# Introduction

A relational database is to be designed and implemented in SQL using the Oracle SQL Developer DBMS that would cater for the information needs of the online business – 10 Green Bottles. The database aims to store information on customers, their orders, the items that are currently on their shopping list/basket, as well as store information on internal orders, warehouses, distribution centres, wines and offers. In order to design this database, the following steps will be undertaken:

* Identify the entities that are to be modelled
* Assign attributes to the entities
* Establish the relationship between entities
* Solve many-to-many relationships by introducing new entities
* Assign primary keys, foreign keys and constraints

# Assumptions

Since the specification did not mention whether cases were sold at different costs, it has been assumed that the cost of cases could be obtained by multiplying the cost per bottle by the number of bottles per case (i.e. cases are not multi-pack).

It has been assumed that each wine is delivered to all centres on specific days of the week and that the distribution centres are to decide in the frequency of deliveries (weekly/monthly). A special Order is a one-off delivery; it has one date of delivery, which is to be specified by the user of the application. The database is to ensure that the delivery date entered is of a future date. A SQL view or a back-end programming language of an application could be developed to calculate the next date of delivery (Figure 5).

Offers only apply to bottles of wine, not cases; each wine can have many offers (over time) and an offer can apply to many wines. Since each attribute in a table can only store a single value, defining offerID as a foreign key in the wine table would be inappropriate as a wine could have many offers. Similarly, defining wineID in the Offer table would be inappropriate as an offer could apply to many wines. This many-to-many relationship has been resolved by the WineOffer table, where the wineID and offerID are foreign keys. Since the period in which the offer is valid depends on both, the wine and the offer, the start date and end date are both attributes of the WineOffer table. All offers must be recorded even if it no longer applies as this will be an important source of information, in case a customer would like a refund or makes a complaint on issues such as being overcharged.

# Design Decisions

The wine table will store all products - including the ones that have not yet been introduced onto the market. The attributes (colour, country, sparkling?) will allow application programmers to develop a search feature enabling users to find wines by specifying their preferences. Similarly, a date field has been added allowing users to view wines that will be arriving onto the market shortly. In regards to error prevention, several check constraints have been put in place e.g. the sweet rating of a wine must be between 1 and 6 (inclusive), the colour of the wine must be red, white or rose etc.

It is essential that the database stores information on the customers. Each customer will have a password to their online account; the length of this field has been set to thirty-two characters since the assumption has been made that the md5 () hashing function will be used to encrypt passwords, which will return a string of thirty-two characters. As well as the name of the customers, their address details will be stored for usability purposes. For example, software developers may decide to provide users with an option: to enter an address or use their home address, on the click of a button. Storing the home address of customers will prevent errors as the system is less dependent on user input, thus making the system easier to use.

Since the Offer table has *description* and *discount* as its attributes, it is possible to use SQL or an application program to calculate the new cost of items, after a specified discount e.g. 20%. However, it will not allow the business to have offers such as “Buy 6 for 5,” where the offer only applies for every six bottles that the customer buys. Although, it is possible to store the description of the offer, no mathematical functions can be applied to determine the new cost. Therefore, it has been decided that two more attributes are to be added – one storing the minimum quantity for which the offer applies and another storing the number of bottles that are to be paid for (Figure 6).

The initial design modelled *Order* as having one item; this has been corrected in the final version, as an order could have many items; an item could be in many orders. This many-to-many relationship has been resolved by the introduction of a new table – OrderWine, where the orderID and wineID will be foreign keys. Since quantity is dependant on both - the order and the item, this has been defined as an attribute of the OrderWine table, along with delivery charge and subtotal, which would be calculated by an application program.

If a user is to remove an item from the shopping list/basket, it has been decided that the record is to be deleted, since it is not useful to store information on every item that a customer places into a basket/list, thus saving disk space. It is a business rule that items in a shopping list are to be removed after sixty days, unless it has been manually removed by the customer themselves. Therefore, it is necessary to record the date on which the item was placed onto the shopping list. The initial idea was to use a SQL query that displays items on the shopping list that have been on the list for up to 60 days (Figure 4). However, storing every item placed onto the shopping list is unnecessary. A scheduled program such as a chron job or scheduler (provided by the DBMS) can be used to delete all shopping list items that have been on the list for more than sixty days; these programs should be executed on a daily basis. To prevent errors and to minimise the workload for customers, the database is to record the date without user input by obtaining this information from SYSDATE.

The shopping basket is not to be modelled as a customer will only have one basket, which will be emptied out by removing items manually or automatically at purchase. Since a basket could have many items, it would be unnecessary to model this as an entity, as attributes cannot hold collections of values. This would leave the customerID as the only attribute of the table, unless an object-oriented database was to be designed, where attributes can store collections of values of the same data type using the SET keyword. Modelling the basket will complicate the relational database in that a more complex *join* query will be necessary to find the items in the basket of a particular customer. The customerID has been defined as a foreign key and stored for each record in the BasketItem table.

Each record in any table is to be identified uniquely by a primary key. The primary key is to auto-increment in most tables, as prompting the user for a unique ID for an Order, for example, would be inappropriate. The primary key will not auto-increment in the Card table, as the card number will be provided by the customer and will be used as the primary key. Having done some research, it has been decided that sequences will be used to develop a program that will handle the auto-increment functionality, and triggers will be used to call these functions, on inserting a record.

The design proposed shows inheritance (or generalisation); the Standing Order and SpecialOrder tables extend the InternalOrder table, i.e. they are types of internal orders. This will be modelled by having an attribute (“orderType”), in the InternalOrder table, which will specify what type of order it is. The child tables will have an InternalOrderID as foreign keys, pointing to the associated record in parent table.

# Constraints

Entity Integrity has been assured by defining primary keys in all tables; this rejects any entries where the primary key is already in use. For example, in the Card table, a card number can only be added onto the table once, since it is unique, thus preventing duplicate data. Similarly, referential integrity has been assured by defining foreign keys, where appropriate. For example, a customer can purchase a wine. Due to the foreign keys, the database ensures that the wineID and customerID are primary keys of records in the referenced table, i.e. it ensures that a purchase is made by an existing customer and that the wine purchased actually exists. Check constraints have been put in place to prevent human error.

# Conclusion

The database could have been improved if more data was stored. Since the database will serve an online business, storing the customers’ email addresses could have allowed the business to keep their regular customers up-to-date on offers etc. Despite the email address being unique, it is not a suitable primary key, as it is possible that users may wish to update their email address. This would be a problem as it may require updating all attributes where the email address appears as a foreign key. However, it could have been a potential username for customers when logging onto the site; in terms of usability, the email address would be more memorable for the user, rather than having to remember their customer ID, which would be an auto-generated number. Furthermore, the email address could potentially be used to implement features such as account recovery, allowing customers to generate new passwords at any time of the day.

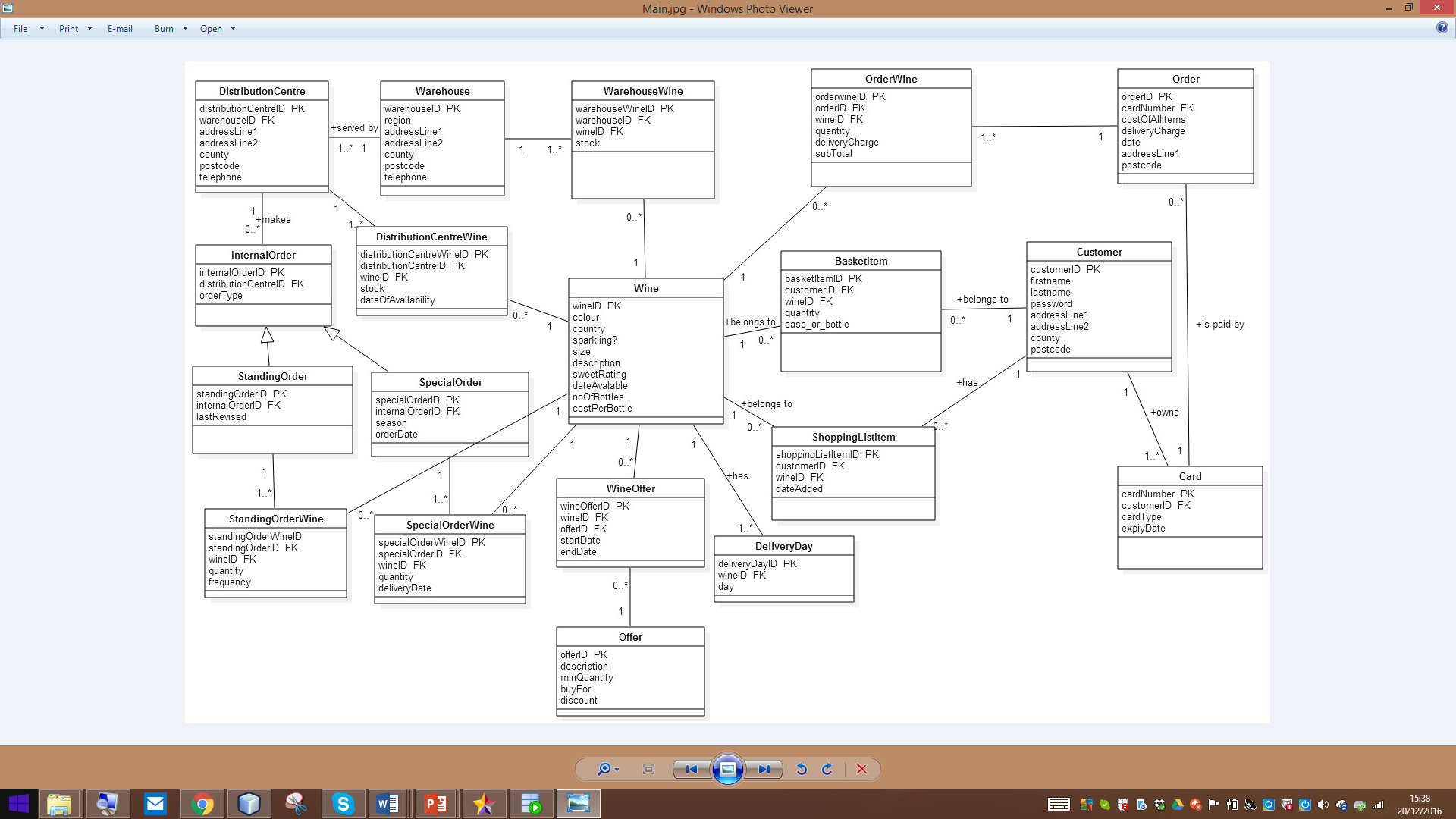
Another issue that was identified was that the md5 () hashing function is vulnerable to attacks. Therefore, it is rarely used by programmers. Instead, more secure functions such as the SHA-256 is used, which the database will not support as the password field holds up to thirty-two characters, whilst the SHA-256 function returns a string of sixty-four characters.

When making an order, a trigger could have been applied to records that are to be inserted into the Order table. The trigger could have included some validation, by comparing the current date with the expiry date of the card; any entries made to the Order table should only proceed if the expiry date is greater than the current date. Other validation that have been missed include the cost of all items, which must be greater than 0. Another issue regarding the Order table, is that it fails to record whether the sum of money has been deducted from the customers’ bank accounts.

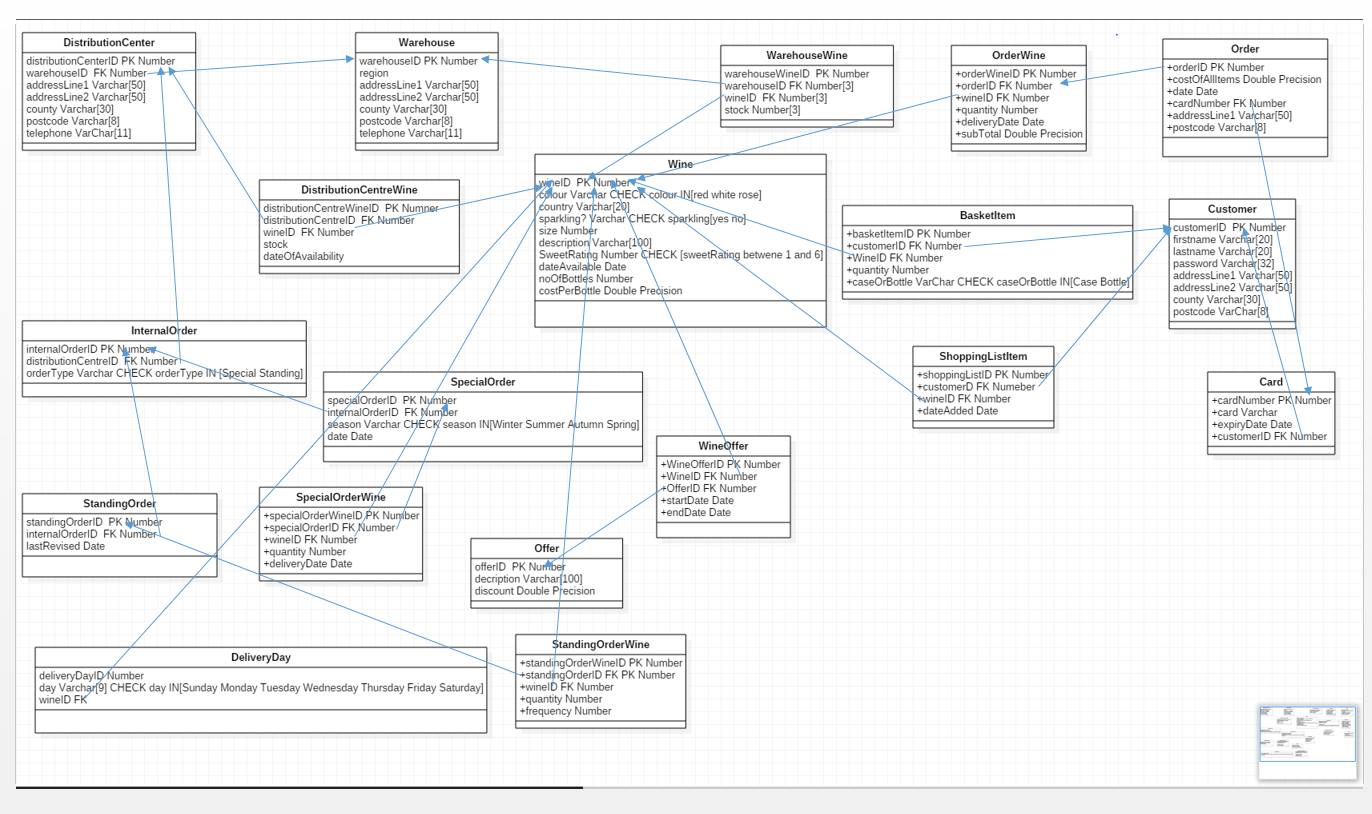
After implementation, it was identified that no constraints were put in place to manage null values. For example, when a customer registers an account, but fails to provide their details, the database will fail to recognise this. An alternative solution is to have front-end validation, e.g. use the *required* attribute for all text field in HTML5 and JavaScript/AJAX to ensure that the user has not entered a sequence of blank spaces. However, the validation will not be applied to records that are inserted directly via SQL. By using VARCHAR2, instead of VARCHAR, empty strings will be treated equally to null values and will be rejected, provided that attributes are set to NOT NULL. There are many attributes that must not accept null values including address line1 and postcode.

Finally, it was found that the check constraints on attributes are case-sensitive, and will therefore fail to tolerate human errors. There are many ways to prevent this. For example, the lower () function, that is provided by SQL, converts all characters to lower-case, before comparing it to the given string. An alternative solution is to introduce new tables into the database design. For example, a table that records the type of orders. Each type would then be referenced by a primary key, which would appear as a foreign key on the InternalOrder table. This would form a one-to-many relationship between the InternalOrder class and the InternalOrderType class. However, this could complicate the design and lead to many complex join queries, which could possibly affect the performance.

# Conceptual Model



# Relational Model



Implementation: Create Statements

Create Statements

CREATE TABLE WAREHOUSE

(

warehouseID NUMBER PRIMARY KEY,

region VARCHAR (20),

addressLine1 VARCHAR(50),

addressLine2 VARCHAR(50),

county VARCHAR(30),

postcode VARCHAR(8),

telephone Varchar(11)

)

CREATE TABLE WINE

(

wineID NUMBR PRIMARY KEY,

colour VARCHAR

CHECK colour IN(“Red”, “White”, “Rose”),

country VARCHAR(20),

sparkling VARCHAR

CHECK sparkling IN(“Yes”,”No”),

sizeOfBottle NUMBER,

description VARCHAR(100),

sweetRating NUMBE

CHECK sweetRating (BETWEEN 1 AND 6),

dateAvailable date,

numberOfBottles NUMBER,

costPerBottle double precision

)

CREATE TABLE CUSTOMER

(

customerID NUMBER PRIMARY KEY,

firstname VARCHAR(20),

lastname VARCHAR(20),

password VARCHAR(32),

addressLine1 VARCHAR(50),

addressLine2 VARCHAR(50),

county VARCHAR(30),

postcode VARCHAR(8),

)

CREATE TABLE DISTRIBUTIONCENTER

(

distributionCenter NUMBER PRIMARY KEY,

warehouseID NUMBER,

constraint fk\_warehouseID FROEIGN KEY(warehouseID) REFERENCES Warehouse(WAREHOUSEID),

addressLine1 VARCHAR(50),

addressLine2 VARCHAR(50),

county VARCHAR(30),

postcode VARCHAR(8),

telephone VARCHAR(11)

)

CREATE TABLE CARD

(

cardNumber NUMBER PRIMARY KEY,

card VARCHAR,

expiryDate date,

customerID NUMBER,

constraint fk\_customerID FOREIGN KEY(customerID) REFERENCES CUSTOMER(CustomerID),

)

CREATE TABLE INTERNALORDER

(

internalOrderID NUMBER PRIMARY KEY,

distributionCenterID NUMBER,

constraint fk\_distributionCenterID FROEIGN KEY(distributionCenterID) REFERENCES DistributionCenter(distributionCenterID),

orderType VARCHAR check orderType IN(“Standing Order”, “Special Order”),

)

CREATE TABLE SPECIALORDER

(

specialOrderID NUMBER PRIMARY KEY,

internalOrderID NUMBER,

constraint fk\_ internalOrderID FROEIGN KEY(internalOrderID) REFERENCES InternalOrder (internalOrderID),

season varchar check season IN(“Winter”, “Spring, “Summer” ”, “Autumn” ),

dateOfOrder DATE

)

CREATE TABLE STANDINGORDER

(

standingOrderID NUMBER PRIMARY KEY,

internalOrderID NUMBER,

constraint fk\_ internalOrderNumber FROEIGN KEY(internalOrderID) REFERENCES InternalOrder (internalOrderID),

lastRevised DATE

)

CREATE TABLE DELIVERYDAY

(

deliveryDayID NUMBER PRIMARY KEY,

day Varchar(9) check day IN(“Sunday”, “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday”),

wineID NUMBER,

constraint fk\_ wineID FROEIGN KEY(wineID) REFERENCES Wine (wineID),

)

CREATE TABLE OFFER

(

offerID NUMBER PRIMARY KEY,

description Varchar(100),

discount double precision,

minQuantity NUMBER,

buyFor NUMBER

)

CREATE TABLE WINEOFFER

(

wineOfferID NUMBER PRIMARY KEY,

wineID NUMBER,

constraint fk\_ wineNo FROEIGN KEY(wineID) REFERENCES Wine (wineID),

offerID NUMBER,

constraint fk\_ offerID FROEIGN KEY(offerID) REFERENCES Offer (offerID),

startDate Date,

endDate Date

)

CREATE YABLE SPECIALORDERWINE

(

specialOrderWine NUMBER PRIMARY KEY,

specialOrderID NUMBEE,

CONSTRAINT fk\_specialOrderID FOREIGN KEY(specialOrderID) REFERNCES SPECIALORDER(specialrderID),

WineID fk\_wine FOREIGN KEY(wineID) REFERENCES Wine(wineID),

Quantity NUMBER,

deliveryDate date,

)

CREATE TABLE STANDINGORDERWINE

(

srandingOrderWine NUMBER PRIMARY KEY,

standingOrderID NUMBER,

CONSTRAINT fk\_standingOrderID FOREIGN KEY(standingOrderID) REFERENCES StandingOrder(StandingOrderID),

wineID NUMBER,

CONSTRAINT fk\_wineNumber FOREIGN KEY(WineID) Wine(WineID),

quantity NUMBER,

frequency NUMBER

)

CREATE TABLE DISTRIBUTIONCENTERWINE

(

distributionCenterWineID NUMBER PRIMARY KEY,

distributionCenterID NUMBER,

CONSTRAINT fk\_distributionCenterID FOREIGN KEY(distributionCenterID) REFERNCES DistributionCenter(distributionCenterID),

wineID NUMBER,

CONSTRAINT fk\_wineNum FOREIGN KEY(WineID) REFERENCES Wine(WineID),

Stock NUMBER,

dateOfAvailability date,

)

CREATE TABLE WAREHOUSEWINE

(

warehouseWineID NUMBER PRIMARY KEY,

warehouseID NUMBER,

wineID NUMBER,

CONSTRAINT fk\_wineIDNumber FOREIGN KEY(WineID) REFERENCES Wine(WineID),

Stock NUMBER

)

CREATE TABLE OrderTable

(

orderID NUMBER PRIMARY KEY,

costOfAllItems double precision,

orderDate Date,

cardNumber NUMBER,

CONSTRAINT fk\_ cardNumber FOREIGN KEY(cardNumber) REFERENCES Card(cardNumber),

addressLine1 Varch(50),

postcode VarChar(8)

)

CREATE TABLE OrderWine

(

orderWineID NUMBER PRIMARY KEY,

orderID NUMBER,

CONSTRAINT fk\_ orderID FOREIGN KEY(orderID) REFERENCES OrderTable(orderID),

wineID NUMBER,

CONSTRAINT fk\_wines FOREIGN KEY(WineID) REFERENCES Wine(WineID),

quantity NUMBER,

deliveryDate Date,

subtotal double precision

)

CREATE TABLE BasketItem

(

BasketItemID NUMBER PRIMARY KEY,

customerID NUMBER,

CONSTRAINT fk\_ customerNumber FOREIGN KEY(customerID) REFERENCES Customer(customerID),

wineID NUMBER,

CONSTRAINT fk\_wineIDNum FOREIGN KEY(WineID) REFERENCES Wine(WineID),

quantity NUMBER,

case\_or\_bottle Varchar check case\_or\_bottle IN (“Case”, “Bottle”),

subtotal double precision

)

CREATE TABLE ShoppingListItem

(

ShoppingListItemID NUMBER PRIMARY KEY,

customerID NUMBER,

CONSTRAINT fk\_ customerIDNumber FOREIGN KEY(customerID) REFERENCES Customer(customerID),

wineID NUMBER,

CONSTRAINT fk\_wineIdentity FOREIGN KEY(WineID) REFERENCES Wine(WineID),

dateAdded Date,

)

Queries

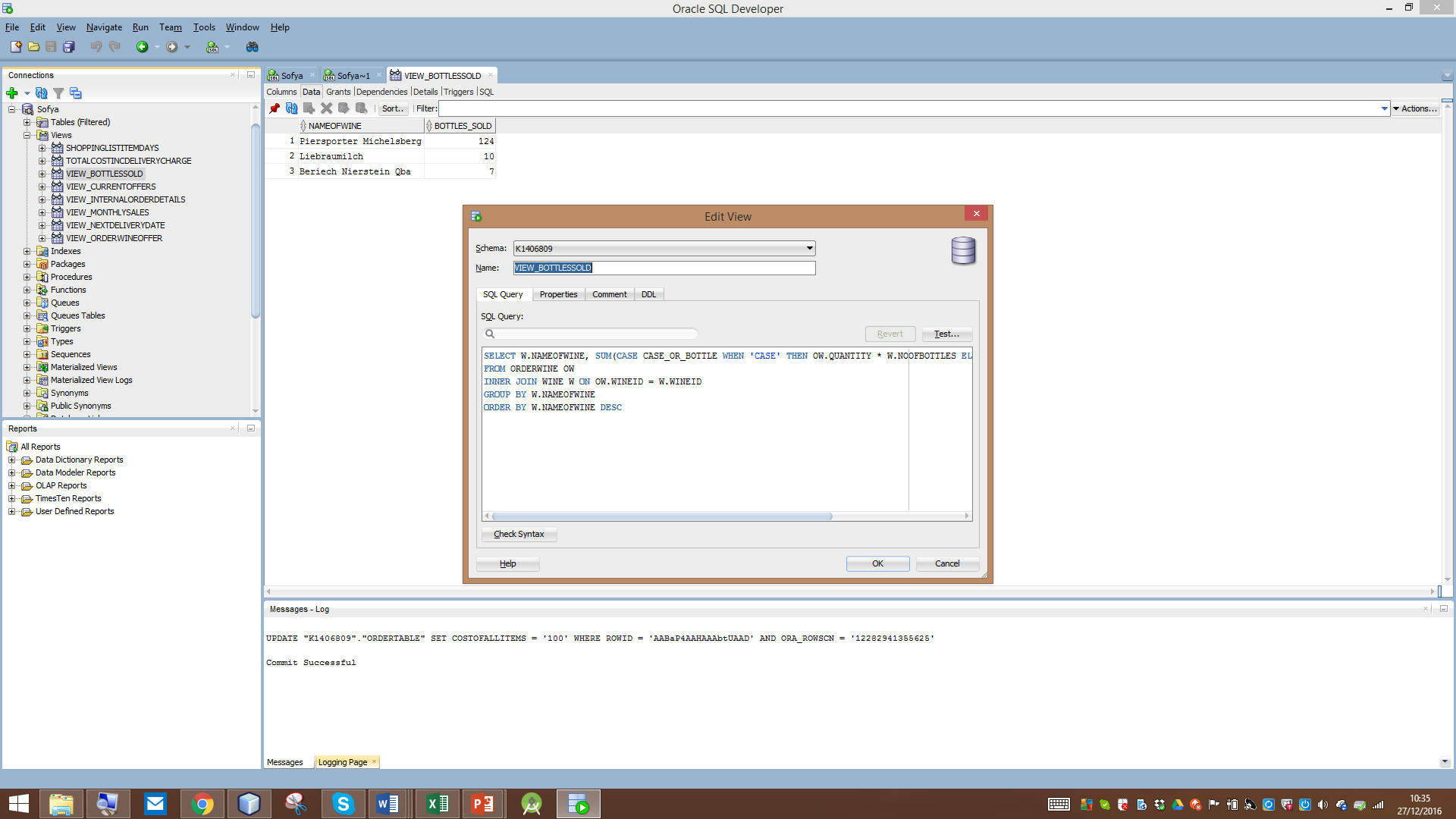


Figure 1: Calculates total no. of bottles sold; taking the no. of bottles in cases into account

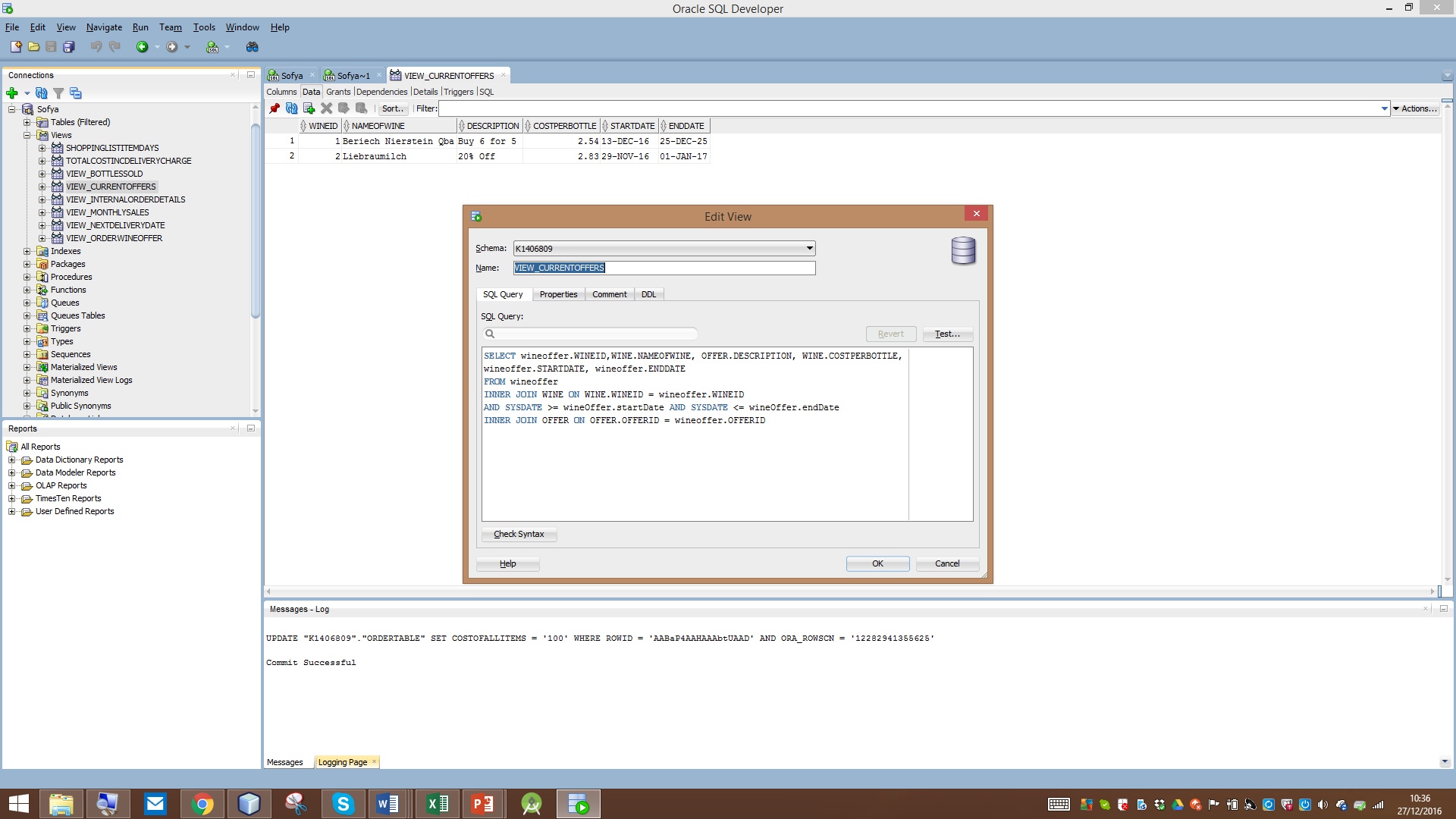


Figure 2: Query executed on 27/12/16

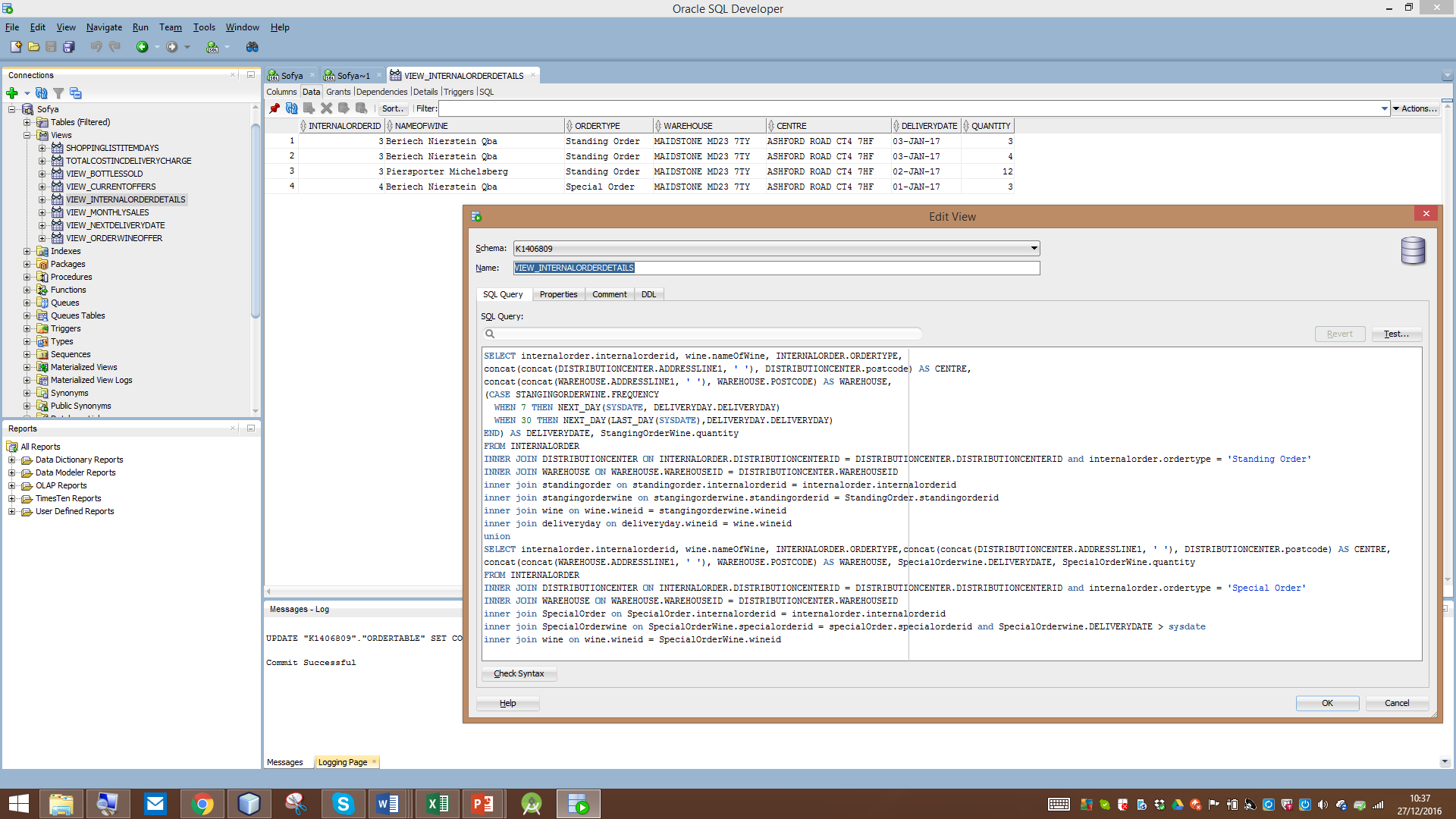


Figure 3: Viewing all internal orders:

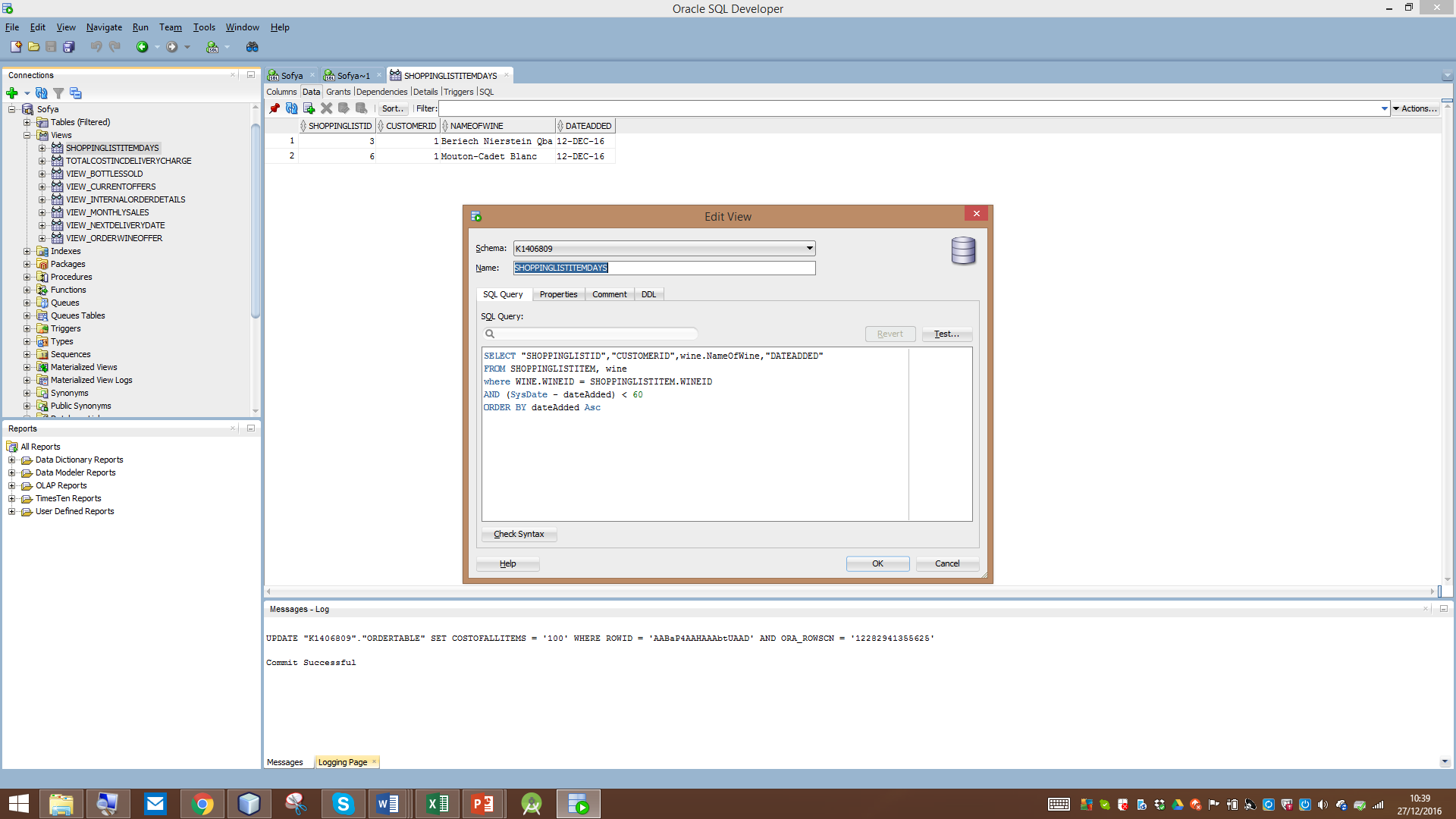


Figure 4: Checking the items in shopping list; query executed on 27/12/16

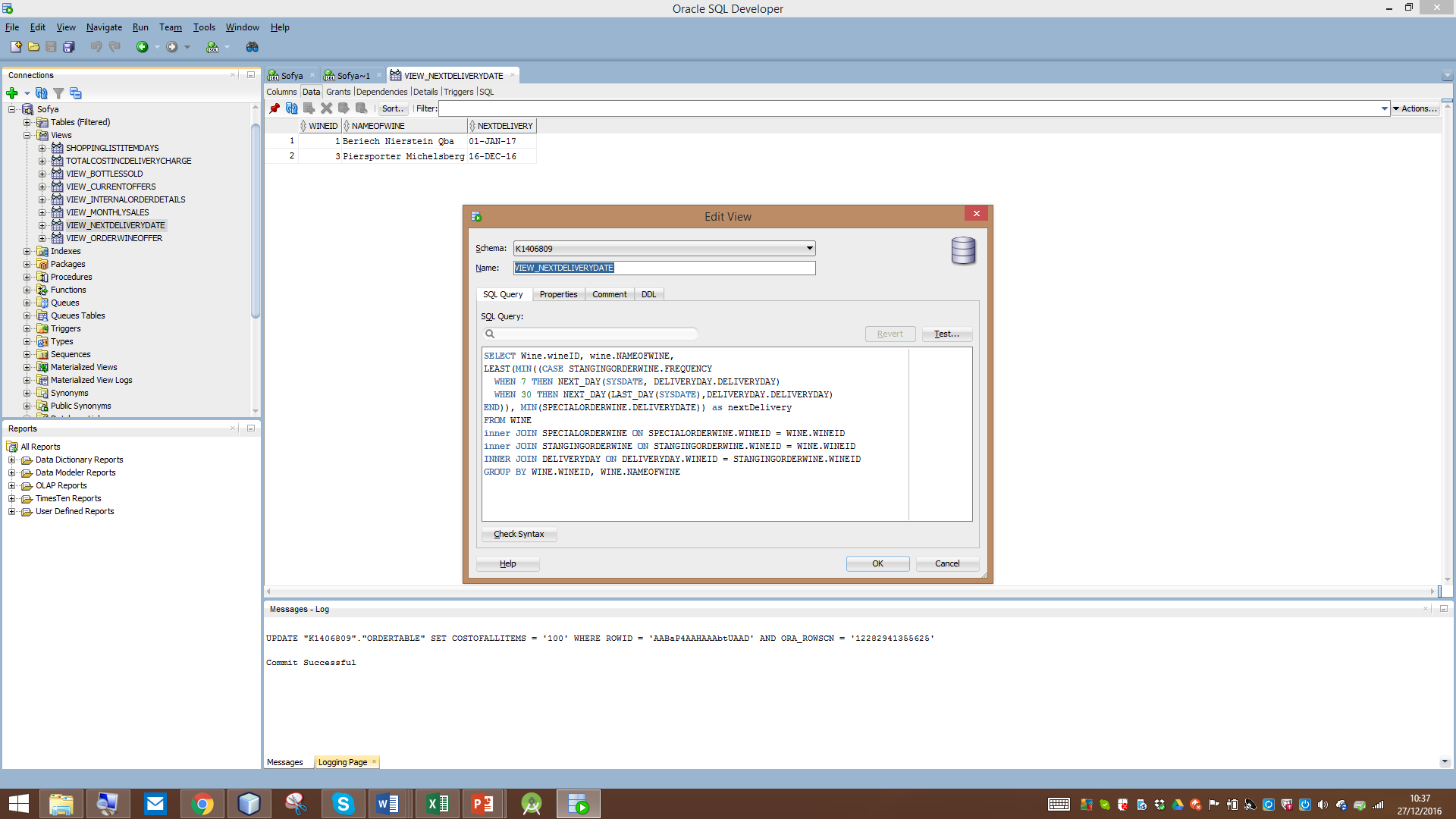


Figure 5: Finds next delivery date; calculates the earliest date between standing order and special order

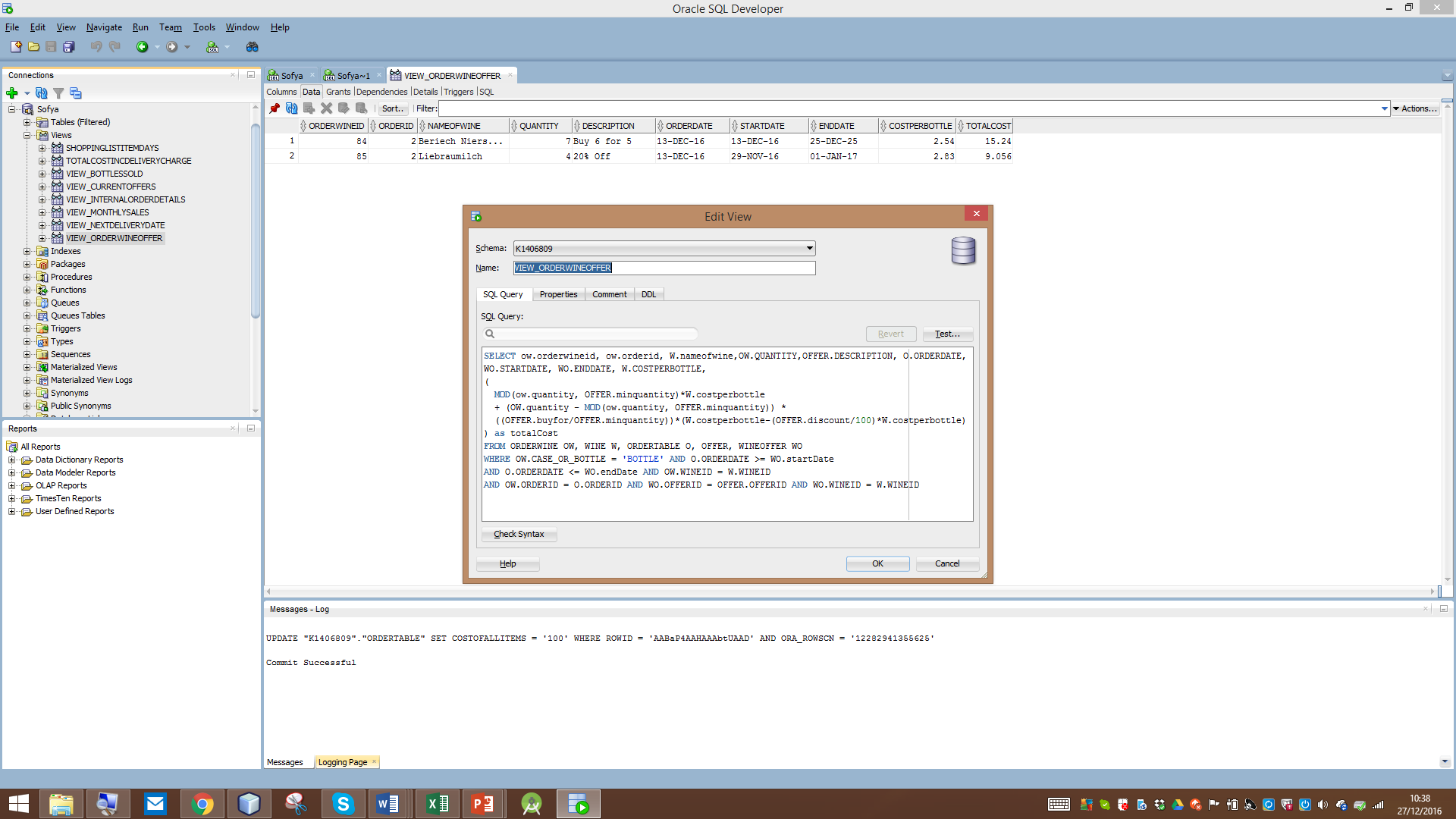


Figure 6: Calculates new cost of items, after offers apply

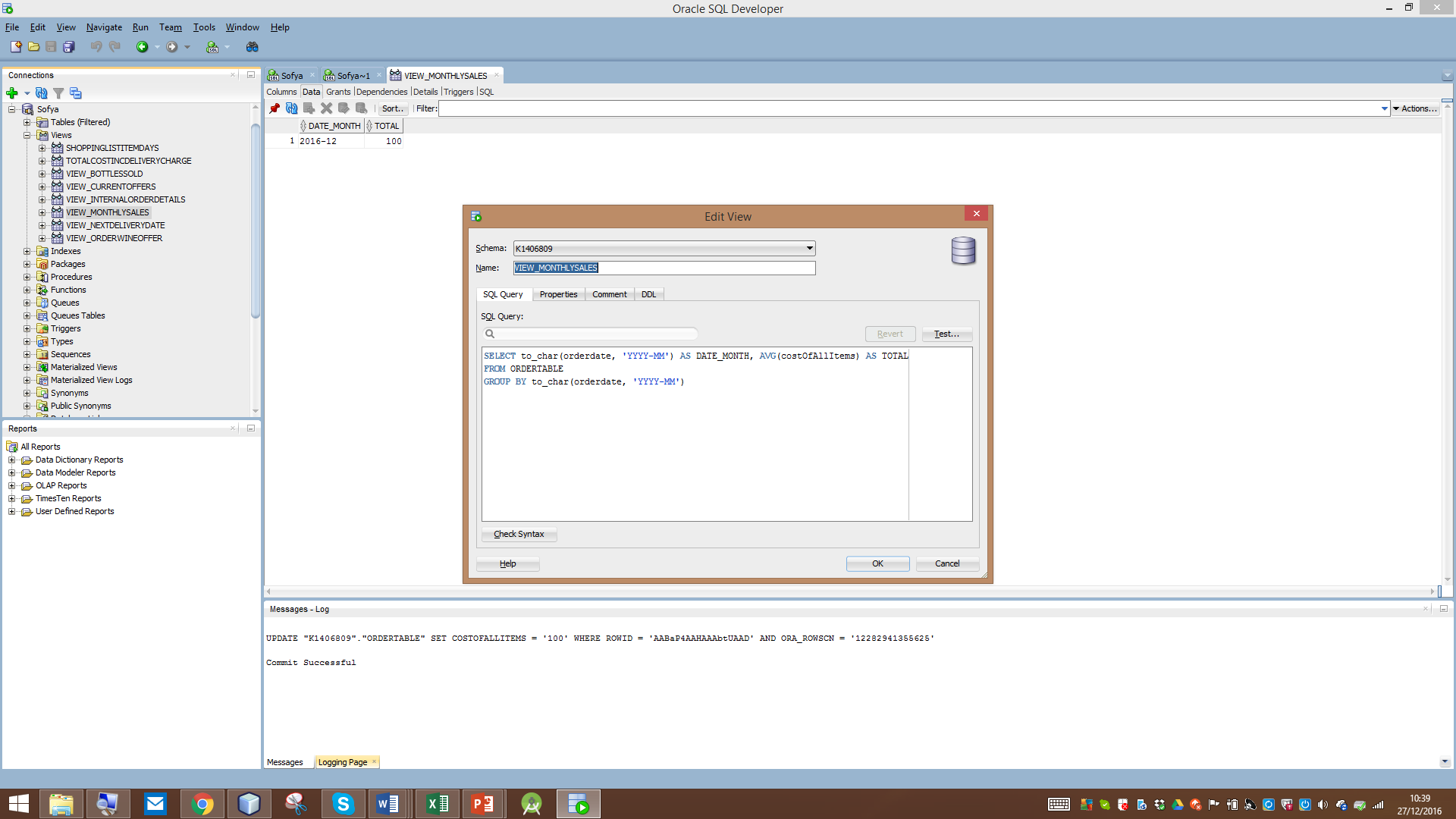


Figure 7: No. of bottles sold each month

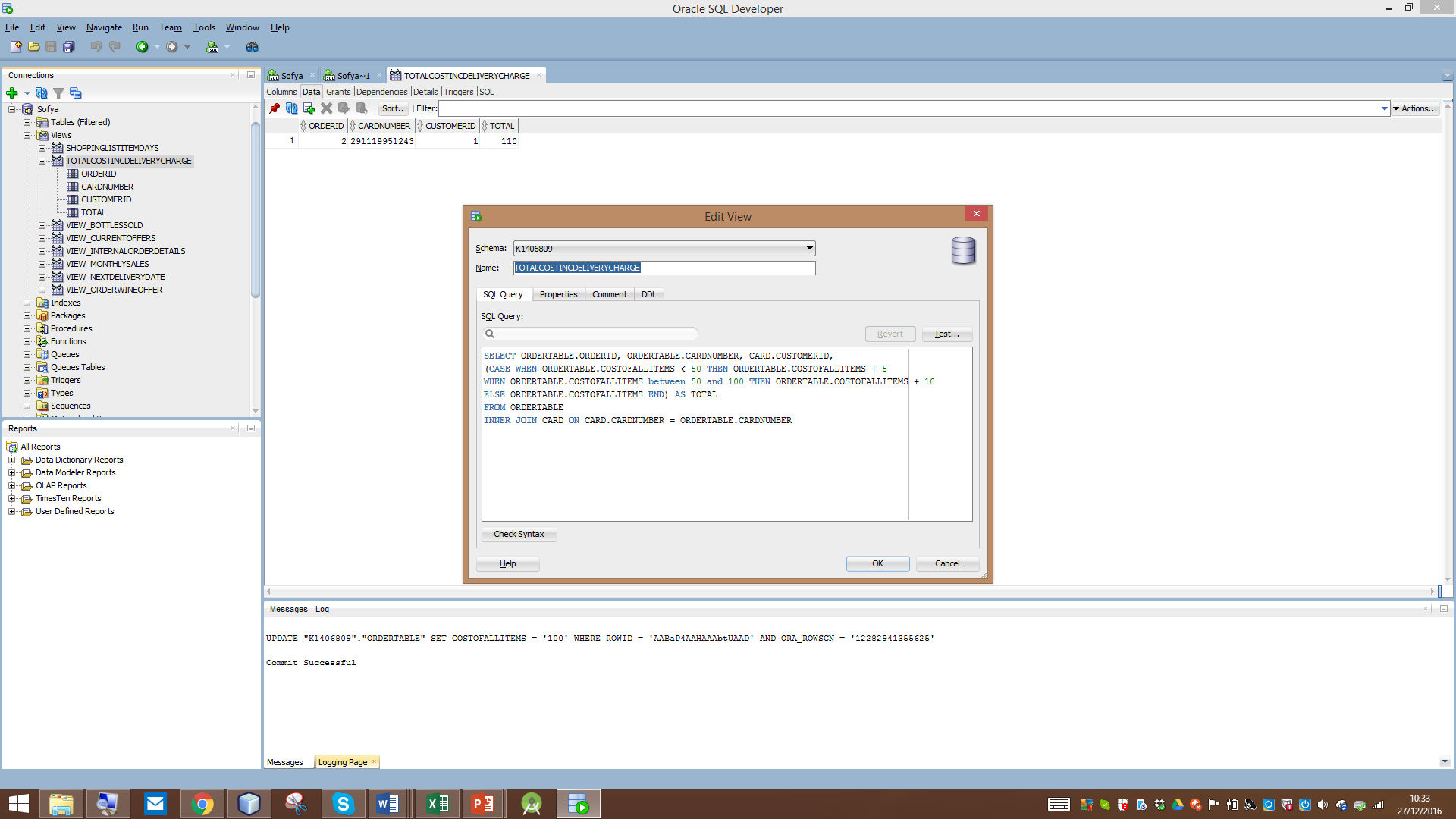


Figure 8: Calculates Cost for each order, including delivery charges