(a) Create a view showing the first and last names of customers with shopping carts, then write a query that return its full extent. Ensure that your result does not include any customer with no cart (or with only an empty shopping cart).

Note that when creating a view, the view may already exist, in which case Oracle offers a **CREATE OR REPLACE** form of the statements that is the one you should always use for creating views.

(b) Create a view showing the code, item number, category id and quantity in stock of inventory items that need to be reordered (where an inventory item needs to be reordered if the quantity in stock drops below 25), then write a query that return its full extent.

(c) Create a view showing the login name, first and last names, order cart id and total price of each order, then write a query that return its full extent.

(d) Create a view showing the login name, first and last names, and total of all orders by a customer, then write a query that return its full extent.

(e) Create a view to return the number of carts per customer, then use this view in a query with a **CASE** statement in the **SELECT** clause that, for each customer, returns the login name and an outcome (represented as a string), which is either **‘BR-1 satisfied’** if that customer has no more than two carts in the database, or **‘BR-1 violated’** otherwise.

(f) Now, rather than define a view, use query nesting in a similar problem, as follows. Firstly, define a query (we’ll refer to it as **Q2**) that returns item num, item size, item colour and a count of how many items of a given color and size there are. Secondly, define a query (we’ll refer to it as **Q1**) that nests **Q2** in its **FROM** clause and (somewhat similarly to the previous task) has a **CASE** statement in the **SELECT** clause that returns as outcome the string ’**BR-2 satisfied’** if the item number does not occur more than once with the same color and size, or **’BR-2 violated’** otherwise. Finally, define a (top-level, as it were) query that uses **Q1** and when executed returns only the item number, color and size that violate the **BR-2** business rule and the string output from **Q2**.

Do not take the terms count and sum to be loose synonyms. In SQL (and in computer science, more generally), they denote strictly different computations.

So, assume a list **L = [1, 2, 3, 4].** Then, **count(L) = 4** and **sum(L) = 10.**

In this part of the exercise, the first step (i.e., the innermost query) is to do a count of how many distinct combinations of a triple of values there are. This is a straight group-by aggregation query. The second step is to use a query to wrap that innermost query and, using a **CASE** construct in its **SELECT** clause, to project out a value that denotes whether the tuples produced by the innermost query reveal a violation (the violation being that the count is greater than 1) and, thus, flag any violation. Finally, the third step is to use an outermost query to only return (i.e., print out) the violations flagged by the (so to speak) midmost query.

(g) Code a SQL trigger that raises an error if the price of an item is set to a value that is more than four times the value of the least expensive item.

In doing this task, make sure that in your final submitted script you include the following tests cases, with their respective resultant output:

i. a SQL query that retrieves all the tuples in the itemType table (so that you know the state before the trigger is fired)

ii. one or more SQL **INSERT** statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the **INSERT** was attempted on a valid and an invalid case)

iii. one or more SQL **UPDATE** statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the **INSERT** was attempted on a valid and an invalid case)

Note that, as with views, when creating a trigger, the trigger may already exist, in which case Oracle offers a **CREATE OR REPLACE** form of the statement that is the one you should always use for creating triggers.

Note also that you should add a single slash as the first and single character in the final line of a trigger. This is so that the parser does not take any ’;’ character in the body of the trigger as the end of the statement.

One important additional piece of information that the SQL\*Plus tutorial didn’t cover is that if you get an error on a trigger named **<triggerName>** (and, of course, you’re likely to), you can get more information on what caused the error by using **SHOW ERRORS TRIGGER <triggerName>**

One additional thing to pay attention to here is that when you’re debugging your triggers, an error may leave the database in an unexpected and undesirable state. Because of this, always think it through and consider the need to take action to reverse the changes and how to do so