddress = NULL
     WHERE CustomerID % 7 = 1;
     ```

3. Retrieve shipping status

   - You have been asked to create a query that returns a list of sales order IDs and order dates with a column named **ShippingStatus** that contains the text *Shipped* for orders with a known ship date, and *Awaiting Shipment* for orders with no ship date.

     IMPORTANT: In the sample data provided, there are no sales order header records without a ship date. Therefore, to verify that your query works as expected, run the following UPDATE statement to remove some existing ship dates before creating your query.

     ```
     UPDATE SalesLT.SalesOrderHeader
     SET ShipDate = NULL
     WHERE SalesOrderID > 71899;
     ```

9.6 Challenge Solutions

This section contains suggested solutions for the challenge queries.

9.6.1 Challenge 1

1. Retrieve customer details:

   ```
   SELECT * FROM SalesLT.Customer;
   ```

2. Retrieve customer name data:

   ```
   SELECT Title, FirstName, MiddleName, LastName, Suffix
   FROM SalesLT.Customer;
   ```

3. Retrieve customer names and phone numbers:

   ```
   SELECT Salesperson, Title + ' ' + LastName AS CustomerName, Phone
   FROM SalesLT.Customer;
   ```

9.6.2 Challenge 2

1. Retrieve a list of customer companies:

   ```
   SELECT CAST(CustomerID AS varchar) + ': ' + CompanyName AS CustomerCompany
   FROM SalesLT.Customer;
   ```

2. Retrieve a list of sales order revisions:

```
SELECT SalesOrderNumber + ' (' + STR(RevisionNumber, 1) + ')' AS OrderRevision,
   CONVERT(nvarchar(30), OrderDate, 102) AS OrderDate
FROM SalesLT.SalesOrderHeader;
```

9.6.3 Challenge 3:

1. Retrieve customer contact names with middle names if known:

```
SELECT FirstName + ' ' + ISNULL(MiddleName + ' ', '') + LastName AS CustomerName
FROM SalesLT.Customer;
```

2. Retrieve primary contact details:

```
SELECT CustomerID, COALESCE(EmailAddress, Phone) AS PrimaryContact
FROM SalesLT.Customer;
```

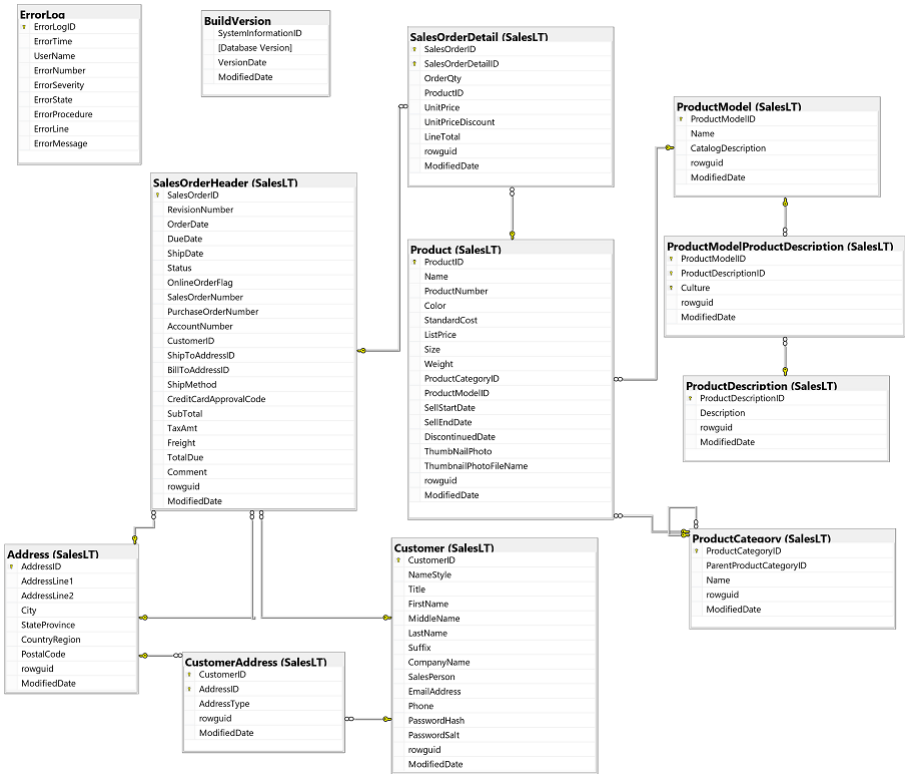
3. Retrieve shipping status:

```
SELECT SalesOrderID, OrderDate,
    CASE
        WHEN ShipDate IS NULL THEN 'Awaiting Shipment'
        ELSE 'Shipped'
    END AS ShippingStatus
FROM SalesLT.SalesOrderHeader;
```

9.7 lab: title: 'Sort and Filter Query Results' module: 'Module 2: Sorting and Filtering Query Results'

10 Sort and Filter Query Results

In this lab, you'll use the Transact-SQL **SELECT** statement to query and filter data in the **adventureworks** database. For your reference, the following diagram shows the tables in the database (you may need to resize the pane to see them clearly).



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

10.1 Sort results using the ORDER BY clause

It's often useful to sort query results into a particular order.

1. Modify the existing query to return the **Name** and **ListPrice** of all products:

```
SELECT Name, ListPrice
FROM SalesLT.Product;
```

2. Run the query and note that the results are presented in no particular order.
3. Modify the query to add an **ORDER BY** clause that sorts the results by **Name**, as shown here:

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY Name;
```

4. Run the query and review the results. This time the products are listed in alphabetical order by **Name**.
5. Modify the query as shown below to sort the results by **ListPrice**.

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice;
```

6. Run the query and note that the results are listed in ascending order of **ListPrice**. By default, the **ORDER BY** clause applies an ascending sort order to the specified field.
7. Modify the query as shown below to sort the results into descending order of **ListPrice**.

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC;
```

8. Run the query and note that the results now show the most expensive items first.
9. Modify the query as shown below to sort the results into descending order of **ListPrice**, and then into ascending order of **Name**.

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC, Name ASC;
```

10. Run the query and review the results. Note that they are sorted into descending order of **ListPrice**, and each set of products with the same price is sorted in ascending order of **Name**.

10.2 Restrict results using TOP

Sometimes you only want to return a specific number of rows. For example, you might want to find the twenty most expensive products.

1. Modify the existing query to return the **Name** and **ListPrice** of all products:

```
SELECT TOP 20 Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC;
```

2. Run the query and note that the results contain the first twenty products in descending order of **ListPrice**. Typically, you include an **ORDER BY** clause when using the **TOP** parameter; otherwise the query just returns the first specified number of rows in an arbitrary order.
3. Modify the query to add the **WITH TIES** parameter as shown here, and re-run it.

```
SELECT TOP 20 WITH TIES Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC;
```

4. This time, there are 21 rows in the results, because there are multiple products that share the same price, one of which wasn't included when ties were ignored by the previous query.
5. Modify the query to add the **PERCENT** parameter as shown here, and re-run it.

```
SELECT TOP 20 PERCENT WITH TIES Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC;
```

6. Note that this time the results contain the 20% most expensive products.

10.3 Retrieve pages of results with OFFSET and FETCH

User interfaces and reports often present large volumes of data as pages, you make them easier to navigate on a screen.

1. Modify the existing query to return product **Name** and **ListPrice** values:

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY Name OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Run the query and note the effect of the **OFFSET** and **FETCH** parameters of the **ORDER BY** clause. The results start at the 0 position (the beginning of the result set) and include only the next 10 rows, essentially defining the first page of results with 10 rows per page.
3. Modify the query as shown here, and run it to retrieve the next page of results.

```
SELECT Name, ListPrice
FROM SalesLT.Product
ORDER BY Name OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

10.4 Use the ALL and DISTINCT options

Often, multiple rows in a table may contain the same values for a given subset of fields. For example, a table of products might contain a **Color** field that identifies the color of a given product. It's not unreasonable to assume that there may be multiple products of the same color. Similarly, the table might contain a **Size** field; and again it's not unreasonable to assume that there may be multiple products of the same size; or even multiple products with the same combination of size and color.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

```
SELECT Color
FROM SalesLT.Product;
```

4. Use the **Run** button to run the query, and and after a few seconds, review the results, which includes the color of each product in the table.
5. Modify the query as follows, and re-run it.

```
SELECT ALL Color
FROM SalesLT.Product;
```

   The results should be the same as before. The **ALL** parameter is the default behavior, and is applied implicitly to return a row for every record that meets the query criteria.

6. Modify the query to replace **ALL** with **DISTINCT**, as shown here:

```
SELECT DISTINCT Color
FROM SalesLT.Product;
```

7. Run the modified query and note that the results include one row for each unique **Color** value. This ability to remove duplicates from the results can often be useful - for example to retrieve values in order to populate a drop-down list of color options in a user interface.

8. Modify the query to add the **Size** field as shown here:

```
SELECT DISTINCT Color, Size
FROM SalesLT.Product;
```

9. Run the modified query and note that it returns each unique combination of color and size.

10.5 Filter results with the WHERE clause

Most queries for application development or reporting involve filtering the data to match specified criteria. You typically apply filtering criteria as a predicate in a **WHERE** clause of a query.

1. In Azure Data Studio, replace the existing query with the following code:

```
SELECT Name, Color, Size
FROM SalesLT.Product
WHERE ProductModelID = 6
ORDER BY Name;
```

2. Run the query and review the results, which contain the **Name**, **Color**, and **Size** for each product with a **ProductModelID** value of *6* (this is the ID for the *HL Road Frame* product model, of which there are multiple variants).
3. Replace the query with the following code, which uses the *not equal* (<>) operator, and run it.

```
SELECT Name, Color, Size
FROM SalesLT.Product
WHERE ProductModelID <> 6
ORDER BY Name;
```

4. Review the results, noting that they contain all products with a **ProductModelID** other than **6**.
5. Replace the query with the following code, and run it.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice > 1000.00
ORDER BY ListPrice;
```

6. Review the results, noting that they contain all products with a **ListPrice** greater than 1000.00.
7. Modify the query as follows, and run it.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE Name LIKE 'HL Road Frame %';
```

8. Review the results, noting that the **LIKE** operator enables you to match string patterns. The % character in the predicate is a wildcard for one or more characters, so the query returns all rows where the **Name** is *HL Road Frame* followed by any string.

   The **LIKE** operator can be used to define complex pattern matches based on regular expressions, which can be useful when dealing with string data that follows a predictable pattern.

9. Modify the query as follows, and run it.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE ProductNumber LIKE 'FR-_[0-9][0-9]_-[0-9][0-9]';
```

10. Review the results. This time the results include products with a **ProductNumber** that matches patterns similar to FR-*xNNx-NN* (in which *x* is a letter and *N* is a numeral).

    Tip: To learn more about pattern-matching with the **LIKE** operator, see the Transact-SQL documentation.

11. Modify the query as follows, and run it.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE SellEndDate IS NOT NULL;
```

12. Note that to filter based on *NULL* values you must use **IS NULL** (or **IS NOT NULL**) you cannot compare a *NULL* value using the = operator.
13. Now try the following query, which uses the **BETWEEN** operator to define a filter based on values within a defined range.

```
SELECT Name
FROM SalesLT.Product
WHERE SellEndDate BETWEEN '2006/1/1' AND '2006/12/31';
```

14. Review the results, which contain products that the company stopped selling in 2006.
15. Run the following query, which retrieves products with a **ProductCategoryID** value that is in a specified list.

```
SELECT ProductCategoryID, Name, ListPrice
FROM SalesLT.Product
WHERE ProductCategoryID IN (5,6,7);
```

16. Now try the following query, which uses the **AND** operator to combine two criteria.

```
SELECT ProductCategoryID, Name, ListPrice, SellEndDate
FROM SalesLT.Product
WHERE ProductCategoryID IN (5,6,7) AND SellEndDate IS NULL;
```

17. Try the following query, which filters the results to include rows that match one (or both) of two criteria.

```
SELECT Name, ProductCategoryID, ProductNumber
FROM SalesLT.Product
WHERE ProductNumber LIKE 'FR%' OR ProductCategoryID IN (5,6,7);
```

10.6 Challenges

Now that you've seen some examples of filtering and sorting data, it's your chance to put what you've learned into practice.

> **Tip**: Try to determine the appropriate queries for yourself. If you get stuck, suggested answers are provided at the end of this lab.

10.6.1 Challenge 1: Retrieve data for transportation reports

The logistics manager at Adventure Works has asked you to generate some reports containing details of the company's customers to help to reduce transportation costs.

1. Retrieve a list of cities
   - Initially, you need to produce a list of all of you customers' locations. Write a Transact-SQL query that queries the **SalesLT.Address** table and retrieves the values for **City** and **StateProvince**, removing duplicates and sorted in ascending order of city.
2. Retrieve the heaviest products
   - Transportation costs are increasing and you need to identify the heaviest products. Retrieve the names of the top ten percent of products by weight.

10.6.2 Challenge 2: Retrieve product data

The Production Manager at Adventure Works would like you to create some reports listing details of the products that you sell.

1. Retrieve product details for product model 1
   - Initially, you need to find the names, colors, and sizes of the products with a product model ID 1.
2. Filter products by color and size
   - Retrieve the product number and name of the products that have a color of *black*, *red*, or *white* and a size of *S* or *M*.
3. Filter products by product number
   - Retrieve the product number, name, and list price of products whose product number begins *BK-*
4. Retrieve specific products by product number
   - Modify your previous query to retrieve the product number, name, and list price of products whose product number begins *BK-* followed by any character other than *R*, and ends with a - followed by any two numerals.

10.7 Challenge Solutions

This section contains suggested solutions for the challenge queries.

10.7.1 Challenge 1

1. Retrieve a list of cities:

```
SELECT DISTINCT City, StateProvince
FROM SalesLT.Address
ORDFER BY City
```

2. Retrieve the heaviest products:

```
SELECT TOP 10 PERCENT WITH TIES Name
FROM SalesLT.Product
ORDER BY Weight DESC;
```

10.7.2 Challenge 2

1. Retrieve product details for product model 1:

```
SELECT Name, Color, Size
FROM SalesLT.Product
WHERE ProductModelID = 1;
```

2. Filter products by color and size:

```
SELECT ProductNumber, Name
FROM SalesLT.Product
WHERE Color IN ('Black','Red','White') AND Size IN ('S','M');
```

3. Filter products by product number:

```
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
WHERE ProductNumber LIKE 'BK-%';
```

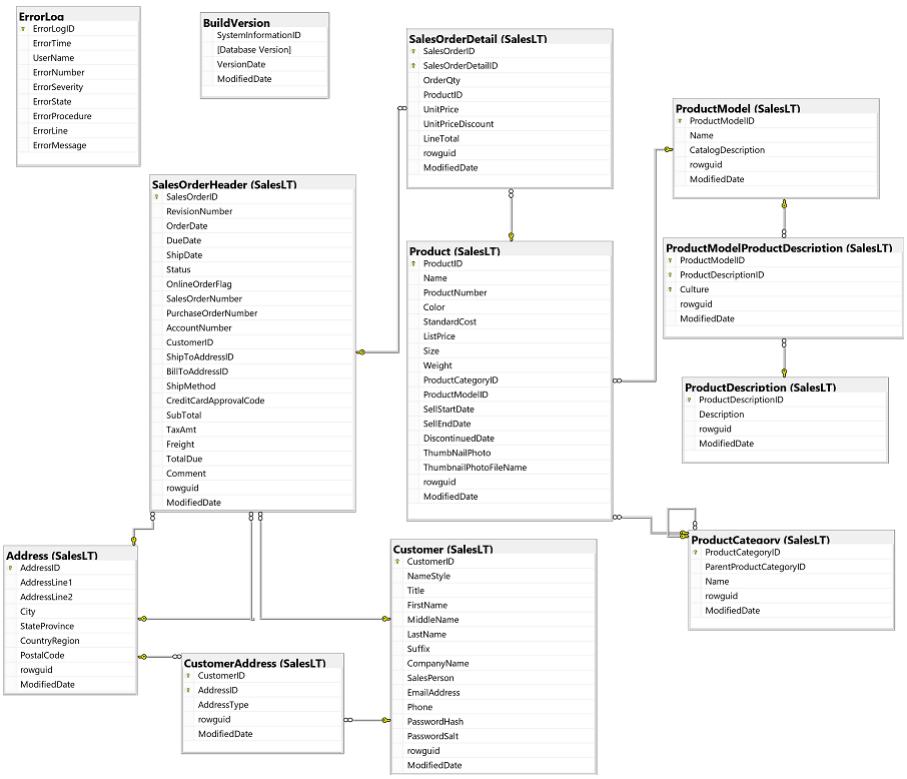
4. Retrieve specific products by product number:

```
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
WHERE ProductNumber LIKE 'BK-[^R]%-[0-9][0-9]';
```

10.8 lab: title: 'Query Multiple Tables with Joins' module: 'Module 3: Using Joins and Subqueries'

11 Query Multiple Tables with Joins

In this lab, you'll use the Transact-SQL **SELECT** statement to query multiple tables in the **adventureworks** database. For your reference, the following diagram shows the tables in the database (you may need to resize the pane to see them clearly).



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

11.1 Use inner joins

An inner join is used to find related data in two tables. For example, suppose you need to retrieve data about a product and its category from the **SalesLT.Product** and **SalesLT.ProductCategory** tables. You can find the relevant product category record for a product based on its **ProductCategoryID** field; which is a foreign-key in the product table that matches a primary key in the product category table.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

```
SELECT SalesLT.Product.Name As ProductName, SalesLT.ProductCategory.Name AS Category
FROM SalesLT.Product
INNER JOIN SalesLT.ProductCategory
ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;
```

4. Use the **Run** button to run the query, and and after a few seconds, review the results, which include the **ProductName** from the products table and the corresponding **Category** from the product category table. Because the query uses an **INNER** join, any products that do not have corresponding categories, and any categories that contain no products are omitted from the results.
5. Modify the query as follows to remove the **INNER** keyword, and re-run it.

```
SELECT SalesLT.Product.Name As ProductName, SalesLT.ProductCategory.Name AS Category
FROM SalesLT.Product
JOIN SalesLT.ProductCategory
    ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;
```

   The results should be the same as before. **INNER** joins are the default kind of join.
6. Modify the query to assign aliases to the tables in the **JOIN** clause, as shown here:

```
SELECT p.Name As ProductName, c.Name AS Category
```

```
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory As c
    ON p.ProductCategoryID = c.ProductCategoryID;
```

7. Run the modified query and confirm that it returns the same results as before. The use of table aliases can greatly simplify a query, particularly when multiple joins must be used.
8. Replace the query with the following code, which retrieves sales order data from the **SalesLT.SalesOrderHeader**, **SalesLT.SalesOrderDetail**, and **SalesLT.Product** tables:

```
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.Lin
FROM SalesLT.SalesOrderHeader AS oh
JOIN SalesLT.SalesOrderDetail AS od
    ON od.SalesOrderID = oh.SalesOrderID
JOIN SalesLT.Product AS p
    ON od.ProductID = p.ProductID
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

9. Run the modified query and note that it returns data from all three tables.

11.2 Use outer joins

An outer join is used to retrieve all rows from one table, and any corresponding rows from a related table. In cases where a row in the outer table has no corresponding rows in the related table, *NULL* values are returned for the related table fields. For example, suppose you want to retrieve a list of all customers and any orders they have placed, including customers who have registered but never placed an order.

1. Replace the existing query with the following code:

```
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
    ON c.CustomerID = oh.CustomerID
ORDER BY c.CustomerID;
```

2. Run the query and note that the results contain data for every customer. If a customer has placed an order, the order number is shown. Customers who have registered but not placed an order are shown with a *NULL* order number.

   Note the use of the **LEFT** keyword. This identifies which of the tables in the join is the *outer* table (the one from which all rows should be preserved). In this case, the join is between the **Customer** and **SalesOrderHeader** tables, so a **LEFT** join designates **Customer** as the outer table. Had a **RIGHT** join been used, the query would have returned all records from the **SalesOrderHeader** table and only matching data from the **Customer**table (in other words, all orders including those for which there was no matching customer record). You can also use a **FULL** outer join to preserve unmatched rows from *both* sides of the join (all customers, including those who haven't placed an order; and all orders, including those with no matching customer), though in practice this is used less frequently.

3. Modify the query to remove the **OUTER** keyword, as shown here:

```
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS oh
    ON c.CustomerID = oh.CustomerID
ORDER BY c.CustomerID;
```

4. Run the query and review the results, which should be the same as before. Using the **LEFT** (or **RIGHT**) keyword automatically identifies the join as an **OUTER** join.
5. Modify the query as shown below to take advantage of the fact that it identifies non-matching rows and return only the customers who have not placed any orders.

```
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS oh
    ON c.CustomerID = oh.CustomerID
WHERE oh.SalesOrderNumber IS NULL
```

```
ORDER BY c.CustomerID;
```

6. Run the query and review the results, which contain data for customers who have not placed any orders.
7. Replace the query with the following one, which uses outer joins to retrieve data from three tables.

```
SELECT p.Name As ProductName, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT JOIN SalesLT.SalesOrderDetail AS od
    ON p.ProductID = od.ProductID
LEFT JOIN SalesLT.SalesOrderHeader AS oh
    ON od.SalesOrderID = oh.SalesOrderID
ORDER BY p.ProductID;
```

8. Run the query and note that the results include all products, with order numbers for any that have been purchased. This required a sequence of joins from **Product** to **SalesOrderDetail** to **SalesOrderHeader**. Note that when you join multiple tables like this, after an outer join has been specified in the join sequence, all subsequent outer joins must be of the same direction (**LEFT** or **RIGHT**).
9. Modify the query as shown below to add an inner join to return category information. When mixing inner and outer joins, it can be helpful to be explicit about the join types by using the **INNER** and **OUTER** keywords.

```
SELECT p.Name As ProductName, c.Name AS Category, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT OUTER JOIN SalesLT.SalesOrderDetail AS od
    ON p.ProductID = od.ProductID
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
    ON od.SalesOrderID = oh.SalesOrderID
INNER JOIN SalesLT.ProductCategory AS c
    ON p.ProductCategoryID = c.ProductCategoryID
ORDER BY p.ProductID;
```

10. Run the query and review the results, which include product names, categories, and sales order numbers.

11.3 Use a cross join

A *cross* join matches all possible combinations of rows from the tables being joined. In practice, it's rarely used; but there are some specialized cases where it is useful.

1. Replace the existing query with the following code:

```
SELECT p.Name, c.FirstName, c.LastName, c.EmailAddress
FROM SalesLT.Product as p
CROSS JOIN SalesLT.Customer as c;
```

2. Run the query and note that the results contain a row for every product and customer combination (which might be used to create a mailing campaign in which an indivdual advertisement for each product is emailed to each customer - a strategy that may not endear the company to its customers!).

11.4 Use a self join

A *self* join isn't actually a specific kind of join, but it's a technique used to join a table to itself by defining two instances of the table, each with its own alias. This approach can be useful when a row in the table includes a foreign key field that references the primary key of the same table; for example in a table of employees where an employee's manager is also an employee, or a table of product categories where each category might be a subcategory of another category.

1. Replace the existing query with the following code, which includes a self join between two instances of the **SalesLT.ProductCategory** table (with aliases **cat** and **pcat**):

```
SELECT pcat.Name AS ParentCategory, cat.Name AS SubCategory
FROM SalesLT.ProductCategory as cat
JOIN SalesLT.ProductCategory pcat
    ON cat.ParentProductCategoryID = pcat.ProductCategoryID
ORDER BY ParentCategory, SubCategory;
```

2. Run the query and review the results, which reflect the hierarchy of parent and sub categories.

11.5 Challenges

Now that you've seen some examples of joins, it's your turn to try retrieving data from multiple tables for yourself.

> **Tip**: Try to determine the appropriate queries for yourself. If you get stuck, suggested answers are provided at the end of this lab.

11.5.1 Challenge 1: Generate invoice reports

Adventure Works Cycles sells directly to retailers, who must be invoiced for their orders. You have been tasked with writing a query to generate a list of invoices to be sent to customers.

1. Retrieve customer orders
   - As an initial step towards generating the invoice report, write a query that returns the company name from the **SalesLT.Customer** table, and the sales order ID and total due from the **SalesLT.SalesOrderHeader** table.
2. Retrieve customer orders with addresses
   - Extend your customer orders query to include the *Main Office* address for each customer, including the full street address, city, state or province, postal code, and country or region
   - **Tip**: Note that each customer can have multiple addressees in the **SalesLT.Address** table, so the database developer has created the **SalesLT.CustomerAddress** table to enable a many-to-many relationship between customers and addresses. Your query will need to include both of these tables, and should filter the results so that only *Main Office* addresses are included.

11.5.2 Challenge 2: Retrieve customer data

As you continue to work with the Adventure Works customer and sales data, you must create queries for reports that have been requested by the sales team.

1. Retrieve a list of all customers and their orders
   - The sales manager wants a list of all customer companies and their contacts (first name and last name), showing the sales order ID and total due for each order they have placed. Customers who have not placed any orders should be included at the bottom of the list with NULL values for the order ID and total due.
2. Retrieve a list of customers with no address
   - A sales employee has noticed that Adventure Works does not have address information for all customers. You must write a query that returns a list of customer IDs, company names, contact names (first name and last name), and phone numbers for customers with no address stored in the database.

11.5.3 Challenge 3: Create a product catalog

The marketing team has asked you to retrieve data for a new product catalog.

1. Retrieve product information by category
   - The product catalog will list products by parent category and subcategory, so you must write a query that retrieves the parent category name, subcategory name, and product name fields for the catalog.

11.6 Challenge Solutions

This section contains suggested solutions for the challenge queries.

11.6.1 Challenge 1

1. Retrieve customer orders:

```
SELECT c.CompanyName, oh.SalesOrderID, oh.TotalDue
FROM SalesLT.Customer AS c
JOIN SalesLT.SalesOrderHeader AS oh
    ON oh.CustomerID = c.CustomerID;
```

2. Retrieve customer orders with addresses:

```
SELECT c.CompanyName,
       a.AddressLine1,
       ISNULL(a.AddressLine2, '') AS AddressLine2,
       a.City,
       a.StateProvince,
       a.PostalCode,
       a.CountryRegion,
       oh.SalesOrderID,
       oh.TotalDue
FROM SalesLT.Customer AS c
JOIN SalesLT.SalesOrderHeader AS oh
    ON oh.CustomerID = c.CustomerID
JOIN SalesLT.CustomerAddress AS ca
    ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
    ON ca.AddressID = a.AddressID
WHERE ca.AddressType = 'Main Office';
```

11.6.2 Challenge 2

1. Retrieve a list of all customers and their orders:

```
SELECT c.CompanyName, c.FirstName, c.LastName,
       oh.SalesOrderID, oh.TotalDue
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS oh
    ON c.CustomerID = oh.CustomerID
ORDER BY oh.SalesOrderID DESC;
```

2. Retrieve a list of customers with no address:

```
SELECT c.CompanyName, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.CustomerAddress AS ca
    ON c.CustomerID = ca.CustomerID
WHERE ca.AddressID IS NULL;
```

11.6.3 Challenge 3

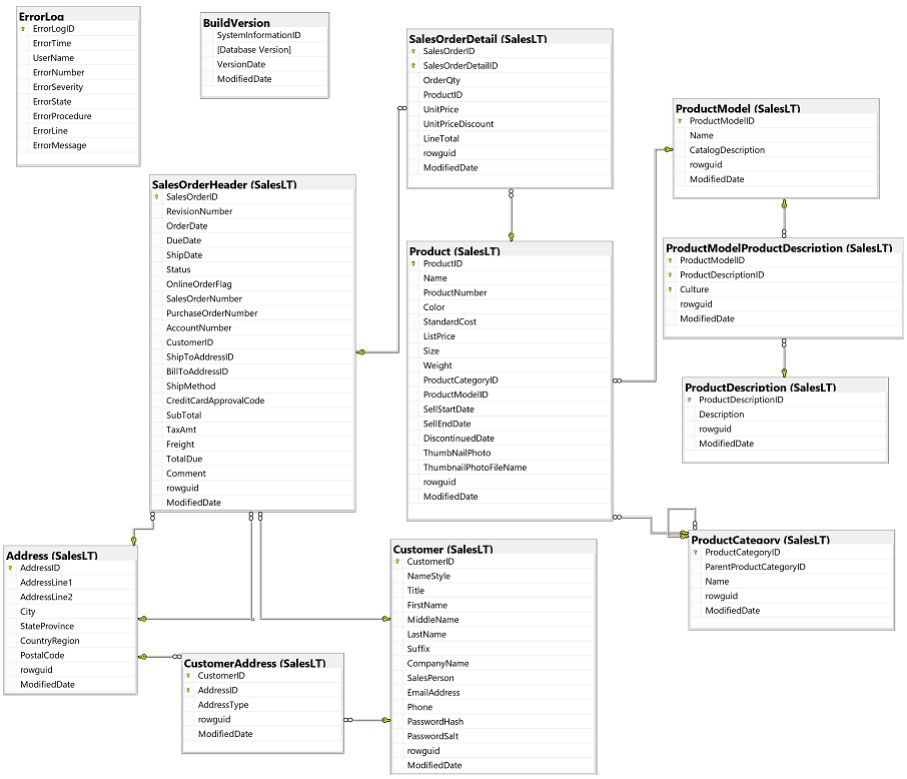
1. Retrieve product information by category:

```
SELECT pcat.Name AS ParentCategory, cat.Name AS SubCategory, prd.Name AS ProductName
FROM SalesLT.ProductCategory pcat
JOIN SalesLT.ProductCategory as cat
    ON pcat.ProductCategoryID = cat.ProductCategoryID
JOIN SalesLT.Product as prd
    ON prd.ProductCategoryID = cat.ProductCategoryID
ORDER BY ParentCategory, SubCategory, ProductName;
```

11.7 lab: title: 'Use Subqueries' module: 'Module 3: Using Joins and Subqueries'

12 Use Subqueries

In this lab, you'll use subqueries to retrieve data from tables in the **adventureworks** database. For your reference, the following diagram shows the tables in the database (you may need to resize the pane to see them clearly).



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

12.1 Use simple subqueries

A subquery is a query that is nested within another query. The subquery is often referred to as the *inner* query, and the query within which it is nested is referred to as the *outer* query.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

```
SELECT MAX(UnitPrice)
FROM SalesLT.SalesOrderDetail;
```

4. Use the **Run** button to run the query, and and after a few seconds, review the results, which consists of the maximum **UnitPrice** in the **SalesLT.SalesOrderDetail** (the highest price for which any individual product has been sold).
5. Modify the query as follows to use the query you just ran as a subquery in an outer query that retrieves products with a **ListPrice** higher than the maximum selling price.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice >
    (SELECT MAX(UnitPrice)
     FROM SalesLT.SalesOrderDetail);
```

6. Run the query and review the results, which include all products that have a **listPrice** that is higher than the maximum price for which any product has been sold.
7. Replace the existing query with the following code:

```
SELECT DISTINCT ProductID
FROM SalesLT.SalesOrderDetail
WHERE OrderQty >= 20;
```

8. Run the query and note that it returns the **ProductID** for each product that has been ordered in quantities of 20 or more.
9. Modify the query as follows to use it in a subquery that finds the names of the products that have been ordered in quantities of 20 or more.

```
SELECT Name FROM SalesLT.Product
WHERE ProductID IN
    (SELECT DISTINCT ProductID
     FROM SalesLT.SalesOrderDetail
     WHERE OrderQty >= 20);
```

10. Run the query and note that it returns the product names.
11. Replace the query with the following code:

```
SELECT DISTINCT Name
FROM SalesLT.Product AS p
JOIN SalesLT.SalesOrderDetail AS o
    ON p.ProductID = o.ProductID
WHERE OrderQty >= 20;
```

12. Run the query and note that it returns the same results. Often you can achieve the same outcome with a subquery or a join, and often a subquery approach can be more easily interpreted by a developer looking at the code than a complex join query because the operation can be broken down into discrete components. In most cases, the performance of equivalent join or subquery operations is similar, but in some cases where existence checks need to be performed, joins perform better.

12.2 Use correlated subqueries

So far, the subqueries we've used have been independent of the outer query. In some cases, you might need to use an inner subquery that references a value in the outer query. Conceptually, the inner query runs once for each row returned by the outer query (which is why correlated subqueries are sometimes referred to as *repeating subqueries*).

1. Replace the existing query with the following code:

```
SELECT od.SalesOrderID, od.ProductID, od.OrderQty
FROM SalesLT.SalesOrderDetail AS od
ORDER BY od.ProductID;
```

2. Run the query and note that the results contain the order ID, product ID, and quantity for each sale of a product.
3. Modify the query as follows to filter it using a subquery in the **WHERE** clause that retrieves the maximum purchased quantity for each product retrieved by the outer query. Note that the inner query references a table alias that is declared in the outer query.

```
SELECT od.SalesOrderID, od.ProductID, od.OrderQty
FROM SalesLT.SalesOrderDetail AS od
WHERE od.OrderQty =
    (SELECT MAX(OrderQty)
     FROM SalesLT.SalesOrderDetail AS d
     WHERE od.ProductID = d.ProductID)
ORDER BY od.ProductID;
```

4. Run the query and review the results, which should only contain product order records for which the quantity ordered is the maximum ordered for that product.
5. Replace the query with the following code:

```
SELECT o.SalesOrderID, o.OrderDate, o.CustomerID
FROM SalesLT.SalesOrderHeader AS o
ORDER BY o.SalesOrderID;
```

6. Run the query and note that it returns the order ID, order date, and customer ID for each order that has been placed.

7. Modify the query as follows to retrieve the company name for each customer using a correlated subquery in the **SELECT** clause.

```
SELECT o.SalesOrderID, o.OrderDate,
      (SELECT CompanyName
      FROM SalesLT.Customer AS c
      WHERE c.CustomerID = o.CustomerID) AS CompanyName
FROM SalesLT.SalesOrderHeader AS o
ORDER BY o.SalesOrderID;
```

8. Run the query, and verify that the company name is returned for each customer found by the outer query.

12.3 Challenges

Now it's your opportunity to try using subqueries to retrieve data.

> **Tip**: Try to determine the appropriate queries for yourself. If you get stuck, suggested answers are provided at the end of this lab.

12.3.1 Challenge 1: Retrieve product price information

Adventure Works products each have a standard cost price that indicates the cost of manufacturing the product, and a list price that indicates the recommended selling price for the product. This data is stored in the **SalesLT.Product** table. Whenever a product is ordered, the actual unit price at which it was sold is also recorded in the **SalesLT.SalesOrderDetail** table. You must use subqueries to compare the cost and list prices for each product with the unit prices charged in each sale.

1. Retrieve products whose list price is higher than the average unit price.
   - Retrieve the product ID, name, and list price for each product where the list price is higher than the average unit price for all products that have been sold.
   - **Tip**: Use the **AVG** function to retrieve an average value.
2. Retrieve Products with a list price of 100 or more that have been sold for less than 100.
   - Retrieve the product ID, name, and list price for each product where the list price is 100 or more, and the product has been sold for less than 100.

12.3.2 Challenge 2: Analyze profitability

The standard cost of a product and the unit price at which it is sold determine its profitability. You must use correlated subqueries to compare the cost and average selling price for each product.

1. Retrieve the cost, list price, and average selling price for each product
   - Retrieve the product ID, name, cost, and list price for each product along with the average unit price for which that product has been sold.
2. Retrieve products that have an average selling price that is lower than the cost.
   - Filter your previous query to include only products where the cost price is higher than the average selling price.

12.4 Challenge Solutions

This section contains suggested solutions for the challenge queries.

12.4.1 Challenge 1

1. Retrieve products whose list price is higher than the average unit price:

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice >
    (SELECT AVG(UnitPrice)
     FROM SalesLT.SalesOrderDetail)
ORDER BY ProductID;
```

2. Retrieve Products with a list price of 100 or more that have been sold for less than 100:

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
```

```
WHERE ProductID IN
    (SELECT ProductID
     FROM SalesLT.SalesOrderDetail
     WHERE UnitPrice < 100.00)
AND ListPrice >= 100.00
ORDER BY ProductID;
```

12.4.2 Challenge 2

1. Retrieve the cost, list price, and average selling price for each product:

```
SELECT p.ProductID, p.Name, p.StandardCost, p.ListPrice,
    (SELECT AVG(o.UnitPrice)
     FROM SalesLT.SalesOrderDetail AS o
     WHERE p.ProductID = o.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS p
ORDER BY p.ProductID;
```

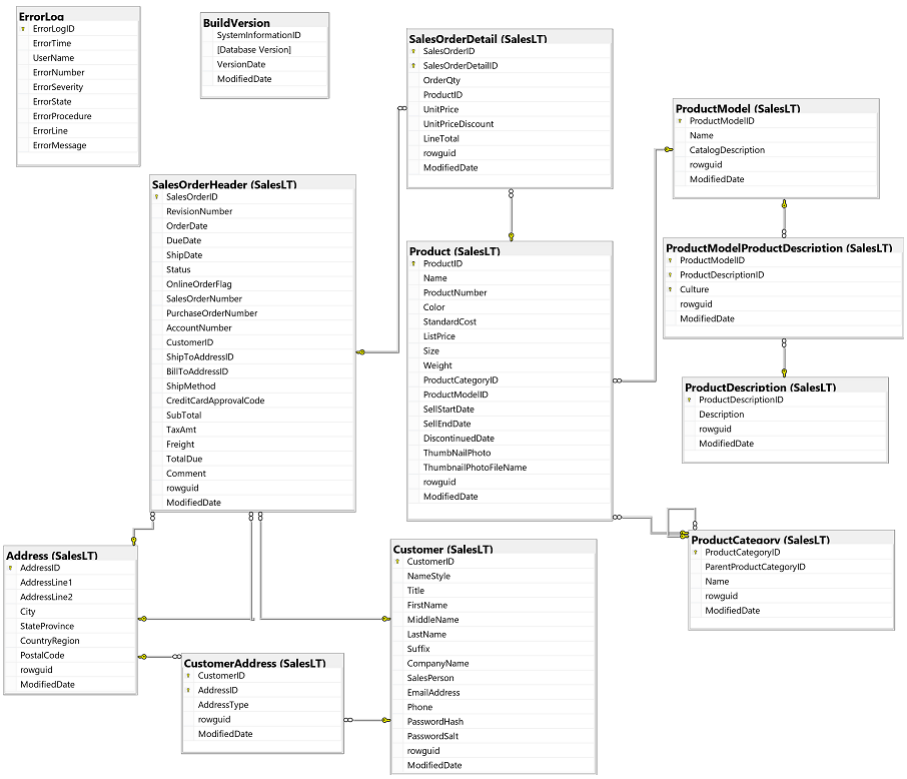
2. Retrieve products that have an average selling price that is lower than the cost:

```
SELECT p.ProductID, p.Name, p.StandardCost, p.ListPrice,
    (SELECT AVG(o.UnitPrice)
    FROM SalesLT.SalesOrderDetail AS o
    WHERE p.ProductID = o.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS p
WHERE StandardCost >
    (SELECT AVG(od.UnitPrice)
     FROM SalesLT.SalesOrderDetail AS od
     WHERE p.ProductID = od.ProductID)
ORDER BY p.ProductID;
```

12.5 lab: title: 'Use Built-in Functions' module: 'Module 4: Using Built-in Functions'

13 Use Built-in Functions

In this lab, you'll use built-in functions to retrieve and aggregate data in the **adventureworks** database. For your reference, the following diagram shows the tables in the database (you may need to resize the pane to see them clearly).



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

13.1 Scalar functions

Transact-SQL provides a large number of functions that you can use to extract additional information from your data. Most of these functions are *scalar* functions that return a single value based on one or more input parameters, often a data field.

Tip: We don't have enough time in this exercise to explore every function available in Transact-SQL. To learn more about the functions covered int his exercise, and more, view the Transact-SQL documentation.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code.

```
SELECT YEAR(SellStartDate) AS SellStartYear, ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

4. Use the **Run** button to run the query, and and after a few seconds, review the results, noting that the **YEAR** function has retrieved the year from the **SellStartDate** field.
5. Modify the query as follows to use some additional scalar functions that operate on *datetime* values.

```
SELECT YEAR(SellStartDate) AS SellStartYear,
       DATENAME(mm,SellStartDate) AS SellStartMonth,
       DAY(SellStartDate) AS SellStartDay,
       DATENAME(dw, SellStartDate) AS SellStartWeekday,
       DATEDIFF(yy,SellStartDate, GETDATE()) AS YearsSold,
       ProductID,
       Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

6. Run the query and review the results.

   Note that the **DATENAME** function returns a different value depending on the *datepart* parameter that is passed to it. In this example, **mm** returns the month name, and **dw** returns the weekday name.

   Note also that the **DATEDIFF** function returns the specified time interval between and start date and an end date. In this case the interval is measured in years (**yy**), and the end date is determined by the **GETDATE** function; which when used with no parameters returns the current date and time.

7. Replace the existing query with the following code.

```
SELECT CONCAT(FirstName + ' ', LastName) AS FullName
FROM SalesLT.Customer;
```

8. Run the query and note that it returns the concatenated first and last name for each customer.

9. Replace the query with the following code to explore some more functions that manipulate string-based values.

```
SELECT UPPER(Name) AS ProductName,
       ProductNumber,
       ROUND(Weight, 0) AS ApproxWeight,
       LEFT(ProductNumber, 2) AS ProductType,
       SUBSTRING(ProductNumber,CHARINDEX('-', ProductNumber) + 1, 4) AS ModelCode,
       SUBSTRING(ProductNumber, LEN(ProductNumber) - CHARINDEX('-', REVERSE(RIGHT(ProductNumber, 3
FROM SalesLT.Product;
```

10. Run the query and note that it returns the following data:

    - The product name, converted to upper case by the **UPPER** function.
    - The product number, which is a string code that encapsulates details of the product.
    - The weight of the product, rounded to the nearest whole number by using the **ROUND** function.
    - The product type, which is indicated by the first two characters of the product number, starting from the left (using the **LEFT** function).
    - The model code, which is extracted from the product number by using the **SUBSTRING** function, which extracts the four characters immediately following the first - character, which is found using the **CHARINDEX** function.
    - The size code, which is extracted using the **SUBSTRING** function to extract the two characters following the last - in the product code. The last - character is found by taking the total length (**LEN**) of the product ID and finding its index (**CHARINDEX**) in the reversed (**REVERSE**) first three characters from the right (**RIGHT**). This example shows how you can combine functions to apply fairly complex logic to extract the values you need.

13.2 Use logical functions

Logical functions are used to apply logical tests to values, and return an appropriate value based on the results of the logical evaluation.

1. Replace the existing query with the following code.

```
SELECT Name, Size AS NumericSize
FROM SalesLT.Product
WHERE ISNUMERIC(Size) = 1;
```

2. Run the query and note that the results only products with a numeric size.

3. Replace the query with the following code, which nests the **ISNUMERIC** function used previously in an **IIF** function; which in turn evaluates the result of the **ISNUMERIC** function and returns *Numeric* if the result is **1** (*true*), and *Non-Numeric* otherwise.

```
SELECT Name, IIF(ISNUMERIC(Size) = 1, 'Numeric', 'Non-Numeric') AS SizeType
FROM SalesLT.Product;
```

4. Run the query and review the results.

5. Replace the query with the following code:

```
SELECT prd.Name AS ProductName,
       cat.Name AS Category,
```

```
       CHOOSE (cat.ParentProductCategoryID, 'Bikes','Components','Clothing','Accessories') AS Produ
FROM SalesLT.Product AS prd
JOIN SalesLT.ProductCategory AS cat
    ON prd.ProductCategoryID = cat.ProductCategoryID;
```

6. Run the query and note that the **CHOOSE** function returns the value in the ordinal position in a list based on the a specified index value. The list index is 1-based so in this query the function returns *Bikes* for category 1, *Components* for category 2, and so on.

13.3 Use aggregate functions

Aggregate functions return an aggregated value, such as a sum, count, average, minimum, or maximum.

1. Replace the existing query with the following code.

```
SELECT COUNT(*) AS Products,
       COUNT(DISTINCT ProductCategoryID) AS Categories,
       AVG(ListPrice) AS AveragePrice
FROM SalesLT.Product;
```

2. Run the query and note that the following aggregations are returned:
   - The number of products in the table. This is returned by using the **COUNT** function to count the number of rows (*).
   - The number of categories. This is returned by using rhe **COUNT** function to count the number of distinct category IDs in the table.
   - The average price of a product. This is returned by using the **AVG** function with the **ListPrice** field.
3. Replace the query with the following code.

```
SELECT COUNT(p.ProductID) AS BikeModels, AVG(p.ListPrice) AS AveragePrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
    ON p.ProductCategoryID = c.ProductCategoryID
WHERE c.Name LIKE '%Bikes';
```

4. Run the query, noting that it returns the number of models and the average price for products with category names that end in ”bikes”.

13.4 Group aggregated results with the GROUP BY clause

Aggregate functions are especially useful when combined with the **GROUP BY** clause to calculate aggregations for different groups of data.

1. Replace the existing query with the following code.

```
SELECT Salesperson, COUNT(CustomerID) AS Customers
FROM SalesLT.Customer
GROUP BY Salesperson
ORDER BY Salesperson;
```

2. Run the query and note that it returns the number of customers assigned to each salesperson.
3. Replace the query with the following code:

```
SELECT c.Salesperson, SUM(oh.SubTotal) AS SalesRevenue
FROM SalesLT.Customer c
JOIN SalesLT.SalesOrderHeader oh
    ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

4. Run the query, noting that it returns the total sales revenue for each salesperson who has completed any sales.
5. Modify the query as follows to use an outer join:

```
SELECT c.Salesperson, ISNULL(SUM(oh.SubTotal), 0.00) AS SalesRevenue
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
    ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

6. Run the query, noting that it returns the sales totals for salespeople who have sold items, and 0.00 for those who haven't.

13.5 Filter groups with the HAVING clause

After grouping data, you may want to filter the results to include only the groups that meet specified criteria. For example, you may want to return only salespeople with more than 100 customers.

1. Replace the existing query with the following code, which you may think would return salespeople with more than 100 customers (but you'd be wrong, as you will see!)

```
SELECT Salesperson, COUNT(CustomerID) AS Customers
FROM SalesLT.Customer
WHERE COUNT(CustomerID) > 100
GROUP BY Salesperson
ORDER BY Salesperson;
```

2. Run the query and note that it returns an error. The **WHERE** clause is applied *before* the aggregations and the **GROUP BY** clause, so you can't use it to filter on the aggregated value.
3. Modify the query as follows to add a **HAVING** clause, which is applied *after* the aggregations and **GROUP BY** clause.

```
SELECT Salesperson, COUNT(CustomerID) AS Customers
FROM SalesLT.Customer
GROUP BY Salesperson
HAVING COUNT(CustomerID) > 100
ORDER BY Salesperson;
```

4. Run the query, and note that it returns only salespeople who have more than 100 customers assigned to them.

13.6 Challenges

Now it's time to try using functions to retrieve data in some queries of your own.

> **Tip**: Try to determine the appropriate queries for yourself. If you get stuck, suggested answers are provided at the end of this lab.

13.6.1 Challenge 1: Retrieve order shipping information

The operations manager wants reports about order shipping based on data in the **SalesLT.SalesOrderHeader** table.

1. Retrieve the order ID and freight cost of each order.
   - Write a query to return the order ID for each order, together with the the **Freight** value rounded to two decimal places in a column named **FreightCost**.
2. Add the shipping method.
   - Extend your query to include a column named **ShippingMethod** that contains the **ShipMethod** field, formatted in lower case.
3. Add shipping date details.
   - Extend your query to include columns named **ShipYear**, **ShipMonth**, and **ShipDay** that contain the year, month, and day of the **ShipDate**. The **ShipMonth** value should be displayed as the month name (for example, *June*)

13.6.2 Challenge 2: Aggregate product sales

The sales manager would like reports that include aggregated information about product sales.

1. Retrieve total sales by product

- Write a query to retrieve a list of the product names from the **SalesLT.Product** table and the total revenue for each product calculated as the sum of **LineTotal** from the **SalesLT.SalesOrderDetail** table, with the results sorted in descending order of total revenue.

2. Filter the product sales list to include only products that cost over 1,000
   - Modify the previous query to include sales totals for products that have a list price of more than 1000.
3. Filter the product sales groups to include only total sales over 20,000
   - Modify the previous query to only include only product groups with a total sales value greater than 20,000.

13.7 Challenge Solutions

This section contains suggested solutions for the challenge queries.

13.7.1 Challenge 1

1. Retrieve the order ID and freight cost of each order:

```
SELECT SalesOrderID,
       ROUND(Freight, 2) AS FreightCost
FROM SalesLT.SalesOrderHeader;
```

2. Add the shipping method:

```
SELECT SalesOrderID,
       ROUND(Freight, 2) AS FreightCost,
       LOWER(ShipMethod) AS ShippingMethod
FROM SalesLT.SalesOrderHeader;
```

3. Add shipping date details:

```
SELECT SalesOrderID,
       ROUND(Freight, 2) AS FreightCost,
       LOWER(ShipMethod) AS ShippingMethod,
       YEAR(ShipDate) AS ShipYear,
       DATENAME(mm, ShipDate) AS ShipMonth,
       DAY(ShipDate) AS ShipDay
FROM SalesLT.SalesOrderHeader;
```

13.7.2 Challenge 2

The product manager would like reports that include aggregated information about product sales.

1. Retrieve total sales by product:

```
SELECT p.Name,SUM(o.LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS o
JOIN SalesLT.Product AS p
    ON o.ProductID = p.ProductID
GROUP BY p.Name
ORDER BY TotalRevenue DESC;
```

2. Filter the product sales list to include only products that cost over 1,000:

```
SELECT p.Name,SUM(o.LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS o
JOIN SalesLT.Product AS p
    ON o.ProductID = p.ProductID
WHERE p.ListPrice > 1000
GROUP BY p.Name
ORDER BY TotalRevenue DESC;
```

3. Filter the product sales groups to include only total sales over 20,000:

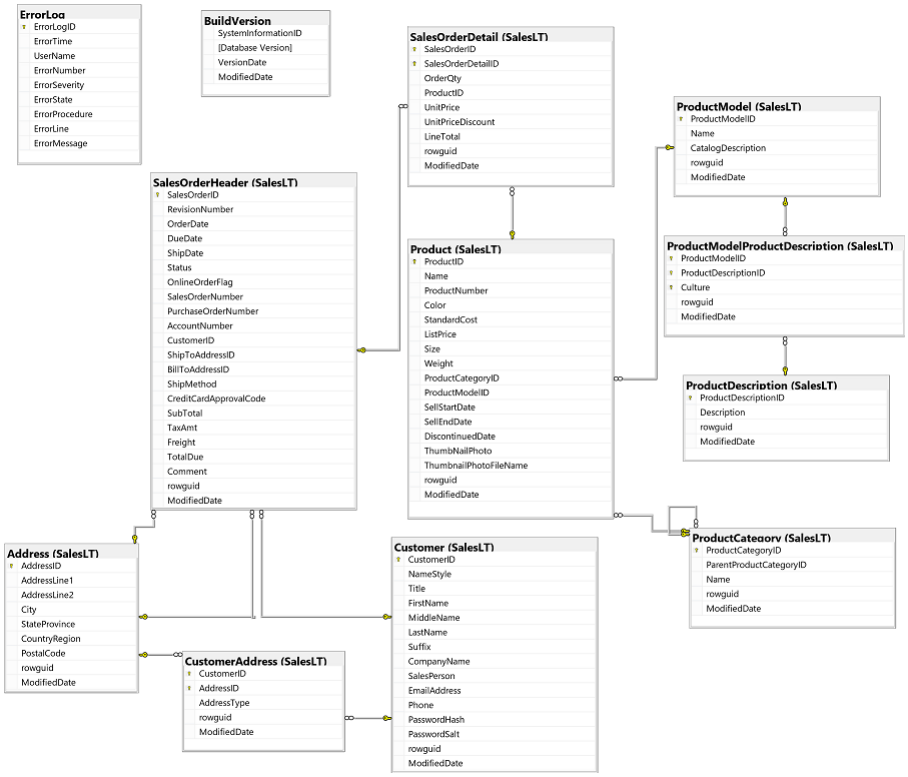
```
SELECT p.Name,SUM(o.LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS o
JOIN SalesLT.Product AS p
```

```
    ON o.ProductID = p.ProductID
WHERE p.ListPrice > 1000
GROUP BY p.Name
HAVING SUM(o.LineTotal) > 20000
ORDER BY TotalRevenue DESC;
```

13.8 lab: title: 'Modify Data' module: 'Module 5: Modifying Data'

14 Modify Data

In this lab, you'll insert, update, and delete data in the **adventureworks** database. For your reference, the following diagram shows the tables in the database (you may need to resize the pane to see them clearly).



Note: If you're familiar with the standard **AdventureWorks** sample database, you may notice that in this lab we are using a simplified version that makes it easier to focus on learning Transact-SQL syntax.

14.1 Insert data

You use the **INSERT** statement to insert data into a table.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the *welcome* page).
2. In the new **SQLQuery_...** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code to create a new table named **SalesLT.CallLog**, which we'll use in this lab.

```
CREATE TABLE SalesLT.CallLog
(
    CallID int IDENTITY PRIMARY KEY NOT NULL,
    CallTime datetime NOT NULL DEFAULT GETDATE(),
    SalesPerson nvarchar(256) NOT NULL,
    CustomerID int NOT NULL REFERENCES SalesLT.Customer(CustomerID),
    PhoneNumber nvarchar(25) NOT NULL,
```

```
    Notes nvarchar(max) NULL
);
```

4. Use the **Run** button to run the code and create the table. Don't worry too much about the details of the **CREATE TABLE** statement - it creates a table with some fields that we'll use in subsequent tasks to insert, update, and delete data.

5. Create a new query, so you have two **SQLQuery_...** panes, and in the new pane, enter the following code to query the **SalesLT.CallLog** you just created.

```
SELECT * FROM SalesLT.CallLog;
```

6. Run the **SELECT** query and view the results, which show the columns in the new table but no rows, because the table is empty.

7. Switch back to the **SQLQuery_...** pane containing the **CREATE TABLE** statement, and replace it with the following **INSERT** statement to insert a new row into the **SalesLT.CallLog** table.

```
INSERT INTO SalesLT.CallLog
VALUES
('2015-01-01T12:30:00', 'adventure-works\pamela0', 1, '245-555-0173', 'Returning call re: enquiry
```

8. Run the query and review the message, which should indicate that 1 row was affected.

9. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that the results contain the row you inserted. The **CallID** column is an *identity* column that is automatically incremented (so the first row has the value **1**), and the remaining columns contain the values you specified in the **INSERT** statement

10. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code to insert another row. This time, the **INSERT** statement takes advantage of the fact that the table has a default value defined for the **CallTime** field, and allows *NULL* values in the **Notes** field.

```
INSERT INTO SalesLT.CallLog
VALUES
(DEFAULT, 'adventure-works\david8', 2, '170-555-0127', NULL);
```

11. Run the query and review the message, which should indicate that 1 row was affected.

12. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that the second row has been inserted, with the default value for the **CallTime** field (the current time when the row was inserted) and *NULL* for the **Notes** field.

13. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code to insert another row. This time, the **INSERT** statement explicitly lists the columns into which the new values will be inserted. The columns not specified in the statement support either default or *NULL* values, so they can be omitted.

```
INSERT INTO SalesLT.CallLog (SalesPerson, CustomerID, PhoneNumber)
VALUES
('adventure-works\jillian0', 3, '279-555-0130');
```

14. Run the query and review the message, which should indicate that 1 row was affected.

15. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that the third row has been inserted, once again using the default value for the **CallTime** field and *NULL* for the **Notes** field.

16. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code, which inserts two rows of data into the **SalesLT.CallLog** table.

```
INSERT INTO SalesLT.CallLog
VALUES
(DATEADD(mi,-2, GETDATE()), 'adventure-works\jillian0', 4, '710-555-0173', NULL),
(DEFAULT, 'adventure-works\shu0', 5, '828-555-0186', 'Called to arrange deliver of order 10987');
```

17. Run the query and review the message, which should indicate that 2 rows were affected.

18. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that two new rows have been added to the table.

19. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code, which inserts the results of a **SELECT** query into the **SalesLT.CallLog** table.

```
INSERT INTO SalesLT.CallLog (SalesPerson, CustomerID, PhoneNumber, Notes)
SELECT SalesPerson, CustomerID, Phone, 'Sales promotion call'
FROM SalesLT.Customer
WHERE CompanyName = 'Big-Time Bike Store';
```

20. Run the query and review the message, which should indicate that 2 rows were affected.
21. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that two new rows have been added to the table. These are the rows that were retrieved by the **SELECT** query.
22. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code, which inserts a row and then uses the **SCOPE_IDENTITY** function to retrieve the most recent *identity* value that has been assigned in the database (to any table), and also the **IDENT_CURRENT** function, which retrieves the latest *identity* value in the specified table.

```
INSERT INTO SalesLT.CallLog (SalesPerson, CustomerID, PhoneNumber)
VALUES
('adventure-works\josé1', 10, '150-555-0127');

SELECT SCOPE_IDENTITY() AS LatestIdentityInDB,
       IDENT_CURRENT('SalesLT.CallLog') AS LatestCallID;
```

23. Run the code and review the results, which should be two numeric values, both the same.
24. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it to validate that the new row that has been inserted has a **CallID** value that matches the *identity* value returned when you inserted it.
25. Switch back to the **SQLQuery_...** pane containing the **INSERT** statement, and replace it with the following code, which enables explicit insertion of *identity* values and inserts a new row with a specified **CallID** value, before disabling explicit *identity* insertion again.

```
SET IDENTITY_INSERT SalesLT.CallLog ON;

INSERT INTO SalesLT.CallLog (CallID, SalesPerson, CustomerID, PhoneNumber)
VALUES
(20, 'adventure-works\josé1', 11, '926-555-0159');

SET IDENTITY_INSERT SalesLT.CallLog OFF;
```

26. Run the code and review the results, which should affect 1 row.
27. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it to validate that a new row has been inserted with the especific **CallID** value you specified in the **INSERT** statement (9).

14.2 Update data

To modify existing rows in a table, use the **UPDATE** statement.

1. On the **SQLQuery_...** pane containing the **INSERT** statement, replace the existing code with the following code.

```
UPDATE SalesLT.CallLog
SET Notes = 'No notes'
WHERE Notes IS NULL;
```

2. Run the **UPDATE** statement and review the message, which should indicate the number of rows affected.
3. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that the rows that previously had *NULL* values for the **Notes** field now contain the text *No notes*.
4. Switch back to the **SQLQuery_...** pane containing the **UPDATE** statement, and replace it with the following code, which updates multiple columns.

```
UPDATE SalesLT.CallLog
SET SalesPerson = '', PhoneNumber = ''
```

5. Run the **UPDATE** statement and note the number of rows affected.
6. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that *all* rows have been updated to remove the **SalesPerson** and **PhoneNumber** fields - this emphasizes the danger of accidentally omitting a **WHERE** clause in an **UPDATE** statement.
7. Switch back to the **SQLQuery_...** pane containing the **UPDATE** statement, and replace it with the following code, which updates the **SalesLT.CallLog** table based on the results of a **SELECT** query.

```
UPDATE SalesLT.CallLog
SET SalesPerson = c.SalesPerson, PhoneNumber = c.Phone
FROM SalesLT.Customer AS c
WHERE c.CustomerID = SalesLT.CallLog.CustomerID;
```

8. Run the **UPDATE** statement and note the number of rows affected.
9. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that the table has been updated using the values returned by the **SELECT** statement.

14.3 Delete data

To delete rows in the table, you generally use the **DELETE** statement; though you can also remove all rows from a table by using the **TRUNCATE TABLE** statement.

1. On the **SQLQuery_...** pane containing the **UPDATE** statement, replace the existing code with the following code.

```
DELETE FROM SalesLT.CallLog
WHERE CallTime < DATEADD(dd, -7, GETDATE());
```

2. Run the **DELETE** statement and review the message, which should indicate the number of rows affected.
3. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that rows with a **CallDate** older than 7 days have been deleted.
4. Switch back to the **SQLQuery_...** pane containing the **DELETE** statement, and replace it with the following code, which uses the **TRUNCATE TABLE** statement to remove all rows in the table.

```
TRUNCATE TABLE SalesLT.CallLog;
```

5. Run the **TRUNCATE TABLE** statement and note the number of rows affected.
6. Switch to the **SQLQuery_...** pane containing the **SELECT** query and run it. Note that *all* rows have been deleted from the table.

14.4 Challenges

Now it's your turn to try modifying some data.

> **Tip**: Try to determine the appropriate code for yourself. If you get stuck, suggested answers are provided at the end of this lab.

14.4.1 Challenge 1: Insert products

Each Adventure Works product is stored in the **SalesLT.Product** table, and each product has a unique **ProductID** identifier, which is implemented as an *identity* column in the **SalesLT.Product** table. Products are organized into categories, which are defined in the **SalesLT.ProductCategory** table. The products and product category records are related by a common **ProductCategoryID** identifier, which is an *identity* column in the **SalesLT.ProductCategory** table.

1. Insert a product
   - Adventure Works has started selling the following new product. Insert it into the **SalesLT.Product** table, using default or *NULL* values for unspecified columns:
     - **Name**: LED Lights
     - **ProductNumber**: LT-L123
     - **StandardCost**: 2.56
     - **ListPrice**: 12.99
     - **ProductCategoryID**: 37
     - **SellStartDate**: *Today's date*

- After you have inserted the product, run a query to determine the **ProductID** that was generated.
- Then run a query to view the row for the product in the **SalesLT.Product** table.

2. Insert a new category with two products
   - Adventure Works is adding a product category for *Bells and Horns* to its catalog. The parent category for the new category is **4** (*Accessories*). This new category includes the following two new products:
     - First product:
       * **Name**: Bicycle Bell
       * **ProductNumber**: BB-RING
       * **StandardCost**: 2.47
       * **ListPrice**: 4.99
       * **ProductCategoryID**: *The **ProductCategoryID** for the new Bells and Horns category*
       * **SellStartDate**: *Today's date*
     - Second product:
       * **Name**: Bicycle Horn
       * **ProductNumber**: BB-PARP
       * **StandardCost**: 1.29
       * **ListPrice**: 3.75
       * **ProductCategoryID**: *The **ProductCategoryID** for the new Bells and Horns category*
       * **SellStartDate**: *Today's date*
   - Write a query to insert the new product category, and then insert the two new products with the appropriate **ProductCategoryID** value.
   - After you have inserted the products, query the **SalesLT.Product** and **SalesLT.ProductCategory** tables to verify that the data has been inserted.

14.4.2 Challenge 2: Update products

You have inserted data for a product, but the pricing details are not correct. You must now update the records you have previously inserted to reflect the correct pricing. Tip: Review the documentation for UPDATE in the Transact-SQL Language Reference.

1. Update product prices
   - The sales manager at Adventure Works has mandated a 10% price increase for all products in the *Bells and Horns* category. Update the rows in the **SalesLT.Product** table for these products to increase their price by 10%.
2. Discontinue products
   - The new LED lights you inserted in the previous challenge are to replace all previous light products. Update the **SalesLT.Product** table to set the **DiscontinuedDate** to today's date for all products in the Lights category (product category ID **37**) other than the LED Lights product you inserted previously.

14.4.3 Challenge 3: Delete products

The Bells and Horns category has not been successful, and it must be deleted from the database.

1. Delete a product category and its products
   - Delete the records for the *Bells and Horns* category and its products. You must ensure that you delete the records from the tables in the correct order to avoid a foreign-key constraint violation.

14.5 Challenge Solutions

This section contains suggested solutions for the challenge queries.

14.5.1 Challenge 1

1. Insert a product:

```
INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost, ListPrice, ProductCategoryID, Sell
VALUES
('LED Lights', 'LT-L123', 2.56, 12.99, 37, GETDATE());

SELECT SCOPE_IDENTITY();
```

```
SELECT * FROM SalesLT.Product
WHERE ProductID = SCOPE_IDENTITY();
```

2. Insert a new category with two products:

```
INSERT INTO SalesLT.ProductCategory (ParentProductCategoryID, Name)
VALUES
(4, 'Bells and Horns');

INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost, ListPrice, ProductCategoryID, Sell
VALUES
('Bicycle Bell', 'BB-RING', 2.47, 4.99, IDENT_CURRENT('SalesLT.ProductCategory'), GETDATE()),
('Bicycle Horn', 'BH-PARP', 1.29, 3.75, IDENT_CURRENT('SalesLT.ProductCategory'), GETDATE());

SELECT c.Name As Category, p.Name AS Product
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory as c
    ON p.ProductCategoryID = c.ProductCategoryID
WHERE p.ProductCategoryID = IDENT_CURRENT('SalesLT.ProductCategory');
```

14.5.2 Challenge 2

1. Update product prices:

```
UPDATE SalesLT.Product
SET ListPrice = ListPrice * 1.1
WHERE ProductCategoryID =
    (SELECT ProductCategoryID
     FROM SalesLT.ProductCategory
     WHERE Name = 'Bells and Horns');
```

2. Discontinue products:

```
UPDATE SalesLT.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductCategoryID = 37
AND ProductNumber <> 'LT-L123';
```

14.5.3 Challenge 3

1. Delete a product category and its products:

```
DELETE FROM SalesLT.Product
WHERE ProductCategoryID =
    (SELECT ProductCategoryID
     FROM SalesLT.ProductCategory
     WHERE Name = 'Bells and Horns');

DELETE FROM SalesLT.ProductCategory
WHERE ProductCategoryID =
    (SELECT ProductCategoryID
     FROM SalesLT.ProductCategory
     WHERE Name = 'Bells and Horns');
```