

2022

# DP-080- TRANSACT-SQL



GIANG TRANVN.COM

# DP-080-TRANSACT-SQL

## 1.1 Labs

Module	Lab
Setup	<a href="#">Lab Environment Setup</a>
Module 1: Getting Started with Transact-SQL	<a href="#">Get Started with Transact-SQL</a>
Module 2: Sorting and Filtering Query Results	<a href="#">Sort and Filter Query Results</a>
Module 3: Using Joins and Subqueries	<a href="#">Query Multiple Tables with Joins</a>
Module 3: Using Joins and Subqueries	<a href="#">Use Subqueries</a>
Module 4: Using Built-in Functions	<a href="#">Use Built-in Functions</a>
Module 5: Modifying Data	<a href="#">Modify Data</a>
Additional exercises	<a href="#">Create queries with table expressions</a>
Additional exercises	<a href="#">Combine query results with set operators</a>
Additional exercises	<a href="#">Use window functions</a>
Additional exercises	<a href="#">Use pivoting and grouping sets</a>
Additional exercises	<a href="#">Introduction to programming with T-SQL</a>
Additional exercises	<a href="#">Create stored procedures in T-SQL</a>
Additional exercises	<a href="#">Implement error handling with T-SQL</a>
Additional exercises	<a href="#">Implement transactions with Transact SQL</a>

## 1.2 Demos

Module	Demo
<b>Module 1: Getting Started with Transact-SQL</b>	<a href="#">Module 1 Demonstrations</a>
<b>Module 2: Sorting and Filtering Query Results</b>	<a href="#">Module 2 Demonstrations</a>
<b>Module 3: Using Joins and Subqueries</b>	<a href="#">Module 3 Demonstrations</a>
<b>Module 4: Using Built-in Functions</b>	<a href="#">Module 4 Demonstrations</a>
<b>Module 5: Modifying Data</b>	<a href="#">Module 5 Demonstrations</a>

## 1 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.

### 1.1 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.

### 1.2 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.

### 1.3 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.

## 1.4 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 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

## 2 Get Started with Transact-SQL

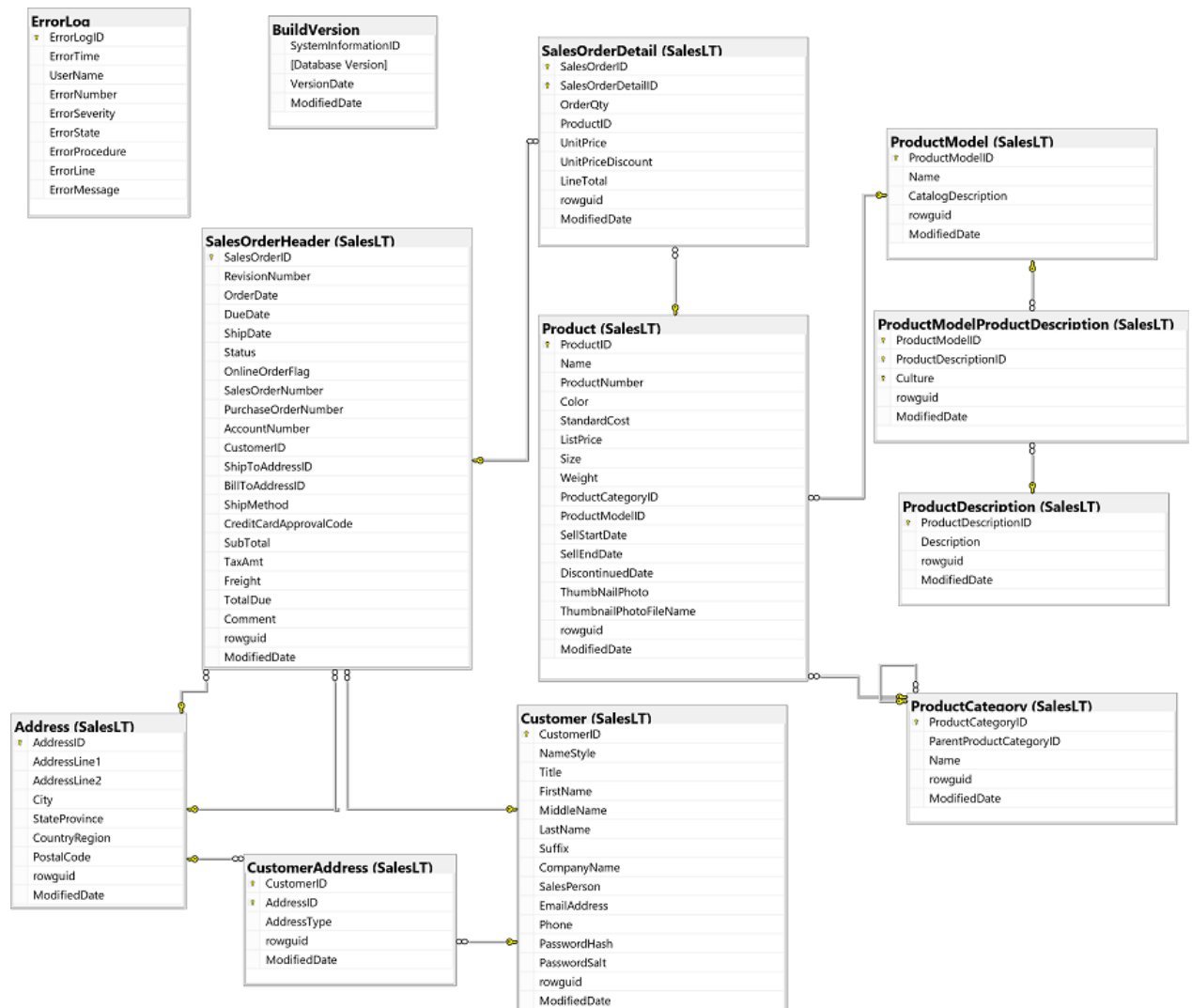
In this lab, you will use some basic **SELECT** queries to retrieve data from the **AdventureWorks** database.

### 2.1 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.

## 2.2 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:

Code

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

4. Use the ☐ **Run** button to run the query, 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:

Code

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

6. Use the ☐ **Run** button to re-run the query, 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:



Code

```
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.

Code

```
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:

Code

```
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.

## 2.3 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:

Code

```
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:

Code

```
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:

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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*.

## 2.4 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:

Code

```
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:

Code

```
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 **NULLIF** function.

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

Code

```
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.

Code

```
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, our **CASE** expression 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.

Code

```
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** predicates 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.

Code

```
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.

## 2.5 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.

### 2.5.1 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.

### 2.5.2 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*).

### 2.5.3 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:

Code

```
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.

Code

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

## 2.6 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 2.6.1 Challenge 1

1. Retrieve customer details:

Code

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

2. Retrieve customer name data:

Code

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

3. Retrieve customer names and phone numbers:

Code

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

### 2.6.2 Challenge 2

1. Retrieve a list of customer companies:

Code

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

2. Retrieve a list of sales order revisions:

Code

```
SELECT SalesOrderNumber + ' (' + STR(RevisionNumber, 1) + ')' AS
OrderRevision,
    CONVERT(nvarchar(30), OrderDate, 102) AS OrderDate
```

```
FROM SalesLT.SalesOrderHeader;
```

### 2.6.3 Challenge 3:

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

Code

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

2. Retrieve primary contact details:

Code

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

3. Retrieve shipping status:

Code

```
SELECT SalesOrderID, OrderDate,
CASE
    WHEN ShipDate IS NULL THEN 'Awaiting Shipment'
    ELSE 'Shipped'
```

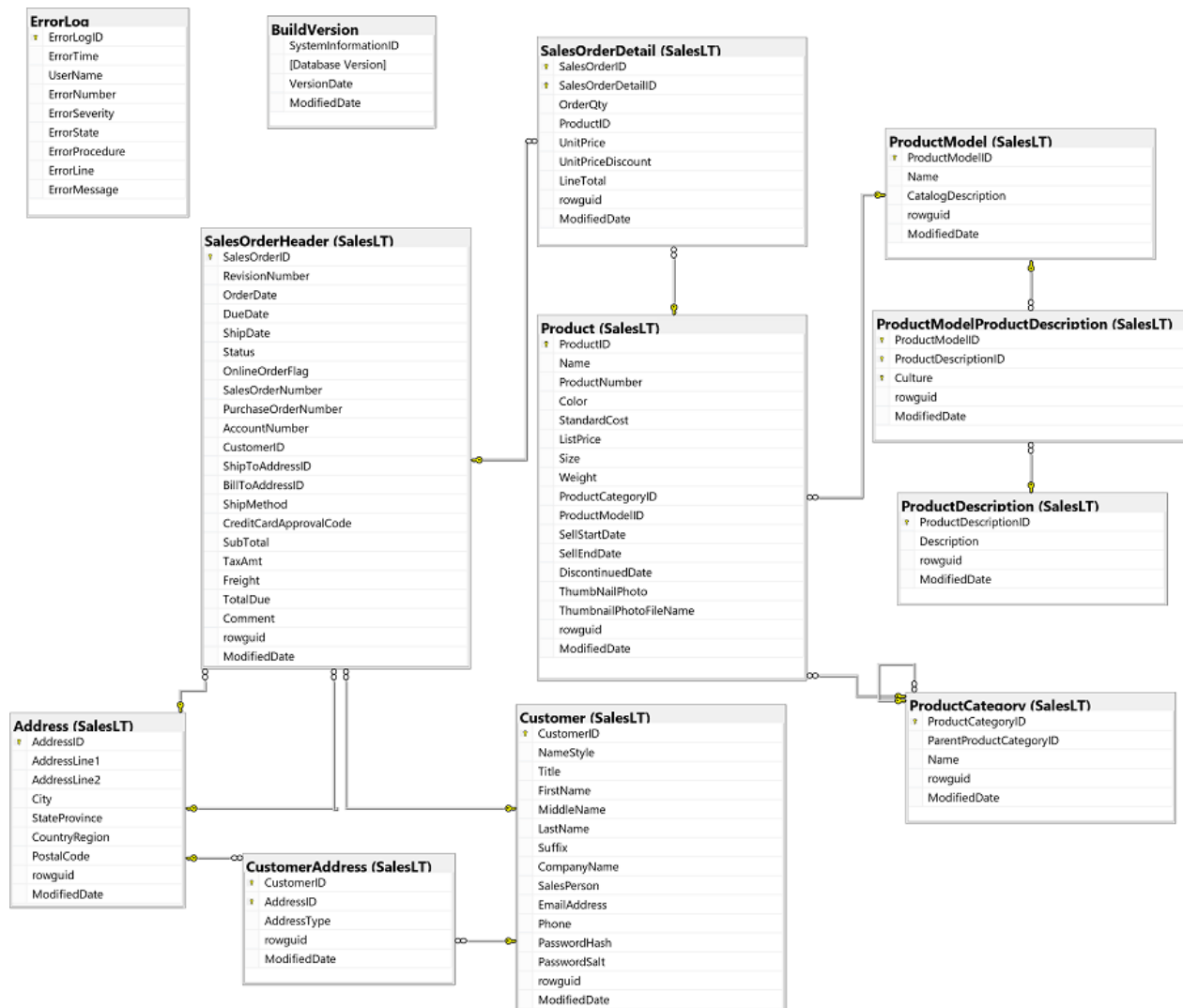
```

END AS ShippingStatus
FROM SalesLT.SalesOrderHeader;

```

### 3 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.

### 3.1 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:

Code

```
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:

Code

```
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**.



Code

```
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**.

Code

```
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**.

Code

```
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**.

### 3.2 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:

Code

```
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.

Code

```
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.

Code

```
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.

### 3.3 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:

Code

```
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.

Code

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

### 3.4 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:

Code

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

4. Use the  **Run** button to run the query, 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.

Code

```
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:

Code

```
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:

Code

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

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

### 3.5 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:

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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

```
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.

Code

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

### 3.6 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.

### 3.6.1 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.

### 3.6.2 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.

### 3.7 Challenge Solutions

This section contains suggested solutions for the challenge queries.

#### 3.7.1 Challenge 1

1. Retrieve a list of cities:

Code

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

2. Retrieve the heaviest products:

Code

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

### 3.7.2 Challenge 2

1. Retrieve product details for product model 1:

Code

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

2. Filter products by color and size:

Code

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

3. Filter products by product number:

Code

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

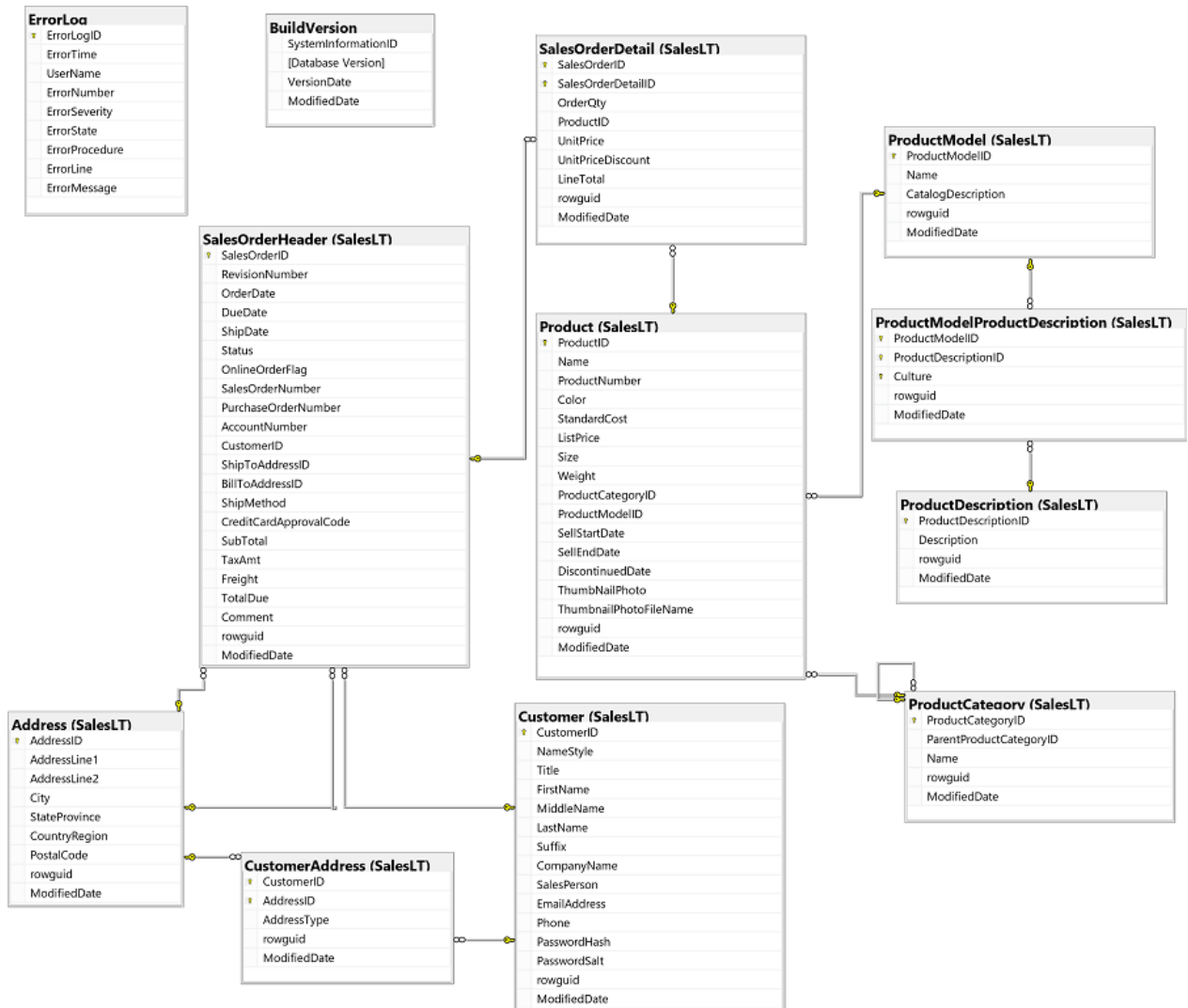
4. Retrieve specific products by product number:

Code

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

## 4 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.

## 4.1 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:

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 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.

Code



```

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:

Code

```

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:

Code

```
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName,  
od.OrderQty, od.UnitPrice, od.LineTotal  
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.

## 4.2 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:

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:

Code

```
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.

Code

```
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.

Code

```
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.

Code

```

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.

### 4.3 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:

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 individual advertisement for each product is emailed to each customer - a strategy that may not endear the company to its customers!).

### 4.4 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**):

Code

```
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.

## 4.5 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.

### 4.5.1 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.

#### 4.5.2 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.

### 4.5.3 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.

## 4.6 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 4.6.1 Challenge 1

1. Retrieve customer orders:

Code

```
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:

Code

```
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';
```

#### 4.6.2 Challenge 2

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

Code

```

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:

Code

```

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;

```

### 4.6.3 Challenge 3

1. Retrieve product information by category:

Code

```

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.ParentProductCategoryID

```

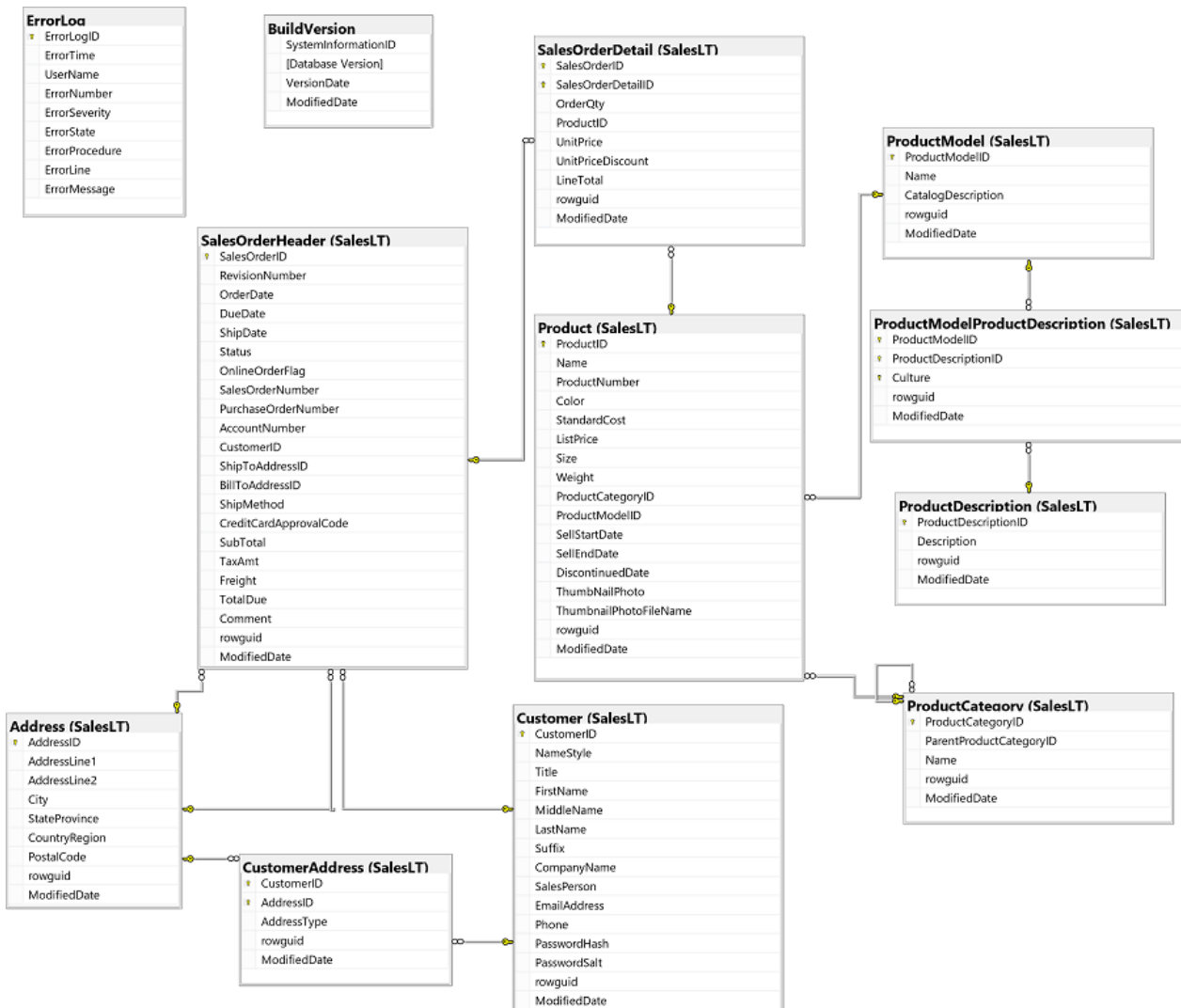
```

JOIN SalesLT.Product as prd
    ON prd.ProductCategoryID = cat.ProductCategoryID
ORDER BY ParentCategory, SubCategory, ProductName;

```

## 5 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.

## 5.1 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:

Code

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

4. Use the ☐ **Run** button to run the query, 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.

Code

```
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:

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.

Code

```
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:

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.

## 5.2 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:

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.

Code

```
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:

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.

Code

```
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.

### 5.3 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.

#### 5.3.1 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.

### 5.3.2 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.

## 5.4 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 5.4.1 Challenge 1

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

Code

```
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:

Code

```
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;
```

### 5.4.2 Challenge 2

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

Code

```
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;
```

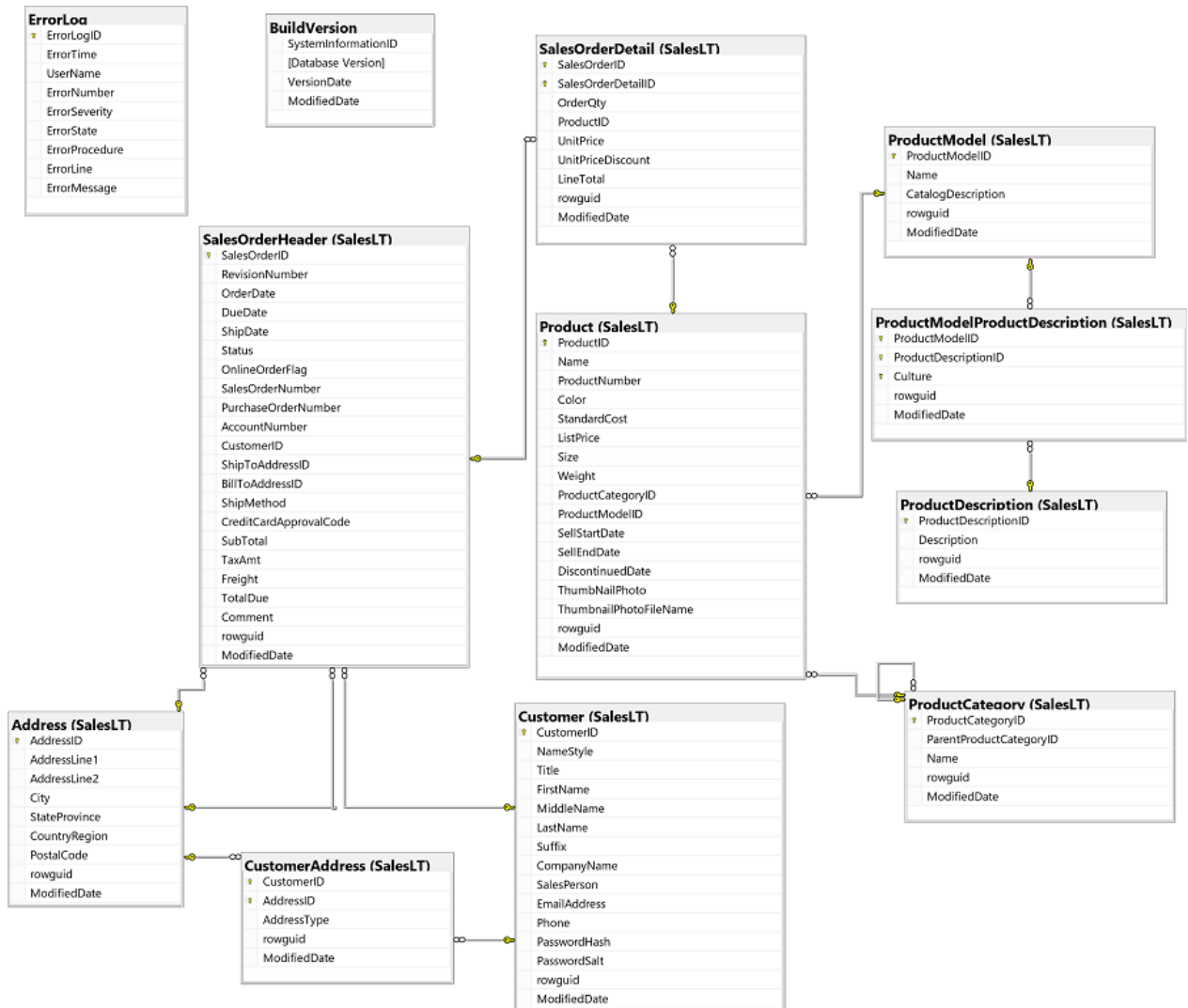
2. Retrieve products that have an average selling price that is lower than the cost:

## Code

```
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;
```

## 6 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.

## 6.1 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 in this 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.

Code

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

4. Use the  **Run** button to run the query, 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.

Code

```
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.

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.



## Code

```

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))) + 2, 2) AS
SizeCode
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.

## 6.2 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.

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.

Code

```
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:

Code

```
SELECT prd.Name AS ProductName,
       cat.Name AS Category,
       CHOOSE                                (cat.ParentProductCategoryID,
       'Bikes','Components','Clothing','Accessories') AS ProductType
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.

### 6.3 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.

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 the **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.

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”.

## 6.4 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.

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:

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:

Code

```
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.

## 6.5 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!)

Code

```
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.

Code

```
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.

## 6.6 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.

### 6.6.1 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 **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*)

### 6.6.2 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.

## 6.7 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 6.7.1 Challenge 1

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

Code

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

2. Add the shipping method:

Code

```
SELECT SalesOrderID,  
       ROUND(Freight, 2) AS FreightCost,  
       LOWER(ShipMethod) AS ShippingMethod
```

```
FROM SalesLT.SalesOrderHeader;
```

3. Add shipping date details:

Code

```
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;
```

### 6.7.2 Challenge 2

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

1. Retrieve total sales by product:

Code

```
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:

Code

```
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;
```

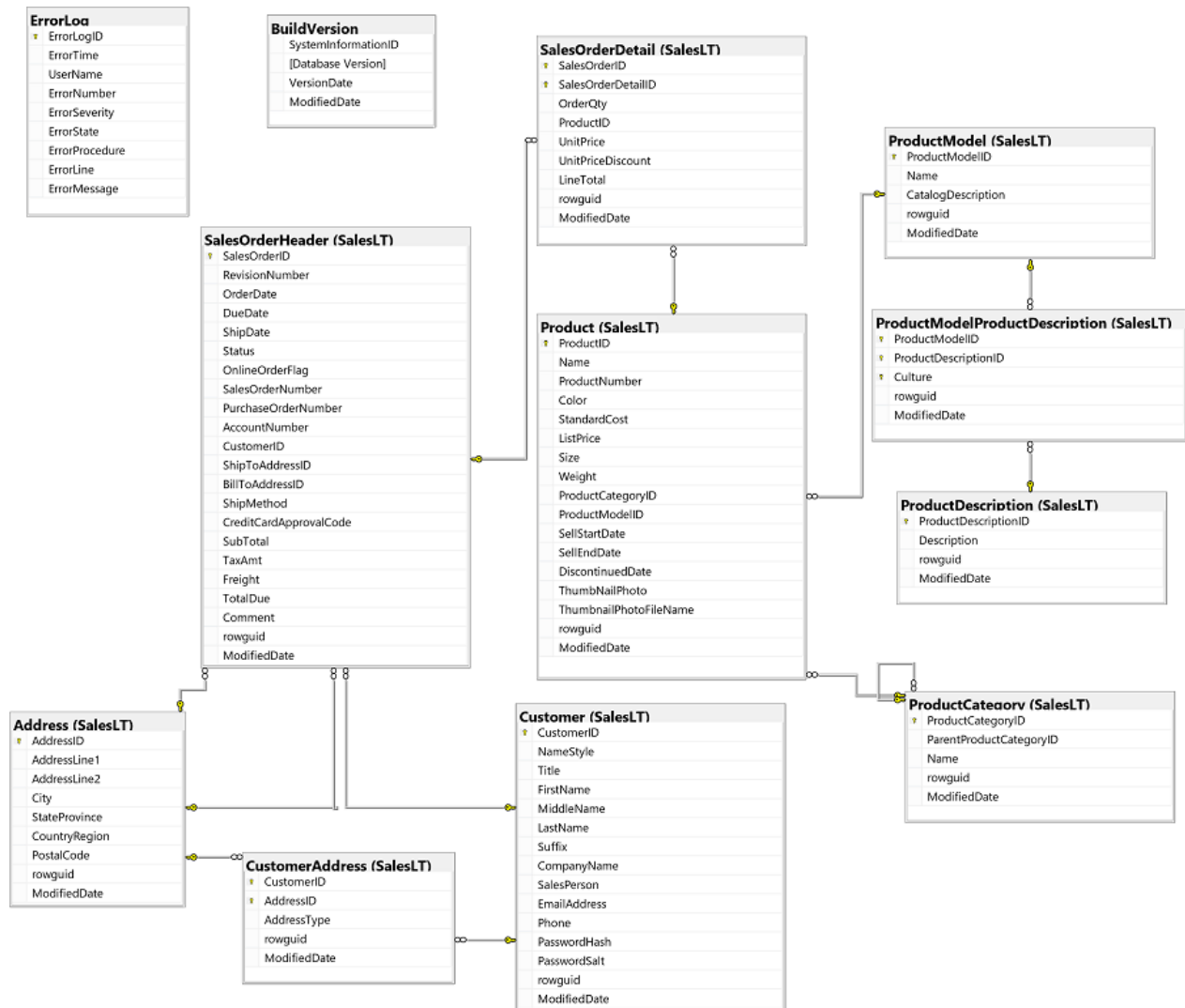
3. Filter the product sales groups to include only total sales over 20,000:

Code

```
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;
```

## 7 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.


### 7.1 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.

Code

```
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.

Code

```
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.

Code

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

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.

Code

```
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.

Code

```
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.

## Code

```
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.

Code

```
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.

Code

```
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.

## Code

```
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 specific **CallID** value you specified in the **INSERT** statement (9).

## 7.2 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.

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.

Code

```
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.

Code

```
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.

### 7.3 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.

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.

Code

```
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.

## 7.4 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.

### 7.4.1 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.

### 7.4.2 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.

### 7.4.3 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.



## 7.5 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 7.5.1 Challenge 1

1. Insert a product:

Code

```
INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost,
ListPrice, ProductCategoryID, SellStartDate)
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:

Code

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

```

INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost,
ListPrice, ProductCategoryID, SellStartDate)
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');

```

### 7.5.2 Challenge 2

1. Update product prices:

Code

```

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

```

## 2. Discontinue products:

Code

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

### 7.5.3 Challenge 3

## 1. Delete a product category and its products:

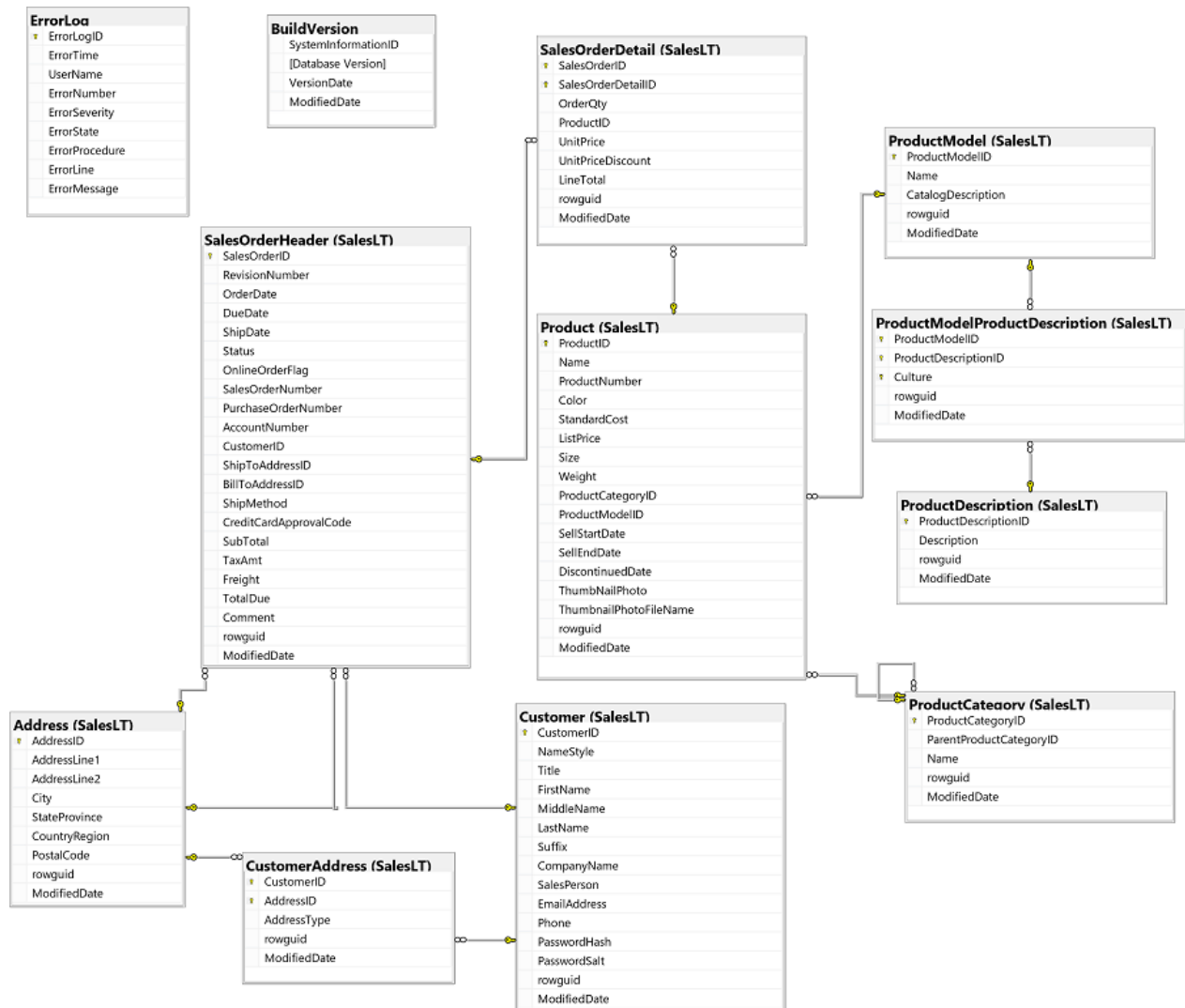
Code

```
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');
```

## 8 Create queries with table expressions

In this lab, you'll use table expressions to query 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.

### 8.1 Create a view

You use the **CREATE VIEW** statement to create a view.

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. After you've connected to your database, you can query it. Let's fetch some data. Enter the following query to retrieve all products that are classified as road bikes (*ProductCategoryID=6*) from the **SalesLT.Products** table:

Code

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ProductCategoryID = 6;
```

4. Select ☐ **Run** to run your query.
5. The query returned all products that are categorized as road bikes. But what if you wanted to use a view for this data to ensure applications don't need to access the underlying table to fetch it? Replace your previous code with the code shown below:

Code

```
CREATE VIEW SalesLT.vProductsRoadBikes AS
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
```

```
WHERE ProductCategoryID = 6;
```

6. This code creates a view called **vProductsRoadBikes** for all road bikes. Select ☐ **Run** to run the code and create the view.

## 8.2 Query a view

You've created your view. Now you can use it. For example, you can use your view to get a list of any road bikes based on their **ListPrice**.

1. In the query editor, replace the code you entered previously with the following code:

Code

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.vProductsRoadBikes
WHERE ListPrice < 1000;
```

2. Select ☐ **Run**.
3. Review the results. You've queried your view and retrieved a list of any road bikes that have a **ListPrice** under 1000. Your query uses your view as a source for the data. This means your applications can use your view for specific searches like this, and won't need to access the underlying table to fetch the data they need.

## 8.3 Use a derived table

Sometimes you might end up having to rely on complex queries. You can use derived tables in place of those complex queries to avoid adding to their complexity.

1. In the query editor, replace the code you entered previously with the following code:

Code

```
SELECT ProductID, Name, ListPrice,
       CASE WHEN ListPrice > 1000 THEN N'High' ELSE N'Normal' END
AS PriceType
FROM SalesLT.Product;
```

2. Select ☐ **Run**.
3. The query calculates whether the price of a product is considered high or normal. But you'd like to be able to further build on this query based on additional criteria, without further adding to its complexity. In order to do this, you can create a derived table for it. Replace the previous code with the code below:

Code

```
SELECT      DerivedTable.ProductID,      DerivedTable.Name,
DerivedTable.ListPrice
FROM
(
    SELECT
        ProductID, Name, ListPrice,
```

```

CASE WHEN ListPrice > 1000 THEN N'High' ELSE N'Normal' END
AS PriceType
FROM SalesLT.Product
) AS DerivedTable
WHERE DerivedTable.PriceType = N'High';

```

4. Select ☐ **Run**.
5. You've created derived table based on your previous query. Your new code uses that derived table and fetches the **ProductID**, **Name**, and **ListPrice** of products that have a **PriceType** of *High* only. Your derived table enabled you to easily build on top of your initial query based on your additional criteria, without making the initial query any more complex.

## 8.4 Challenges

Now it's your turn to use table expressions.

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

### 8.4.1 Challenge 1: Create a view

Adventure Works is forming a new sales team located in Canada. The team wants to create a map of all of the customer addresses in Canada. This team will need access to address details on Canadian customers only. Your manager has asked you to make sure that the team can get the data they require, but ensure that they don't access the underlying source data when getting their information.

To carry out the task do the following:



1. Write a Transact-SQL query to create a view for customer addresses in Canada.
  - Create a view based on the following columns in the **SalesLT.Address** table:
    - **AddressLine1**
    - **City**
    - **StateProvince**
    - **CountryRegion**
  - In your query, use the **CountryRegion** column to filter for addresses located in *Canada* only.
2. Query your new view.
  - Fetch the rows in your newly created view to ensure it was created successfully. Notice that it only shows address in Canada.

#### 8.4.2 Challenge 2: Use a derived table

The transportation team at Adventure Works wants to optimize its processes. Products that weigh more than 1000 are considered to be heavy products, and will also need to use a new transportation method if their list price is over 2000. You've been asked to classify products according to their weight, and then provide a list of products that meet both these weight and list price criteria.

To help, you'll:

1. Write a query that classifies products as heavy and normal based on their weight.
  - Use the **Weight** column to decide whether a product is heavy or normal.
2. Create a derived table based on your query

- Use your derived table to find any heavy products with a list price over 2000.
- Make sure to select the following columns: **ProductID, Name, Weight, ListPrice.**

## 8.5 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 8.5.1 Challenge 1

1. Write a Transact-SQL query to create a view for customer addresses in Canada.

Code

```
CREATE VIEW SalesLT.vAddressCA AS
SELECT AddressLine1, City, StateProvince, CountryRegion
FROM SalesLT.Address
WHERE CountryRegion = 'Canada';
```

2. Query your new view.

Code

```
SELECT * FROM SalesLT.vAddressCA;
```

### 8.5.2 Challenge 2

1. Write a query that classifies products as heavy and normal based on their weight.

Code

```
SELECT ProductID, Name, Weight, ListPrice,
       CASE WHEN Weight > 1000 THEN N'Heavy' ELSE N'Normal' END
AS WeightType
FROM SalesLT.Product;
```

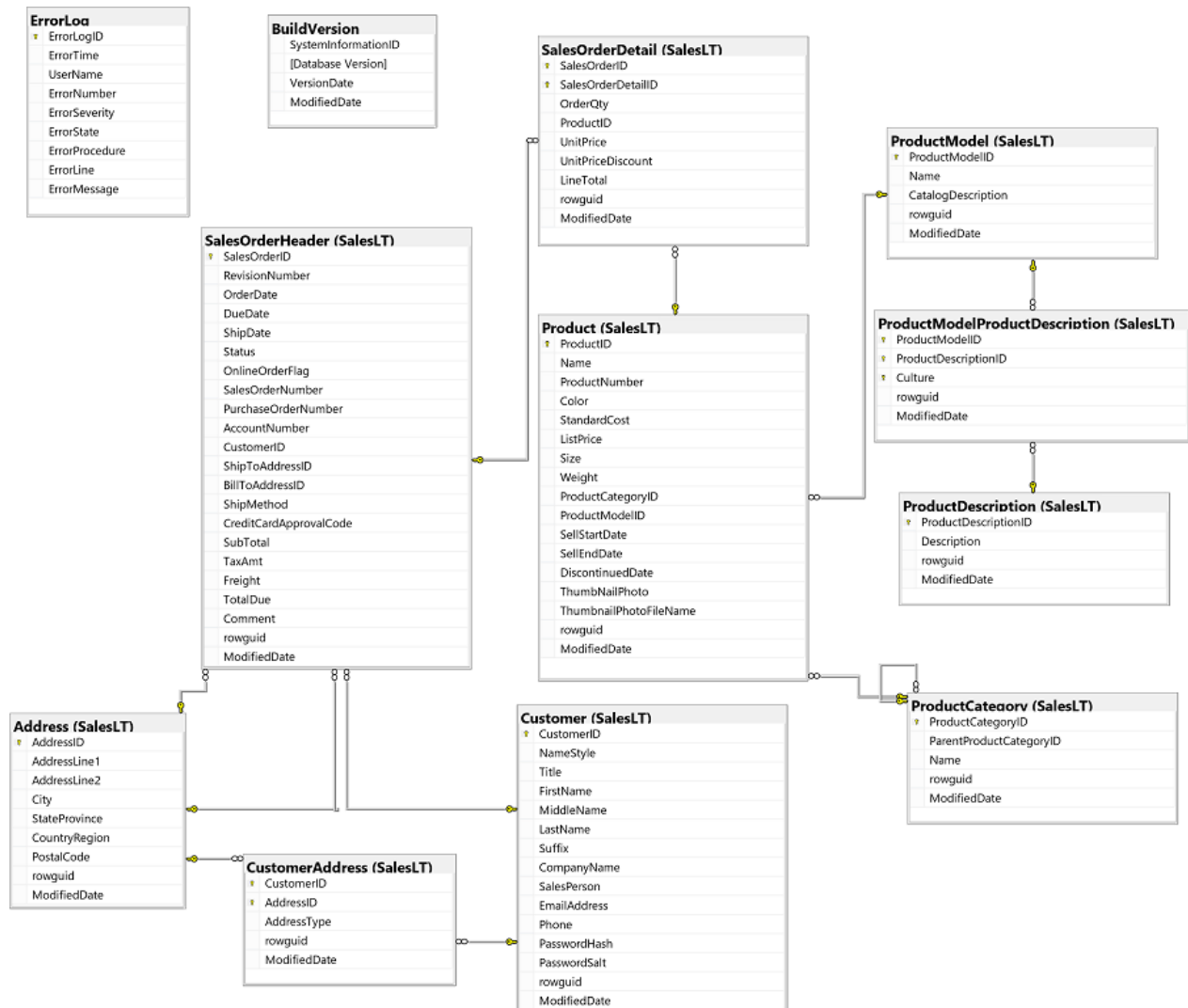
2. Create a derived table based on your query.

Code

```
SELECT      DerivedTable.ProductID,      DerivedTable.Name,
DerivedTable.Weight, DerivedTable.ListPrice
FROM
(
    SELECT ProductID, Name, Weight, ListPrice,
           CASE WHEN Weight > 1000. THEN N'Heavy' ELSE N'Normal'
END AS WeightType
    FROM SalesLT.Product
) AS DerivedTable
WHERE      DerivedTable.WeightType      =      N'Heavy'      AND
DerivedTable.ListPrice > 2000;
```

## 9 Combine query results with set operators

In this lab, you will use set operators to retrieve results from 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.

### 9.1 Write a query that uses the UNION operator


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:

Code

```

SELECT CompanyName
FROM SalesLt.Customer
WHERE CustomerID BETWEEN 1 and 20000
UNION
SELECT CompanyName
FROM SalesLt.Customer
WHERE CustomerID BETWEEN 20000 and 40000;

```

4. Highlight the T-SQL code and select  **Run**. Notice that the result set contains **CompanyNames** from both result sets.

## 9.2 Write a query that uses the INTERSECT operator

Now let's try a query using the INTERSECT operator.

1. In the query editor, below the existing code, enter the following code:

Code

```

-- Prepare tables
DECLARE @t1 AS table

```

```

(Name nvarchar(30) NOT NULL);
DECLARE @t2 AS table
([Name] nvarchar(30) NOT NULL);
INSERT INTO @t1 ([Name])
VALUES
    (N'Daffodil'),
    (N'Camembert'),
    (N'Neddy'),
    (N'Smudge'),
    (N'Molly');
INSERT INTO @t2 ([Name])
VALUES
    (N'Daffodil'),
    (N'Neddy'),
    (N'Molly'),
    (N'Spooky');
SELECT [Name]
FROM @t1
INTERSECT
SELECT [Name]
FROM @t2
ORDER BY [Name];

```

2. Highlight the code and select  **Run** to execute it. Notice that values in both **t1** and **t2** are returned.

### 9.3 Write a query that uses the CROSS APPLY operator

Now you will write a table-valued function to return the product category and quantity ordered by specific customers. You will pass the **CustomerID** from the select statement to the table-valued function in a CROSS APPLY statement.

1. In the query editor, enter the following code:

Code

```
CREATE OR ALTER FUNCTION dbo.ProductSales (@CustomerID int)
RETURNS TABLE
RETURN
    SELECT C.[Name] AS 'Category', SUM(D.OrderQty) AS 'Quantity'
    FROM SalesLT.SalesOrderHeader AS H
    INNER JOIN SalesLT.SalesOrderDetail AS D
    ON H.SalesOrderID = D.SalesOrderID
    INNER JOIN SalesLT.Product AS P
    ON D.ProductID = P.ProductID
    INNER JOIN SalesLT.ProductCategory AS C
    ON P.ProductCategoryID = C.ProductCategoryID
    WHERE H.CustomerID = @CustomerID
    GROUP BY C.[Name]
```

2. Highlight the code and select ☐ **Run** to execute it.
3. Then, enter the following code on a new line:

Code

```
SELECT C.CustomerID, C.CompanyName, P.Category, P.Quantity
```

```
FROM SalesLT.Customer AS C
CROSS APPLY dbo.ProductSales(C.CustomerID) AS P;
```

4. Highlight the code and select  **Run** to execute it.

## 9.4 Challenges

Now it's your turn to use set operators.

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

### 9.4.1 Challenge 1: Return all company names

Amend the T-SQL code containing the UNION operator, to return ALL company names, including duplicates.

### 9.4.2 Challenge 2: Return names from t1

Amend the T-SQL code containing the INTERSECT operator to return names from **t1** that do not appear in **t2**.

## 9.5 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 9.5.1 Challenge 1

Code

```
SELECT CompanyName
FROM SalesLT.Customer
WHERE CustomerID BETWEEN 1 and 20000
UNION ALL
```



```
SELECT CompanyName
FROM SalesLt.Customer
WHERE CustomerID BETWEEN 20000 and 40000;
```

### 9.5.2 Challenge 2

Code

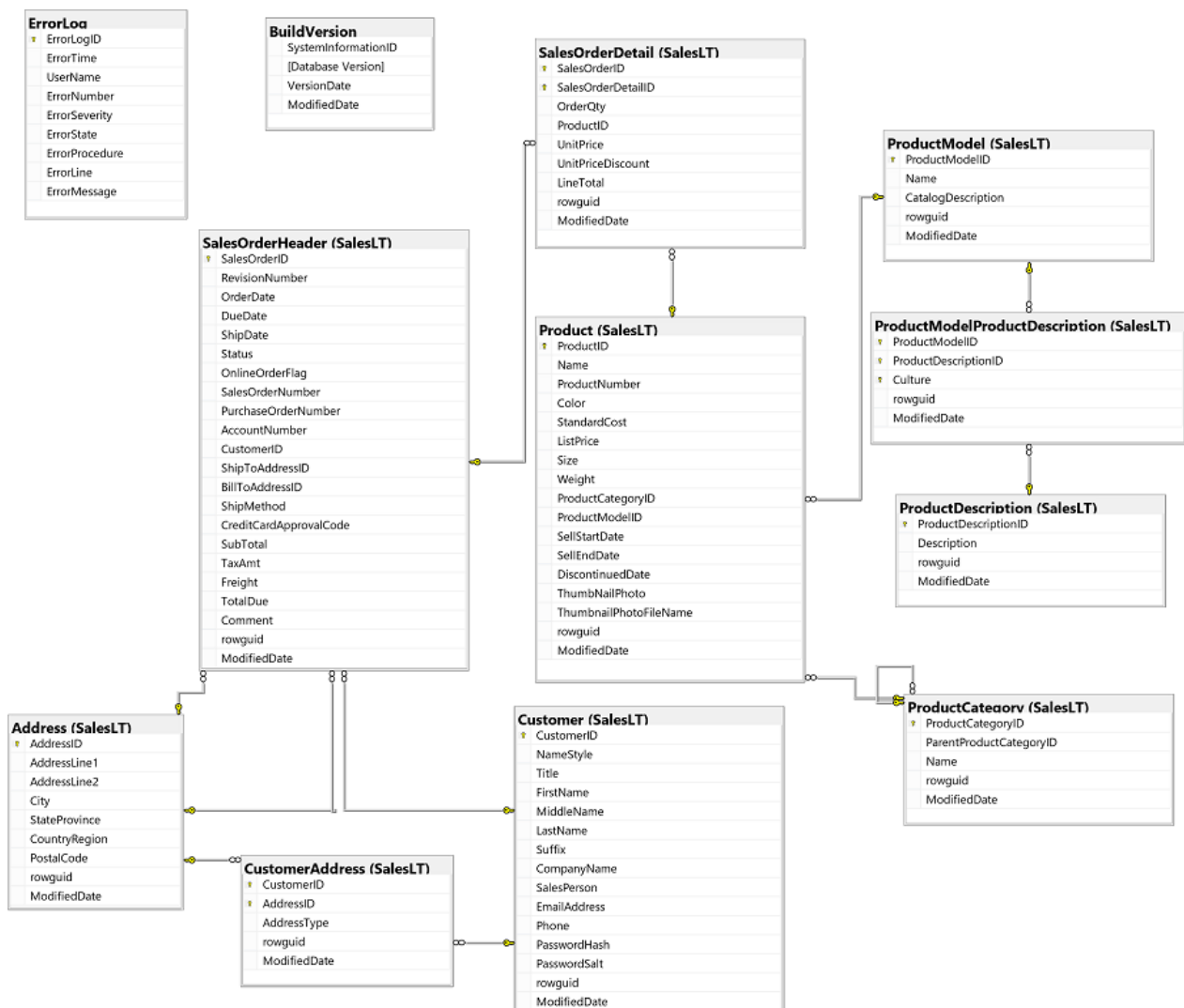
```
DECLARE @t1 AS table
(Name nvarchar(30) NOT NULL);
DECLARE @t2 AS table
([Name] nvarchar(30) NOT NULL);
INSERT INTO @t1 ([Name])
VALUES
    (N'Daffodil'),
    (N'Camembert'),
    (N'Neddy'),
    (N'Smudge'),
    (N'Molly');
INSERT INTO @t2 ([Name])
VALUES
    (N'Daffodil'),
    (N'Neddy'),
    (N'Molly'),
    (N'Spooky');
SELECT [Name]
FROM @t1
EXCEPT
SELECT [Name]
```

FROM @t2

ORDER BY [Name];

## 10 Use window functions

In this lab, you'll run window functions on 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 Ranking function

In this exercise you will create a query that uses a window function to return a ranking value. The query uses a CTE (common table expression) called **sales**. You then use the **sales** CTE to add the RANK window function.

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. Copy the following T-SQL code into the query window, highlight it and select ☐ **Run**.

Code

```
WITH sales AS
(
    SELECT  C.Name  AS  'Category',  CAST(SUM(D.LineTotal)  AS
numeric(12, 2)) AS 'SalesValue'
    FROM SalesLT.SalesOrderDetail AS D
    INNER JOIN SalesLT.Product AS P
        ON D.ProductID = P.ProductID
    INNER JOIN SalesLT.ProductCategory AS C
        ON P.ProductCategoryID = C.ProductCategoryID
```

```

WHERE C.ParentProductCategoryID = 4
GROUP BY C.Name
)
SELECT Category, SalesValue, RANK() OVER(ORDER BY SalesValue
DESC) AS 'Rank'
FROM sales
ORDER BY Category;

```

The product categories now have a rank number according to the **SalesValue** for each category. Notice that the RANK function required the rows to be ordered by **SalesValue**, but the final result set was ordered by **Category**.

## 10.2 Offset function

In this exercise you will create a new table called **Budget** populated with budget values for five years. You will then use the LAG window function to return each year's budget, together with the previous year's budget value.

1. In the query editor, under the existing code enter the following code:

Code

```

CREATE TABLE dbo.Budget
(
    [Year] int NOT NULL PRIMARY KEY,
    Budget int NOT NULL
);

```

```

INSERT INTO dbo.Budget ([Year], Budget)
VALUES
    (2017, 14600),
    (2018, 16300),
    (2019, 18200),
    (2020, 21500),
    (2021, 22800);

SELECT [Year], Budget, LAG(Budget, 1, 0) OVER (ORDER BY [Year])
AS 'Previous'
FROM dbo.Budget
ORDER BY [Year];

```

2. Highlight the code and select ☐ **Run**.

### 10.3 Aggregation function

In this exercise you will create a query that uses PARTITION BY to count the number of subcategories in each category.

1. In the query editor, under the existing code enter the following code to return a count of products in each category:

Code

```

SELECT  C.Name  AS  'Category',  SC.Name  AS  'Subcategory',
COUNT(SC.Name) OVER (PARTITION BY C.Name) AS 'SubcatCount'
FROM SalesLT.SalesOrderDetail AS D

```

```

INNER JOIN SalesLT.Product AS P
    ON D.ProductID = P.ProductID
INNER JOIN SalesLT.ProductCategory AS SC
    ON P.ProductCategoryID = SC.ProductCategoryID
INNER JOIN SalesLT.ProductCategory AS C
    ON SC.ParentProductCategoryID = C.ProductCategoryID
GROUP BY C.Name, SC.Name
ORDER BY C.Name, SC.Name;

```

2. Highlight the code and select ☐ **Run**.

## 10.4 Challenges

Now it's your turn to use window functions.

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

### 10.4.1 Challenge 1: Return a RANK value for products

Amend the T-SQL code with the RANK clause so that it returns a Rank value for products within each category.

### 10.4.2 Challenge 2: Return the next year's budget value

Using the **Budget** table you have already created, amend the SELECT statement to return the following year's budget value as "Next".

### 10.4.3 Challenge 3: Return the first and last budget values for each year

Using the **Budget** table you have already created, amend the select statement to return the first budget value in one column, and the last budget value in another column, where budget values are ordered by year in ascending order.

#### 10.4.4 Challenge 4: Count the products in each category

Amend the code containing the aggregation function to return a count of products by category.

### 10.5 Challenge Solutions

This section contains suggested solutions for the challenge queries.

#### 10.5.1 Challenge 1

Code

```
WITH sales AS
(
    SELECT    C.Name    AS    'Category',    SC.Name    AS    'Subcategory',
    CAST(SUM(D.LineTotal) AS numeric(12, 2)) AS 'SalesValue'
    FROM SalesLT.SalesOrderDetail AS D
    INNER JOIN SalesLT.Product AS P
        ON D.ProductID = P.ProductID
    INNER JOIN SalesLT.ProductCategory AS SC
        ON P.ProductCategoryID = SC.ProductCategoryID
    INNER JOIN SalesLT.ProductCategory AS C
        ON SC.ParentProductCategoryID = C.ProductCategoryID
    GROUP BY C.Name, SC.Name
)
SELECT Category, Subcategory, SalesValue, RANK() OVER(PARTITION BY
Category ORDER BY SalesValue DESC) AS 'Rank'
FROM sales
    ORDER BY Category, SalesValue DESC;
```

### 10.5.2 Challenge 2

Code

```
SELECT [Year], Budget, LEAD(Budget, 1, 0) OVER (ORDER BY [Year]) AS  
'Next'  
FROM dbo.Budget  
ORDER BY [Year];
```

### 10.5.3 Challenge 3

Code

```
SELECT [Year], Budget,  
FIRST_VALUE(Budget) OVER (ORDER BY [Year]) AS 'First_Value',  
LAST_VALUE(Budget) OVER (ORDER BY [Year] ROWS BETWEEN  
CURRENT ROW AND UNBOUNDED FOLLOWING) AS 'Last_Value'  
FROM dbo.Budget  
ORDER BY [Year];
```

### 10.5.4 Challenge 4

Code

```
SELECT C.Name AS 'Category', SC.Name AS 'Subcategory', COUNT(P.Name)  
OVER (PARTITION BY C.Name) AS 'ProductCount'  
FROM SalesLT.SalesOrderDetail AS D  
INNER JOIN SalesLT.Product AS P  
ON D.ProductID = P.ProductID  
INNER JOIN SalesLT.ProductCategory AS SC  
ON P.ProductCategoryID = SC.ProductCategoryID  
INNER JOIN SalesLT.ProductCategory AS C
```



```

ON SC.ParentProductCategoryID = C.ProductCategoryID

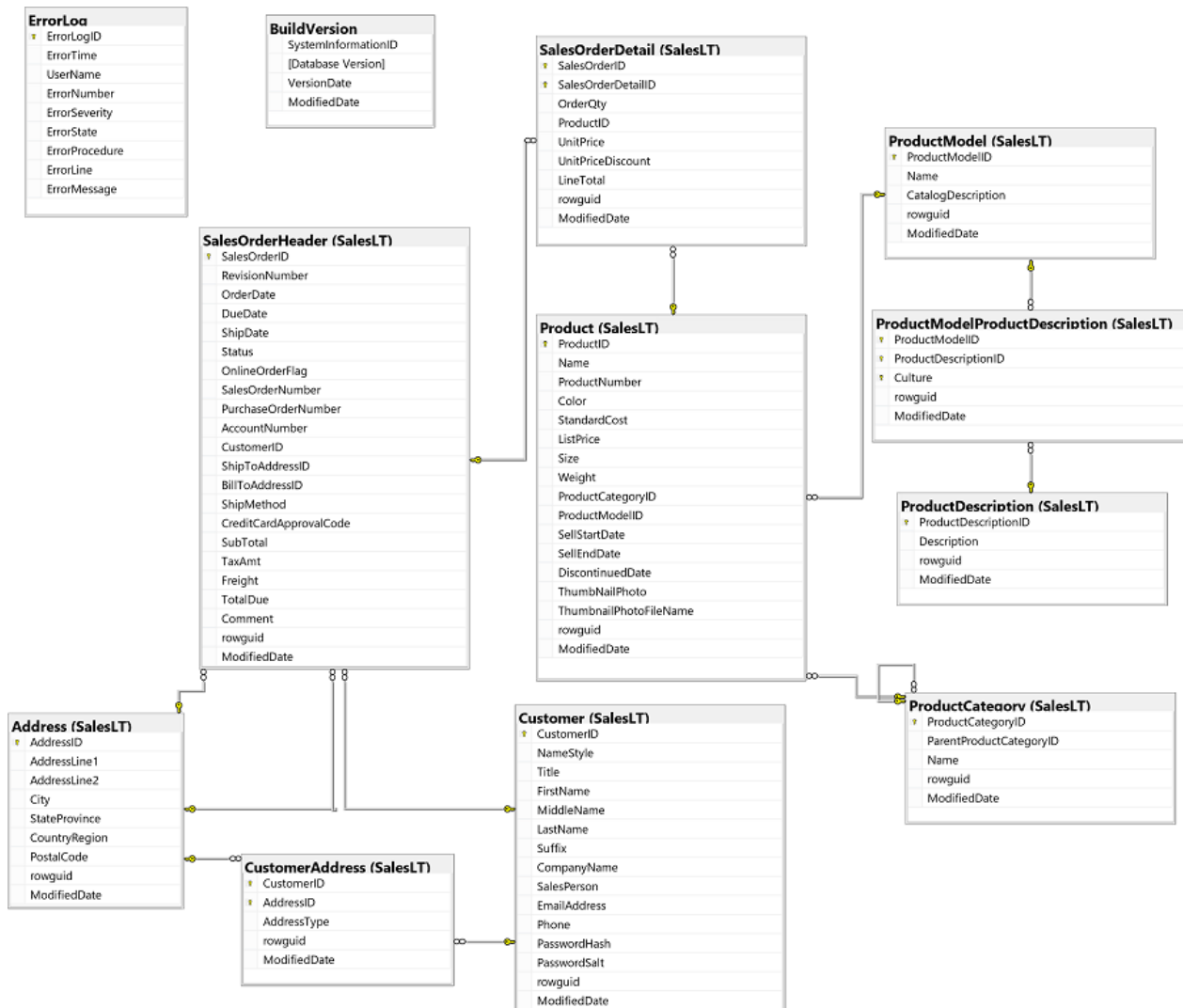
GROUP BY C.Name, SC.Name, P.Name

ORDER BY C.Name, SC.Name, P.Name;

```

## 11 Use pivoting and grouping sets

In this lab, you'll use pivoting and grouping sets to query 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 Pivot data using the PIVOT operator

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. Once you've connected to your database, you can use it. First, let's create a view that contains the information we want to pivot. Write the following code in the query pane:

Code

```
CREATE VIEW SalesLT.vCustGroups AS
SELECT AddressID, CHOOSE(AddressID % 3 + 1, N'A', N'B', N'C') AS
custgroup, CountryRegion
FROM SalesLT.Address;
```

4. Select ☐ **Run** to run your query.
5. The code creates a custom view named **SalesLT.CustGroups** from the **SalesLT.Address** table that groups customers based on their address ID. You can query this view to retrieve results from it. Replace the previous code with the code below:

Code

```
SELECT AddressID, custgroup, CountryRegion
FROM SalesLT.vCustGroups;
```

6. Select ☐ **Run**.
7. The query returns a list of records from the view that shows each customer's address ID, their assigned customer group based on their address ID, and their country. Let's pivot the data using the **PIVOT** operator. Replace the code with the code below:

Code

```
SELECT CountryRegion, p.A, p.B, p.C
FROM SalesLT.vCustGroups PIVOT (
    COUNT(AddressID) FOR custgroup IN (A, B, C)
) AS p;
```

8. Select ☐ **Run** to run your query.
9. Review the results. The result set shows the total number of customers in each customer group for each country. Notice that the result set has changed its orientation. Each customer group (A, B, and C) has a dedicated column, alongside the **CountryRegion** column. With this new orientation, it's easier to understand how many customers are in each group, across all countries.

## 11.2 Group data using a grouping subclause

Use subclauses like **GROUPING SETS**, **ROLLUP**, and **CUBE** to group data in different ways. Each subclause allows you to group data in a unique way. For

instance, **ROLLUP** allows you to dictate a hierarchy and provides a grand total for your groupings. Alternatively, you can use **CUBE** to get all possible combinations for groupings.

For example, let's see how you can use **ROLLUP** to group a set of data.

1. Create a view that captures sales information based on details from the **SalesLT.Customer** and **SalesLT.SalesOrderHeader** tables. To do this, replace the previous code with the code below, then select ☐ **Run**:

Code

```
CREATE VIEW SalesLT.vCustomerSales AS
SELECT      Customer.CustomerID,      Customer.CompanyName,
Customer.SalesPerson, SalesOrderHeader.TotalDue
FROM SalesLT.Customer
INNER JOIN SalesLT.SalesOrderHeader
ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

2. Your view (**SalesLT.vCustomerSales**) cross-references information from two different tables, to display the **TotalDue** amount for customer companies who have made orders, along with their assigned sales representative. Have a look at the view by replacing the previous code with the code below:

Code

```
SELECT * FROM SalesLT.vCustomerSales;
```

3. Select ☐ **Run**.
4. Let's use **ROLLUP** to group this data. Replace your previous code with the code below:

Code

```
SELECT SalesPerson, CompanyName, SUM(TotalDue) AS TotalSales
FROM SalesLT.vCustomerSales
GROUP BY ROLLUP (SalesPerson, CompanyName);
```

5. Select ☐ **Run**.
6. Review the results. You were able you to retrieve customer sales data, and the use of **ROLLUP** enabled you to group the data in a way that allowed you to get the subtotal for historical sales for each sales person, and a final grand total for all sales at the bottom of the result set.

## 11.3 Challenges

Now it's your turn to pivot and group data.

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

### 11.3.1 Challenge 1: Pivot product data

The Adventure Works marketing team wants to conduct research into the relationship between colors and products. To give them a starting point, you've been asked to provide information on how many products are available across the different color types.

1. For each product category, count how many products are available across all the color types.
  - Use the **SalesLT.Product** and **SalesLT.ProductCategory** tables to get a list of products, their colors, and product categories.
  - Pivot your data so that the color types become columns.

### 11.3.2 Challenge 2: Group sales data

The sales team at Adventure Works wants to write a report on sales data. To help them, your manager has asked if you can group historical sales data for them using all possible combinations of groupings based on the **CompanyName** and **SalesPerson** columns.

To help, you're going to write a query to:

1. Retrieve customer sales data, and group the data.
  - In your query, fetch all data from the **Sales.vCustomerSales** view.
  - Group the data using **CompanyName** and **SalesPerson**.
  - Use the appropriate subclause that allows you to create groupings for all possible combinations of your columns.

## 11.4 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 11.4.1 Challenge 1

1. For each product category, count how many products are available across the different color types. Pivot the data so the color types become columns.

Code

```

SELECT *
FROM
(
    SELECT P.ProductID, PC.Name, ISNULL(P.Color, 'Uncolored') AS
Color
    FROM SalesLT.ProductCategory AS PC
    JOIN SalesLT.Product AS P
        ON PC.ProductCategoryID = P.ProductCategoryID
) AS PPC PIVOT(
    COUNT(ProductID) FOR Color IN(
        [Red], [Blue], [Black], [Silver], [Yellow],
        [Grey], [Multi], [Uncolored]
    )
) AS pvt
ORDER BY Name;

```

### 11.4.2 Challenge 2

1. Retrieve customer sales data, and group the data.

Code

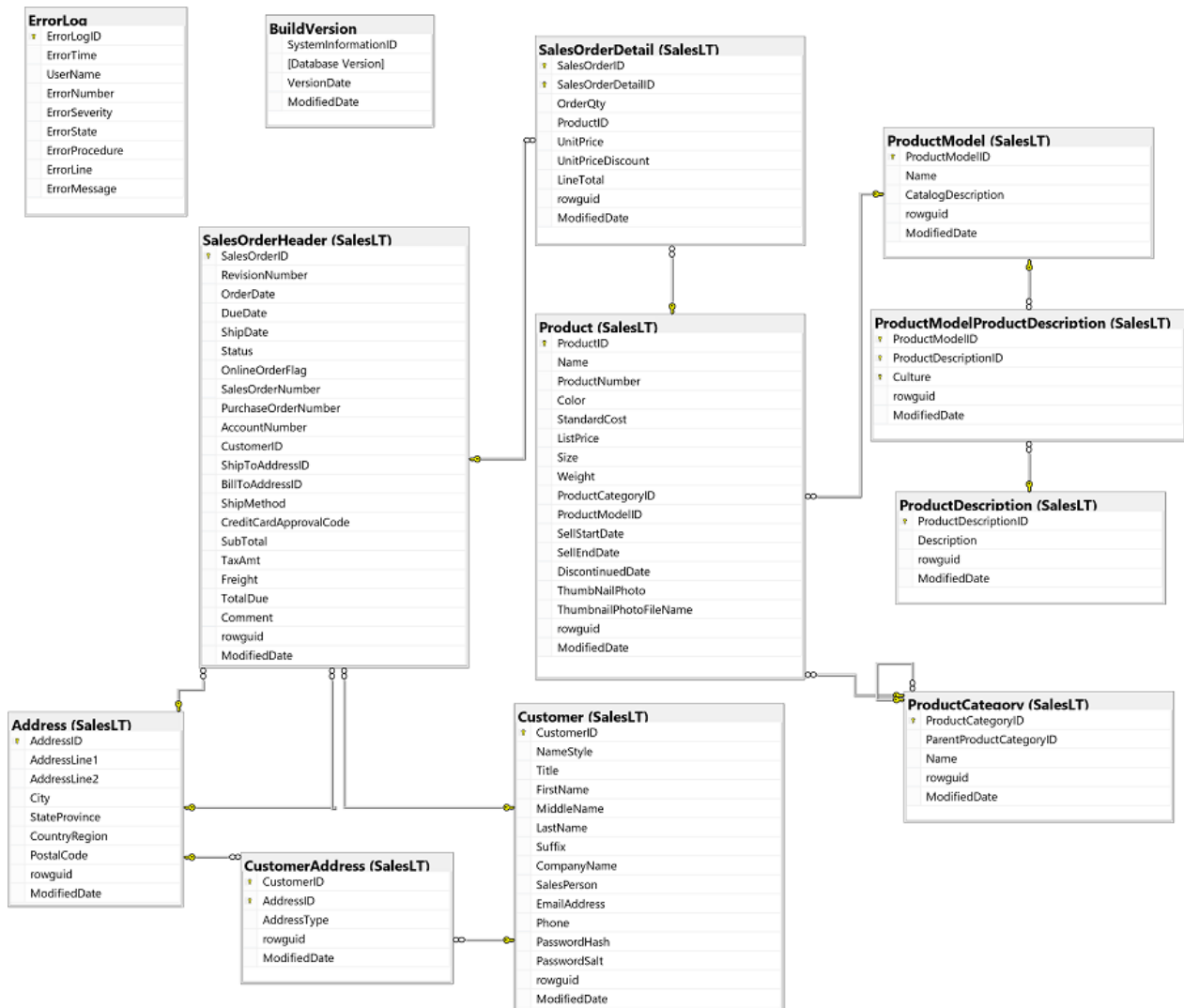
```

SELECT CompanyName, SalesPerson, SUM(TotalDue) AS TotalSales
FROM SalesLT.vCustomerSales
GROUP BY CUBE (CompanyName, SalesPerson);

```

## 12 Introduction to programming with T-SQL

In this lab, you'll use get an introduction to programming using T-SQL techniques using 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 Declare variables and retrieve values



1. Start Azure Data Studio
2. From the Servers pane, double-click the **AdventureWorks connection**. A green dot will appear when the connection is successful.
3. Right click on the AdventureWorks connection and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
4. The previous step will open a query screen that is connected to the adventureworks database.
5. In the query pane, type the following T-SQL code:

Code

```
DECLARE @num int = 5;  
  
SELECT @num AS mynumber;
```

6. Highlight the above T-SQL code and select ☐ **Run**.
7. This will give the result:

mynumber

5

8. In the query pane, type the following T-SQL code after the previous one:

Code

```
DECLARE
```

```

@num1 int,
@num2 int;

SET @num1 = 4;
SET @num2 = 6;

SELECT @num1 + @num2 AS totalnum;

```

9. Highlight the written T-SQL code and select ☐ **Run**.

10. This will give the result:

totalnum
10

11. You've now seen how to declare variables and how to retrieve values.

## 12.2 Use variables with batches

Now, we'll look at how to declare variables in batches.

1. Right click on the TSQL connection and select **New Query**
2. In the query pane, type the following T-SQL code:

Code

```

DECLARE
@empname nvarchar(30),
@empid int;

```

```
SET @empid = 5;
```

```
SET @empname = (SELECT FirstName + N' ' + LastName FROM
SalesLT.Customer WHERE CustomerID = @empid)
```

```
SELECT @empname AS employee;
```

3. Highlight the written T-SQL code and Select ☐ **Run**.
4. This will give you this result:

employee
Lucy Harrington

5. Change the [@empid](#) variable's value from 5 to 2 and execute the modified T-SQL code you'll get:

employee
Keith Harris

6. Now, in the code you just copied, add the batch delimiter GO before this statement:

```
Code
```

```
SELECT @empname AS employee;
```

7. Make sure your T-SQL code looks like this:

Code

```

DECLARE
@empname nvarchar(30),
@empid int;

SET @empid = 5;

SET @empname = (SELECT FirstName + N' ' + LastName FROM
SalesLT.Customer WHERE CustomerID = @empid)

GO

SELECT @empname AS employee;

```

8. Highlight the written T-SQL code and select ☐ **Run\*\***.
9. Observe the error:

Must declare the scalar variable “[@empname](#)”.

Variables are local to the batch in which they’re defined. If you try to refer to a variable that was defined in another batch, you get an error saying that the variable wasn’t defined. Also, keep in mind that GO is a client command, not a server T-SQL command.

### 12.3 Write basic conditional logic

1. Right click on the TSQL connection and select **New Query**

2. In the query pane, type the following T-SQL code:

Code

```
DECLARE
    @i int = 8,
    @result nvarchar(20);

IF @i < 5
    SET @result = N'Less than 5'
ELSE IF @i <= 10
    SET @result = N'Between 5 and 10'
ELSE if @i > 10
    SET @result = N'More than 10'
ELSE
    SET @result = N'Unknown';

SELECT @result AS result;
```

3. Highlight the written T-SQL code and select ☐ **Run**.
4. Which should result in:

result

Between 5 and 10

5. In the query pane, type the following T-SQL code after the previous code:

## Code

```
DECLARE
    @i int = 8,
    @result nvarchar(20);

SET @result =
CASE
WHEN @i < 5 THEN
    N'Less than 5'
WHEN @i <= 10 THEN
    N'Between 5 and 10'
WHEN @i > 10 THEN
    N'More than 10'
ELSE
    N'Unknown'
END;

SELECT @result AS result;
```

This code uses a CASE expression and only one SET expression to get the same result as the previous T-SQL code. Remember to use a CASE expression when it's a matter of returning an expression. However, if you need to execute multiple statements, you can't replace IF with CASE.

1. Highlight the written T-SQL code and select ☐ **Run**.
2. Which should result in the same answer that we had before:

result
Between 5 and 10

## 12.4 Execute loops with WHILE statements

1. Right click on the TSQL connection and select **New Query**
2. In the query pane, type the following T-SQL code:

Code

```
DECLARE @i int = 1;

WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

3. Highlight the written T-SQL code and select ☐ **Run**.
4. This will result in:

Started executing query at Line 1
1
2

Started executing query at Line 1

3

4

5

6

7

8

9

10

## 12.5 Return to Microsoft Learn

When you've finished the exercise, make sure to end the lab environment before you complete the knowledge check in Microsoft Learn.

## 12.6 Challenges

Now it's time to try using what you've learnt.

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



### 12.6.1 Challenge 1: Assignment of values to variables

You are developing a new T-SQL application that needs to temporarily store values drawn from the database, and depending on their values, display the outcome to the user.

1. Create your variables.
  - Write a T-SQL statement to declare two variables. The first is an `nvarchar` with length 30 called `salesOrderNumber`, and the other is an integer called `customerID`.
2. Assign a value to the integer variable.
  - Extend your TSQL code to assign the value 29847 to the `customerID`.
3. Assign a value from the database and display the result.
  - Extend your TSQL to set the value of the variable `salesOrderNumber` using the column **`salesOrderNumber`** from the `SalesOrderHeader` table, filter using the **`customerID`** column and the `customerID` variable. Display the result to the user as `OrderNumber`.

### 12.6.2 Challenge 2: Aggregate product sales

The sales manager would like a list of the first 10 customers that registered and made purchases online as part of a promotion. You've been asked to build the list.

1. Declare the variables:
  - Write a T-SQL statement to declare three variables. The first is called **`customerID`** and will be an Integer with an initial value of 1. The next two variables will be called **`fname`** and **`lname`**. Both will be `NVARCHAR`, give `fname` a length 20 and `lname` a length 30.
2. Construct a terminating loop:

- Extend your T-SQL code and create a WHILE loop that will stop when the customerID variable reaches 10.
3. Select the customer first name and last name and display:
- Extend the T-SQL code, adding a SELECT statement to retrieve the **FirstName** and **LastName** columns and assign them respectively to fname and lname. Combine and PRINT the fname and lname. Filter using the **customerID** column and the customerID variable.

## 12.7 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 12.7.1 Challenge 1

1. Create your variables

Code

```
DECLARE
@salesOrderNumber nvarchar(30),
@customerID int;
```

2. Assign a value to the integer variable.

Code

```
DECLARE
@salesOrderNumber nvarchar(30),
@customerID int;
```

```
SET @customerID = 29847;
```

3. Assign a value from the database and display the result

Code

```
DECLARE
@salesOrderNUmber nvarchar(30),
@customerID int;

SET @customerID = 29847;

SET @salesOrderNUmber = (SELECT salesOrderNumber FROM
SalesLT.SalesOrderHeader WHERE CustomerID = @customerID)

SELECT @salesOrderNUmber as OrderNumber;
```

### 12.7.2 Challenge 2

The sales manager would like a list of the first 10 customers that registered and made purchases online as part of a promotion. You've been asked to build the list.

1. Declare the variables:

Code

```
DECLARE @customerID AS INT = 1;
DECLARE @fname AS NVARCHAR(20);
```

```
DECLARE @lname AS NVARCHAR(30);
```

- Construct a terminating loop:

Code

```
DECLARE @customerID AS INT = 1;
DECLARE @fname AS NVARCHAR(20);
DECLARE @lname AS NVARCHAR(30);

WHILE @customerID <=10
BEGIN
    SET @customerID += 1;
END;
```

- Select the customer first name and last name and display:

Code

```
DECLARE @customerID AS INT = 1;
DECLARE @fname AS NVARCHAR(20);
DECLARE @lname AS NVARCHAR(30);

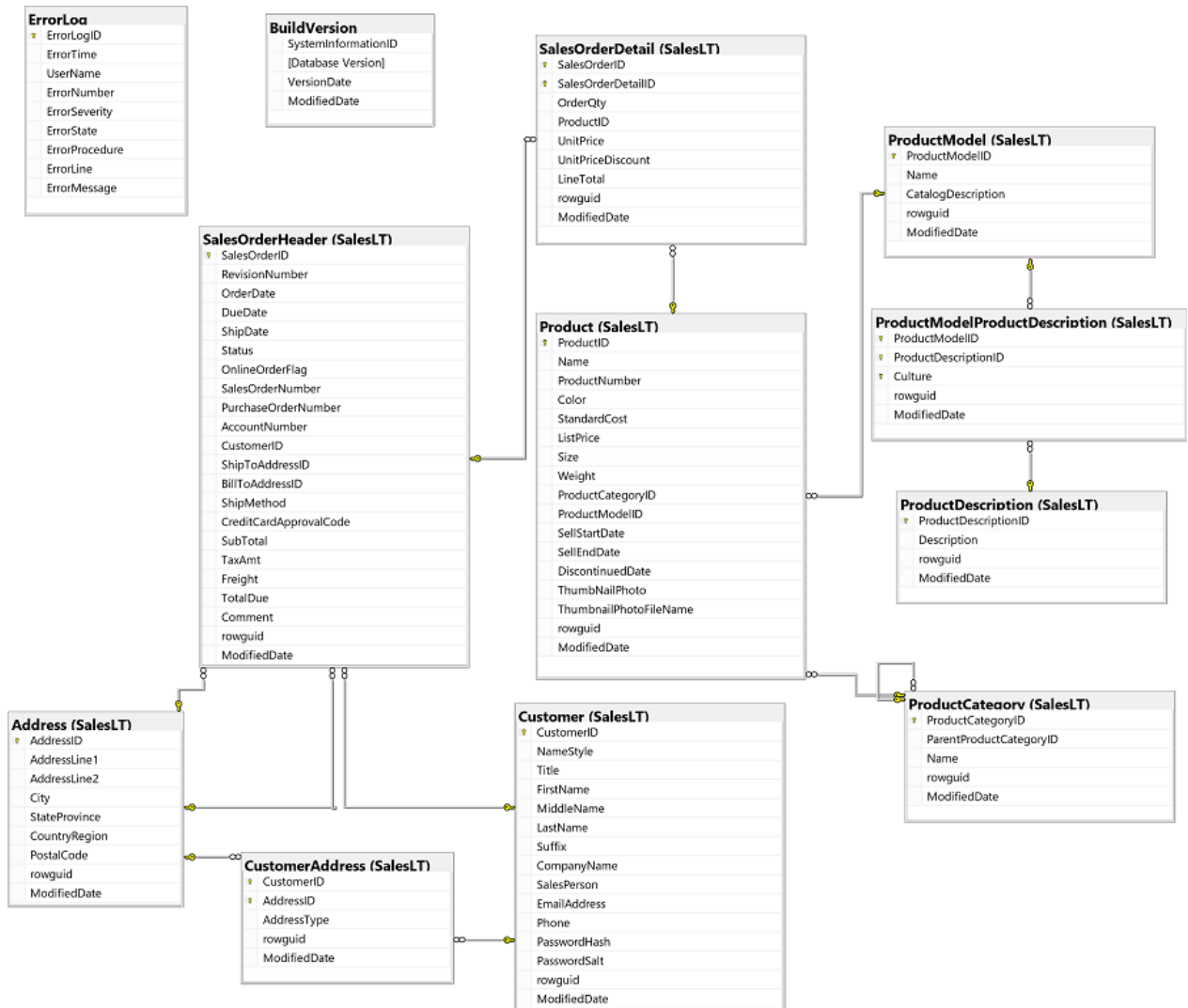
WHILE @customerID <=10
BEGIN
    SELECT @fname = FirstName, @lname = LastName FROM
    SalesLT.Customer
```

```
WHERE CustomerID = @CustomerID;  
  
PRINT @fname + N' ' + @lname;  
  
SET @customerID += 1;  
  
END;
```

- Challenge answer
- Challenge
- Challenge answer

## 13 Create stored procedures in T-SQL

In this lab, you'll use T-SQL statements to create and understand stored procedure techniques 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 Create and execute stored procedures

1. Start Azure Data Studio.
2. From the Servers pane, double-click the **AdventureWorks** connection. A green dot will appear when the connection is successful.

3. Right click on the AdventureWorks connection and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
4. Type the following T-SQL code:

Code

```
CREATE PROCEDURE SalesLT.TopProducts AS
SELECT TOP(10) name, listprice
           FROM SalesLT.Product
           GROUP BY name, listprice
           ORDER BY listprice DESC;
```

5. Select ☐ **Run**. You've created a stored procedure named SalesLT.TopProducts.
6. In the query pane, type the following T-SQL code after the previous code:

Code

```
EXECUTE SalesLT.TopProducts;
```

7. Highlight the written T-SQL code and click ☐ **Run**. You've now executed the stored procedure.
8. Now modify the stored procedure so that it returns only products from a specific product category by adding an input parameter. In the query pane, type the following T-SQL code:

Code

```

ALTER PROCEDURE SalesLT.TopProducts @ProductCategoryID int
AS
SELECT TOP(10) name, listprice
      FROM SalesLT.Product
     WHERE ProductCategoryID = @ProductCategoryID
    GROUP BY name, listprice
    ORDER BY listprice DESC;

```

9. In the query pane, type the following T-SQL code:

Code

```
EXECUTE SalesLT.TopProducts @ProductCategoryID = 18;
```

10. Highlight the written T-SQL code and click ☐ **Run** to execute the stored procedure, passing the parameter value by name.

### 13.1.1 Challenge

1. Pass a value to the stored procedure by position instead of by name. Try Product Category 41.

### 13.1.2 Challenge answer

Code

```
EXECUTE SalesLT.TopProducts 41;
```

## 13.2 Create an inline table valued function



1. In the query pane, type the following T-SQL code:

Code

```
CREATE FUNCTION SalesLT.GetFreightbyCustomer
(@orderyear AS INT) RETURNS TABLE
AS
RETURN
SELECT
customerid, SUM(freight) AS totalfreight
FROM SalesLT.SalesOrderHeader
WHERE YEAR(orderdate) = @orderyear
GROUP BY customerid;
```

2. Highlight the written T-SQL code and click ☐ **Run** to create the table-valued function.

### 13.2.1 Challenge

1. Run the table-valued function to return data for the year 2008.

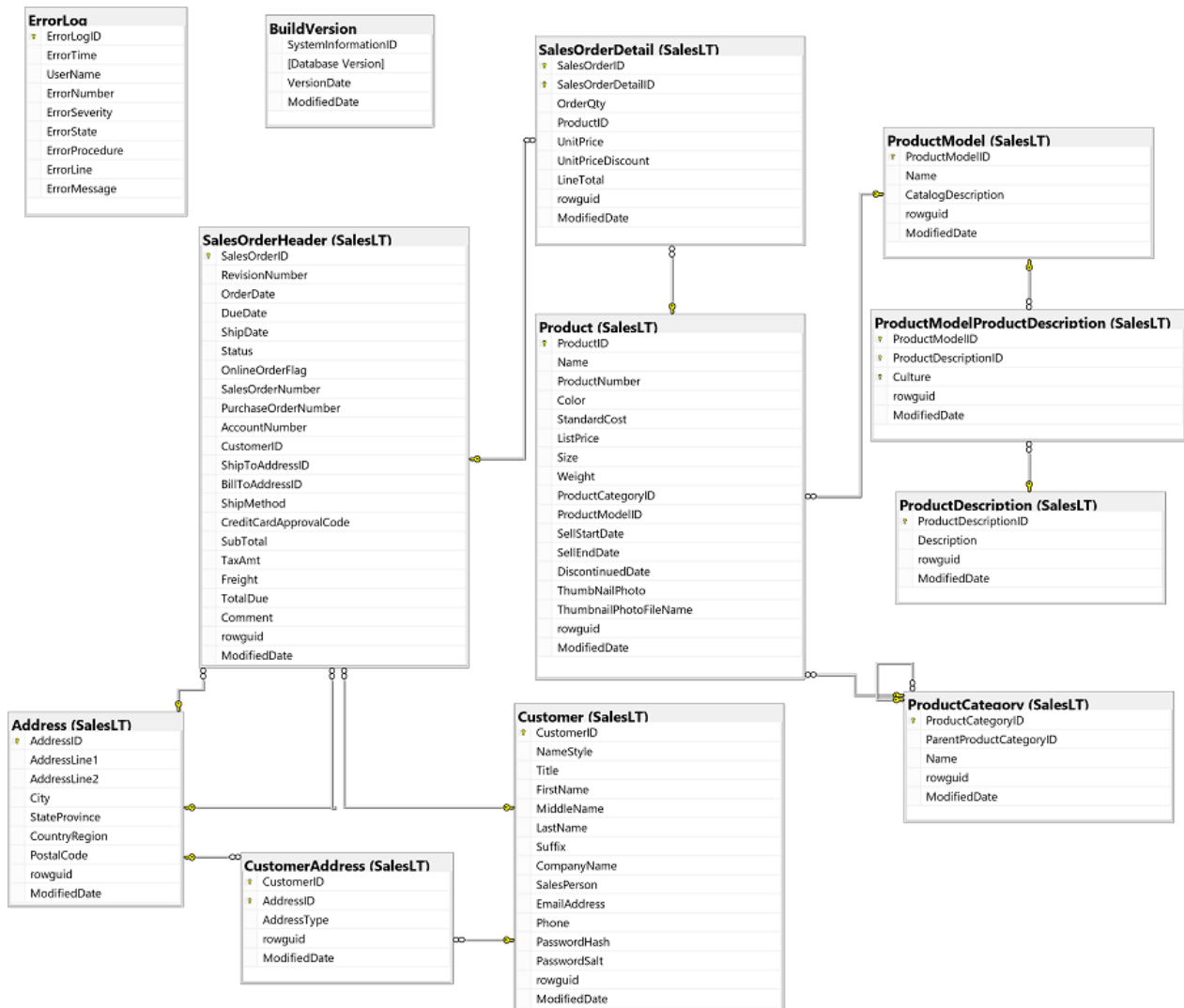
### 13.2.2 Challenge answer

Code

```
SELECT * FROM SalesLT.GetFreightbyCustomer(2008)
```

## 14 Implement error handling with T-SQL

In this lab, you'll use T-SQL statements to test various error handling techniques 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 Write a basic TRY/CATCH construct

1. Start Azure Data Studio
2. From the Servers pane, double-click the **AdventureWorks connection**. A green dot will appear when the connection is successful.
3. Right click the AdventureWorks connection and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
4. The previous step will open a query screen that is connected to the TSQL database.
5. In the query pane, type the following T-SQL code:

Code

```
SELECT CAST(N'Some text' AS int);
```

1. Select ☐ **Run** to run the code.
2. Notice the conversion error:

Result

Conversion failed when converting the nvarchar value 'Some text' to data type int.

3. Write a TRY/CATCH construct. Your T-SQL code should look like this:

Code

```
BEGIN TRY
    SELECT CAST(N'Some text' AS int);
END TRY
BEGIN CATCH
```

```
PRINT 'Error';
END CATCH;
```

1. Run the modified code, and review the response. The results should include no rows, and the **Messages** tab should include the text **Error**.

## 14.2 Display an error number and an error message

1. Right click the AdventureWorks connection and select New Query
2. Enter the following T-SQL code:

Code

```
DECLARE @num varchar(20) = '0';

BEGIN TRY
    PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH

END CATCH;
```

1. Select ☐ **Run**. Notice that you didn't get an error because you used the TRY/CATCH construct.
2. Modify the T-SQL code by adding two PRINT statements. The T-SQL code should look like this:

Code

```
DECLARE @num varchar(20) = '0';

BEGIN TRY
    PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
    PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

1. Run the modified code, and notice that an error is produced, but it's one that you defined.

Started executing query at line 1

Error Number: 8134

Error Message: Divide by zero error encountered.

2. Now change the value of the [@num](#) variable to look like this:

Code

```
DECLARE @num varchar(20) = 'A';
```

1. Run the modified code. Notice that you get a different error number and message.

Started executing query at line 1

Error Message: Error converting data type varchar to numeric.

Error Number: 8114

2. Change the value of the [@num](#) variable to look like this:

Code

```
DECLARE @num varchar(20) = ' 1000000000';
```

1. Run the modified code. Notice that you get a different error number and message.

Started executing query at line 1

Error Number: 8115

Error Message: Arithmetic overflow error converting varchar to data type numeric.

### 14.3 Add conditional logic to a CATCH block

1. Modify the T-SQL code you used previously so it looks like this:

Code

```
DECLARE @num varchar(20) = 'A';
```

```
BEGIN TRY
```

```
    PRINT 5. / CAST(@num AS numeric(10,4));
```

```

END TRY
BEGIN CATCH
    IF ERROR_NUMBER() IN (245, 8114)
    BEGIN
        PRINT 'Handling conversion error...'
    END
    ELSE
    BEGIN
        PRINT 'Handling non-conversion error...';
    END;

    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
    PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;

```

1. Run the modified code. You'll see that message returned now contains more information:

Started executing query at line 1

Handling conversion error...

Error Number: 8114

Error Message: Error converting data type varchar to numeric.

2. Change the value of the [@num](#) variable to look like this:

Code

```
DECLARE @num varchar(20) = '0';
```

1. Run the modified code. This produces a different type of error message:

Started executing query at line 1

Handling non-conversion error...

Error Number: 8134

Error Message: Divide by zero error encountered.

#### 14.4 Create a stored procedure to display an error message

1. Right click the AdventureWorks connection and select New Query
2. Enter the following T-SQL code:

Code

```
CREATE PROCEDURE dbo.GetErrorInfo AS
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
PRINT 'Error State: ' + CAST(ERROR_STATE() AS varchar(10));
PRINT 'Error Line: ' + CAST(ERROR_LINE() AS varchar(10));
PRINT 'Error Proc: ' + COALESCE(ERROR_PROCEDURE(), 'Not within
procedure');
```



1. Select ☐ **Run.** to run the code, which creates a stored procedure named **dbo.GetErrorInfo**.
2. Return to the query that previously resulted in a “Divide by zero” error, and modify it as follows:

Code

```
DECLARE @num varchar(20) = '0';

BEGIN TRY
    PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
    EXECUTE dbo.GetErrorInfo;
END CATCH;
```

1. Run the code. This will trigger the stored procedure and display:

Started executing query at line 1

Error Number: 8134

Error Message: Divide by zero error encountered.

Error Severity: 16

Error State: 1

Error Line: 4

Started executing query at line 1

Error Proc: Not within procedure

## 14.5 Rethrow the Existing Error Back to a Client

1. Modify the CATCH block of your code to include a THROW command, so that your code looks like this:

Code

```
DECLARE @num varchar(20) = '0';

BEGIN TRY
    PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
    EXECUTE dbo.GetErrorInfo;
    THROW;
END CATCH;
```

1. Run the modified code. Here you'll see that it executes the stored procedure, and then throws the error message again (so a client application can catch and process it).

Started executing query at line 1

Error Number: 8134

Started executing query at line 1

Error Message: Divide by zero error encountered.

Error Severity: 16

Error State: 1

Error Line: 4

Error Proc: Not within procedure

Msg 8134, Level 16, State 1, Line 4

Divide by zero error encountered.

## 14.6 Add an Error Handling Routine

1. Modify your code to look like this:

Code

```
DECLARE @num varchar(20) = 'A';

BEGIN TRY
    PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
    EXECUTE dbo.GetErrorInfo;
```

```
IF ERROR_NUMBER() = 8134
BEGIN
    PRINT 'Handling devision by zero...';
END
ELSE
BEGIN
    PRINT 'Throwing original error';
    THROW;
END;

END CATCH;
```

1. Run the modified code As you'll see, it executes the stored procedure to display the error, identifies that it isn't error number 8134, and throws the error again.

Started executing query at line 1

Error Number: 8114

Error Message: Error converting data type varchar to numeric.

Error Severity: 16

Error State: 5

Error Line: 5

Started executing query at line 1

Error Proc: Not within procedure

Throwing original error

Msg 8114, Level 16, State 5, Line 5

Error converting data type varchar to numeric.

## 14.7 Challenges

Now it's time to try using what you've learned.

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

### 14.7.1 Challenge 1: Catch errors and display only valid records

The marketing manager is using the following T-SQL query, but they are getting unexpected results. They have asked you to make the code more resilient, to stop it crashing and to not display duplicates when there is no data.

Code

```
DECLARE @customerID AS INT = 30110;  
DECLARE @fname AS NVARCHAR(20);  
DECLARE @lname AS NVARCHAR(30);  
DECLARE @maxReturns AS INT = 1;
```

```
WHILE @maxReturns <= 10  
BEGIN
```

```

SELECT @fname = FirstName, @lname = LastName FROM SalesLT.Customer
WHERE CustomerID = @CustomerID;

PRINT @fname + N' ' + @lname;

SET @maxReturns += 1;

SET @CustomerID += 1;

END;

```

1. Catch the error
  - Add a TRY .. CATCH block around the SELECT query.
2. Warn the user that an error has occurred
  - Extend your TSQL code to display a warning to the user that their is an error.
3. Only display valid customer records
  - Extend the T-SQL using the `@@ROWCOUNT > 0` check to only display a result if the customer ID exists.

### 14.7.2 Challenge 2: Create a simple error display procedure

Error messages and error handling are essential for good code. Your manager has asked you to develop a common error display procedure. Use this sample code as your base.

Code

```

DECLARE @num varchar(20) = 'Challenge 2';

PRINT 'Casting: ' + CAST(@num AS numeric(10,4));

```

1. Catch the error

- Add a TRY...CATCH around the PRINT statement.
2. Create a stored procedure
    - Create a stored procedure called `dbo.DisplayErrorDetails`. It should display a title and the value for **ERROR\_NUMBER**, **ERROR\_MESSAGE** and **ERROR\_SEVERITY**.
  3. Display the error information
    - Use the stored procedure to display the error information when an error occurs.

## 14.8 Challenge Solutions

This section contains suggested solutions for the challenge queries.

### 14.8.1 Challenge 1

1. Catch the error

#### Code

```

DECLARE @customerID AS INT = 30110;
DECLARE @fname AS NVARCHAR(20);
DECLARE @lname AS NVARCHAR(30);
DECLARE @maxReturns AS INT = 1;

WHILE @maxReturns <= 10
BEGIN
    BEGIN TRY
        SELECT  @fname = FirstName, @lname = LastName FROM
SalesLT.Customer
    
```

```

WHERE CustomerID = @CustomerID;

PRINT CAST(@customerID as NVARCHAR(20)) + N' ' + @fname + N' ' +
@lname;

END TRY
BEGIN CATCH

END CATCH;

SET @maxReturns += 1;
SET @CustomerID += 1;
END;

```

1. Warn the user that an error has occurred

#### Code

```

DECLARE @customerID AS INT = 30110;
DECLARE @fname AS NVARCHAR(20);
DECLARE @lname AS NVARCHAR(30);
DECLARE @maxReturns AS INT = 1;

WHILE @maxReturns <= 10
BEGIN
    BEGIN TRY
        SELECT  @fname = FirstName, @lname = LastName FROM
SalesLT.Customer

        WHERE CustomerID = @CustomerID;

```



```

        PRINT CAST(@customerID as NVARCHAR(20)) + N' ' + @fname + N' ' +
        @lname;
    END TRY
    BEGIN CATCH
        PRINT 'Unable to run query'
    END CATCH;

    SET @maxReturns += 1;
    SET @CustomerID += 1;
END;

```

### 1. Only display valid customer records

#### Code

```

DECLARE @customerID AS INT = 30110;
DECLARE @fname AS NVARCHAR(20);
DECLARE @lname AS NVARCHAR(30);
DECLARE @maxReturns AS INT = 1;

WHILE @maxReturns <= 10
BEGIN
    BEGIN TRY
        SELECT  @fname = FirstName, @lname = LastName FROM
        SalesLT.Customer
        WHERE CustomerID = @CustomerID;
    
```

```

IF @@ROWCOUNT > 0
BEGIN
    PRINT CAST(@customerID as NVARCHAR(20)) + N' ' + @fname + N' ' +
@lname;
END
END TRY
BEGIN CATCH
    PRINT 'Unable to run query'
END CATCH

SET @maxReturns += 1;
SET @CustomerID += 1;
END;

```

### 14.8.2 Challenge 2

1. Catch the error

Code

```

DECLARE @num varchar(20) = 'Challenge 2';

BEGIN TRY
    PRINT 'Casting: ' + CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH

END CATCH;

```

### 1. Create a stored procedure

#### Code

```
CREATE PROCEDURE dbo.DisplayErrorDetails AS
PRINT 'ERROR INFORMATION';
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
```

### 1. Display the error information

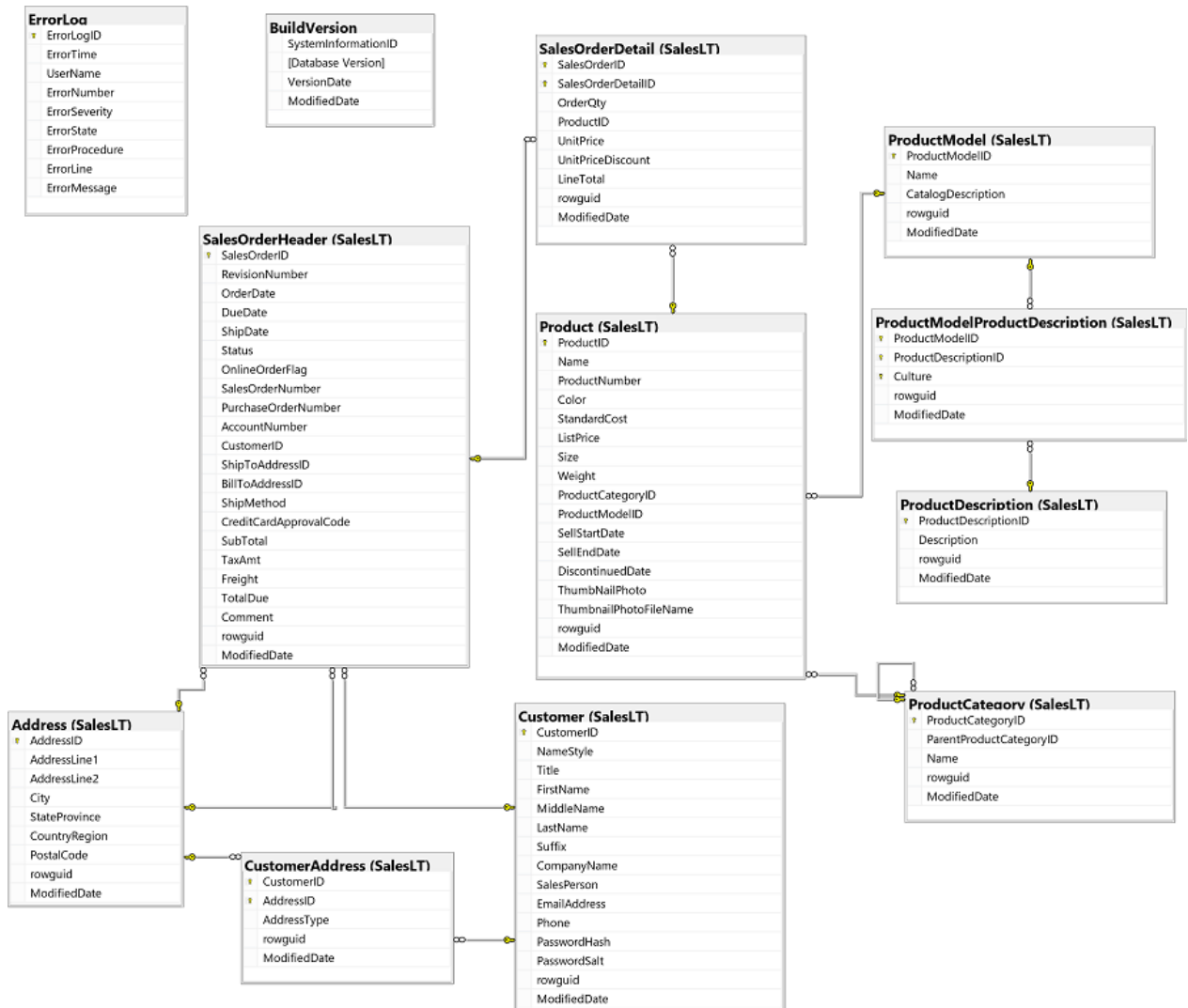
#### Code

```
DECLARE @num varchar(20) = 'Challenge 2';

BEGIN TRY
    PRINT 'Casting: ' + CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
    EXECUTE dbo.DisplayErrorDetails;
END CATCH;
```

## 15 Implement transactions with Transact SQL

In this lab, you'll use T-SQL statements to see the impact of using transactions 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.

## 15.1 Insert data without transactions

Consider a website that needs to store customer information. As part of the customer registration, data about a customer and their address need to be stored. A customer without an address will cause problems for the shipping when orders are made.

In this exercise you'll use a transaction to ensure that when a row is inserted into the **Customer** and **Address** tables, a row is also added to the **CustomerAddress** table. If one insert fails, then all will fail.


1. Start Azure Data Studio.
2. In the **Connections** pane, double-click the **AdventureWorks** server. A green dot will appear when the connection is successful.
3. Right click the **AdventureWorks** server and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
4. Enter the following T-SQL code into the query window:

Code

```
INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
VALUES (0, 'Norman','Newcustomer','norman0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=
',NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
3A6',NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID, AddressType,
rowguid, ModifiedDate)
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', NEWID(), '12-1-20212');
```

1. Select  **Run** at the top of the query window, or press the **F5** key to run the code.
2. Note the output messages, which should look like this:

(1 row affected)

(1 row affected)

Conversion failed when converting date and/or time from character string.

Two of the statements appear to have succeeded, but the third failed.

1. Right click the **AdventureWorks** server and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
2. Enter the following T-SQL code into the new query window:

Code

```
SELECT * FROM SalesLT.Customer ORDER BY ModifiedDate DESC;
```

A row for *Norman Newcustomer* was inserted into the Customer table (and another was inserted into the Address table). However, the insert for the CustomerAddress table failed with a duplicate key error. The database is now inconsistent as there's no link between the new customer and their address.

To fix this, you'll need to delete the two rows that were inserted.

1. Right click the **AdventureWorks** server and select **New Query**. A new query window is displayed with a connection to the AdventureWorks database.
2. Enter the following T-SQL code into the new query window and run it to delete the inconsistent data:

Code

```
DELETE SalesLT.Customer
WHERE CustomerID = IDENT_CURRENT('SalesLT.Customer');

DELETE SalesLT.Address
WHERE AddressID = IDENT_CURRENT('SalesLT.Address');
```

**Note:** This code only works because you are the only user working in the database. In a real scenario, you would need to ascertain the IDs of the records that were inserted and specify them explicitly in case new customer and address records had been inserted since you ran your original code.

## 15.2 Insert data as using a transaction

All of these statements need to run as a single atomic transaction. If any one of them fails, then all statements should fail. Let's group them together in a transaction.

1. Switch back to the original query window with the INSERT statements, and modify the code to enclose the original INSERT statements in a transaction, like this:

Code

```
BEGIN TRANSACTION;

INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
```

```
VALUES (0, 'Norman','Newcustomer','norman0@adventure-works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=', NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince, CountryRegion, PostalCode, rowguid, ModifiedDate)
```

```
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A3A6', NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID, AddressType, rowguid, ModifiedDate)
```

```
VALUES (IDENT_CURRENT('SalesLT.Customer'), IDENT_CURRENT('SalesLT.Address'), 'Home', NEWID(), '12-1-20212');
```

```
COMMIT TRANSACTION;
```

1. Run the code, and review the output message:

(1 row affected)

(1 row affected)

Msg 241, Level 16, State 1, Line 9 Conversion failed when converting date and/or time from character string.

Again, it looks like the first two statements succeeded and the third one failed.

1. Switch to the query containing the SELECT statement to check for a new customer record, and run it. This time, there should be no record for *Norman Newcustomer*. Using a transaction with these statements has triggered an



automatic rollback. The level 16 conversion error is high enough to cause all statements to be rolled back.

### 15.3 Handle errors and explicitly rollback transactions

Lower level errors can require that you explicitly handle the error and rollback any active transactions.

1. Switch back to the original INSERT query script, and modify the transaction as follows:

Code

```
BEGIN TRANSACTION;
```

```
INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
VALUES (0, 'Norman','Newcustomer','norman0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8='
, NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
3A6', NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
AddressType, rowguid, ModifiedDate)
```

```
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', '16765338-dbe4-4421-b5e9-
3836b9278e63', GETDATE());

COMMIT TRANSACTION;
```

1. Run the modified code. The output message this time is:

```
(1 row affected)
(1 row affected)
Msg 2627, Level 14, State 1, Line 9 Violation of UNIQUE KEY constraint
'AK_CustomerAddress_rowguid'. Cannot insert duplicate key in object
'SalesLT.CustomerAddress'. The duplicate key value is (16765338-dbe4-4421-
b5e9-3836b9278e63).
```

1. Switch back to the query window containing the SELECT customer statement and run the query to see if the *Norman Newcustomer* row was added.

Even though an error occurred in the transaction, a new record has been added and the database is once again inconsistent.

2. Switch back to the query window containing the DELETE statements, and run it to delete the new inconsistent data.

Enclosing the statements in a transaction isn't enough to deal with lower priority errors. You need to catch these errors and explicitly use a ROLLBACK statement. We need to combine batch error handling and transactions to resolve our data consistency issue.

3. Switch back to the original query window, and modify the code to enclose the transaction in a TRY/CATCH block, and use the ROLLBACK TRANSACTION statement if an error occurs.

## Code

```
BEGIN TRY
```

```
BEGIN TRANSACTION;
```

```
INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
```

```
VALUES (0, 'Norman','Newcustomer','norman0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=
',NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
```

```
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
3A6',NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
AddressType, rowguid, ModifiedDate)
```

```
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', '16765338-dbe4-4421-b5e9-
3836b9278e63', GETDATE());
```

```
COMMIT TRANSACTION;
```

```
PRINT 'Transaction committed.';
```

```

END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Transaction rolled back.';
END CATCH;

```

1. Run the code and review the results:

```

Started executing query at Line 1
(1 row affected)
(1 row affected)
(0 rows affected)

```

Now there isn't any error message so it looks like two rows were affected.

1. Switch back to the query window containing the `SELECT` customer statement and run the query to see if the *Norman Newcustomer* row was added.

Note that the most recently modified customer record is not for *Norman Newcustomer* - the `INSERT` statement that succeeded has been rolled back to ensure the database remains consistent.

## 15.4 Check the transaction state before rolling back

The `CATCH` block will handle errors that occur anywhere in the `TRY` block, so if an error were to occur outside of the `BEGIN TRANSACTION...COMMIT TRANSACTION` block, there would be no active transaction to roll back. To avoid

this issue, you can check the current transaction state with XACT\_STATE(), which returns one of the following values:

- **-1**: There is an active transaction in process that cannot be committed.
- **0**: There are no transactions in process.
- **1**: There is an active transaction in process that can be committed or rolled back.

1. Back in the original query window, surround the ROLLBACK statements with an IF statement checking the value, so your code looks like this.

#### Code

```
BEGIN TRY
```

```
BEGIN TRANSACTION;
```

```
    INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
    EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
```

```
    VALUES          (0,          'Norman','Newcustomer','norman0@adventure-
    works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8='
    ,NEWID(), GETDATE());
```

```
    INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
    CountryRegion, PostalCode, rowguid, ModifiedDate)
```

```
    VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
    3A6',NEWID(), GETDATE());
```

```
    INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
    AddressType, rowguid, ModifiedDate)
```

```
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', '16765338-dbe4-4421-b5e9-
3836b9278e63', GETDATE());
```

```
COMMIT TRANSACTION;
PRINT 'Transaction committed.';
```

```
END TRY
BEGIN CATCH
    PRINT 'An error occurred.'
    IF (XACT_STATE()) <> 0
    BEGIN
        PRINT 'Transaction in process.'
        ROLLBACK TRANSACTION;
        PRINT 'Transaction rolled back.';
    END;
END CATCH
```

1. Run the modified code, and review the output - noting that an in-process transaction was detected and rolled back.
2. Modify the code as follows to avoid specifying an explicit **rowid** (which was caused the duplicate key error)

Code

```
BEGIN TRY
BEGIN TRANSACTION;
```

```

INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
VALUES (0, 'Norman','Newcustomer','norman0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=
',NEWID(), GETDATE());

```

```

INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
3A6',NEWID(), GETDATE());

```

```

INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
AddressType, ModifiedDate)
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', GETDATE());

```

```

COMMIT TRANSACTION;
PRINT 'Transaction committed.';
END TRY
BEGIN CATCH
PRINT 'An error occurred.'
IF (XACT_STATE()) <> 0
BEGIN
PRINT 'Transaction in process.'
ROLLBACK TRANSACTION;
PRINT 'Transaction rolled back.';
END;

```

**END CATCH**

1. Run the code, and note that this time, all three INSERT statement succeed.
2. Switch back to the query window containing the SELECT customer statement and run the query to verify that the *Norman Newcustomer* row was added.
3. Back in the original query window, modify the code to insert another customer - this time throwing an error within the TRY block after the transaction has been committed:

## Code

**BEGIN TRY****BEGIN TRANSACTION;**

```
INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate) VALUES
(0, 'Ann','Othercustomr','ann0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=
',NEWID(), GETDATE());;
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
VALUES ('6388 Lake City Way', 'Burnaby','British Columbia','Canada','V5A
3A6',NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
AddressType, ModifiedDate)
```



```

VALUES                                     (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', GETDATE());

COMMIT TRANSACTION;
PRINT 'Transaction committed.';

THROW 51000, 'Some kind of error', 1;

END TRY
BEGIN CATCH
    PRINT 'An error occurred.'
    IF (XACT_STATE()) <> 0
    BEGIN
        PRINT 'Transaction in process.'
        ROLLBACK TRANSACTION;
        PRINT 'Transaction rolled back.';
    END;
END CATCH

```

1. Run the code and review the output. All three INSERT statements should succeed, but an error is caught by the CATCH block.
2. Switch back to the query window containing the SELECT customer statement and run the query to verify that a record for *Ann Othercustomer* has been inserted. The transaction succeeded and was not rolled back, even though an error subsequently occurred.

## 15.5 Explore transaction concurrency

Isolation levels determine the visibility of data modifications made in transactions across multiple sessions with the database. On-premises SQL Server has a default isolation level of **READ\_COMMITTED\_SNAPSHOT\_OFF**. This level of isolation will hold locks on rows while a transaction is acting on it. For example, inserting a customer into the customer table. If the update takes a long time to run, any queries on that table from other sessions will be blocked from running until the transaction is committed or rolled back.

1. In Azure Data Studio, close all open query panes.
2. In the **Connections** pane, ensure that the **AdventureWorks** server has a green icon indicating an active connection. Then, in the header for the **Servers** section, use the **new Connection** icon to create a new connection with the following properties:
  - **Connection type:** Microsoft SQL Server
  - **Server:** (local)\sqlexpress
  - **Authentication type:** Windows Authentication
  - **Database:** adventureworks
3. After the new connection has been made, verify that the **Connections** pane now includes two connections (which represent two different connections to the same database):
  - **AdventureWorks**
  - **(local)\sqlexpress**
4. Right-click the **AdventureWorks** and create a new query. Then enter the following code (but do not run it yet):

Code

```
BEGIN TRANSACTION;
```

```
INSERT INTO SalesLT.Customer (NameStyle, FirstName, LastName,
EmailAddress, PasswordHash, PasswordSalt, rowguid, ModifiedDate)
VALUES (0, 'Yeta','Nothercustomer','yeta0@adventure-
works.com','U1/CrPqSzwLTtwgBehfpII7f1LHSFpZw1qnG1sMzFjo=','QhHP+y8=
', NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.Address (AddressLine1, City, StateProvince,
CountryRegion, PostalCode, rowguid, ModifiedDate)
VALUES ('2067 Park Lane', 'Redmond','Washington','United States','98007',
NEWID(), GETDATE());
```

```
INSERT INTO SalesLT.CustomerAddress (CustomerID, AddressID,
AddressType, rowguid, ModifiedDate)
VALUES (IDENT_CURRENT('SalesLT.Customer'),
IDENT_CURRENT('SalesLT.Address'), 'Home', NEWID(), GETDATE());
```


```
COMMIT TRANSACTION;
```

1. Right-click the **(loval)\sqlexpress** connection and create a new query. Then enter the following code:

Code

```
SELECT COUNT(*) FROM SalesLT.Customer
```

1. Run the SELECT COUNT(\*) query and note the number of counted rows.  
Notice the query returns results quickly.

2. Return the transaction query for the **AdventureWorks** connection and highlight BEGIN TRANSACTION and INSERT statements (but not the COMMIT TRANSACTION; statement) Then use the  **Run** to run only the highlighted code.

As this is in a transaction that hasn't been committed, running the SELECT COUNT(\*) query in the other window will be blocked.

3. In the other query pane, run the SELECT COUNT(\*) query and note that the query doesn't finish.
4. To prove the query is blocked by the transaction, highlight and run the COMMIT TRANSACTION; statement in the other window.
5. Switch back to the SELECT COUNT(\*)`query, and verify that it completes and returns the correct number of customers.

## 15.6 Change how concurrency is handled on a database

Concurrency can be changed to allow database queries on tables while there are transactions inserting or updating them. To change this enable READ\_COMMITTED\_SNAPSHOT\_ON.

1. Create a new query from the **AdventureWorks** connection, and run this Transact-SQL code in it.

Code

```
ALTER DATABASE AdventureWorks SET READ_COMMITTED_SNAPSHOT  
ON WITH ROLLBACK IMMEDIATE  
GO
```

1. Switch to the transaction query pane that's connected to the **AdventureWorks** connection, and once again highlight the **BEGIN TRANSACTION** and **INSERT** statements (but not the **COMMIT TRANSACTION**; statement) and run the ighlighted code.
2. Switch to the query pane with the **SELECT COUNT(\*)** query (connected to the **(local)\sqlexpress** connection) and run it. The query results reflect the current state of the database as the **INSERT** statement hasn't been committed yet.
3. In the transaction query pane, highlight the **COMMIT TRANSACTION**; statement and run it.
4. Re-run the **SELECT COUNT(\*)** query and note that the total customers has increased by 1.

Be careful when selecting what isolation levels to use in a database. In some scenarios returning the current state of data before a transaction has been committed is worse than a query being blocked and waiting for all the data to be in the correct state.

## 15.7 Challenge

Now it's time to try using what you've learned.

**Tip:** Try to determine the appropriate solution for yourself. If you get stuck, a suggested solution is provided at the end of this lab.

### 15.7.1 Use a transaction to insert data into multiple tables

When a sales order header is inserted, it must have at least one corresponding sales order detail record. Currently, you use the following code to accomplish this:

Code

-- Get the highest order ID and add 1

DECLARE @OrderID INT;

SELECT @OrderID = MAX(SalesOrderID) + 1 FROM SalesLT.SalesOrderHeader;

-- Insert the order header

INSERT INTO SalesLT.SalesOrderHeader (SalesOrderID, OrderDate, DueDate, CustomerID, ShipMethod)

VALUES (@OrderID, GETDATE(), DATEADD(month, 1, GETDATE()), 1, 'CARGO TRANSPORT');

-- Insert one or more order details

INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID, UnitPrice)

VALUES (@OrderID, 1, 712, 8.99);

You need to encapsulate this code in a transaction so that all inserts succeed or fail as an atomic unit or work.

## 15.8 Challenge solution

### 15.8.1 Use a transaction to insert data into multiple tables

The following code encloses the logic to insert a new order and order detail in a transaction, rolling back the transaction if an error occurs.

Code

BEGIN TRY

BEGIN TRANSACTION;

```

-- Get the highest order ID and add 1
DECLARE @OrderID INT;
SELECT      @OrderID      =      MAX(SalesOrderID)      +      1      FROM
SalesLT.SalesOrderHeader;

-- Insert the order header
INSERT INTO SalesLT.SalesOrderHeader (SalesOrderID, OrderDate, DueDate,
CustomerID, ShipMethod)
VALUES (@OrderID, GETDATE(), DATEADD(month, 1, GETDATE()), 1,
'CARGO TRANSPORT');

-- Insert one or more order details
INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID,
UnitPrice)
VALUES (@OrderID, 1, 712, 8.99);

COMMIT TRANSACTION;
PRINT 'Transaction committed.';

END TRY
BEGIN CATCH
PRINT 'An error occurred.'
IF (XACT_STATE()) <> 0
BEGIN
PRINT 'Transaction in process.'
ROLLBACK TRANSACTION;
PRINT 'Transaction rolled back.';

```

```
END;
END CATCH
```

To test the transaction, try to insert an order detail with an invalid product ID, like this:

Code

```
BEGIN TRY
BEGIN TRANSACTION;
    -- Get the highest order ID and add 1
    DECLARE @OrderID INT;
    SELECT      @OrderID      =      MAX(SalesOrderID)      +      1      FROM
SalesLT.SalesOrderHeader;

    -- Insert the order header
    INSERT INTO SalesLT.SalesOrderHeader (SalesOrderID, OrderDate, DueDate,
CustomerID, ShipMethod)
    VALUES (@OrderID, GETDATE(), DATEADD(month, 1, GETDATE()), 1,
'CARGO TRANSPORT');

    -- Insert one or more order details
    INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID,
UnitPrice)
    VALUES (@OrderID, 1, 'Invalid product', 8.99);

COMMIT TRANSACTION;
PRINT 'Transaction committed.';
```



```
END TRY
BEGIN CATCH
    PRINT 'An error occurred.'
    IF (XACT_STATE()) <> 0
    BEGIN
        PRINT 'Transaction in process.'
        ROLLBACK TRANSACTION;
        PRINT 'Transaction rolled back.';
    END;
END CATCH
```

## 16 Module 1 Demonstrations

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

### 16.1 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:


Code

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

12. Use the  **Run** button to run the query, 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.

## 16.2 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).

### 16.3 Module01-demos.sql

#### -- BASIC QUERIES

-- Select all columns

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

-- Select specific columns

```
SELECT CustomerID, FirstName, LastName
FROM SalesLT.Customer;
```

-- Select an expression

```
SELECT CustomerID, FirstName + ' ' + LastName
FROM SalesLT.Customer;
```

-- Apply an alias

```
SELECT CustomerID, FirstName + ' ' + LastName AS CustomerName
FROM SalesLT.Customer;
```

#### -- DATA TYPES

-- Try to combine incompatible data types (results in error)

```
SELECT CustomerID + ':' + EmailAddress AS CustomerIdentifier
FROM SalesLT.Customer;
```

-- Use cast

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

-- Use convert

```
SELECT CONVERT(varchar, CustomerID) + ':' + EmailAddress AS
CustomerIdentifier
FROM SalesLT.Customer;
```

-- convert dates

```
SELECT CustomerID,
       CONVERT(nvarchar(30), ModifiedDate) AS ConvertedDate,
       CONVERT(nvarchar(30), ModifiedDate, 126) AS ISO8601FormatDate
FROM SalesLT.Customer;
```

-- NULL VALUES

-- See the effect of expressions with NULL values

```
SELECT CustomerID, Title + ' ' + LastName AS Greeting
FROM SalesLT.Customer;
```

-- Replace NULL value (use ? if Title is NULL)

```
SELECT CustomerID, ISNULL(Title, '?') + ' ' + LastName AS Greeting
FROM SalesLT.Customer;
```

-- Coalesce (use first non-NULL value)

```
SELECT CustomerID, COALESCE(Title, FirstName) + ' ' + LastName AS Greeting
FROM SalesLT.Customer;
```

-- Convert specific values to NULL

```
SELECT SalesOrderID, ProductID, UnitPrice, NULLIF(UnitPriceDiscount, 0) AS
Discount
FROM SalesLT.SalesOrderDetail;
```

-- CASE statement

--Simple case

```
SELECT CustomerID,
       CASE
         WHEN Title IS NOT NULL AND MiddleName IS NOT NULL
           THEN Title + ' ' + FirstName + ' ' + MiddleName + ' ' + LastName
         WHEN Title IS NOT NULL AND MiddleName IS NULL
           THEN Title + ' ' + FirstName + ' ' + LastName
         ELSE FirstName + ' ' + LastName
       END AS CustomerName
FROM SalesLT.Customer;
```

-- Searched case


```
SELECT FirstName, LastName,
       CASE Suffix
         WHEN 'Sr.' THEN 'Senior'
         WHEN 'Jr.' THEN 'Junior'
         ELSE ISNULL(Suffix, "")
       END AS NameSuffix
FROM SalesLT.Customer;
```

## 17 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).

## 17.1 Module02-demos.sql

-- This script contains demo code for Module 2 of the Transact-SQL course

-- ORDER BY

-- Sort by column

```
SELECT AddressLine1, City, PostalCode, CountryRegion
FROM SalesLT.Address
ORDER BY CountryRegion;
```

-- Sort and subsort

```
SELECT AddressLine1, City, PostalCode, CountryRegion
FROM SalesLT.Address
ORDER BY CountryRegion, City;
```

-- Descending

```
SELECT AddressLine1, City, PostalCode, CountryRegion
FROM SalesLT.Address
ORDER BY CountryRegion DESC, City ASC;
```

-- TOP

-- Top records

```
SELECT TOP 10 AddressLine1, ModifiedDate
FROM SalesLT.Address
ORDER BY ModifiedDate DESC;
```

-- Top with ties

```
SELECT TOP 10 WITH TIES AddressLine1, ModifiedDate
FROM SalesLT.Address
ORDER BY ModifiedDate DESC;
```

-- Top percent

```
SELECT TOP 10 PERCENT AddressLine1, ModifiedDate
FROM SalesLT.Address
ORDER BY ModifiedDate DESC;
```

-- OFFSET and FETCH

-- First 10 rows

```
SELECT AddressLine1, ModifiedDate
FROM SalesLT.Address
ORDER BY ModifiedDate DESC OFFSET 0 ROWS FETCH NEXT 10 ROWS
ONLY;
```

-- Next page

```
SELECT AddressLine1, ModifiedDate
FROM SalesLT.Address
ORDER BY ModifiedDate DESC OFFSET 10 ROWS FETCH NEXT 10 ROWS
ONLY;
```

-- ALL and DISTINCT

-- Implicit all

```
SELECT City
FROM SalesLT.Address;
```

-- Explicit all

```
SELECT ALL City
FROM SalesLT.Address;
```

-- Distinct

```
SELECT DISTINCT City
FROM SalesLT.Address;
```

-- Distinct combination

```
SELECT DISTINCT City, PostalCode
FROM SalesLT.Address;
```



-- WHERE CLAUSE

-- Simple filter

```
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
ORDER BY City, PostalCode;
```

-- Multiple criteria (and)

```
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND City = 'London'
ORDER BY PostalCode;
```

-- Multiple criteria (or)

```
SELECT AddressLine1, City, PostalCode, CountryRegion
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      OR CountryRegion = 'Canada'
ORDER BY CountryRegion, PostalCode;
```

-- Nested conditions

```
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND (City = 'London' OR City = 'Oxford')
ORDER BY City, PostalCode;
```

-- Not equal to

```
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND City <> 'London'
ORDER BY City, PostalCode;
```

-- Greater than

```
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND City = 'London'
      AND PostalCode > 'S'
ORDER BY PostalCode;
```

```
-- Like with wildcard
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND City = 'London'
      AND PostalCode LIKE 'SW%'
ORDER BY PostalCode;
```

```
-- Like with regex pattern
SELECT AddressLine1, City, PostalCode
FROM SalesLT.Address
WHERE CountryRegion = 'United Kingdom'
      AND City = 'London'
      AND PostalCode LIKE 'SW[0-9] [0-9]__'
ORDER BY PostalCode;
```

```
-- check for null
SELECT AddressLine1, AddressLine2, City, PostalCode
FROM SalesLT.Address
WHERE AddressLine2 IS NOT NULL
ORDER BY City, PostalCode;
```


```
-- within a range
SELECT AddressLine1, ModifiedDate
FROM SalesLT.Address
WHERE ModifiedDate BETWEEN '01/01/2005' AND '12/31/2005'
ORDER BY ModifiedDate;
```

```
-- In a list
SELECT AddressLine1, City, CountryRegion
FROM SalesLT.Address
WHERE CountryRegion IN ('Canada', 'United States')
ORDER BY City;
```

## 18 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).

### 18.1 Module03-demos.sql

-- This script contains demo code for Module 3 of the Transact-SQL course

-- INNER joins

-- Implicit

```
SELECT p.ProductID, m.Name AS Model, p.Name AS Product
FROM SalesLT.Product AS p
JOIN SalesLT.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID
ORDER BY p.ProductID;
```

-- Explicit

```
SELECT p.ProductID, m.Name AS Model, p.Name AS Product
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID
ORDER BY p.ProductID;
```

-- Multiple joins

```
SELECT od.SalesOrderID, m.Name AS Model, p.Name AS ProductName,
od.OrderQty
FROM SalesLT.Product AS p
JOIN SalesLT.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID
JOIN SalesLT.SalesOrderDetail AS od
    ON p.ProductID = od.ProductID
ORDER BY od.SalesOrderID;
```

-- OUTER Joins

-- Left outer join

```
SELECT od.SalesOrderID, p.Name AS ProductName, od.OrderQty
FROM SalesLT.Product AS p
LEFT OUTER JOIN SalesLT.SalesOrderDetail AS od
    ON p.ProductID = od.ProductID
ORDER BY od.SalesOrderID;
```

-- Outer keyword is optional

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

-- CROSS JOIN

-- Every product/city combination

```
SELECT p.Name AS Product, a.City
FROM SalesLT.Product AS p
CROSS JOIN SalesLT.Address AS a;
```

-- SELF JOIN

```
-- Prepare the demo
-- There's no employee table, so we'll create one for this example
CREATE TABLE SalesLT.Employee
(EmployeeID int IDENTITY PRIMARY KEY,
EmployeeName nvarchar(256),
ManagerID int);
GO
-- Get salesperson from Customer table and generate managers
INSERT INTO SalesLT.Employee (EmployeeName, ManagerID)
SELECT DISTINCT Salesperson, NULLIF(CAST(RIGHT(SalesPerson, 1) as INT),
0)
FROM SalesLT.Customer;
GO
UPDATE SalesLT.Employee
SET ManagerID = (SELECT MIN(EmployeeID) FROM SalesLT.Employee
WHERE ManagerID IS NULL)
WHERE ManagerID IS NULL
AND EmployeeID > (SELECT MIN(EmployeeID) FROM SalesLT.Employee
WHERE ManagerID IS NULL);
GO

-- Here's the actual self-join demo
SELECT e.EmployeeName, m.EmployeeName AS ManagerName
FROM SalesLT.Employee AS e
LEFT JOIN SalesLT.Employee AS m
ON e.ManagerID = m.EmployeeID
ORDER BY e.ManagerID;
```

## -- SIMPLE SUBQUERIES

```
-- Scalar subquery
-- Outer query
SELECT p.Name, p.StandardCost
FROM SalesLT.Product AS p
WHERE StandardCost <
    -- Inner query
    (SELECT AVG(StandardCost)
```

```
FROM SalesLT.Product)
ORDER BY p.StandardCost DESC;
```

```
--Multivalue subquery
-- Outer query
SELECT p.Name, p.StandardCost
FROM SalesLT.Product AS p
WHERE p.ProductID IN
    -- Inner query
    (SELECT ProductID
     FROM SalesLT.SalesOrderDetail)
ORDER BY p.StandardCost DESC;
```

#### -- CORRELATED SUBQUERY

```
-- Outer query
SELECT SalesOrderID, CustomerID, OrderDate
FROM SalesLT.SalesOrderHeader AS o1
WHERE SalesOrderID =
    -- Inner query
    (SELECT MAX(SalesOrderID)
     FROM SalesLT.SalesOrderHeader AS o2
     --References alias in outer query
     WHERE o2.CustomerID = o1.CustomerID)
ORDER BY CustomerID, OrderDate;
```

```
-- Outer query
SELECT od.SalesOrderID, od.OrderQty,
    -- Inner query
    (SELECT Name
     FROM SalesLT.Product AS p
     --References alias in outer query
     WHERE p.ProductID = od.ProductID) AS ProductName
FROM SalesLT.SalesOrderDetail AS od
ORDER BY od.SalesOrderID
```

```
-- Using EXISTS
-- Outer query
```

```


SELECT CustomerID, CompanyName, EmailAddress
FROM SalesLT.Customer AS c
WHERE EXISTS
    -- Inner query
    (SELECT *
      FROM SalesLT.SalesOrderHeader AS o
    --References alias in outer query
      WHERE o.CustomerID = c.CustomerID);

```

## 19 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).

### 19.1 Module04-demos.sql

```

-- This script contains demo code for Module 5 of the Transact-SQL course
-- Basic scalar functions

```

```

-- Dates
SELECT SalesOrderID,
       OrderDate,
       YEAR(OrderDate) AS OrderYear,
       DATENAME(mm,OrderDate) AS OrderMonth,
       DAY(OrderDate) AS OrderDay,

```

```

    DATENAME(dw, OrderDate) AS OrderWeekDay,
    DATEDIFF(yy, OrderDate, GETDATE()) AS YearsSinceOrder
FROM SalesLT.SalesOrderHeader;

```

-- Math

```

SELECT TaxAmt,
    ROUND(TaxAmt, 0) AS Rounded,
    FLOOR(TaxAmt) AS Floor,
    CEILING(TaxAmt) AS Ceiling,
    SQUARE(TaxAmt) AS Squared,
    SQRT(TaxAmt) AS Root,
    LOG(TaxAmt) AS Log,
    TaxAmt * RAND() AS Randomized
FROM SalesLT.SalesOrderHeader;

```

-- Text

```

SELECT CompanyName,
    UPPER(CompanyName) AS UpperCase,
    LOWER(CompanyName) AS LowerCase,
    LEN(CompanyName) AS Length,
    REVERSE(CompanyName) AS Reversed,
    CHARINDEX(' ', CompanyName) AS FirstSpace,
    LEFT(CompanyName, CHARINDEX(' ', CompanyName)) AS FirstWord,
    SUBSTRING(CompanyName, CHARINDEX(' ', CompanyName) + 1,
    LEN(CompanyName)) AS RestOfName
FROM SalesLT.Customer;

```

-- Logical

-- IIF

```

SELECT AddressType, -- Evaluation    if True    if False
    IIF(AddressType = 'Main Office', 'Billing', 'Mailing') AS UseAddressFor
FROM SalesLT.CustomerAddress;

```



-- CHOOSE

-- Prepare by updating status to a value between 1 and 5

```
UPDATE SalesLT.SalesOrderHeader
SET Status = SalesOrderID % 5 + 1
```

-- Now use CHOOSE to map the status code to a value in a list

```
SELECT SalesOrderID, Status,
       CHOOSE(Status, 'Ordered', 'Confirmed', 'Shipped', 'Delivered', 'Completed') AS
OrderStatus
FROM SalesLT.SalesOrderHeader;
```

-- RANKING Functions

-- Ranking

```
SELECT TOP 100 ProductID, Name, ListPrice,
       RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
ORDER BY RankByPrice;
```

-- Partitioning

```
SELECT c.Name AS Category, p.Name AS Product, ListPrice,
       RANK() OVER(PARTITION BY c.Name ORDER BY ListPrice DESC)
AS RankByPrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductcategoryID
ORDER BY Category, RankByPrice;
```

-- ROWSET Functions

-- Use OPENROWSET to retrieve external data

-- (Advanced option needs to be enabled to allow this)

```
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
```

```

GO
EXEC sp_configure 'Ad Hoc Distributed Queries', 1;
RECONFIGURE;
GO
-- Now we can use OPENROWSET to connect to an external data source and return
a rowset
SELECT a.*
FROM                                OPENROWSET('SQLNCLI',
'Server=localhost\SQLEXPRESS;Trusted_Connection=yes;',
'SELECT Name, ListPrice
FROM adventureworks.SalesLT.Product') AS a;

```

```

-- Use OPENXML to read data from an XML document into a rowset
-- First prepare an XML document
DECLARE @idoc INT, @doc VARCHAR(1000);
SET @doc = '<Reviews>
    <Review ProductID="1" Reviewer="Paul Henriot">
        <ReviewText>This is a really great product!</ReviewText>
    </Review>
    <Review ProductID="7" Reviewer="Carlos Gonzlez">
        <ReviewText>Fantasic - I love this product!!</ReviewText>
    </Review>
</Reviews>';
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc;
-- Now use OPENXML to read attributes and elements into a rowset
SELECT ProductID, Reviewer, ReviewText
FROM OPENXML (@idoc, '/Reviews/Review',3)
    WITH (ProductID INT,
    Reviewer VARCHAR(20),
    ReviewText VARCHAR(MAX));

```

```

-- Use OPENJSON to read a JSON document
-- First prepare a JSON document
DECLARE @jsonCustomer NVARCHAR(max) = N'{
    "id" : 1,
    "firstName": "John",

```

```

        "lastName": "Smith",
        "dateOfBirth": "2015-03-25T12:00:00"
    }';
-- Now read the JSON values into a rowset
SELECT *
FROM OPENJSON(@jsonCustomer)
WITH (id INT,
      firstName NVARCHAR(50),
      lastName NVARCHAR(50),
      dateOfBirth DATETIME);

-- Aggregate functions and GROUP BY

-- Aggergate functions
SELECT COUNT(*) AS ProductCount,
       MIN(ListPrice) AS MinPrice,
       MAX(ListPrice) AS MaxPrice,
       AVG(ListPrice) AS AvgPrice
FROM SalesLT.Product

-- Group by
SELECT c.Name AS Category,
       COUNT(*) AS ProductCount,
       MIN(p.ListPrice) AS MinPrice,
       MAX(p.ListPrice) AS MaxPrice,
       AVG(p.ListPrice) AS AvgPrice
FROM SalesLT.ProductCategory AS c
JOIN SalesLT.Product AS p
  ON p.ProductCategoryID = c.ProductCategoryID
GROUP BY c.Name -- (can't use alias because GROUP BY happens before
SELECT)
ORDER BY Category; -- (can use alias because ORDER BY happens after SELECT)

-- Filter aggregated groups
-- How NOT to do it!
SELECT c.Name AS Category,
       COUNT(*) AS ProductCount,

```

```

    MIN(p.ListPrice) AS MinPrice,
    MAX(p.ListPrice) AS MaxPrice,
    AVG(p.ListPrice) AS AvgPrice
FROM SalesLT.ProductCategory AS c
JOIN SalesLT.Product AS p
    ON p.ProductCategoryID = c.ProductCategoryID
WHERE COUNT(*) > 1 -- Attempt to filter on grouped aggregate = error!
GROUP BY c.Name
ORDER BY Category;

```

-- How to do it

```

SELECT c.Name AS Category,
    COUNT(*) AS ProductCount,
    MIN(p.ListPrice) AS MinPrice,
    MAX(p.ListPrice) AS MaxPrice,
    AVG(p.ListPrice) AS AvgPrice
FROM SalesLT.ProductCategory AS c
JOIN SalesLT.Product AS p
    ON p.ProductCategoryID = c.ProductCategoryID
GROUP BY c.Name
HAVING COUNT(*) > 1 -- Use HAVING to filter after grouping
ORDER BY Category;


```

## 20 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).

## 20.1 Module05-demos.sql

```
-- This script contains demo code for Module 6 of the Transact-SQL course
-- CREATE A TABLE FOR THE DEMOS
```

```
CREATE TABLE SalesLT.Promotion
(
    PromotionID int IDENTITY PRIMARY KEY,
    PromotionName varchar(20),
    StartDate datetime NOT NULL DEFAULT GETDATE(),
    ProductModelID int NOT NULL REFERENCES
SalesLT.ProductModel(ProductModelID),
    Discount decimal(4,2) NOT NULL,
    Notes nvarchar(max) NULL
);
```

```
-- Show it's empty
SELECT * FROM SalesLT.Promotion;
```

```
-- INSERT
```

```
-- Basic insert with all columns by position
INSERT INTO SalesLT.Promotion
VALUES
('Clearance Sale', '01/01/2021', 23, 0.1, '10% discount')
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Use defaults and NULLs
INSERT INTO SalesLT.Promotion
VALUES
('Pull your socks up', DEFAULT, 24, 0.25, NULL)
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Explicit columns
```

```
INSERT INTO SalesLT.Promotion (PromotionName, ProductModelID, Discount)
VALUES
('Caps Locked', 2, 0.2)
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Multiple rows
```

```
INSERT INTO SalesLT.Promotion
VALUES
('The gloves are off!', DEFAULT, 3, 0.25, NULL),
('The gloves are off!', DEFAULT, 4, 0.25, NULL)
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Insert from query
```

```
INSERT INTO SalesLT.Promotion (PromotionName, ProductModelID, Discount,
Notes)
SELECT DISTINCT 'Get Framed', m.ProductModelID, 0.1, '10% off ' + m.Name
FROM SalesLT.ProductModel AS m
WHERE m.Name LIKE '%frame%';
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- SELECT...INTO
```

```
SELECT SalesOrderID, CustomerID, OrderDate, PurchaseOrderNumber, TotalDue
INTO SalesLT.Invoice
FROM SalesLT.SalesOrderHeader;
```

```
SELECT * FROM SalesLT.Invoice;
```

```
-- Retrieve inserted identity value
```

```
INSERT INTO SalesLT.Promotion (PromotionName, ProductModelID, Discount)
VALUES
```

```
('A short sale',13, 0.3);
```

```
SELECT SCOPE_IDENTITY() AS LatestIdentityInDB;
```

```
SELECT IDENT_CURRENT('SalesLT.Promotion') AS LatestPromotionID;
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Override Identity
```

```
SET IDENTITY_INSERT SalesLT.Promotion ON;
```

```
INSERT INTO SalesLT.Promotion (PromotionID, PromotionName,  
ProductModelID, Discount)
```

```
VALUES
```

```
(20, 'Another short sale',37, 0.3);
```

```
SET IDENTITY_INSERT SalesLT.Promotion OFF;
```

```
SELECT * FROM SalesLT.Promotion;
```

```
-- Sequences
```

```
-- Create sequence
```

```
CREATE SEQUENCE SalesLT.InvoiceNumbers AS INT  
START WITH 72000 INCREMENT BY 1;
```

```
-- Get next value
```

```
SELECT NEXT VALUE FOR SalesLT.InvoiceNumbers;
```

```
-- Get next value again (automatically increments on each retrieval)
```

```
SELECT NEXT VALUE FOR SalesLT.InvoiceNumbers;
```

```
-- Insert using next sequence value
```

```
INSERT INTO SalesLT.Invoice
```

```
VALUES
```

```
(NEXT VALUE FOR SalesLT.InvoiceNumbers, 2, GETDATE(), 'PO12345',  
107.99);
```

```
SELECT * FROM SalesLT.Invoice;
```

-- UPDATE

-- Update a single field

```
UPDATE SalesLT.Promotion
SET Notes = '25% off socks'
WHERE PromotionID = 2;
```

```
SELECT * FROM SalesLT.Promotion;
```

-- Update multiple fields

```
UPDATE SalesLT.Promotion
SET Discount = 0.2, Notes = REPLACE(Notes, '10%', '20%')
WHERE PromotionName = 'Get Framed';
```

```
SELECT * FROM SalesLT.Promotion;
```

-- Update from query

```
UPDATE SalesLT.Promotion
SET Notes = FORMAT(Discount, 'P') + ' off ' + m.Name
FROM SalesLT.ProductModel AS m
WHERE Notes IS NULL
      AND SalesLT.Promotion.ProductModelID = m.ProductModelID;
```

```
SELECT * FROM SalesLT.Promotion;
```

-- Delete data

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

```
SELECT * FROM SalesLT.Promotion;
```

-- Truncate to remove all rows

```
TRUNCATE TABLE SalesLT.Promotion;
```



```
SELECT * FROM SalesLT.Promotion;
```

```
-- Merge insert and update
```

```
-- Create a source table with staged changes (don't worry about the details)
```

```
SELECT SalesOrderID, CustomerID, OrderDate, PurchaseOrderNumber, TotalDue  
* 1.1 AS TotalDue
```

```
INTO #InvoiceStaging
```

```
FROM SalesLT.SalesOrderHeader
```

```
WHERE PurchaseOrderNumber = 'PO29111718'
```

```
UNION
```

```
SELECT 79999, 1, GETDATE(), 'PO54321', 202.99;
```

```
-- Here's the staged data
```

```
SELECT * FROM #InvoiceStaging;
```

```
-- Now merge the staged changes
```

```
MERGE INTO SalesLT.Invoice as i
```

```
USING #InvoiceStaging as s
```

```
ON i.SalesOrderID = s.SalesOrderID
```

```
WHEN MATCHED THEN
```

```
    UPDATE SET i.CustomerID = s.CustomerID,
```

```
            i.OrderDate = GETDATE(),
```

```
            i.PurchaseOrderNumber = s.PurchaseOrderNumber,
```

```
            i.TotalDue = s.TotalDue
```

```
WHEN NOT MATCHED THEN
```

```
    INSERT (SalesOrderID, CustomerID, OrderDate, PurchaseOrderNumber,  
TotalDue)
```

```
    VALUES (s.SalesOrderID, s.CustomerID, s.OrderDate, s.PurchaseOrderNumber,  
s.TotalDue);
```

```
-- View the merged table
```

```
SELECT * FROM SalesLT.Invoice
```

```
ORDER BY OrderDate DESC;
```