COMP 1630
Relational Database Design and SQL

Project 2

by

Yuzhe (Stan) Chen

Table of Contents

Introduction

This project is intended to use as a tutorial for Structured Query Language (SQL). In this report, the basic features of SQL, such as commands that allow the creation of databases and table structures, performing various types of data manipulation and administration, and querying the database to extract useful information, will be demonstrated using the Microsoft SQL Server Management Studio (SSMS). Sample commands and results of execution will be provided for various tasks. These commands and results of execution should be similar across different relational DBMS software that supports SQL. At the end of this report, in the Appendix section, a script that contains the complete set of SQL statements used is provided. The majority of this tutorial is based on the textbook Database Systems: Design, Implementation, & Management, 12th edn (1), if there is anything that is confusing or not easily understood, please consult this textbook for more details.

Part A - Database and Tables

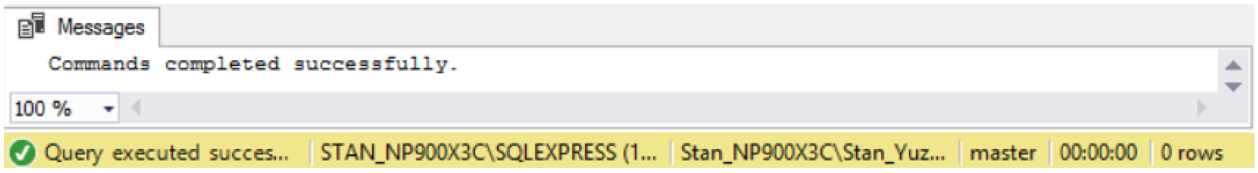
1. Create a database called **Cus_Orders**.

To start with, we need to make sure we have selected the MASTER database. This MASTER database records all the system-level information for a SQL Server system. Moreover, the MASTER is the database that records the existence of all other databases and the location of those database files and records the initialization information for SQL Server. Hence, in SSMS we cannot possibly create any other databases without selecting the MASTER. Note that, the commands used here inside SSMS also refer as SQL statements.

SQL statements used:

```
USE MASTER;
GO
```

Producing the following results:



```
Commands completed successfully.
```

Query executed succes... | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | master | 00:00:00 | 0 rows

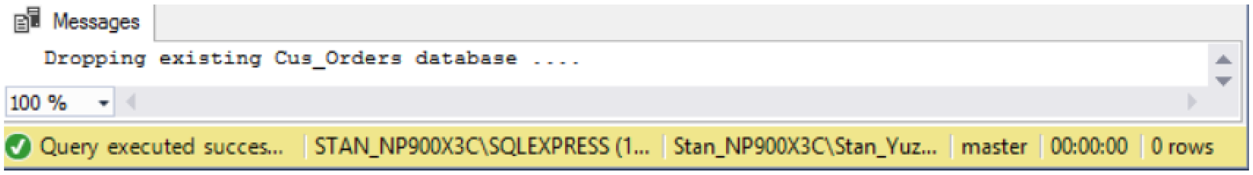
Please note about the use of semi-colon "; " and key word "GO". The semi-colon indicate the end of a statement, while the key word "GO" behaves like the EXECUTE button, in which it will execute all SQL statements before it if encountered. The key word "GO" only relates to SSMS, and it is not an actual Transact SQL. It only tells SSMS to execute the SQL statements between each GO in individual batches sequentially. If we want to include comments in scripts, between each statements, we can either start the comment with double hyphens, "--…", for single line ones, or include them inside the slash and asterisk blocks, "/* … */", for multi-line ones.

The second thing we need to check before creating the new database **Cus_Orders** is to check the existence of old ones with the same name. If it does exist, we need to delete it before we can create the new ones.

SQL statements used:

```
IF EXISTS (SELECT * FROM sysdatabases WHERE name='Cus_Orders')
begin
   raiserror('Dropping existing Cus_Orders database ....',0,1)
   DROP DATABASE Cus_Orders;
end;
GO
```

If the database **Cus_Orders** does exist, it will produce the following results:



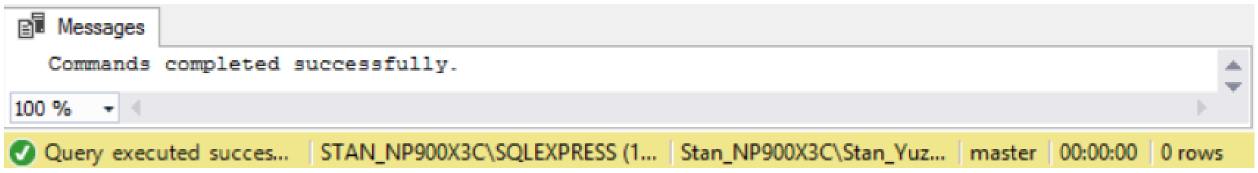
Messages

```
Dropping existing Cus_Orders database ....
```

100 %

Query executed succes... | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | master | 00:00:00 | 0 rows

Otherwise, the following will be produced:



Messages

```
Commands completed successfully.
```

100 %

Query executed succes... | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | master | 00:00:00 | 0 rows

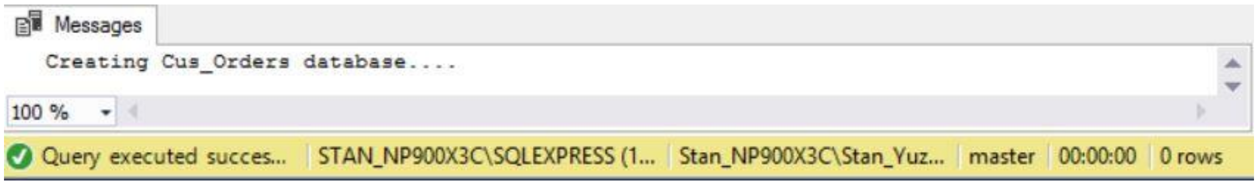
The above statements asks the system databases whether there exists a database called **Cus_Orders**, if it does, then it will print an error message and remove the old database via "DROP" command. Note that the commands for print the error message and dropping the old database were contained in the same "Begin – End" block, this is because IF statements can only take a single SQL statement. If we want to include more statements in a single IF, we need to include these statement in a Begin-End block. Additionally, this error message will not cause the abortion of commands, it is only used to deliver a message as we set the severe level low. Other higher severe levels of the "raiserror" function may cause the termination of client connection to the sever side. For details of using the "raiserror", please consult the Microsoft's Transact SQL websites (2).

Now, we can proceed to the creation of database **Cus_Orders**.

SQL statements used:

```
print 'Creating Cus_Orders database....';
CREATE DATABASE Cus_Orders;
GO
```

Producing the following results:



Messages

```
Creating Cus_Orders database....
```

100 %

Query executed succes... | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | master | 00:00:00 | 0 rows

From the above statement, the database **Cus_Orders** was created using the command "`CREATE DATABASE ...` ". Prior to this, we asked the system to print a message to indicate that the creation process is about to start, using "print", follow by the text message we want to display contained by the single quotes ' **. . .** '.

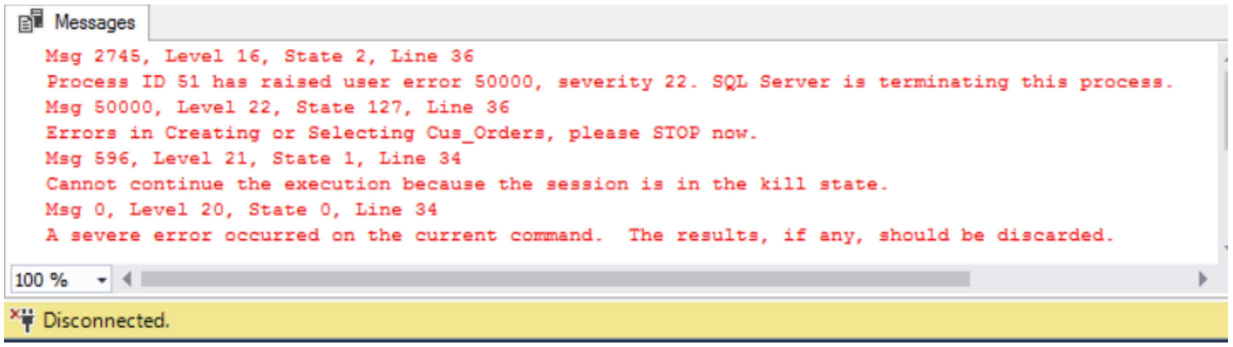
2. Create user defined data types for all similar Primary Key attribute columns (e.g. order_id, product_id, title_id), to ensure the same data type, length and nullability.

For this task, we are going to create user defined data types, which are database objects, for our newly created database **Cus_Orders**, so we want to make sure that we have selected our new database **Cus_Orders** by issuing command 'USE Cus_Orders' before we do anything. Otherwise, we will be creating new objects for the system database or other existing databases, which is not desirable and dangerous. We can add a database selection check to ensure we selected the intended database and not others.

SQL statements used:

```
USE Cus_Orders;
if db_name() <> 'Cus_Orders'
   raiserror('Errors in Creating or Selecting Cus_Orders, please STOP now.'
            ,22,127) with log
else print 'Checked: Cus_Orders in USE!'
GO
```

Producing the following results if **Cus_Orders** IS NOT in use:



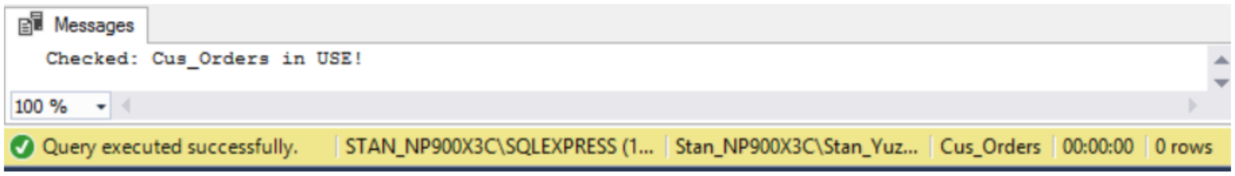
Messages

```
Msg 2745, Level 16, State 2, Line 36
Process ID 51 has raised user error 50000, severity 22. SQL Server is terminating this process.
Msg 50000, Level 22, State 127, Line 36
Errors in Creating or Selecting Cus_Orders, please STOP now.
Msg 596, Level 21, State 1, Line 34
Cannot continue the execution because the session is in the kill state.
Msg 0, Level 20, State 0, Line 34
A severe error occurred on the current command.  The results, if any, should be discarded.
```

100 %

Disconnected.

Alternatively, it will produce the following results if **Cus_Orders** IS in use:



```
Checked: Cus_Orders in USE!
```

Messages
100 %
Query executed successfully.
STAN_NP900X3C\SQLEXPRESS (1...
Stan_NP900X3C\Stan_Yuz...
Cus_Orders
00:00:00
0 rows

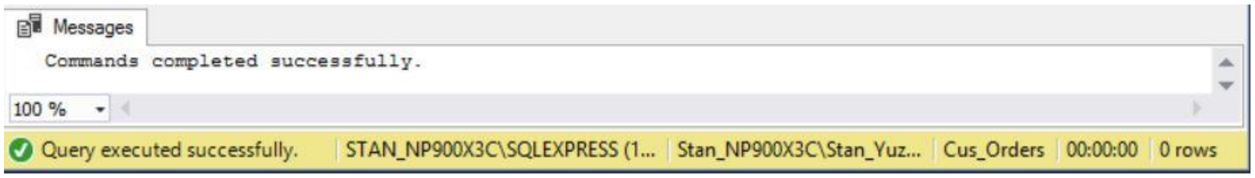
From the above results, we can see that our database checking method works as the process was stopped and connection to the server was closed, as indicated at the status bar, while the **Cus_Orders** database was not selected. As previously mentioned, this is done by setting the severe level to very high for the "raiserror" command, and this can protect the existing databases from mistakenly altered.

After we selected the right database, Cus_Orders, and double check we have done it correctly, we can proceed to the creation of user defined data types with "`CREATE TYPE`". Additionally, since the user defined data types are database objects, we need to check the existence of old ones with the same names and drop them, otherwise the system will prevent the creation of new objects. This is also how we re-create these objects without removing the whole database.

SQL statements used:

```
DROP TYPE IF EXISTS dbo.csid_ch5;
DROP TYPE IF EXISTS dbo.csid_int;
CREATE TYPE csid_ch5 FROM char(5) NOT NULL;
CREATE TYPE csid_int FROM int NOT NULL;
GO
```

Producing the following results:



```
Commands completed successfully.
```

Messages
100 %
Query executed successfully.
STAN_NP900X3C\SQLEXPRESS (1...
Stan_NP900X3C\Stan_Yuz...
Cus_Orders
00:00:00
0 rows

For the above "`CREATE TYPE`" statements, the '`csid_ch5`' was created from the "`char(5)`" data type, which means it is of fixed length of 5 characters, with the restriction of not allowing null values. Similarly, the '`csid_int`' was created from the '`int`' data type, which is just integers.

3. Create the following tables:

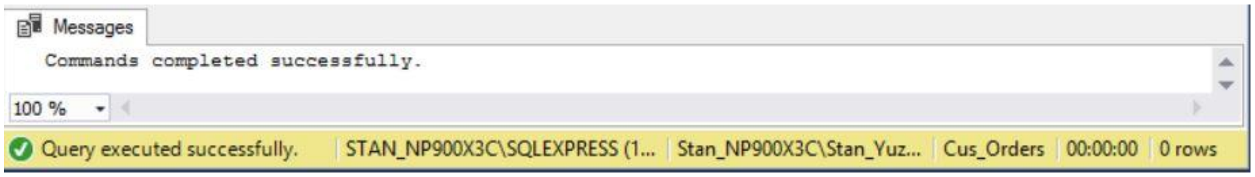
customers
orders
order_details
products
shippers
suppliers
titles

Again, since these tables are database objects, we need to check their existence prior to creating them, and drop them if needed. This is very useful when we want to re-create some of the tables while keeping others unchanged, and without removing the whole database.

SQL statements used:

```
DROP TABLE IF EXISTS dbo.customers;
DROP TABLE IF EXISTS dbo.orders;
DROP TABLE IF EXISTS dbo.order_details;
DROP TABLE IF EXISTS dbo.products;
DROP TABLE IF EXISTS dbo.shippers;
DROP TABLE IF EXISTS dbo.suppliers;
DROP TABLE IF EXISTS dbo.titles;
GO
```

Producing the following results:



```
Commands completed successfully.
```

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 0 rows

After issuing commands to check and remove the pre-existing tables, we can now create new tables we want. An important note about creating tables is that, we need to specify the column names of the table being created, as well as the associated data types and the nullability, and all these information need to be included in a single statement, that is, follow by a single semi-colon "; ". We can also set constrains, primary keys (PKs) or foreign keys (FKs) when we are creating the table, yet in this tutorial, for the easier demonstration of the use of SQL, we will set these variables latter. However, note that, after placing the FK references, the parent tables being referenced cannot be drop before the FK constrains dropped.

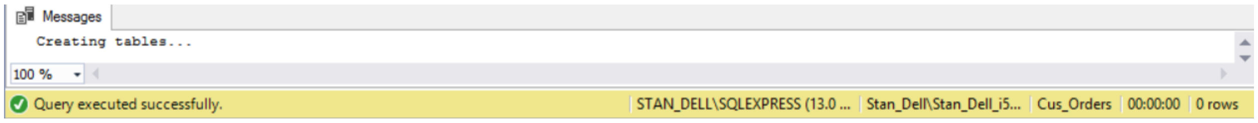
SQL statements used:

```
print 'Creating tables... ';

CREATE TABLE customers(
customer_id csid_ch5,
name varchar(50) NOT NULL,
contact_name varchar(30),
title_id char(3) NOT NULL,
address varchar(50),
city varchar(20),
region varchar(15),
country_code varchar(10),
country varchar(15),
phone varchar(20),
fax varchar(20)
);
GO
```

```
CREATE TABLE orders(
order_id csid_int,
customer_id csid_ch5,
employee_id int NOT NULL,
shipping_name varchar(50),
shipping_address varchar(50),
shipping_city varchar(20),
shipping_region varchar(15),
shipping_country_code varchar(10),
shipping_country varchar(15),
shipper_id int NOT NULL,
order_date datetime,
required_date datetime,
shipped_date datetime,
freight_charge money
);
GO

CREATE TABLE order_details(
order_id csid_int,
product_id int NOT NULL,
quantity int NOT NULL,
discount float NOT NULL
);
GO

CREATE TABLE products(
product_id  csid_int,
supplier_id int NOT NULL,
name varchar(40) NOT NULL,
alternate_name varchar(40),
quantity_per_unit varchar(25),
unit_price money,
quantity_in_stock int,
units_on_order int,
reorder_level int,
);
GO
```

```
CREATE TABLE shippers(
shipper_id int IDENTITY(1,1),
name varchar(20) NOT NULL
);
GO

CREATE TABLE suppliers(
supplier_id int   IDENTITY(1,1) NOT NULL,
name varchar(40) NOT NULL,
address varchar(30),
city varchar(20),
province char(2)
);
GO

CREATE TABLE titles(
title_id char(3) NOT NULL,
description varchar(35) NOT NULL
);
GO
```

Producing the following results:



```
Creating tables...
```

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   0 rows
```

Note that, for the above statements, we included a print command to indicate the start of the creation of the new tables, since the creation of tables will not show any additional messages.

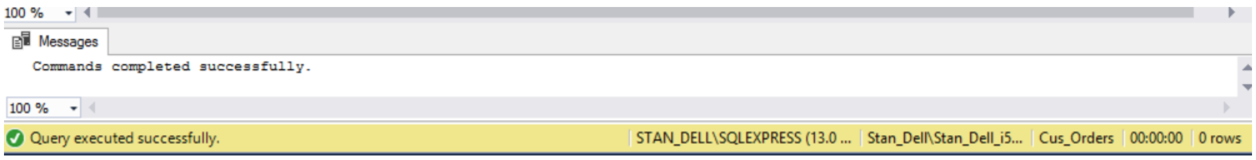
4. Set the **primary keys** and **foreign keys** for the tables.

At this stage, we have created tables with desired columns as well as associated data types and nullability, now we can consider adding PKs and KFs for each table. In order to do this, we first start with adding the PKs, and we have to do this via the "`ALTER TABLE` ..." command as the tables were already created. We need to specify the name of the table that we want to add the PK, after that command. Then we need to supply the column name to which we want it to be the PK, using " `ADD PRIMARY KEY` (...)". For complete commands, please refer below.

SQL statements used:

```
ALTER TABLE customers
ADD PRIMARY KEY ( customer_id );
ALTER TABLE orders
ADD PRIMARY KEY ( order_id );
ALTER TABLE order_details
ADD PRIMARY KEY ( order_id, product_id );
ALTER TABLE titles
ADD PRIMARY KEY ( title_id );
ALTER TABLE shippers
ADD PRIMARY KEY ( shipper_id );
ALTER TABLE suppliers
ADD PRIMARY KEY ( supplier_id );
ALTER TABLE products
ADD PRIMARY KEY ( product_id );
Go
```

Producing the following results:



```
Commands completed successfully.
```

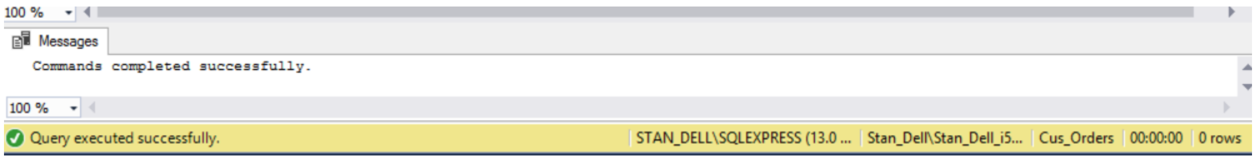
Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

As to the addition of FKs, we also need to do it via the "`ALTER TABLE` …" command, as we are modifying existing tables. However, for the FKs, we are altering the child table to add a constraint for the desired column to ask for reference to the PK of the parent table. Hence the FKs of a table is like a constraint for that table.

SQL statements used:

```
ALTER TABLE customers
ADD CONSTRAINT FK_customer_title FOREIGN KEY (title_id)
REFERENCES titles (title_id);
ALTER TABLE orders
ADD CONSTRAINT FK_orders_customers FOREIGN KEY (customer_id)
REFERENCES customers (customer_id);
ALTER TABLE orders
ADD CONSTRAINT FK_orders_shippers FOREIGN KEY (shipper_id)
REFERENCES shippers (shipper_id);
ALTER TABLE order_details
ADD CONSTRAINT FK_order_details_orders FOREIGN KEY (order_id)
REFERENCES orders (order_id);
ALTER TABLE order_details
ADD CONSTRAINT FK_order_details_products FOREIGN KEY (product_id)
REFERENCES products (product_id);
ALTER TABLE products
ADD CONSTRAINT FK_product_supplier FOREIGN KEY (supplier_id)
REFERENCES suppliers (supplier_id);
GO
```

Producing the following results:


Messages

```
Commands completed successfully.
```

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

As we can see, the above commands executed successfully without other additional messages.

5. Set the constraints as follows:

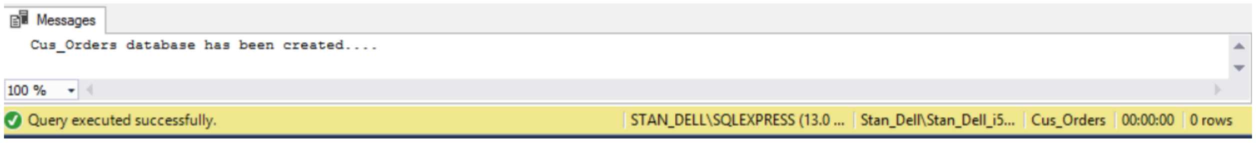
customers table	- country should default to Canada
orders table	- required_date should default to today's date plus ten days
order details table	- quantity must be greater than or equal to 1
products table	- reorder_level must be greater than or equal to 1
	- quantity_in_stock value must not be greater than 150
suppliers table	- province should default to BC

Following the addition of PKs and FKs, we can add other constraints in addition to the FKs, such as the default values and the check of data values, as specify in this task.

SQL statements used:

```
ALTER TABLE customers
ADD CONSTRAINT default_country
      DEFAULT ( 'Canada' ) FOR country;
ALTER TABLE orders
ADD CONSTRAINT default_required_date
      DEFAULT (DATEADD (DAY, 10, getdate())) FOR required_date;
ALTER TABLE order_details
ADD CONSTRAINT ch_min_qty
      CHECK (quantity >= 1);
ALTER TABLE products
ADD CONSTRAINT ch_max_qty_stock
      CHECK (quantity_in_stock <= 150);
ALTER TABLE products
ADD CONSTRAINT ch_min_reorder_lv
      CHECK (reorder_level >= 1);
ALTER TABLE suppliers
ADD CONSTRAINT default_province
      DEFAULT ( 'BC' ) FOR province;
GO


print 'Cus_Orders database has been created....';
Go
```

Producing the following results:



```
Messages
  Cus_Orders database has been created....
```

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

Please note that, while other default values or checks are relatively simple statements, for the default of required date, object `default_required_date`, takes a `GETDATE`() and `DATEADD`() function nested together. The `GETDATE`() will obtain the current system's time, while the `DATEADD` (`DAY`, `10`, …) will add ten days to the result of `GETDATE`().

Moreover, at the end of these commands, we requested a print of the message to indicate the complete creation of the whole data base. At this step, we have created the **Cus_Orders** with required tables with associated columns, PKs, FKs, and constrains and we are now ready to insert data into our database.

6. Load the data into your created tables using the following files:

customers.txt	into the customers table	(91 rows)
orders.txt	into the orders table	(1078 rows)
order_details.txt	into the order_details table	(2820 rows)
products.txt	into the products table	(77 rows)
shippers.txt	into the shippers table	(3 rows)
suppliers.txt	into the suppliers table	(15 rows)
titles.txt	into the titles table	(12 rows)
employees.txt	*into the employees table which is created in* **Part C**	

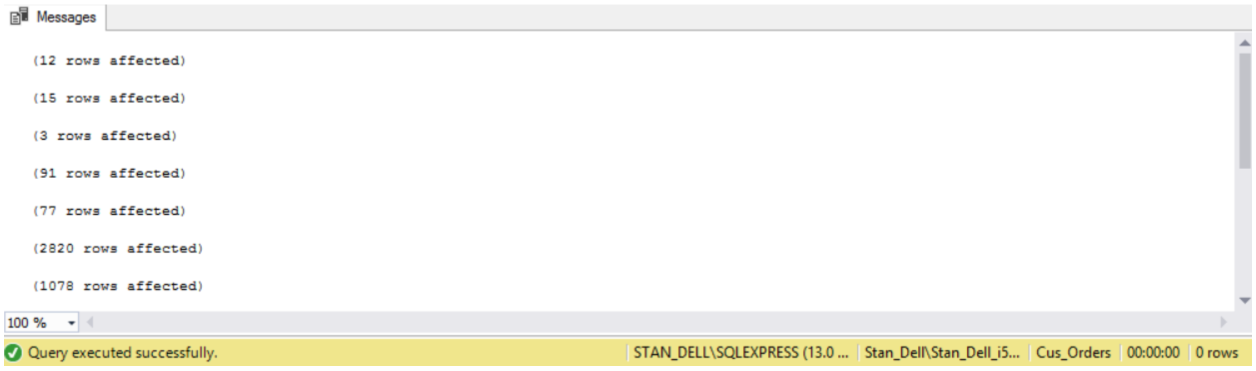
Now, we are going to load the data into our newly created database, **Cus_Orders**. Please note that, the following statements assumes that the data will be picked up from C:\Textfiles\ , so please place the data files into the default location, or change the loading directory in the codes provided.

SQL statements used:

```
BULK INSERT titles
FROM 'C:\TextFiles\titles.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
       )

BULK INSERT suppliers
FROM 'C:\TextFiles\suppliers.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )

BULK INSERT shippers
FROM 'C:\TextFiles\shippers.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )

BULK INSERT customers
FROM 'C:\TextFiles\customers.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )

BULK INSERT products
FROM 'C:\TextFiles\products.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )
```

```
BULK INSERT order_details
FROM 'C:\TextFiles\order_details.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )

BULK INSERT orders
FROM 'C:\TextFiles\orders.txt'
WITH (
            CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
        )
```

Producing the following results:



```
(12 rows affected)

(15 rows affected)

(3 rows affected)

(91 rows affected)

(77 rows affected)

(2820 rows affected)

(1078 rows affected)
```

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

From the above results, we can see that each table of the database **Cus_Orders** have been added with corresponding number of rows of data.

Part B - SQL Statements

1. List the customer id, name, city, and country from the customer table. Order the result set by the customer id. The query should produce the result set listed below.

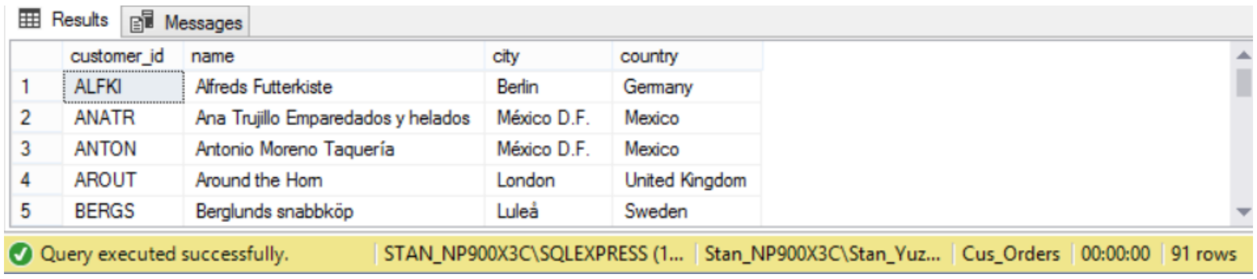
```
customer_id    name                                city           country
---------------  ----------------------------------  -------------  ---------------
ALFKI          Alfreds Futterkiste                 Berlin         Germany
ANATR          Ana Trujillo Emparedados y helados  México D.F.    Mexico
ANTON          Antonio Moreno Taquería             México D.F.    Mexico
AROUT          Around the Horn                     London         United Kingdom
BERGS          Berglunds snabbköp                  Luleå          Sweden
...
WHITC          White Clover Markets                Seattle        United States
WILMK          Wilman Kala                         Helsinki       Finland
WOLZA          Wolski  Zajazd                      Warszawa       Poland

(91 row(s) affected)
```

In this task, we are asked to display information for specific columns from the same customer table, and order them by the customer id. We can use "`SELECT`" command follow by an "`ORDER BY`" clause to complete the task. The SELECT command in general, can be used to extract various information from the columns of specific tables, as will be illustrated latter.

SQL statements used:

```
SELECT customer_id,
       name,
       city,
       country
FROM customers
ORDER BY customer_id;
GO
```

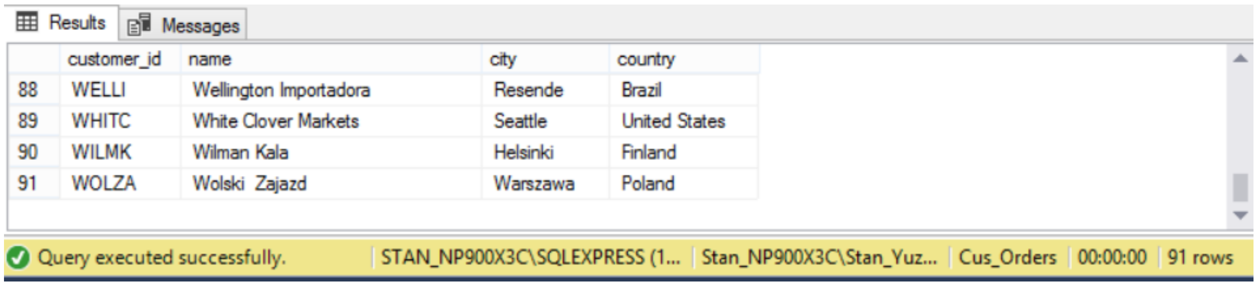
Producing the following results:



	customer_id	name	city	country
1	ALFKI	Alfreds Futterkiste	Berlin	Germany
2	ANATR	Ana Trujillo Emparedados y helados	México D.F.	Mexico
3	ANTON	Antonio Moreno Taquería	México D.F.	Mexico
4	AROUT	Around the Horn	London	United Kingdom
5	BERGS	Berglunds snabbköp	Luleå	Sweden

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 91 rows

…



```
   customer_id  name                      city       country
88 WELLI        Wellington Importadora    Resende    Brazil
89 WHITC        White Clover Markets      Seattle    United States
90 WILMK        Wilman Kala               Helsinki   Finland
91 WOLZA        Wolski  Zajazd            Warszawa   Poland
```

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 91 rows

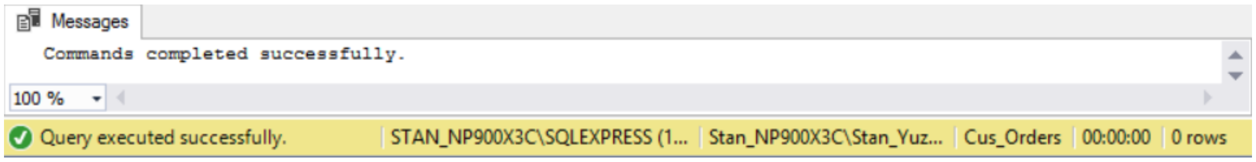
2. Add a new column called active to the customers table using the ALTER statement. The only valid values are 1 or 0. The default should be 1.

In this task, we can use an "ALTER TABLE" command to add the column named "active", and make it into "bit" data type, so that only 0, 1, and NULL are allowed. Next, we apply "`ALTER TABLE`" command again to add a constraint to the "active" column, to make the default values into "1".

SQL statements used:

```
ALTER TABLE customers
ADD active bit;

ALTER TABLE customers
ADD CONSTRAINT default_active
DEFAULT ( '1' ) FOR active;
GO
```

Producing the following results:

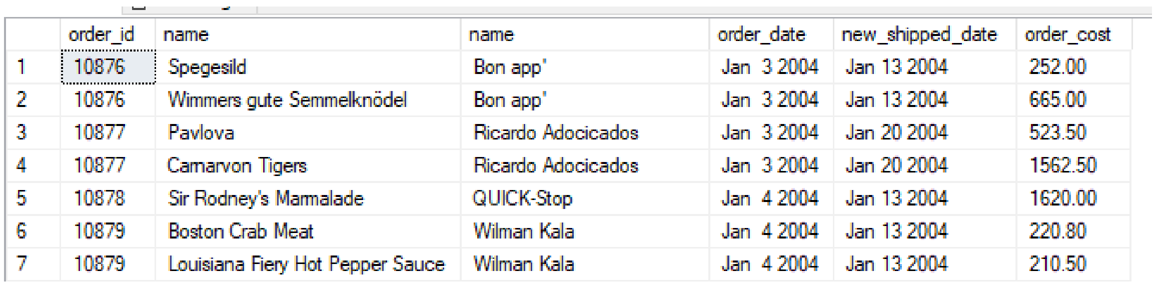


```
Commands completed successfully.
```

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 0 rows

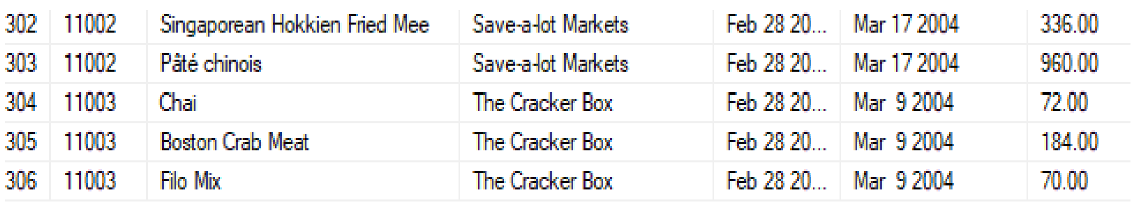
Note, there is no additional data or message to display for this task.

3. List all the orders where the order date is sometime in January or February 2004. Display the order id, order date, and a new shipped date calculated by adding 7 days to the shipped date from the orders table, the product name from the product table, the customer name from the customer table, and the cost of the order. Format the date order date and the shipped date as **MON DD YYYY**. Use the formula (quantity * unit_price) to calculate the cost of the order. The query should produce the result set listed below.



	order_id	name	name	order_date	new_shipped_date	order_cost
1	10876	Spegesild	Bon app'	Jan 3 2004	Jan 13 2004	252.00
2	10876	Wimmers gute Semmelknödel	Bon app'	Jan 3 2004	Jan 13 2004	665.00
3	10877	Pavlova	Ricardo Adocicados	Jan 3 2004	Jan 20 2004	523.50
4	10877	Carnarvon Tigers	Ricardo Adocicados	Jan 3 2004	Jan 20 2004	1562.50
5	10878	Sir Rodney's Marmalade	QUICK-Stop	Jan 4 2004	Jan 13 2004	1620.00
6	10879	Boston Crab Meat	Wilman Kala	Jan 4 2004	Jan 13 2004	220.80
7	10879	Louisiana Fiery Hot Pepper Sauce	Wilman Kala	Jan 4 2004	Jan 13 2004	210.50

…



302	11002	Singaporean Hokkien Fried Mee	Save-a-lot Markets	Feb 28 20...	Mar 17 2004	336.00
303	11002	Pâté chinois	Save-a-lot Markets	Feb 28 20...	Mar 17 2004	960.00
304	11003	Chai	The Cracker Box	Feb 28 20...	Mar 9 2004	72.00
305	11003	Boston Crab Meat	The Cracker Box	Feb 28 20...	Mar 9 2004	184.00
306	11003	Filo Mix	The Cracker Box	Feb 28 20...	Mar 9 2004	70.00

(306 rows should be returned)

In this task, we are going to display information from related tables via the FKs of child tables, using the "`INNER JOIN`" commands. The following tables are to be joint, which are, the orders table, the customers table, and the products table. However, for the customers table and products table to be joint, we need to include the order_details table. Moreover, the quantities of an order can only be obtained from the order_details table. Therefore, four tables will be included for this "`SELECT`" and "`INNER JOIN`" statement.

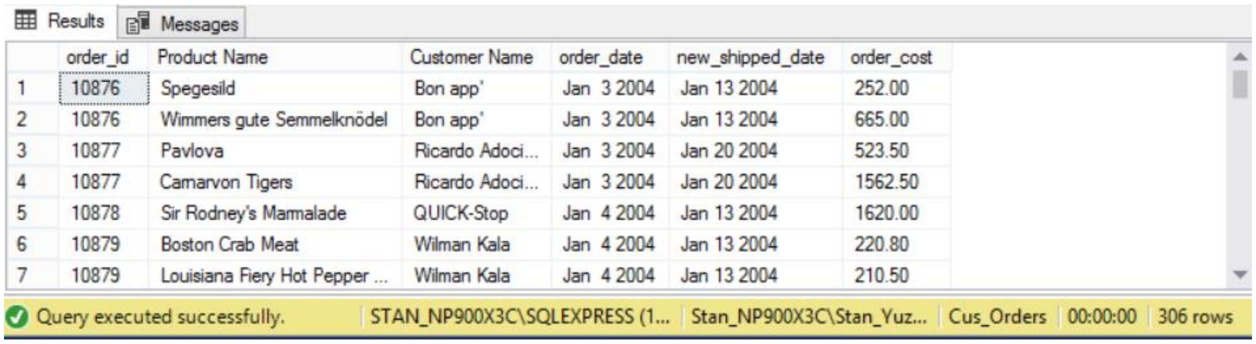
To use the INNER JOIN properly, we need to have the first table listed after the FROM, just as "`SELECT … FROM`" do in the first task of **Part B**. Then the other tables will be listed one by one after each INNER JOIN statement. The FK column of the child table will be referenced to corresponding PK column of parent table after keyword ON of each INNER JOIN using equal sign " = ". Additionally, to specify columns from different tables, we use the follow formats: *tables.specific_columns*, as illustrated below in the codes section.

For the display results to have column headings of our choice, just to include text of choice enclose by single quotation marks ' … '. To display the dates in **MON DD YYYY** format, we need to have "`CONVERT(char(11), … , 109)`" for the intended dates column being asked. After the "`INNER JOIN`", we will include a WHERE command to tell the system which months and years we are looking at.

SQL statements used:

```
SELECT orders.order_id,
       'Product Name' = products.name,
       'Customer Name' = customers.name,
       order_date = CONVERT(char(11), orders.order_date, 109),
       'new_shipped_date'= CONVERT(char(11), DATEADD(DAY, 7,
orders.shipped_date), 109),
       'order_cost' = order_details.quantity*products.unit_price
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id
INNER JOIN order_details ON orders.order_id = order_details.order_id
INNER JOIN products ON order_details.product_id = products.product_id
WHERE ( orders.order_date >= 'January 01 2004' AND orders.order_date <=
'February 29 2004');
GO
```

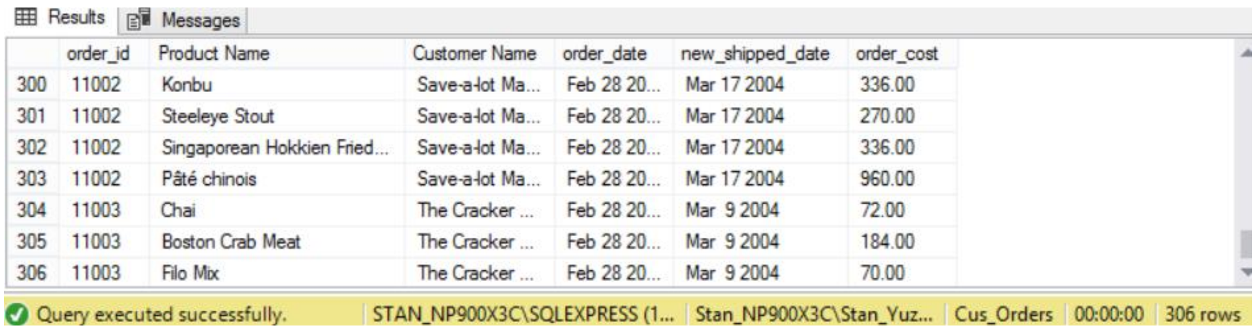
Producing the following results:



	order_id	Product Name	Customer Name	order_date	new_shipped_date	order_cost
1	10876	Spegesild	Bon app'	Jan 3 2004	Jan 13 2004	252.00
2	10876	Wimmers gute Semmelknödel	Bon app'	Jan 3 2004	Jan 13 2004	665.00
3	10877	Pavlova	Ricardo Adoci...	Jan 3 2004	Jan 20 2004	523.50
4	10877	Carnarvon Tigers	Ricardo Adoci...	Jan 3 2004	Jan 20 2004	1562.50
5	10878	Sir Rodney's Marmalade	QUICK-Stop	Jan 4 2004	Jan 13 2004	1620.00
6	10879	Boston Crab Meat	Wilman Kala	Jan 4 2004	Jan 13 2004	220.80
7	10879	Louisiana Fiery Hot Pepper ...	Wilman Kala	Jan 4 2004	Jan 13 2004	210.50

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 306 rows

…



	order_id	Product Name	Customer Name	order_date	new_shipped_date	order_cost
300	11002	Konbu	Save-a-lot Ma...	Feb 28 20...	Mar 17 2004	336.00
301	11002	Steeleye Stout	Save-a-lot Ma...	Feb 28 20...	Mar 17 2004	270.00
302	11002	Singaporean Hokkien Fried...	Save-a-lot Ma...	Feb 28 20...	Mar 17 2004	336.00
303	11002	Pâté chinois	Save-a-lot Ma...	Feb 28 20...	Mar 17 2004	960.00
304	11003	Chai	The Cracker ...	Feb 28 20...	Mar 9 2004	72.00
305	11003	Boston Crab Meat	The Cracker ...	Feb 28 20...	Mar 9 2004	184.00
306	11003	Filo Mix	The Cracker ...	Feb 28 20...	Mar 9 2004	70.00

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 306 rows

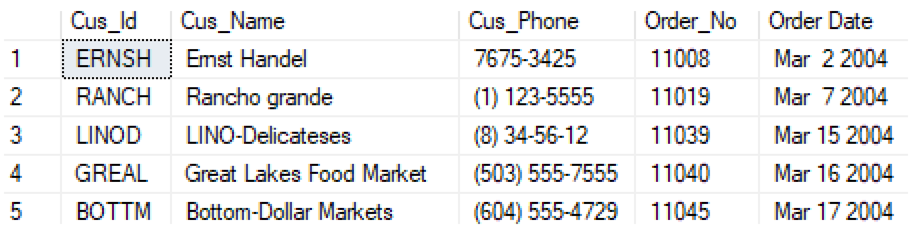
Note that, we used the full date for the range search conditions, and 2004 is a leap year so it has February 29. There is an alternative in which we only specify the months and year in the WHERE clause, in which we use "`DATENAME(MONTH, orders.order_date) = …`" and "`DATENAME(YEAR, orders.order_date) = …`", combined with appropriate use of OR and AND to specify the month and year we interested in. However, as can be seem from below, such statement takes too much to input, so that we do not recommend this alternative, but the full dates for the range search conditions.

Alternative SQL statements that can be used:

```
/* ... (continue from the above, but replace the WHERE clause)*/
WHERE (DATENAME(MONTH, orders.order_date)='January' AND DATENAME(YEAR,
orders.order_date)='2004')
OR (DATENAME(MONTH, orders.order_date)='February' AND DATENAME(YEAR,
orders.order_date)='2004');
```

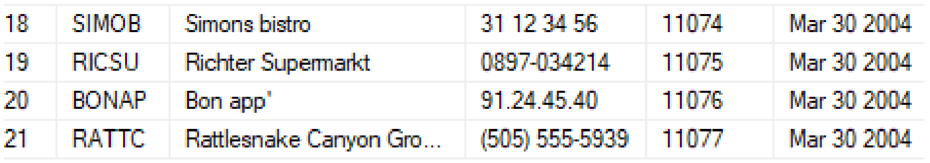
Note, the above alternative WHERE clause will produce exactly the same result as the ones that request the full dates above.

4. List all the orders that have **not** been shipped. Display the customer id, name and phone number from the customers table, and the order id and order date from the orders table. Order the result set by the order date. The query should produce the result set listed below.



	Cus_Id	Cus_Name	Cus_Phone	Order_No	Order Date
1	ERNSH	Ernst Handel	7675-3425	11008	Mar 2 2004
2	RANCH	Rancho grande	(1) 123-5555	11019	Mar 7 2004
3	LINOD	LINO-Delicateses	(8) 34-56-12	11039	Mar 15 2004
4	GREAL	Great Lakes Food Market	(503) 555-7555	11040	Mar 16 2004
5	BOTTM	Bottom-Dollar Markets	(604) 555-4729	11045	Mar 17 2004

…



18	SIMOB	Simons bistro	31 12 34 56	11074	Mar 30 2004
19	RICSU	Richter Supermarkt	0897-034214	11075	Mar 30 2004
20	BONAP	Bon app'	91.24.45.40	11076	Mar 30 2004
21	RATTC	Rattlesnake Canyon Gro...	(505) 555-5939	11077	Mar 30 2004

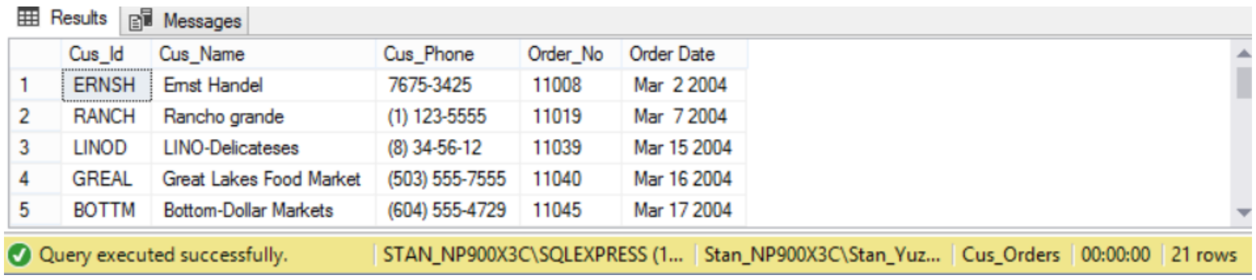
(21 row(s) affected)

This task asked for displaying information from the customers and the orders tables, so the INNER JOIN is going to be used. In addition, this task only asked to display information for orders that have not been shipped, which means that orders.shipped_date column would have to be NULL, so a WHERE clause with key word IS NULL would be required. At the end of statement, we can include an ORDER BY clause to order the results by the order dates. Also for the dates to be in required format, we need to have " CONVERT(char(11), … , 109)" for the appropriate column of dates.

SQL statements used:

```
SELECT 'Cus_Id' = customers.customer_id,
       'Cus_Name' = customers.name,
       'Cus_Phone' = customers.phone,
       'Order_No' = orders.order_id,
       'Order Date' = CONVERT(char(11), orders.order_date, 109)
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.shipped_date IS NULL
ORDER BY orders.order_date;
GO
```

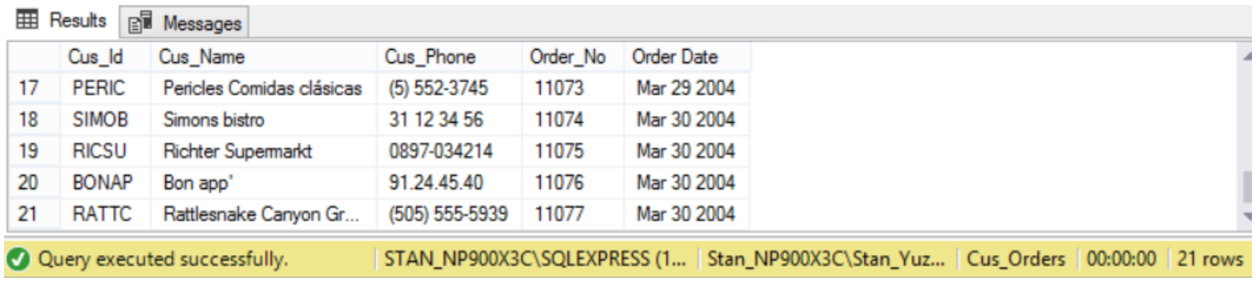
Producing the following results:



	Cus_Id	Cus_Name	Cus_Phone	Order_No	Order Date
1	ERNSH	Ernst Handel	7675-3425	11008	Mar 2 2004
2	RANCH	Rancho grande	(1) 123-5555	11019	Mar 7 2004
3	LINOD	LINO-Delicateses	(8) 34-56-12	11039	Mar 15 2004
4	GREAL	Great Lakes Food Market	(503) 555-7555	11040	Mar 16 2004
5	BOTTM	Bottom-Dollar Markets	(604) 555-4729	11045	Mar 17 2004

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 21 rows
```

…



	Cus_Id	Cus_Name	Cus_Phone	Order_No	Order Date
17	PERIC	Pericles Comidas clásicas	(5) 552-3745	11073	Mar 29 2004
18	SIMOB	Simons bistro	31 12 34 56	11074	Mar 30 2004
19	RICSU	Richter Supermarkt	0897-034214	11075	Mar 30 2004
20	BONAP	Bon app'	91.24.45.40	11076	Mar 30 2004
21	RATTC	Rattlesnake Canyon Gr...	(505) 555-5939	11077	Mar 30 2004

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 21 rows
```

5. List all the customers where the region is **NULL**. Display the customer id, name, and city from the customers table, and the title description from the titles table. The query should produce the result set listed below.

```
customer_id   name                                   city             description
------------  -----------------------------------    ---------------  ----------------------
ALFKI         Alfreds Futterkiste                    Berlin           Sales Representative
ANATR         Ana Trujillo Emparedados y helados     México D.F.      Owner
ANTON         Antonio Moreno Taquería                México D.F.      Owner
AROUT         Around the Horn                        London           Sales Representative
BERGS         Berglunds snabbköp                     Luleå            Order Administrator
...
WARTH         Wartian Herkku                         Oulu             Accounting Manager
WILMK         Wilman Kala                            Helsinki         Owner/Marketing Assistant
WOLZA         Wolski  Zajazd                         Warszawa         Owner
```

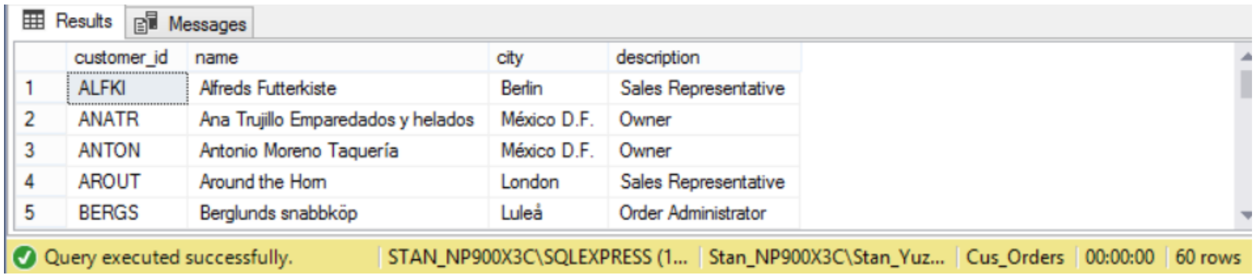
(60 row(s) affected)

This task asked for displaying information from the customers and the titles tables, so the INNER JOIN is going to be used. It also asked to display rows in which the regions for customers to be NULL, so a WHERE clause with IS NULL would be required.

SQL statements used:

```
SELECT customers.customer_id,
       customers.name,
       customers.city,
       titles.description
FROM customers
INNER JOIN titles ON customers.title_id = titles.title_id
WHERE customers.region IS NULL;
GO
```

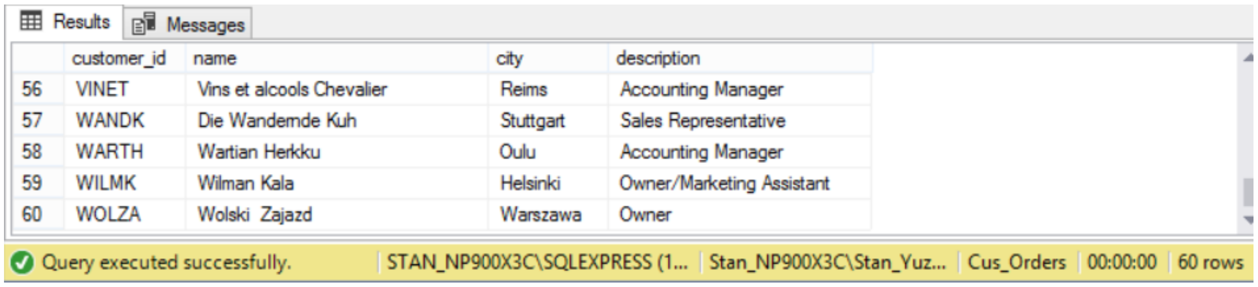
Producing the following results:



	customer_id	name	city	description
1	ALFKI	Alfreds Futterkiste	Berlin	Sales Representative
2	ANATR	Ana Trujillo Emparedados y helados	México D.F.	Owner
3	ANTON	Antonio Moreno Taquería	México D.F.	Owner
4	AROUT	Around the Horn	London	Sales Representative
5	BERGS	Berglunds snabbköp	Luleå	Order Administrator

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 60 rows

…



	customer_id	name	city	description
56	VINET	Vins et alcools Chevalier	Reims	Accounting Manager
57	WANDK	Die Wandernde Kuh	Stuttgart	Sales Representative
58	WARTH	Wartian Herkku	Oulu	Accounting Manager
59	WILMK	Wilman Kala	Helsinki	Owner/Marketing Assistant
60	WOLZA	Wolski Zajazd	Warszawa	Owner

Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 60 rows

6. List the products where the reorder level is higher than the quantity in stock. Display the supplier name from the suppliers table, the product name, reorder level, and quantity in stock from the products table. Order the result set by the supplier name. The query should produce the result set listed below.

```
supplier_name                  product_name             reorder_level    quantity_in_stock
------------------------       ---------------------    ----------------    -----------------
Armstrong Company              Queso Cabrales           30               22
Cadbury Products Ltd.          Ipoh Coffee              25               17
Cadbury Products Ltd.          Røgede sild              15               5
Campbell Company               Gnocchi di nonna Alice   30               21
Dare Manufacturer Ltd.         Scottish Longbreads      15               6
...
Steveston Export Company       Gravad lax               25               11
Steveston Export Company       Outback Lager            30               15
Yves Delorme Ltd.              Longlife Tofu            5                4
```

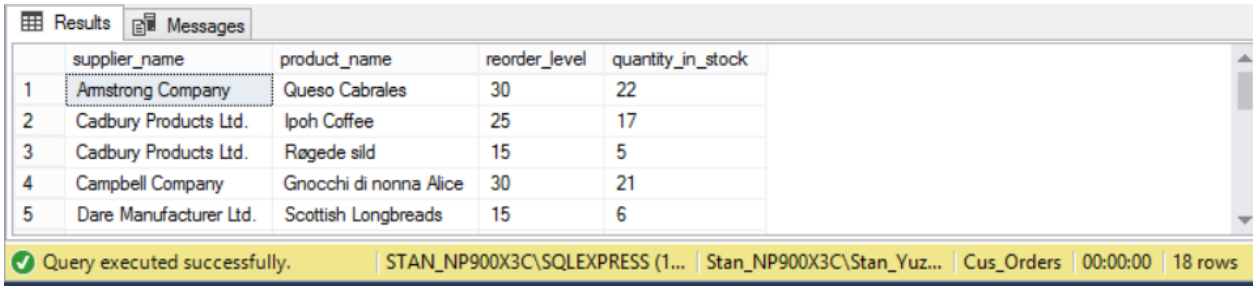
(18 row(s) affected)

This task asked for displaying information from the suppliers and the products tables, so the INNER JOIN is going to be used. In addition, a comparison operation is used in a WHERE clause in order to specify the products where the reorder level is higher than the quantity in stock. Finally, an ORDER BY is used to order the results by supplier name.

SQL statements used:

```
SELECT 'supplier_name' = suppliers.name,
       'product_name' = products.name,
       products.reorder_level,
       products.quantity_in_stock
FROM suppliers
INNER JOIN products ON suppliers.supplier_id = products.supplier_id
WHERE products.reorder_level > products.quantity_in_stock
ORDER BY suppliers.name;
GO
```

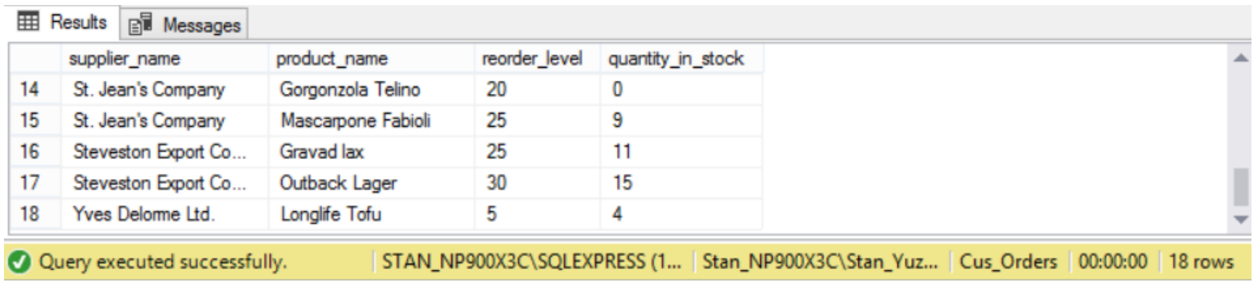
Producing the following results:



	supplier_name	product_name	reorder_level	quantity_in_stock
1	Armstrong Company	Queso Cabrales	30	22
2	Cadbury Products Ltd.	Ipoh Coffee	25	17
3	Cadbury Products Ltd.	Røgede sild	15	5
4	Campbell Company	Gnocchi di nonna Alice	30	21
5	Dare Manufacturer Ltd.	Scottish Longbreads	15	6

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 18 rows
```

…



	supplier_name	product_name	reorder_level	quantity_in_stock
14	St. Jean's Company	Gorgonzola Telino	20	0
15	St. Jean's Company	Mascarpone Fabioli	25	9
16	Steveston Export Co...	Gravad lax	25	11
17	Steveston Export Co...	Outback Lager	30	15
18	Yves Delorme Ltd.	Longlife Tofu	5	4

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 18 rows
```

7. Calculate the length in years from **January 1, 2008** and when an order was shipped where the shipped date is **not null**. Display the order id, and the shipped date from the orders table, the customer name, and the contact name from the customers table, and the length in years for each order. Display the shipped date in the format **MMM DD YYYY**. Order the result set by order id and the calculated years. The query should produce the result set listed below.

```
order_id     name                        contact_name                shipped_date     elapsed
-----------  --------------------------  --------------------------  ------------     -------
10000        Franchi S.p.A.              Paolo Accorti               May 15 2001      7
10001        Mère Paillarde              Jean Fresnière              May 23 2001      7
10002        Folk och fä HB              Maria Larsson               May 17 2001      7
10003        Simons bistro               Jytte Petersen              May 24 2001      7
10004        Vaffeljernet                Palle Ibsen                 May 20 2001      7
...
11066        White Clover Markets        Karl Jablonski              Mar 28 2004      4
11067        Drachenblut Delikatessen    Sven Ottlieb                Mar 30 2004      4
11069        Tortuga Restaurante         Miguel Angel Paolino        Mar 30 2004      4
```

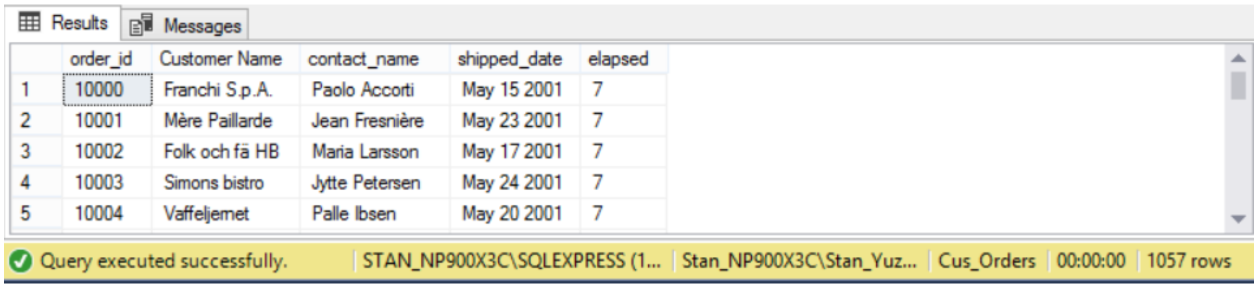
(1057 row(s) affected)

This task asked for displaying information from the orders and the customers tables, so the INNER JOIN is going to be used. It also requested the orders be shipped so the shipped date is not null, which means we need to have a WHERE clause with IS NOT NULL for the shipped dates. To calculate the elapsed years from **January 1, 2008**, we used a DATEDIFF function, which is "`DATEDIFF(YEAR, orders.shipped_date, 'Jan 01 2008')`". The conversion of date format is done in the same fashion as in Task **3** and **4** of this part (**Part B**).

SQL statements used:

```
SELECT orders.order_id,
       'Customer Name' = customers.name,
       customers.contact_name,
       'shipped_date' = CONVERT(char(11),
       orders.shipped_date, 109),
       'elapsed' = DATEDIFF(YEAR, orders.shipped_date, 'Jan 01 2008')
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id
WHERE orders.shipped_date IS NOT NULL
ORDER BY orders.order_id, 'elapsed';
GO
```

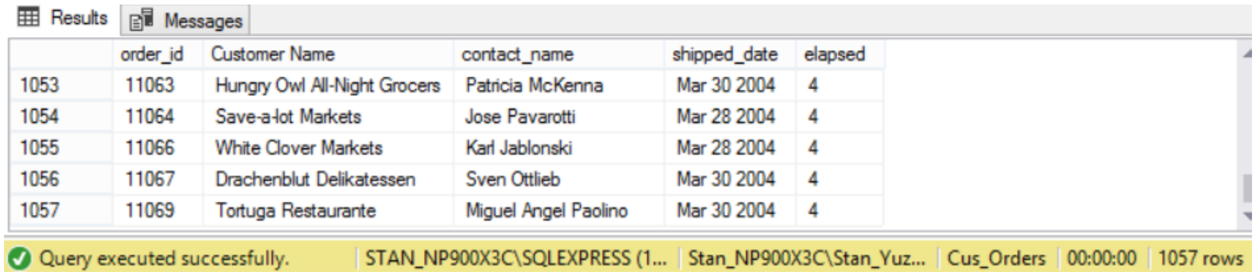
Producing the following results:



	order_id	Customer Name	contact_name	shipped_date	elapsed
1	10000	Franchi S.p.A.	Paolo Accorti	May 15 2001	7
2	10001	Mère Paillarde	Jean Fresnière	May 23 2001	7
3	10002	Folk och fä HB	Maria Larsson	May 17 2001	7
4	10003	Simons bistro	Jytte Petersen	May 24 2001	7
5	10004	Vaffeljernet	Palle Ibsen	May 20 2001	7

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 1057 rows
```

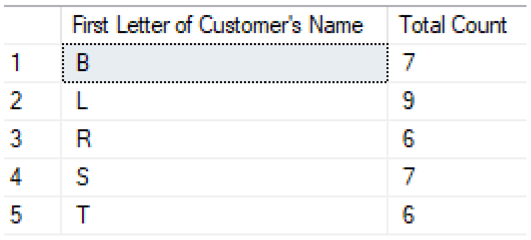
…



	order_id	Customer Name	contact_name	shipped_date	elapsed
1053	11063	Hungry Owl All-Night Grocers	Patricia McKenna	Mar 30 2004	4
1054	11064	Save-a-lot Markets	Jose Pavarotti	Mar 28 2004	4
1055	11066	White Clover Markets	Karl Jablonski	Mar 28 2004	4
1056	11067	Drachenblut Delikatessen	Sven Ottlieb	Mar 30 2004	4
1057	11069	Tortuga Restaurante	Miguel Angel Paolino	Mar 30 2004	4

```
Query executed successfully. | STAN_NP900X3C\SQLEXPRESS (1... | Stan_NP900X3C\Stan_Yuz... | Cus_Orders | 00:00:00 | 1057 rows
```

8. List number of customers with names beginning with each letter of the alphabet. Ignore customers whose name begins with the letter F or G. Do not display the letter and count unless at least six customer's names begin with the letter. The query should produce the result set listed below.



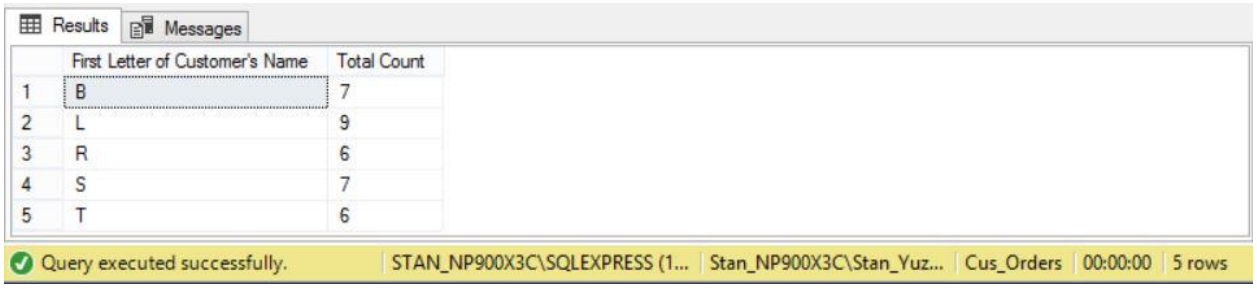
	First Letter of Customer's Name	Total Count
1	B	7
2	L	9
3	R	6
4	S	7
5	T	6

This task asked for the first letter of the customer's name, so " `SUBSTRING(name, 1,1)` " is going to be used, in which the first letter of the rows in name column of customer table will be displayed, starting at the first letter's location. Next, a COUNT is used to count the total occurrences that satisfy the criterions. At the end of the statement, we need a GROUP BY and HAVING clause to sort the results extracted instead of the ORDER BY and WHERE. This is because COUNT is an aggregate function, while HAVING can be used with the aggregate functions, the WHERE clause cannot. GROUP BY is better because ORDER BY will display each individual rows with aggregate functions, while GROUP BY can show the subtotalling. Moreover, we used " != " to represent not equal to.

SQL statements used:

```
SELECT 'First Letter of Customer''s Name' = substring(name, 1,1), 'Total
Count' = count(*)
FROM customers
GROUP by substring(name, 1,1)
HAVING substring(name, 1,1) != 'F' AND substring(name, 1,1) != 'G' AND
count(*) >= 6;
GO
```

Producing the following results:



	First Letter of Customer's Name	Total Count
1	B	7
2	L	9
3	R	6
4	S	7
5	T	6

```
Query executed successfully.   STAN_NP900X3C\SQLEXPRESS (1...   Stan_NP900X3C\Stan_Yuz...   Cus_Orders   00:00:00   5 rows
```

9. List the order details where the quantity is greater than 100. Display the order id and quantity from the order_details table, the product id and reorder level from the products table, and the supplier id from the suppliers table. Order the result set by the order id. The query should produce the result set listed below.

```
order_id     quantity      product_id     reorder_level     supplier_id
-----------  -----------   -------------- ----------------- -----------
10193        110           43             25                10
10226        110           29             0                 12
10398        120           55             20                15
10451        120           55             20                15
10515        120           27             30                11
...
10895        110           24             0                 10
11017        110           59             0                 8
11072        130           64             30                12
```

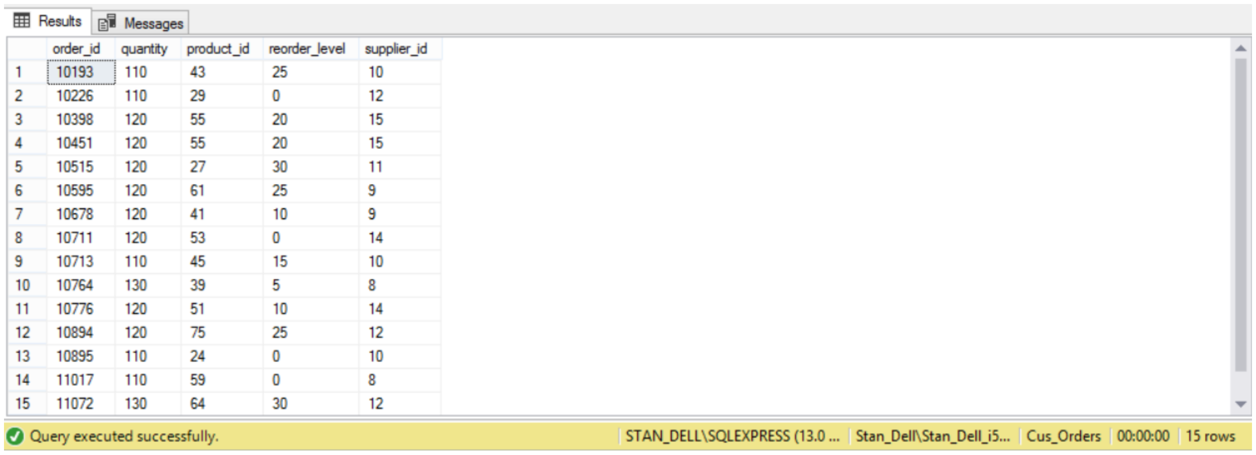
(15 row(s) affected)

In this task, columns from different tables are being asked to display, so INNER JOINs are going to be used. Then, a WHERE clause will be used to only allow orders with quantities greater than 100 to be displayed, follow by an ORDER BY to sort the results by order id.

SQL statements used:

```
SELECT order_details.order_id,
       order_details.quantity,
       products.product_id,
       products.reorder_level,
       suppliers.supplier_id
FROM order_details
INNER JOIN products ON order_details.product_id = products.product_id
INNER JOIN suppliers ON products.supplier_id = suppliers.supplier_id
WHERE order_details.quantity > 100
ORDER BY order_details.order_id;
GO
```

Producing the following results:



	order_id	quantity	product_id	reorder_level	supplier_id
1	10193	110	43	25	10
2	10226	110	29	0	12
3	10398	120	55	20	15
4	10451	120	55	20	15
5	10515	120	27	30	11
6	10595	120	61	25	9
7	10678	120	41	10	9
8	10711	120	53	0	14
9	10713	110	45	15	10
10	10764	130	39	5	8
11	10776	120	51	10	14
12	10894	120	75	25	12
13	10895	110	24	0	10
14	11017	110	59	0	8
15	11072	130	64	30	12

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 15 rows
```

10. List the products which contain tofu or chef in their name. Display the product id, product name, quantity per unit and unit price from the products table. Order the result set by product name. The query should produce the result set listed below.

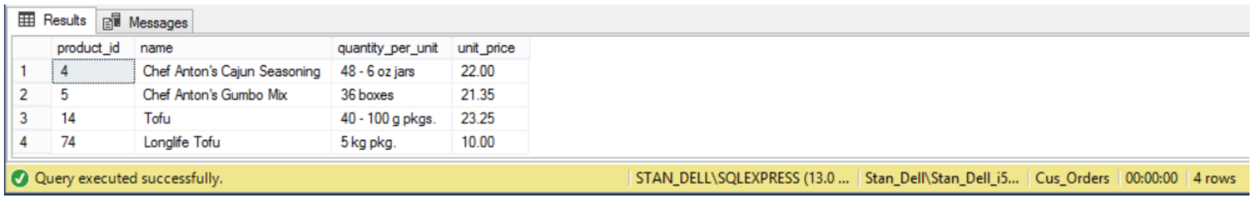
```
product_id     name                                     quantity_per_unit        unit_price
-------------- ---------------------------------------- ------------------------ --------------
4              Chef Anton's Cajun Seasoning             48 - 6 oz jars           22.0000
5              Chef Anton's Gumbo Mix                   36 boxes                 21.3500
74             Longlife Tofu                            5 kg pkg.                10.0000
14             Tofu                                     40 - 100 g pkgs.         23.2500

(4 row(s) affected)
```

This task involves Pattern Match Search Conditions, in which products that contain tofu or chef in their names will be displayed. While other columns to be displayed are also from the products table, we will not use INNER JOIN. To specify the pattern to be matched, we used " `LIKE` " key word in a WHERE clause, and enclose the pattern to be matched by " % ", to indicate the parts that can be any string of zero or more characters.

SQL statements used:

```
SELECT product_id, name, quantity_per_unit, unit_price
FROM products
WHERE name LIKE '%tofu%' OR name LIKE '%chef%';
GO
```

Producing the following results:



	product_id	name	quantity_per_unit	unit_price
1	4	Chef Anton's Cajun Seasoning	48 - 6 oz jars	22.00
2	5	Chef Anton's Gumbo Mix	36 boxes	21.35
3	14	Tofu	40 - 100 g pkgs.	23.25
4	74	Longlife Tofu	5 kg pkg.	10.00

```
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   4 rows
```

Part C - INSERT, UPDATE, DELETE and VIEWS Statements

1. Create an **employee** table with the following columns:

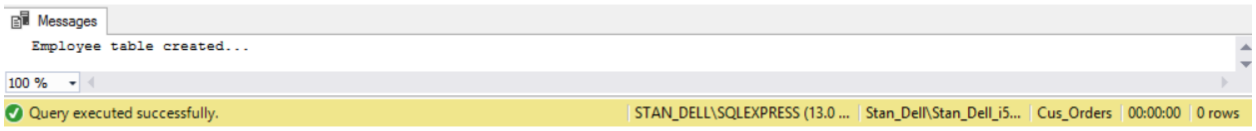
Column Name	*Data Type*	*Length*	*Null Values*
employee_id	int		No
last_name	varchar	30	No
first_name	varchar	15	No
address	varchar	30	
city	varchar	20	
province	char	2	
postal_code	varchar	7	
phone	varchar	10	
birth_date	datetime		No

As previously mention, we need to check the existence of the table, prior to creating the new ones. Then we can create table with the above specified columns and data types. We also included a print statement to indicate the creation of this new employee table.

SQL statements used:

```
DROP TABLE IF EXISTS dbo.employee;

CREATE TABLE employee(
employee_id int NOT NULL,
last_name varchar(30) NOT NULL,
first_name varchar(15)  NOT NULL,
address varchar(30),
city varchar(20),
province char(2),
postal_code varchar(7),
phone varchar(10),
birth_date datetime NOT NULL
);

print 'Employee table created...';
GO
```

Producing the following results:



```
Employee table created...
```

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

2. The **primary key** for the employee table should be the employee id.

As in **Part A**, task **4**, add the PK via ALTER TABLE statements.

SQL statements used:

```
ALTER TABLE employee
ADD PRIMARY KEY ( employee_id );
GO
```

Producing the following results:

```
Commands completed successfully.
```

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

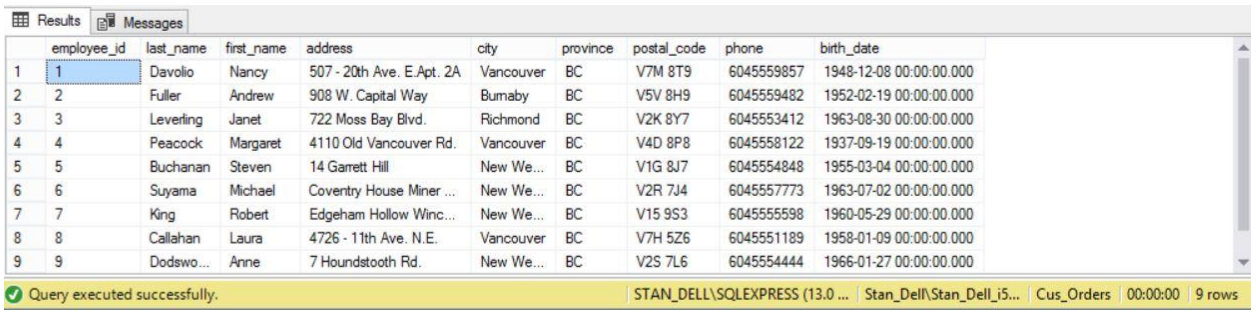
3. Load the data into the employee table using the employee.txt file; **9** rows. In addition, **create the relationship** to enforce referential integrity between the employee and orders tables.

Similar to Part A, task 6, load the data for the employee table. We have also included a statement, " `SELECT *` ", to display all the rows and columns of the employee table to check what we have added.

SQL statements used:

```
BULK INSERT employee
FROM 'C:\TextFiles\employee.txt'
WITH (          CODEPAGE=1252,
            DATAFILETYPE = 'char',
            FIELDTERMINATOR = '\t',
            KEEPNULLS,
            ROWTERMINATOR = '\n'
       );

SELECT *
FROM employee;
GO
```

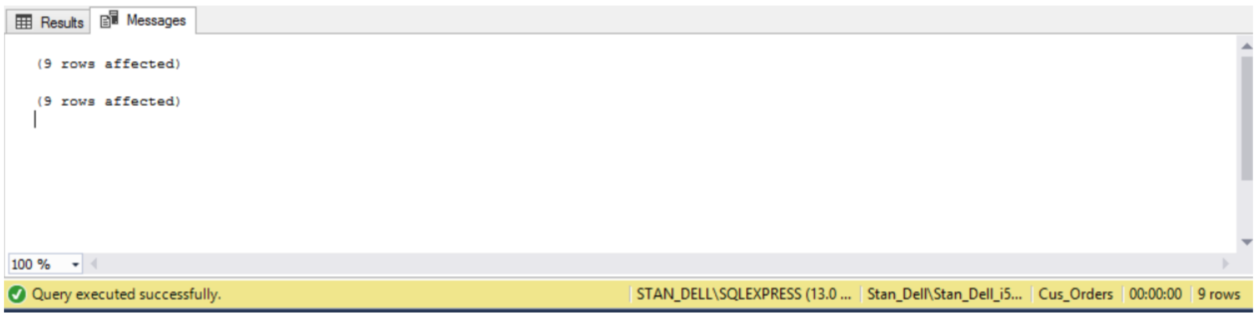
Producing the following results and messages:



Results | Messages

	employee_id	last_name	first_name	address	city	province	postal_code	phone	birth_date
1	1	Davolio	Nancy	507 - 20th Ave. E.Apt. 2A	Vancouver	BC	V7M 8T9	6045559857	1948-12-08 00:00:00.000
2	2	Fuller	Andrew	908 W. Capital Way	Burnaby	BC	V5V 8H9	6045559482	1952-02-19 00:00:00.000
3	3	Leverling	Janet	722 Moss Bay Blvd.	Richmond	BC	V2K 8Y7	6045553412	1963-08-30 00:00:00.000
4	4	Peacock	Margaret	4110 Old Vancouver Rd.	Vancouver	BC	V4D 8P8	6045558122	1937-09-19 00:00:00.000
5	5	Buchanan	Steven	14 Garrett Hill	New We...	BC	V1G 8J7	6045554848	1955-03-04 00:00:00.000
6	6	Suyama	Michael	Coventry House Miner ...	New We...	BC	V2R 7J4	6045557773	1963-07-02 00:00:00.000
7	7	King	Robert	Edgeham Hollow Winc...	New We...	BC	V15 9S3	6045555598	1960-05-29 00:00:00.000
8	8	Callahan	Laura	4726 - 11th Ave. N.E.	Vancouver	BC	V7H 5Z6	6045551189	1958-01-09 00:00:00.000
9	9	Dodswo...	Anne	7 Houndstooth Rd.	New We...	BC	V2S 7L6	6045554444	1966-01-27 00:00:00.000

```
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 9 rows
```



Results | Messages

```
(9 rows affected)

(9 rows affected)
```

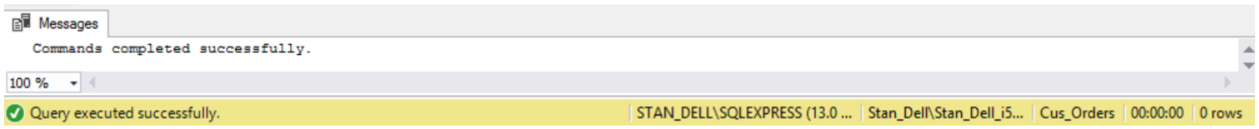
```
100 %
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 9 rows
```

Next, we can create the FK reference between employee table and orders table, similar to what we have done in **Part A**, task **4**. Statement ALTER TABLE is used, followed by addition of FOREIGN KEY constraints to the child table, which is the orders table, to reference the parent, employee table.

SQL statements used:

```
ALTER TABLE orders
ADD CONSTRAINT FK_orders_employee FOREIGN KEY (employee_id)
REFERENCES employee (employee_id);
GO
```

Producing the following results and messages:



Messages

```
Commands completed successfully.
```

```
100 %
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows
```

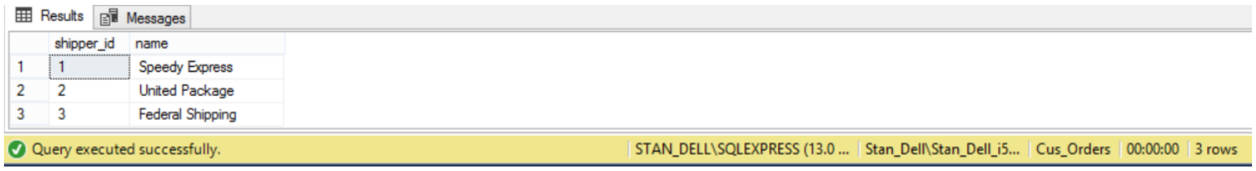
4. Using the INSERT statement, add the shipper **Quick Express** to the shippers table.

First of all, we should have a check on the data of shipper table, before we alter it.

SQL statements used:

```
SELECT *
FROM shippers;
GO
```

Producing the following results:



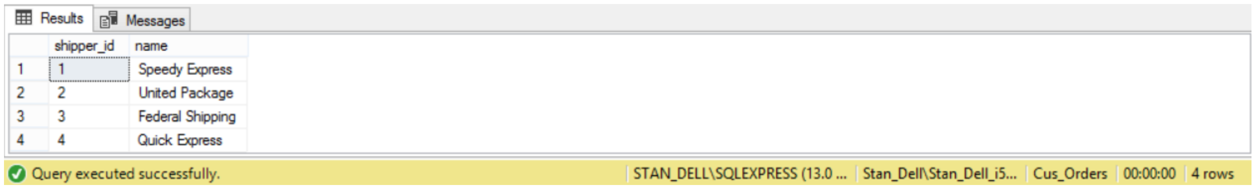
```
Results  Messages
   shipper_id  name
1  1           Speedy Express
2  2           United Package
3  3           Federal Shipping
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...  Stan_Dell\Stan_Dell_i5...  Cus_Orders  00:00:00  3 rows
```

There is nothing special, as there is only one column, the name, besides the auto incremented PK, shipper_id. We are now safe to proceed with the insertion of name " Quick Express ". This follows by another SELECT * statement to check if the desired value have been added.

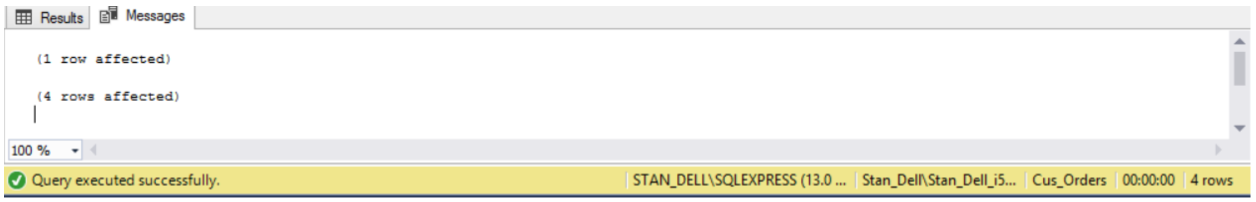
SQL statements used:

```
INSERT INTO shippers (name)
VALUES ('Quick Express ');
SELECT *
FROM shippers;
GO
```

Producing the following results and messages:



```
Results  Messages
   shipper_id  name
1  1           Speedy Express
2  2           United Package
3  3           Federal Shipping
4  4           Quick Express
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...  Stan_Dell\Stan_Dell_i5...  Cus_Orders  00:00:00  4 rows
```



```
Results  Messages

(1 row affected)

(4 rows affected)

100 %
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...  Stan_Dell\Stan_Dell_i5...  Cus_Orders  00:00:00  4 rows
```

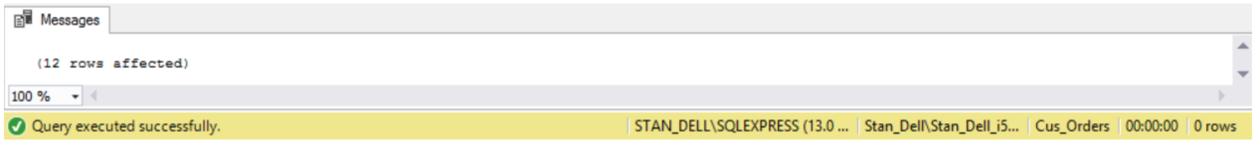
5. Using the UPDATE statement, increate the unit price in the products table of all rows with a current unit price between **$5.00** and **$10.00** by **5%**; 12 rows affected.

This task used an UPDATE statement, to SET the unit prices to be 1.05 times of original, follows by a WHERE clause to restrict unit prices to be between $5 and $10 to be affected.

SQL statements used:

```
UPDATE products
SET unit_price = products.unit_price*1.05
WHERE unit_price >= 5 AND unit_price <= 10;
GO
```

Producing the following results:



```
(12 rows affected)
```

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   0 rows
```

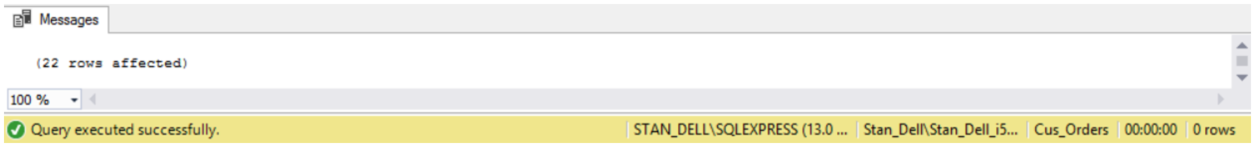
6. Using the UPDATE statement, change the fax value to Unknown for all rows in the customers table where the current fax value is NULL; 22 rows affected.

This task is similar to the last one, except simpler. We only need to UPDATE the customers table to SET the fax to 'Unknow', follow by a WHERE clause to specify that only rows with NULL values in the fax should be affected.

SQL statements used:

```
UPDATE customers
SET fax = 'Unknown'
WHERE fax IS NULL;
GO
```

Producing the following results:



```
(22 rows affected)
```

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   0 rows
```

7. Create a view called **vw_order_cost** to list the cost of the orders. Display the order id and order_date from the orders table, the product id from the products table, the customer name from the customers table, and the order cost. To calculate the cost of the orders, use the formula (order_details.quantity * products.unit_price). Run the view for the order ids between **10000** and **10200**. The view should produce the result set listed below.

```
order_id    order_date                 product_id     name                             order_cost
---------   ------------------------   --------------  -------------------------------  --------------
10000       2001-05-10 00:00:00.000    17             Franchi S.p.A.                   156.0000
10001       2001-05-13 00:00:00.000    25             Mère Paillarde                   420.0000
10001       2001-05-13 00:00:00.000    40             Mère Paillarde                   736.0000
10001       2001-05-13 00:00:00.000    59             Mère Paillarde                   440.0000
10001       2001-05-13 00:00:00.000    64             Mère Paillarde                   498.7500
...
10199       2002-03-27 00:00:00.000    3              Save-a-lot Markets               400.0000
10199       2002-03-27 00:00:00.000    39             Save-a-lot Markets               720.0000
10200       2002-03-30 00:00:00.000    11             Bólido Comidas preparadas        588.0000

(540 row(s) affected)
```

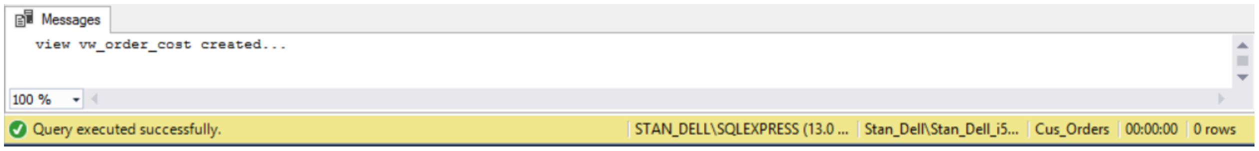
In this task, we are going to create a view first, then we will run this view to display the required information. Note that view is also a database object, so we need to check the existence of the old ones and drop if needed, before we can create the new ones. In terms of the view, it is like a space to store copies of columns from various tables as chosen, for latter manipulation and displaying of data, without touching the original data in each table. This adds some security to the data stored in the original tables in database.

To create the view, just use CREATE VIEW command, yet this statement can only be used in a single batch, so the last statement precedes this will need to be executed by a " GO ". For the creation of view, just SELECT the columns from each table, and do some INNER JOINs as needed, the other clauses, such as WHERE, can be include latter when we are running the view. As usual, we have included a PRINT statement at the end to indicate the creation of this view.

SQL statements used:

```
DROP VIEW IF EXISTS vw_order_cost;
GO
CREATE VIEW vw_order_cost
AS
SELECT  orders.order_id,
        orders.order_date,
        products.product_id,
        customers.name,
        'order_cost' = order_details.quantity * products.unit_price
FROM orders
INNER JOIN order_details ON order_details.order_id = orders.order_id
INNER JOIN products ON order_details.product_id = products.product_id
INNER JOIN customers ON orders.customer_id = customers.customer_id;
GO
print 'view vw_order_cost created...';
GO
```

Producing the following results:



Messages

```
view vw_order_cost created...
```

100 %

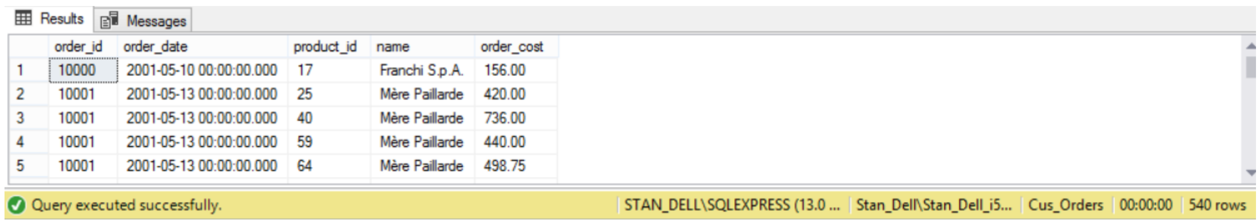
Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows

After the creation of the view, we can run it with the restriction to only display data in which the order ids are between 10000 and 10200.

SQL statements used:

```
SELECT *
FROM vw_order_cost
WHERE order_id >= 10000 AND order_id <= 10200;
GO
```

Producing the following results:

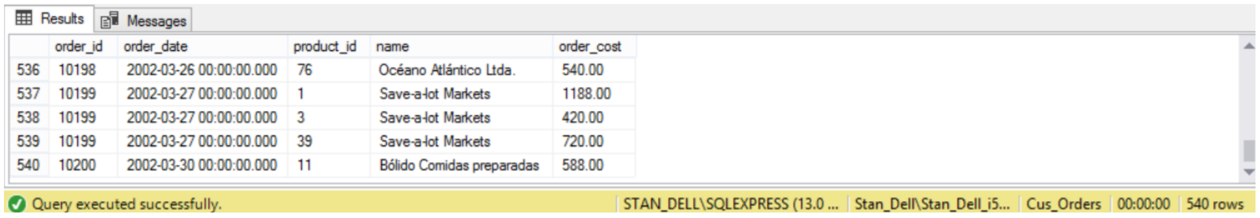


Results | Messages

	order_id	order_date	product_id	name	order_cost
1	10000	2001-05-10 00:00:00.000	17	Franchi S.p.A.	156.00
2	10001	2001-05-13 00:00:00.000	25	Mère Paillarde	420.00
3	10001	2001-05-13 00:00:00.000	40	Mère Paillarde	736.00
4	10001	2001-05-13 00:00:00.000	59	Mère Paillarde	440.00
5	10001	2001-05-13 00:00:00.000	64	Mère Paillarde	498.75

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 540 rows

…



Results | Messages

	order_id	order_date	product_id	name	order_cost
536	10198	2002-03-26 00:00:00.000	76	Océano Atlántico Ltda.	540.00
537	10199	2002-03-27 00:00:00.000	1	Save-a-lot Markets	1188.00
538	10199	2002-03-27 00:00:00.000	3	Save-a-lot Markets	420.00
539	10199	2002-03-27 00:00:00.000	39	Save-a-lot Markets	720.00
540	10200	2002-03-30 00:00:00.000	11	Bólido Comidas preparadas	588.00

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 540 rows

8. Create a view called **vw_list_employees** to list all the employees and all the columns in the employee table. Run the view for employee ids 5, 7, and 9. Display the employee id, last name, first name, and birth date. Format the name as last name followed by a comma and a space followed by the first name. Format the birth date as **YYYY.MM.DD**. The view should produce the result set listed below.

```
employee_id   name                              birth_date
-----------   -------------------------------   ------------
5             Buchanan, Steven                  1955.03.04
7             King, Robert                      1960.05.29
9             Dodsworth, Anne                   1966.01.27
```

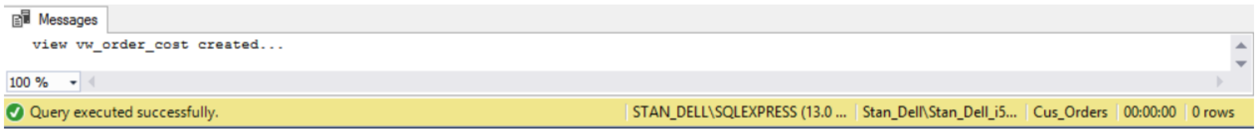
(3 row(s) affected)

This task involves the creation of a new view, which follows similar procedure as in last task, except for this one, there is no need to have INNER JOIN as the columns are from the same employee table, and we have included all the columns in the employee table, even though some of them are not needed for the next running step. We have also included a PRINT statement at the end to indicate the creation of new view.

SQL statements used:

```
DROP VIEW IF EXISTS vw_list_employees;
GO
CREATE VIEW vw_list_employees
AS
SELECT   employee.employee_id,
         employee.last_name,
         employee.first_name,
         employee.address,
         employee.city,
         employee.province,
         employee.postal_code,
         employee.phone,
         employee.birth_date
FROM employee;
GO
print 'view vw_order_cost created...';
GO
```

Producing the following results:



```
view vw_order_cost created...
```

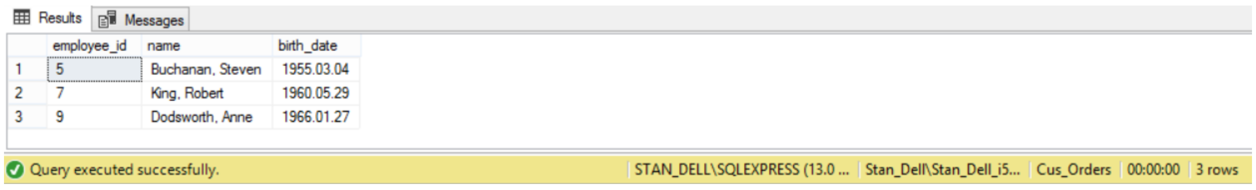
```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   0 rows
```

Now we can run the view with specifications provided by the task. Note that the formation of dates is done at the step of running the view.

SQL statements used:

```
SELECT employee_id,
       'name' = last_name +', '+ first_name,
       'birth_date' = CONVERT(CHAR(10),birth_date,102)
FROM vw_list_employees
WHERE employee_id in (5, 7, 9);
GO
```

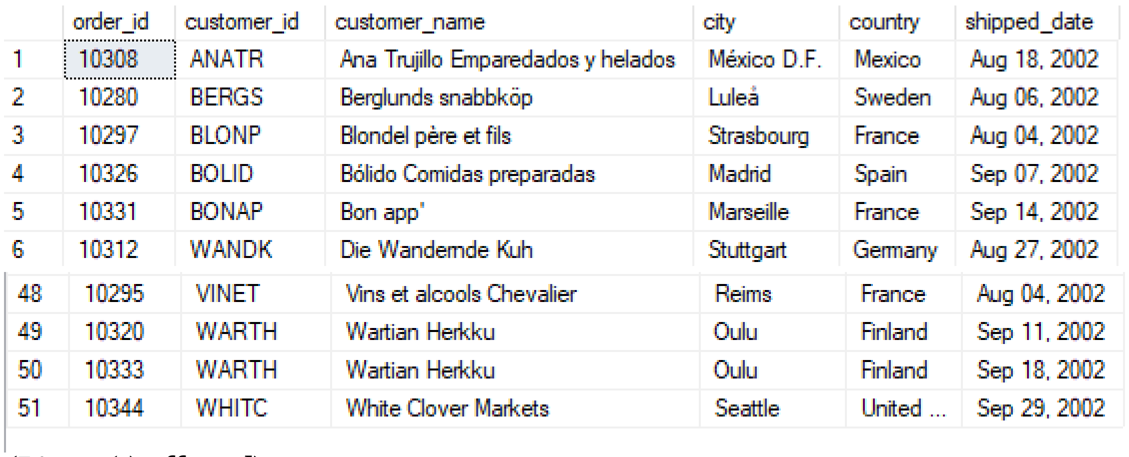
Producing the following results:



	employee_id	name	birth_date
1	5	Buchanan, Steven	1955.03.04
2	7	King, Robert	1960.05.29
3	9	Dodsworth, Anne	1966.01.27

```
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 3 rows
```

9. Create a view called **vw_all_orders** to list the columns shown below. Display the order id and shipped date from the orders table, and the customer id, name, city, and country from the customers table. Run the view for orders shipped from **August 1, 2002** and **September 30, 2002**, formatting the shipped date as shown. Order the result set by customer name and country. The view should produce the result set listed below.



	order_id	customer_id	customer_name	city	country	shipped_date
1	10308	ANATR	Ana Trujillo Emparedados y helados	México D.F.	Mexico	Aug 18, 2002
2	10280	BERGS	Berglunds snabbköp	Luleå	Sweden	Aug 06, 2002
3	10297	BLONP	Blondel père et fils	Strasbourg	France	Aug 04, 2002
4	10326	BOLID	Bólido Comidas preparadas	Madrid	Spain	Sep 07, 2002
5	10331	BONAP	Bon app'	Marseille	France	Sep 14, 2002
6	10312	WANDK	Die Wandernde Kuh	Stuttgart	Germany	Aug 27, 2002
48	10295	VINET	Vins et alcools Chevalier	Reims	France	Aug 04, 2002
49	10320	WARTH	Wartian Herkku	Oulu	Finland	Sep 11, 2002
50	10333	WARTH	Wartian Herkku	Oulu	Finland	Sep 18, 2002
51	10344	WHITC	White Clover Markets	Seattle	United ...	Sep 29, 2002

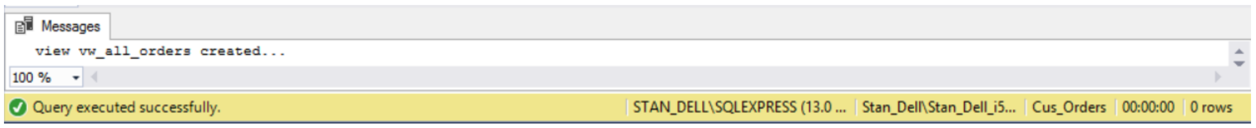
(51 row(s) affected)

In this task, we are going to create a view with columns from different tables, so that INNER JOIN would be needed. We also follow similar procedures as before for the creation of this new view, including the check of existence, and the printing of creation message.

SQL statements used:

```
DROP VIEW IF EXISTS vw_all_orders;
GO
CREATE VIEW vw_all_orders
AS
SELECT orders.order_id,
       orders.shipped_date,
       customers.customer_id,
       customers.name,
       customers.city,
       customers.country
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
GO
print 'view vw_all_orders created...';
GO
```

Producing the following results:



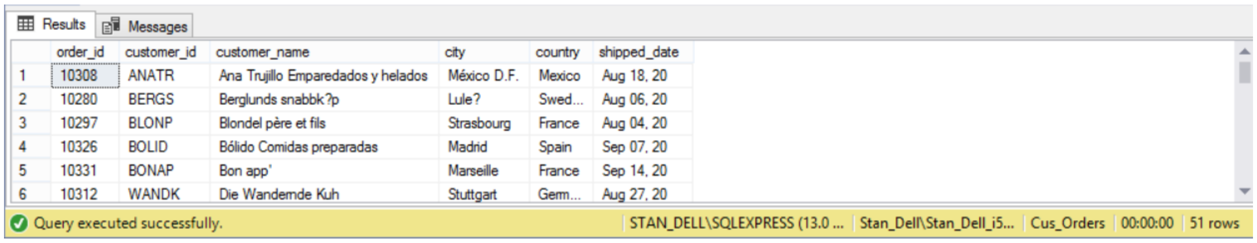
```
Messages
view vw_all_orders created...
100 %
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ...  Stan_Dell\Stan_Dell_i5...  Cus_Orders  00:00:00  0 rows
```

In the running view step, we need to specify to display only the orders shipped from August 1, 2002 to September 30, 2002, and into the desired format.

SQL statements used:

```
SELECT order_id,
       customer_id,
       'customer_name' = name,
       city,
       country,
       'shipped_date' =  CONVERT(CHAR(10),shipped_date,107)
FROM vw_all_orders
WHERE shipped_date >= 'Aug 1 2002' AND shipped_date <= 'Sep 30 2002'
ORDER BY name, country;
GO
```

Producing the following results:

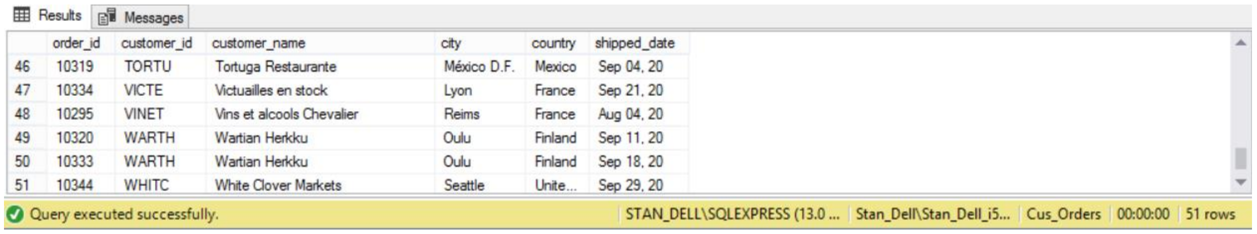


Results | Messages

	order_id	customer_id	customer_name	city	country	shipped_date
1	10308	ANATR	Ana Trujillo Emparedados y helados	México D.F.	Mexico	Aug 18, 20
2	10280	BERGS	Berglunds snabbk?p	Lule?	Swed...	Aug 06, 20
3	10297	BLONP	Blondel père et fils	Strasbourg	France	Aug 04, 20
4	10326	BOLID	Bólido Comidas preparadas	Madrid	Spain	Sep 07, 20
5	10331	BONAP	Bon app'	Marseille	France	Sep 14, 20
6	10312	WANDK	Die Wandernde Kuh	Stuttgart	Germ...	Aug 27, 20

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 51 rows

…



Results | Messages

	order_id	customer_id	customer_name	city	country	shipped_date
46	10319	TORTU	Tortuga Restaurante	México D.F.	Mexico	Sep 04, 20
47	10334	VICTE	Victuailles en stock	Lyon	France	Sep 21, 20
48	10295	VINET	Vins et alcools Chevalier	Reims	France	Aug 04, 20
49	10320	WARTH	Wartian Herkku	Oulu	Finland	Sep 11, 20
50	10333	WARTH	Wartian Herkku	Oulu	Finland	Sep 18, 20
51	10344	WHITC	White Clover Markets	Seattle	Unite...	Sep 29, 20

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 51 rows

10. Create a view listing the suppliers and the items they have shipped. Display the supplier id and name from the suppliers table, and the product id and name from the products table. Run the view. The view should produce the result set listed below, although not necessarily in the same order.

```
supplier_id    supplier_name                              product_id     product_name
---------------  ----------------------------------------   --------------   ------------------------------
9              Silver Spring Wholesale Market             23             Tunnbröd
11             Ovellette Manufacturer Company             46             Spegesild
15             Campbell Company                           69             Gudbrandsdalsost
12             South Harbour Products Ltd.                77             Original Frankfurter grüne Soße
14             St. Jean's Company                         31             Gorgonzola Telino
...
7              Steveston Export Company                   63             Vegie-spread
3              Macaulay Products Company                  8              Northwoods Cranberry Sauce
15             Campbell Company                           55             Pâté chinois
```

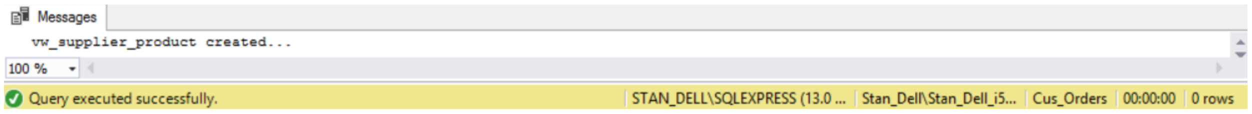
(77 row(s) affected)

For this task, let us call the view vw_supplier_product, and we will follow similar procedure of creating the view for the creation. Since columns from different tables involved, we need to have INNER JOIN for the view.

SQL statements used:

```
DROP VIEW IF EXISTS vw_supplier_product;
GO
CREATE VIEW vw_supplier_product
AS
SELECT   suppliers.supplier_id,
         'supplier_name' = suppliers.name,
         products.product_id,
         'product_name' = products.name
         FROM suppliers
INNER JOIN products ON products.supplier_id = suppliers.supplier_id;
GO
print 'vw_supplier_product created...';
GO
```

Producing the following results:



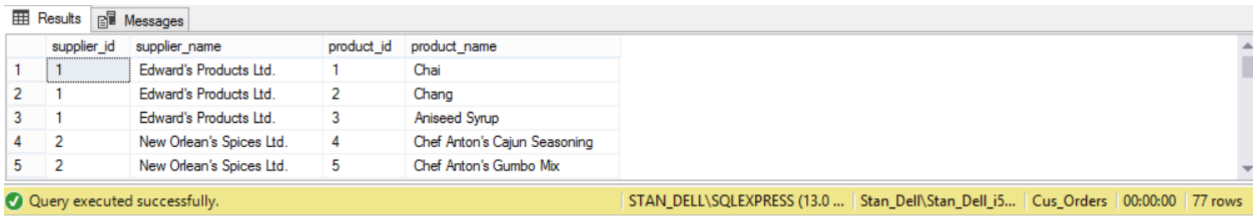
```
Messages
vw_supplier_product created...
100 %
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows
```

Next, we can run the view vw_supplier_product. Since the task states that the order of the results is not important, we can just order them by the supplier id to make the results seen more organisable. Also, there are special symbols that cannot be displayed in some of systems, so will result in displaying the question marks " ? " in place.

SQL statements used:

```
SELECT *
FROM vw_supplier_product
ORDER BY supplier_id;
GO
```

Producing the following results:

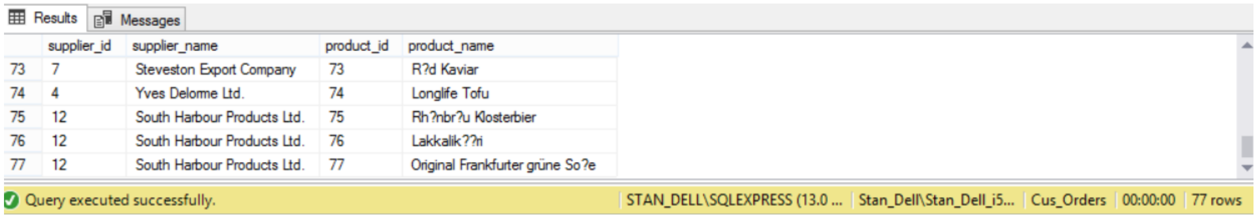


Results | Messages

	supplier_id	supplier_name	product_id	product_name
1	1	Edward's Products Ltd.	1	Chai
2	1	Edward's Products Ltd.	2	Chang
3	1	Edward's Products Ltd.	3	Aniseed Syrup
4	2	New Orlean's Spices Ltd.	4	Chef Anton's Cajun Seasoning
5	2	New Orlean's Spices Ltd.	5	Chef Anton's Gumbo Mix

Query executed successfully. STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 77 rows

...



Results | Messages

	supplier_id	supplier_name	product_id	product_name
73	7	Steveston Export Company	73	R?d Kaviar
74	4	Yves Delorme Ltd.	74	Longlife Tofu
75	12	South Harbour Products Ltd.	75	Rh?nbr?u Klosterbier
76	12	South Harbour Products Ltd.	76	Lakkalik??ri
77	12	South Harbour Products Ltd.	77	Original Frankfurter grüne So?e

Query executed successfully. STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 77 rows

Part D - Stored Procedures and Triggers

1. Create a stored procedure called **sp_customer_city** displaying the customers living in a particular city. The **city** will be an **input parameter** for the stored procedure. Display the customer id, name, address, city and phone from the customers table. Run the stored procedure displaying customers living in **London**. The stored procedure should produce the result set listed below.

```
customer_id    name                              address                               city     phone
---------------  ----------------------------------  -------------------------------------  ------   ------------
AROUT          Around the Horn                   120 Hanover Sq.                       London   (71) 555-7788
BSBEV          B's Beverages                     Fauntleroy Circus                     London   (71) 555-1212
CONSH          Consolidated Holdings             Berkeley Gardens 12  Brewery          London   (71) 555-2282
EASTC          Eastern Connection                35 King George                        London   (71) 555-0297
NORTS          North/South                       South House 300 Queensbridge          London   (71) 555-7733
SEVES          Seven Seas Imports                90 Wadhurst Rd.                       London   (71) 555-1717
```
(6 row(s) affected)

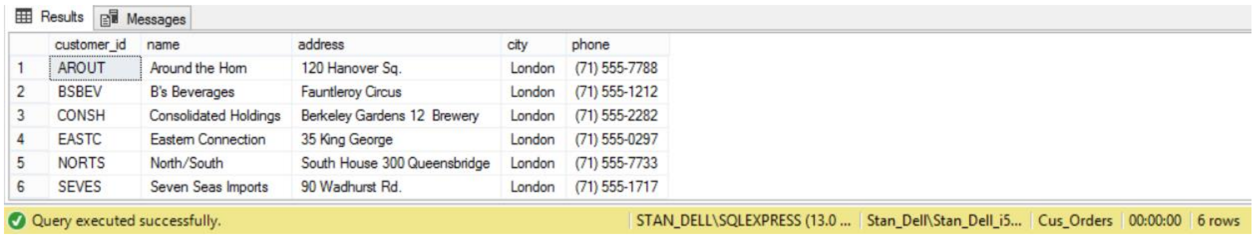
In this task, we will create a stored procedure (SP). Since stored procedures are database objects, we need to check the existence of old one, and drop if needed, prior to creating new ones. This task also asked for an input parameter, which will be created as a local variable, @city, for this SP. Since this variable will receive values later, it needs to be specified in () with appropriate data types at the beginning of SP creation. The input variable helps the SP to locate information that is related to what we interested in. In order to do this, we need a WHERE clause to match the rows with the input variable.

The SPs are similar to the views, in which we can select rows to display. Also, similar to the views that have running steps in order to display the data, the SPs have execution steps. To execute, simply type EXECUTE follow by the name of SP, and the desired city as input, after the SP have been created.

SQL statements used:

```
/* Check existence of sp_customer_city, drop if already exists */
DROP PROCEDURE IF EXISTS sp_customer_city;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_customer_city
(
            @city varchar(30)
)
AS
SELECT      customers.customer_id,
            customers.name,
            customers.address,
            customers.city,
            customers.phone
FROM customers
WHERE @city = customers.city;
GO
/* For execution */
EXECUTE sp_customer_city 'London';
GO
```

Producing the following results:



	customer_id	name	address	city	phone
1	AROUT	Around the Horn	120 Hanover Sq.	London	(71) 555-7788
2	BSBEV	B's Beverages	Fauntleroy Circus	London	(71) 555-1212
3	CONSH	Consolidated Holdings	Berkeley Gardens 12 Brewery	London	(71) 555-2282
4	EASTC	Eastern Connection	35 King George	London	(71) 555-0297
5	NORTS	North/South	South House 300 Queensbridge	London	(71) 555-7733
6	SEVES	Seven Seas Imports	90 Wadhurst Rd.	London	(71) 555-1717

```
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 6 rows
```

2. Create a stored procedure called **sp_orders_by_dates** displaying the orders shipped between particular dates. The **start** and **end** date will be **input parameters** for the stored procedure. Display the order id, customer id, and shipped date from the orders table, the customer name from the customer table, and the shipper name from the shippers table. Run the stored procedure displaying orders from **January 1, 2003** to **June 30, 2003**. The stored procedure should produce the result set listed below.

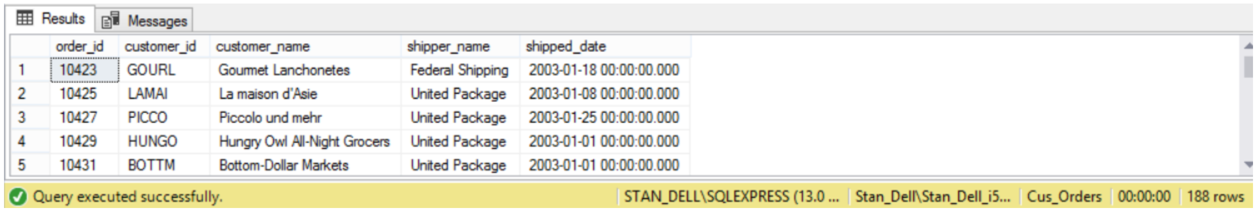
```
order_id    customer_id    customer_name                         shipper_name         shipped_date
----------  ---------------  -------------------------------------  --------------------  ----------------------
10423       GOURL          Gourmet Lanchonetes                   Federal Shipping     2003-01-18 00:00:00.000
10425       LAMAI          La maison d'Asie                      United Package       2003-01-08 00:00:00.000
10427       PICCO          Piccolo und mehr                      United Package       2003-01-25 00:00:00.000
10429       HUNGO          Hungry Owl All-Night Grocers          United Package       2003-01-01 00:00:00.000
10431       BOTTM          Bottom-Dollar Markets                 United Package       2003-01-01 00:00:00.000
...
10615       WILMK          Wilman Kala                           Federal Shipping     2003-06-30 00:00:00.000
10616       GREAL          Great Lakes Food Market               United Package       2003-06-29 00:00:00.000
10617       GREAL          Great Lakes Food Market               United Package       2003-06-28 00:00:00.000

(188 row(s) affected)
```

In this task, we create the SP in a way similar to the last task, yet this time, columns from different tables are involved, so we need to have INNER JOINs. Moreover, there are two input parameters used, so we need to create two local variables, @start_date and @end_date, inside this SP, using datetime as data type. They need to be specified inside () at the beginning of SP creation as they will receive values. A WHERE clause at the end of SP specifies only to display the shipped_date that is between the two local variables. The execution step is similar to the last one, yet two inputs are needed for this SP to be executed.

SQL statements used:

```
/* Check existence of sp_orders_by_dates, drop if already exists */
DROP PROCEDURE IF EXISTS sp_orders_by_dates;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_orders_by_dates
(
            @start_date datetime,
            @end_date datetime
)
AS
SELECT   orders.order_id,
         orders.customer_id,
         'customer_name' = customers.name,
         'shipper_name' = shippers.name,
         orders.shipped_date
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id
INNER JOIN shippers ON orders.shipper_id = shippers.shipper_id
WHERE @start_date <= orders.shipped_date AND @end_date >=
orders.shipped_date;
GO
/* For execution */
EXECUTE sp_orders_by_dates 'January 1 2003', 'June 30 2003';
GO
```

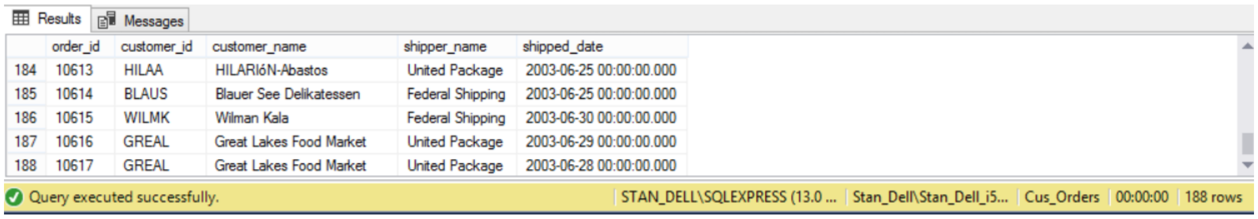
Producing the following results:



	order_id	customer_id	customer_name	shipper_name	shipped_date
1	10423	GOURL	Gourmet Lanchonetes	Federal Shipping	2003-01-18 00:00:00.000
2	10425	LAMAI	La maison d'Asie	United Package	2003-01-08 00:00:00.000
3	10427	PICCO	Piccolo und mehr	United Package	2003-01-25 00:00:00.000
4	10429	HUNGO	Hungry Owl All-Night Grocers	United Package	2003-01-01 00:00:00.000
5	10431	BOTTM	Bottom-Dollar Markets	United Package	2003-01-01 00:00:00.000

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 188 rows

…



	order_id	customer_id	customer_name	shipper_name	shipped_date
184	10613	HILAA	HILARIóN-Abastos	United Package	2003-06-25 00:00:00.000
185	10614	BLAUS	Blauer See Delikatessen	Federal Shipping	2003-06-25 00:00:00.000
186	10615	WILMK	Wilman Kala	Federal Shipping	2003-06-30 00:00:00.000
187	10616	GREAL	Great Lakes Food Market	United Package	2003-06-29 00:00:00.000
188	10617	GREAL	Great Lakes Food Market	United Package	2003-06-28 00:00:00.000

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 188 rows

3. Create a stored procedure called **sp_product_listing** listing a specified product ordered during a specified month and year. The **product** and the **month** and **year** will be **input parameters** for the stored procedure. Display the product name, unit price, and quantity in stock from the products table, and the supplier name from the suppliers table. Run the stored procedure displaying a product name containing **Jack** and the month of the order date is **June** and the year is **2001**. The stored procedure should produce the result set listed below.

```
product_name                            unit_price    quantity_in_stock  supplier_name
--------------------------------------------  -------------  ---------------------  ----------------------------------
Jack's New England Clam Chowder        10.1325        85                  Silver Spring Wholesale Market
Jack's New England Clam Chowder        10.1325       85                  Silver Spring Wholesale Market
Jack's New England Clam Chowder        10.1325       85                  Silver Spring Wholesale Market
Jack's New England Clam Chowder        10.1325       85                  Silver Spring Wholesale Market
```

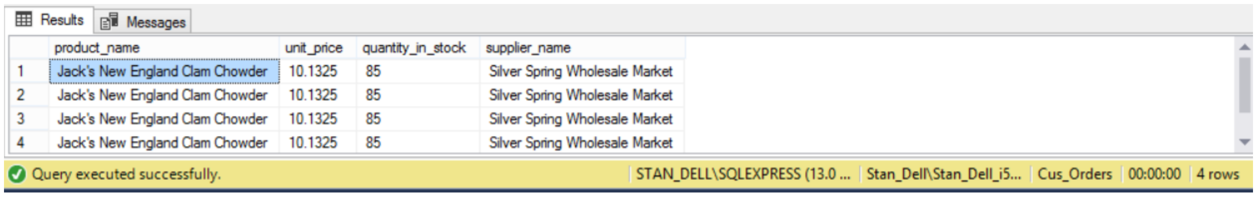
(4 row(s) affected)

In this task, we are will create the SP in a similar fashion as before, yet this time we have three input parameters, so we need to have three local variables, @prodt, @month, and @year, all with varchar(30) as data type. They need to be specified in () at the beginning of SP creation because they will be receiving values. We also designate them with default values of being anything ('%'), which means, when we are executing the SP without giving any inputs, the system will return every product for all months and years. Additionally, in the WHERE clause of the SP, we used LIKE instead of equal sign, " = " to give the flexibility of pattern matching. In order to find any products with names containing Jack, we need to enclose Jack with '%', that is " `%Jack%` ", so that the system will perform a pattern matching for any product names with that criterion.

SQL statements used:

```
/* Check existence of sp_product_listing, drop if already exists */
DROP PROCEDURE IF EXISTS sp_product_listing;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_product_listing
(
            @prodt varchar(30) = '%',
            @month varchar(30) = '%',
            @year varchar(30) = '%'
)
AS
SELECT  'product_name' = products.name,
        products.unit_price,
        products.quantity_in_stock,
        'supplier_name'= suppliers.name
FROM products
INNER JOIN suppliers ON products.supplier_id = suppliers.supplier_id
INNER JOIN order_details ON order_details.product_id = products.product_id
INNER JOIN orders ON order_details.order_id = orders.order_id
WHERE products.name LIKE @prodt
      AND DATENAME(MONTH, orders.order_date)= @month
      AND DATENAME(YEAR, orders.order_date)= @year;
GO
/* For execution */
EXECUTE sp_product_listing '%Jack%', 'June', '2001';
GO
```

Producing the following results:

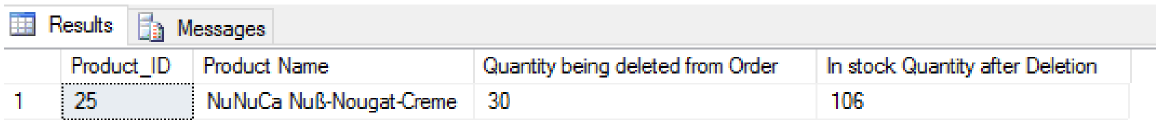


	product_name	unit_price	quantity_in_stock	supplier_name
1	Jack's New England Clam Chowder	10.1325	85	Silver Spring Wholesale Market
2	Jack's New England Clam Chowder	10.1325	85	Silver Spring Wholesale Market
3	Jack's New England Clam Chowder	10.1325	85	Silver Spring Wholesale Market
4	Jack's New England Clam Chowder	10.1325	85	Silver Spring Wholesale Market

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   4 rows
```

4. Create a DELETE trigger on the order_details table to display the information shown below when you issue the following statement:

```
DELETE order_details
WHERE order_id=10001 AND product_id=25
  You should get the following results:
```



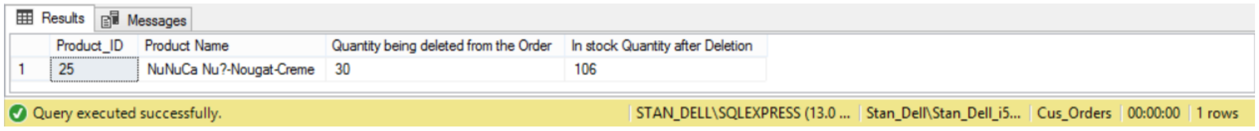
	Product_ID	Product Name	Quantity being deleted from Order	In stock Quantity after Deletion
1	25	NuNuCa Nuß-Nougat-Creme	30	106

This task asked for the creation of a DELETE trigger, let us call it **Deletion_order**. Since the triggers are also database objects, we need to check the existence of old ones before we create it, similar to previous tasks. This trigger will fire when there is any commands that attempt to delete records in order_details table, the quantities being deleted from the order_details table will be automatically added back into the quantity in stock of the products table. This trigger also needs to be able to display the above changes in quantities, as well as to display the Product ID and names of the product that is involved in the deletion.

SQL statements used:

```
/* Check trigger existence, drop if already exists */
DROP TRIGGER IF EXISTS [dbo].[Deletion_order];
GO
/* Create the trigger */
CREATE TRIGGER Deletion_order
ON order_details
FOR DELETE
AS
DECLARE @orderid varchar(30), @prodid varchar(30), @qty int
SELECT @orderid = order_id, @prodid = product_id, @qty = quantity
FROM deleted
UPDATE products
SET products.quantity_in_stock = quantity_in_stock + @qty
WHERE products.product_id = @prodid
SELECT 'Product_ID' = products.product_id,
       'Product Name' = products.name,
       'Quantity being deleted from the Order' = @qty,
       'In stock Quantity after Deletion' = products.quantity_in_stock
FROM products
WHERE product_id = @prodid;
GO

/* For issuing the Delete command */
DELETE order_details
WHERE order_id=10001 AND product_id=25;
GO
```

Producing the following results:



	Product_ID	Product Name	Quantity being deleted from the Order	In stock Quantity after Deletion
1	25	NuNuCa Nu?-Nougat-Creme	30	106

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 1 rows

The ideas behind this trigger is that, the three local variables for this trigger, which are @orderid, @prodid, and @qty, are used to extract the information contained in the deleted table, which is a special virtual table that contains the deleted rows when a DELETE statement is executed. These three variables will take the values of the order id, product id, and quantities in an order, when a row in order_details is being deleted. The product id will later be displayed as one of the results, and will also be used to identify the associated product name, while the quantity of that order is going to be added back into the quantity_in_stock column of that product id in products table.

Also note that, triggers are similar to views and stored procedures. After their creation, in order to show their effects, we need to issue corresponding commands that can fire the trigger. While for views, we can run them, and for SPs, we can execute them.

5. Create an **INSERT** and **UPDATE** trigger called **tr_check_qty** on the order_details table to only allow orders of products that have a quantity in stock greater than or equal to the units ordered. Run the following query to verify your trigger.

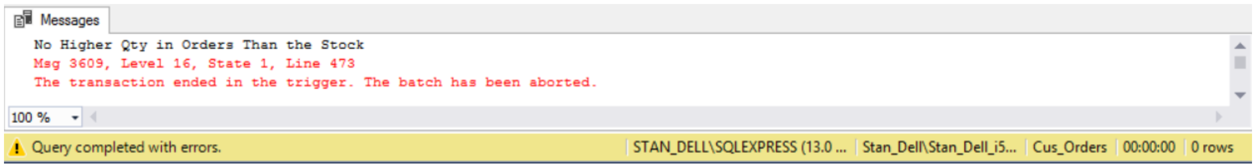
```
UPDATE order_details
SET quantity = 30
WHERE order_id = '10044'
AND product_id = 7
```

We are going to create an INSERT and UPDATE trigger, tr_check_qty, to act as a check to only allow values that satisfy the requirement to be inserted or updated on the order_details table. We will create two local variables, for this trigger, @prodid and @qty, which will be used to extract product id and quantity from the inserted table, which is a virtual table that contains the new or updated rows when an INSERT or UPDATE statement is executed. Copies of the new row stays in the inserted table until the trigger decides how to implement the new data. In this task, we will implement a ROLLBACK TRANSACTION along with a printing message if the quantities of a certain product in an order is greater than its quantity in stock.

SQL statements used:

```
/* Check trigger existence, drop if already exists */
DROP TRIGGER IF EXISTS [dbo].[tr_check_qty];
GO
/* Create the trigger */
CREATE TRIGGER tr_check_qty
ON order_details
FOR INSERT, UPDATE
AS
DECLARE @prodid varchar(30), @qty int
SELECT @prodid = product_id,
       @qty = quantity
FROM inserted
IF @qty > (SELECT products.quantity_in_stock FROM products WHERE @prodid =
products.product_id)
      BEGIN
            PRINT 'No Higher Qty in Orders Than the Stock'
            ROLLBACK TRANSACTION
      END;
GO

/* Test this trigger by the following */
UPDATE order_details
SET quantity = 30
WHERE order_id = '10044' AND product_id = 7;
```

Producing the following results:



```
No Higher Qty in Orders Than the Stock
Msg 3609, Level 16, State 1, Line 473
The transaction ended in the trigger. The batch has been aborted.
```

```
Query completed with errors.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 0 rows
```

From the above output message, we can see that the trigger works when we attempt to update an order for a certain product with a quantity that is greater than the quantity in stock. An error message was displayed and the update attempt was stopped.

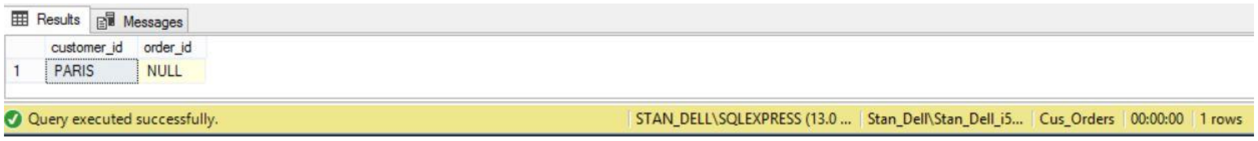
6. Create a stored procedure called **sp_del_inactive_cust** to **delete** customers that have no orders. The stored procedure should delete 1 row.

For this task, we can start with an OUTER JOIN to check if there is any customer that have no orders, or has NULL order id, when the table customers and orders are joint.

SQL statements used:

```
SELECT customers.customer_id, orders.order_id
FROM customers
LEFT OUTER JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.order_id IS NULL;
```

Producing the following results:



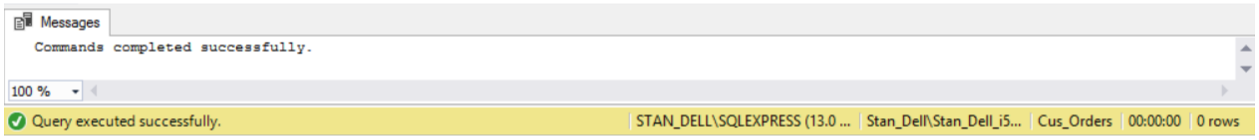
```
Results | Messages
   customer_id   order_id
1  PARIS         NULL
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 1 rows
```

From the above results, we can see there is only one customer that has no orders. Now we can start creating the SP. As usual, we will have an existence check before the creation. Note that, for this SP, we do not have any input parameters, but we still need to have a location variable for local processing in this SP. This variable is @cust_id, and it needs to be declared at the beginning when we create the SP. Its purpose is to obtain the customer id output from the OUTER JOIN of customers and orders, and pass it onto the DELETE command.

SQL statements used:

```
/* Check existence of sp_del_inactive_cust, drop if already exists*/
DROP PROCEDURE IF EXISTS sp_del_inactive_cust;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_del_inactive_cust
AS
DECLARE @cust_id varchar(30)
SELECT @cust_id = customers.customer_id
FROM customers
LEFT OUTER JOIN orders ON customers.customer_id = orders.customer_id
WHERE order_id IS NULL
print @cust_id + ' is being deleted...'
DELETE FROM customers
WHERE customer_id = @cust_id;
GO
```

Producing the following results:



```
Commands completed successfully.
```

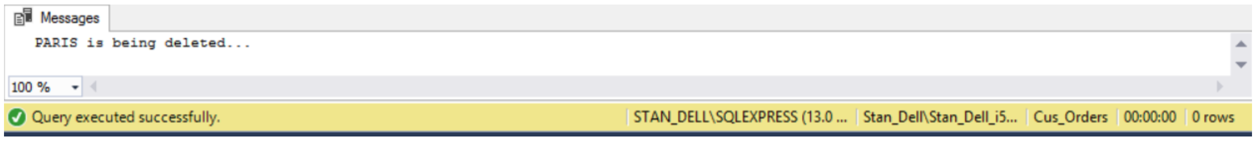
Messages
100 %
Query executed successfully.
STAN_DELL\SQLEXPRESS (13.0 ...
Stan_Dell\Stan_Dell_i5...
Cus_Orders
00:00:00
0 rows

Now the SP have been created, and can use it by execution.

SQL statements used:

```
EXECUTE sp_del_inactive_cust;
GO
```

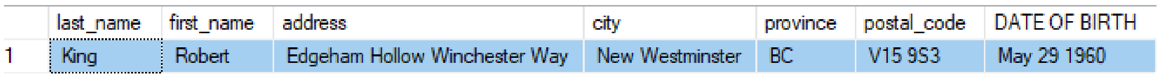
Producing the following results:



```
PARIS is being deleted...
```

Messages
100 %
Query executed successfully.
STAN_DELL\SQLEXPRESS (13.0 ...
Stan_Dell\Stan_Dell_i5...
Cus_Orders
00:00:00
0 rows

7. Create a stored procedure called **sp_employee_information** to display the employee information for a particular employee. The **employee id** will be an **input parameter** for the stored procedure. Run the stored procedure displaying information for employee id of **7**. The stored procedure should produce the result set listed below.



	last_name	first_name	address	city	province	postal_code	DATE OF BIRTH
1	King	Robert	Edgeham Hollow Winchester Way	New Westminster	BC	V15 9S3	May 29 1960

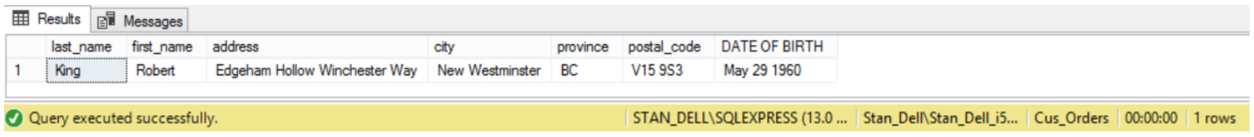
(1 row(s) affected)

In this task, we are going to create sp_employee_information as usual, start with an existence check, and then we will specify the @emp_id as a local variable inside () at the beginning, which will receive the values of employee id as input. The format of date of birth can be controlled by the CONVERT command as before. All columns are from the same table, so there will not have any JOINs. After creation, we can execute this SP with the required employee id.

SQL statements used:

```
/* Check existence of sp_employee_information, drop if already exists*/
DROP PROCEDURE IF EXISTS sp_employee_information;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_employee_information
(
            @emp_id int
)
AS
SELECT last_name,
       first_name,
       address,
       city,
       province,
       postal_code,
       'DATE OF BIRTH' = CONVERT(CHAR(11),birth_date,109)
FROM employee
WHERE @emp_id = employee_id;
GO
/* For execution */
EXECUTE sp_employee_information '7';
GO
```

Producing the following results:



	last_name	first_name	address	city	province	postal_code	DATE OF BIRTH
1	King	Robert	Edgeham Hollow Winchester Way	New Westminster	BC	V15 9S3	May 29 1960

```
Query executed successfully.   STAN_DELL\SQLEXPRESS (13.0 ...   Stan_Dell\Stan_Dell_i5...   Cus_Orders   00:00:00   1 rows
```

8. Create a stored procedure called **sp_reorder_qty** to show when the reorder level subtracted from the quantity in stock is less than a specified value. The **unit** value will be **an input parameter** for the stored procedure. Display the product id, quantity in stock, and reorder level from the products table, and the supplier name, address, city, and province from the suppliers table. Run the stored procedure displaying the information for a value of **5**. The stored procedure should produce the result set listed below.

```
product_id     name                              address                  city               province  qty  reorder_level
-------------- --------------------------------- ------------------------ ------------------ --------  ---  -------------
2              Edward's Products Ltd.            1125 Howe Street         Vancouver          BC        17   25
3              Edward's Products Ltd.            1125 Howe Street         Vancouver          BC        13   25
5              New Orlean's Spices Ltd.          1040 Georgia Street      West Vancouver     BC        0    0
11             Armstrong Company                 1638 Derwent Way         Richmond           BC        22   30
17             Steveston Export Company          2951 Moncton Street      Richmond           BC        0    0
...
68             Dare Manufacturer Ltd.            1603 3rd Avenue          West  Burnaby      BC        6    15
70             Steveston Export Company          2951 Moncton Street      Richmond           BC        15   30
74             Yves Delorme Ltd.                 3050 Granville Street    New Westminster    BC        4    5
```

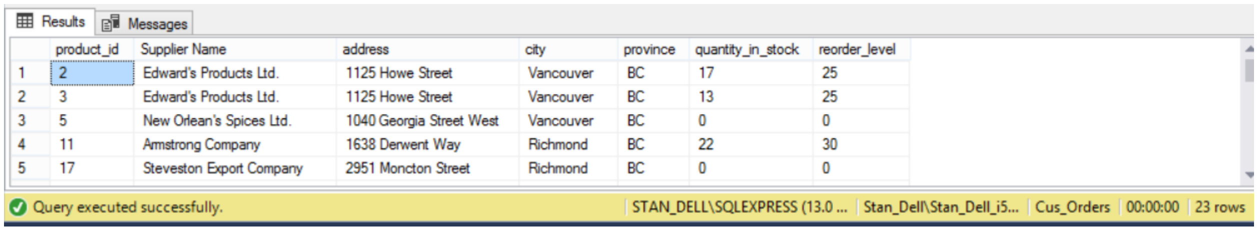
(23 row(s) affected)

In this task, we are going to create sp_reorder_qty as usual, start with an existence check, and then we will specify the @unit_val as a local variable inside () at the beginning, which will receive the value of units as input. The main body of this SP consists of selecting multiple columns from different tables, the products and the suppliers, so an INNER JOIN is needed. At the end of this SP, a WHERE clause will use the input, @unit_val, for the comparison of this input value, and the reorder level subtracted from the quantity in stock, which then determines what to display. After we have created the SP, we can execute it with the desired input values.

SQL statements used:

```
/* Check existence of sp_reorder_qty, drop if already exists*/
DROP PROCEDURE IF EXISTS sp_reorder_qty;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_reorder_qty
(
            @unit_val varchar(30)
)
AS
SELECT products.product_id,
       'Supplier Name' = suppliers.name,
       suppliers.address,
       suppliers.city,
       suppliers.province,
       products.quantity_in_stock,
       products.reorder_level
FROM products
INNER JOIN suppliers ON products.supplier_id = suppliers.supplier_id
WHERE @unit_val > products.quantity_in_stock - products.reorder_level;
GO

/* For execution */
EXECUTE sp_reorder_qty '5';
GO
```

Producing the following results:

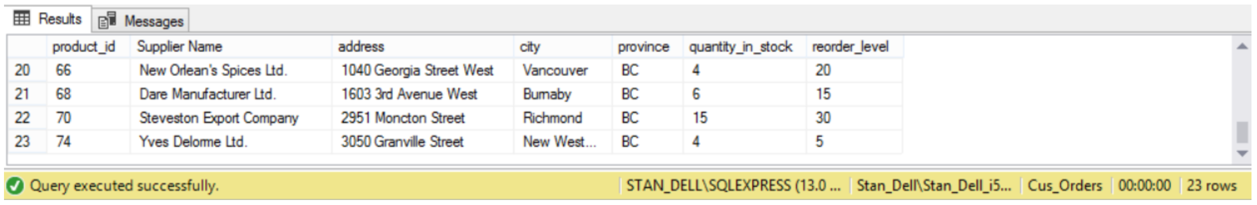


Results | Messages

	product_id	Supplier Name	address	city	province	quantity_in_stock	reorder_level
1	2	Edward's Products Ltd.	1125 Howe Street	Vancouver	BC	17	25
2	3	Edward's Products Ltd.	1125 Howe Street	Vancouver	BC	13	25
3	5	New Orlean's Spices Ltd.	1040 Georgia Street West	Vancouver	BC	0	0
4	11	Amstrong Company	1638 Derwent Way	Richmond	BC	22	30
5	17	Steveston Export Company	2951 Moncton Street	Richmond	BC	0	0

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 23 rows

…



Results | Messages

	product_id	Supplier Name	address	city	province	quantity_in_stock	reorder_level
20	66	New Orlean's Spices Ltd.	1040 Georgia Street West	Vancouver	BC	4	20
21	68	Dare Manufacturer Ltd.	1603 3rd Avenue West	Burnaby	BC	6	15
22	70	Steveston Export Company	2951 Moncton Street	Richmond	BC	15	30
23	74	Yves Delorme Ltd.	3050 Granville Street	New West...	BC	4	5

Query executed successfully. | STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 23 rows

9. Create a stored procedure called **sp_unit_prices** for the product table where the **unit price** is **between particular values**. The **two unit prices** will be **input parameters** for the stored procedure. Display the product id, product name, alternate name, and unit price from the products table. Run the stored procedure to display products where the unit price is between **$5.00** and **$10.00**. The stored procedure should produce the result set listed below.

```
product_id    name                             alternate_name                   unit_price
-------------- --------------------------------- -------------------------------- --------------
13            Konbu                            Kelp Seaweed                     6.30
19            Teatime Chocolate Biscuits       Teatime Chocolate Biscuits       9.66
23            Tunnbr÷d                         Thin Bread                       9.45
45            R°gede sild                      Smoked Herring                   9.975
47            Zaanse koeken                    Zaanse Cookies                   9.975
52            Filo Mix                         Mix for Greek Filo Dough         7.35
54            TourtiÞre                        Pork Pie                         7.8225
75            Rh÷nbrõu Klosterbier             Rh÷nbrõu Beer                    8.1375
```

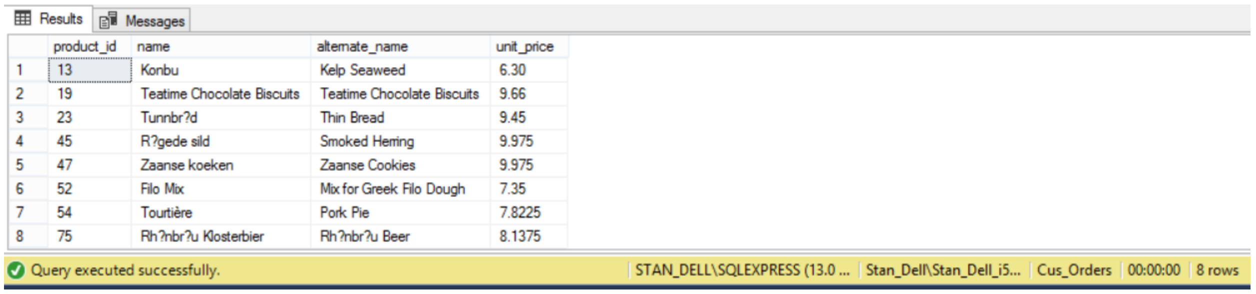
(8 row(s) affected)

In this task, we are going to create sp_unit_prices as usual, start with an existence check. Next we will specify the @unit_pr1 and @unit_pr2 as the local variable with money data type inside () at the beginning, which will receive the value of two unit prices as input. The two unit prices will establish a range of prices for the data to be displayed. Since all the columns to be displayed are from the same products table, there will not have any JOINs. In the execution step, we just need to provide the two end points of the range of interest as the input.

SQL statements used:

```
/* Check existence of sp_unit_prices, drop if already exists*/
DROP PROCEDURE IF EXISTS sp_unit_prices;
GO
/*Now build the procedures */
CREATE PROCEDURE sp_unit_prices
(
            @unit_pr1 money,
            @unit_pr2 money
)
AS
SELECT product_id,
       name,
       alternate_name,
       unit_price
FROM products
WHERE unit_price >= @unit_pr1 AND unit_price <= @unit_pr2;
GO

/* For execution */
EXECUTE sp_unit_prices 5, 10;
GO
```

Producing the following results:



	product_id	name	alternate_name	unit_price
1	13	Konbu	Kelp Seaweed	6.30
2	19	Teatime Chocolate Biscuits	Teatime Chocolate Biscuits	9.66
3	23	Tunnbr?d	Thin Bread	9.45
4	45	R?gede sild	Smoked Herring	9.975
5	47	Zaanse koeken	Zaanse Cookies	9.975
6	52	Filo Mix	Mix for Greek Filo Dough	7.35
7	54	Tourtière	Pork Pie	7.8225
8	75	Rh?nbr?u Klosterbier	Rh?nbr?u Beer	8.1375

```
Query executed successfully.    STAN_DELL\SQLEXPRESS (13.0 ... | Stan_Dell\Stan_Dell_i5... | Cus_Orders | 00:00:00 | 8 rows
```

Conclusion

In this report, we have covered a lot of features of SQL, such as commands, keywords, basic operations and basic functions. In addition, we have done the creation of database and tables, selections, joins, views, stored procedures, triggers and so on, all presented using the Microsoft SQL Server Management Studio (SSMS). For more information or details on this subject, please consult the references below, or refer to other textbooks, or search on the Internet.

Reference

(1) C. Coronel, S. Morris, Database Systems: Design, Implementation, & Management, 12th edn; Cengage Learning, Boston, M.A. United States, **2016**.

(2) Microsoft Transact SQL, https://docs.microsoft.com/en-us/sql/t-sql/language-elements/raiserror-transact-sql, last accessed Nov 28, **2017**.