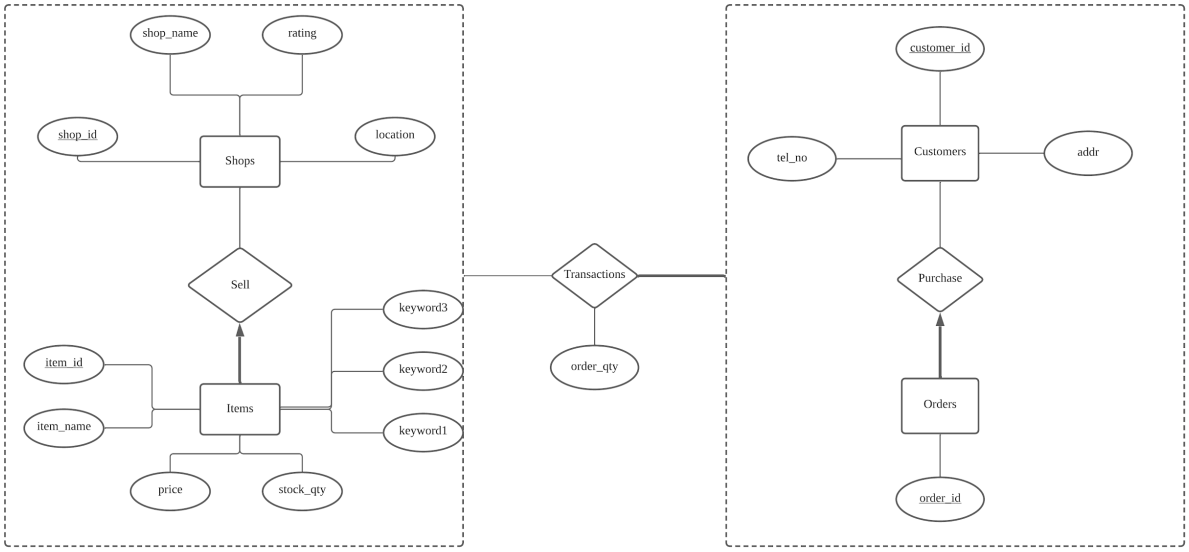
Table of Contents

Group 11 Members

Name	**UID**
CHEUNG Cheuk Yi Gregory	20469659
LEUNG Hei Tung	20440596
LUK Hiu Ying	20467745
FUNG Kwong Kit	21469431
CHAN Kenneth Cheuk Hei	21469970

Design

ER diagram



As shown in the above ER diagram, the '***Shops***' entity set consists of the attributes '***shop_id***', '***shop_name***', '***rating***', and '***location***', in which '***shop_id***' is the key of the entity set. A entity set, namely '***Items***' is characterized by the attributes '***item_id***', '***item_name***', '***price***', '***stock_qty***', '***keyword1***', '***keyword2***' and '***keyword3***', in which '***item_id***' uniquely identifies each entity in this entity set. They are in a one-to-many relationship as items among shops should be unique so as to capture which item is sold by which shop. In addition to that, the '***Items'*** entity set also has a total participation in the '***Sell***' relationship set under the assumption that each item on the online retail shop should belong to an owner.

On the other side of the ER diagram, an entity set called '***Customers***' is also constructed to record attributes '***customer_id***', '***tel_no***' and '***addr***', in which '***customer_id***' is the key for this entity. Another entity set '***Order***' with the attribute '***order_id***' is used to relate to the '***Customers***' entity set for recording the correspondence between customers and orders under the one-to-many settings. As is the case with the '***Items***' entity set, the '***Order***' entity set also has a total participation in the relationship set '***Purchase***' to ensure that each order is owned by a customer.

Treating the relationship sets '***Sell***' and '***Purchase***' as entity sets and joining them together as the '***Transaction***' relationship set with the descriptive attribute being '***order_qty***', we can, therefore, record the items and the quantity purchased for each order. Note also that each entity in the '***Order***' entity set should participate in the relationship set '***Transaction***' as the assumption that an order is not allowed to have an empty set of items should be held.

Table design

```
CREATE TABLE Shops ( shop_id         CHAR(20),
                     shop_name       CHAR(40),
                     rating          INTEGER,
                     location        CHAR(50),
                     PRIMARY KEY (shop_id))
```

As with the ER diagram, the field '***shop_id***' is designated as the primary key.

```
CREATE TABLE Items_Sell (  item_id         CHAR(20),
                           item_name       CHAR(100),
                           price           FLOAT(2),
                           stock_qty       INTEGER,
                           keyword1        CHAR(100),
                           keyword2        CHAR(100),
                           keyword3        CHAR(100),
                           shop_id         CHAR(20) NOT NULL,
                           PRIMARY KEY (item_id),
                           FOREIGN KEY (shop_id)
                           REFERENCES Shops(shop_id) ON DELETE NO
                           ACTION)
```

The relationship set '***Sell***' is created together with the entity set '***Items***', with the '***item_id***' being the primary key of this table and the foreign key '***shop_id***' from the '***Shop***' table.

```
CREATE TABLE Customers (   customer_id     CHAR(20),
                           tel_no          INTEGER(8),
                           addr            CHAR(200),
                           PRIMARY KEY (customer_id))
```

Created the '***Customers***' table with the primary key '***customer_id'*** with the above command.

```
CREATE TABLE Orders_Purchase (  order_id        CHAR(20),
                                customer_id     CHAR(20) NOT NULL,
                                PRIMARY KEY (order_id),
                                FOREIGN KEY (customer_id) REFERENCES
                                Customers(customer_id) ON DELETE NO
                                ACTION)
```

Fusing the relationship set '***Purchase***' and entity set '***order***' into the table '***Orders_Purchase***" with the primary key '***order_id***' and the foregin key '***customer_id***' referencing to the table '***Customers***'

```
CREATE TABLE Transactions (     order_id   CHAR(20),
                            item_id    CHAR(20),
                            order_qty  INTEGER NOT NULL,
                            PRIMARY KEY (order_id, item_id),
                            FOREIGN KEY (order_id) REFERENCES
                            Orders_Purchase(order_id) ON DELETE
                            CASCADE,
                            FOREIGN KEY (item_id REFERENCES
                            Items_Sell(item_id)ON DELETE NO ACTION)
```

Note that the '***Transactions***' table is to be comprising of the four keys from each of the entity set plus the descriptive attribute, i.e. '***order_qty***' in this case, in which, since '***order_id***' and '***item_id***' uniquely identify each record within the table, they are designated as the composite key. In order to cater for the need of order cancellation, a deletion of a particular order would result in deletion of items in that order as well.

Normalization

1. First normal form - Remove multivalued attributes
   In the raw data, some examples of possible multivalued attributes could have been the list of items in shops, or orders from shops, or the three keywords for each item. The data is separated into individual tables to avoid multivalued attributes.
2. Second normal form - Remove partial dependencies
   Partial dependencies could have been in the order table, such as by including shop_id for the orders_purchase table, which would logically be fitting for more convenient referencing. However, this would break the normalization of our data tables. Our database has no partial functional dependencies, as tuples in each table can only be uniquely identified by the whole key.
3. Third normal form - Remove transitive dependencies
   Similar to the second normal form example, including shop_id for the orders_purchase table, which would logically be fitting for more convenient referencing. However the

shop_id could be referenced by the item_id if needed. So the shop_id was excluded to fulfill the third normal form. There are no transitive dependencies.

4. Boyce-Codd normal form - Remove anomalies from functional dependencies
   The Boyce-Codd normal form is generally fulfilled if the third normal form is fulfilled, except under special circumstances, such as overlapping candidate keys. Our database does not have overlapping candidate keys or composite keys, and therefore fulfills the boyce-codd normal form. Note that although item_id may seem like a composite key composed of shop_id and a conventional item_id, it is actually simply a unique key designed for quick human-readability.
5. Fourth normal form - Remove multivalued dependencies
   There are no multivalued dependencies present.
6. Fifth normal form - Remove remaining anomalies
   There are no anomalies present.

Source code explanation

Python

Relevant logical comments are bolded and underlined, and marked with an asterisk (**#***) for easier identification, such as through a find. The asterisk is not in the original python file.

```
import pymysql

#*try get local password file
localDbPassword=''
try:
   f = open("localpw.txt", "r")
   localDbPassword=f.read()
   f.close()
except:
   localDbPassword='password'

#Connect to database
db = pymysql.connect(host='localhost',
                     user='root',
                     password=localDbPassword,
                     database='Comp7640_Proj')
cursor = db.cursor()

#global vars
#table format string
tblFormat1 = "{:<6} {:<40} {:<20} {:<8} {:<40}"
tblFormat2 = "{:<6} {:<20} {:<20} {:<40}"
```

```
#User ID
userid = "NULL"

#*Get number input from user
def getInput(option, mode):
   #*mode values:
   #*0 from main menu
   #*1 from shop menu
   #*2 from

   if mode == 0: #main menu
      for i in range(1,len(option)):
         print("{:<6} {:<50}".format((str(i)+":"), option[i])) #main menu items
      print("{:<6} {:<50}".format((str(0)+":"), option[0])) #quit prog

   elif mode == 1: #all shops, select shop
      for i in range(0,len(option)):
         print("{:<6} {:<50}".format((str(i+1)+":"), option[i]))
      print("{:<6} {:<50}".format(("0:"), "[Go Back]"))

   # elif mode == 2: #
   #     for i in range(0,len(option)):
   #         print("{:<6} {:<40} {:<20} {:<8}".format((str(i+1)+":"), option[i][0], option[i][1], option[i][2]))

   elif mode == 4: #showallcoustomer
      for i in range(0,len(option)):
         print(tblFormat2.format((str(i+1)+":"),option[i][0], option[i][1], option[i][2]))

   elif mode == 5: #getorder
      for i in range(0,len(option)):
         print("{:<6} {:<50}".format((str(i+1)+":"),option[i][0]))

   #*Simple data validation
   while True:
      try:
         val = Input = int(input("Enter number : "))
      except ValueError:
         print("Input must be an integer!")
         continue
      else:
         val = int(val)
         if ((val < 0) or (val > len(option))):
            print("Input must be within 1-"+str(len(option))+"!")
            continue
         break
```

```
    print("")
    return val

#*Show all item of a shop
def showItems(shop_id):

    #*get shop_name
    sql = "SELECT shop_name FROM Shops WHERE shop_id = %s"
    cursor.execute(sql, (shop_id, ))
    shopName = cursor.fetchone()[0]

    print("All items sold by "+ shopName +" [ID: "+shop_id+"]:")
    print("")
    print("{:<6} {:<40} {:<20} {:<8}".format("", "Item Name", "Price(HKD)", "Quantity"))
    try:
        sql = "SELECT item_name, price, stock_qty FROM items_sell WHERE shop_id = %s"
        cursor.execute(sql, (shop_id, ))
        rows = cursor.fetchall()
        result = []
        if len(rows)>0:
            for row in rows:
                temp = []
                temp.append(row[0])
                temp.append(row[1])
                temp.append(row[2])
                result.append(temp)
            for i in range(0,len(result)):
                print("{:<6} {:<40} {:<20} {:<8}".format((str(i+1)+":"), result[i][0], result[i][1], result[i][2]))
        else:
            print("[This shop is empty]")

    except:
        print("Error fetching data!")
    menu()

#*Get all shop
def allShops():
    print("Shops list:")
    sql = "SELECT shop_name, shop_id FROM Shops"
    try:
        cursor.execute(sql)
        rows = cursor.fetchall()
        result = [row[0]+" ("+row[1]+")" for row in rows]
        id = [row[1] for row in rows]
```

```
        input = getInput(result, 1)
        if input > 0:
            shop_id = id[input-1]
            showItems(shop_id)
        elif input == 0:
            menu()

    except:
        print("Error fetching data!")

    #Switch for different shop

def newShop():
    #*Get shop name and check if any duplicate
    name = input("Enter name of the shop: ")
    # sql = "SELECT shop_name FROM Shops WHERE shop_name=%s"
    # check = cursor.execute(sql, name)
    # while check == 0:
    #     print("Shop name already occupied!")
    #     name = input("Enter name of the shop: ")
    #     sql = "SELECT shop_name FROM Shops WHERE shop_name=%s"
    #     check = cursor.execute(sql, name)
    #Get location
    location = input("Enter location of the shop: ")

    #Insert data
    try:
        num = cursor.execute("SELECT shop_name FROM Shops") + 1
        id = str(num).zfill(4)
        id = "S" + id
        sql = "INSERT INTO Shops (shop_id, shop_name, rating, location) VALUES (%s, %s, %s,
%s)"
        cursor.execute(sql,(id, name, 0, location))
        db.commit()
        print(name + " successfully added!")
        print("")
        menu()
    except:
        print("Error fetching data!")

#*Add new items
def newItem():
    shop_id = input("Enter ID of the shop: ")
    try:
        sql = "SELECT shop_name FROM Shops WHERE shop_id=%s"
```

```
        check = cursor.execute(sql, shop_id)
    except Exception as e:
        print(str(e))
    #*Keep checking if shop exist
    while check == 0:
        print("Shop did not exist!")
        shop_id = input("Enter ID of the shop: ")
        sql = "SELECT shop_name FROM Shops WHERE shop_id=%s"
        check = cursor.execute(sql, shop_id)
    name = input("Enter name of new item: ")
    #*Get all required input
    while True:
        try:
            price = int(input("Price of new item: "))
            qty = int(input("Quantity of new item: "))
        except ValueError:
            print("Price and Quantity of new item must be an integer!")
            continue
        else:
            price = int(price)
            qty = int(qty)
            break
    kw1 = input("Keyword 1 of new item: ")
    kw2 = input("Keyword 2 of new item: ")
    kw3 = input("Keyword 3 of new item: ")
    num = cursor.execute("SELECT item_id FROM items_sell WHERE shop_id=%s",shop_id) + 1
    item_id = str(num).zfill(4)
    item_id = shop_id+"I"+item_id
    #item_id parsed in SxxxIxxx format for human-readability

    try:
        if kw1 != "" and kw2 != "" and kw3 != "":
            sql = "INSERT INTO items_sell (item_id, item_name, price, stock_qty, keyword1,
keyword2, keyword3, shop_id) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"
            cursor.execute(sql,(item_id, name, price, qty, kw1, kw2, kw3, shop_id))
        elif kw1 != "" and kw2 != "" and kw3 == "":
            sql = "INSERT INTO items_sell (item_id, item_name, price, stock_qty, keyword1,
keyword2, shop_id) VALUES (%s, %s, %s, %s, %s, %s, %s)"
            cursor.execute(sql,(item_id, name, price, qty, kw1, kw2, shop_id))
        elif kw1 != "" and kw2 == "" and kw3 == "":
            sql = "INSERT INTO items_sell (item_id, item_name, price, stock_qty, keyword1, shop_id)
VALUES (%s, %s, %s, %s, %s, %s)"
            cursor.execute(sql,(item_id, name, price, qty, kw1, shop_id))
        elif kw1 == "" and kw2 == "" and kw3 == "":
```

```
            sql = "INSERT INTO items_sell (item_id, item_name, price, stock_qty, shop_id) VALUES
(%s, %s, %s, %s, %s)"
            cursor.execute(sql,(item_id, name, price, qty, shop_id))
        else:
            print("There is an error!")
        db.commit()
        print(name + " successfully added!")
        print("")
        menu()
    except:
        print("Error fetching data!")

#*Search all items
def searchItems():
    keyword = input("Input keyword: ")
    if keyword != "":
        print("All items that matches the keyword: ", keyword)
        print("")
        print(tblFormat1.format("", "Item Name", "Price(HKD)", "Shop ID", "Shop Name"))
        try:
            #partial matches
            # sqlWhereClause = "("
            # sqlWhereClause = sqlWhereClause+ "item_name LIKE '%" + keyword + "%' OR "
            # sqlWhereClause = sqlWhereClause+  "keyword1 LIKE '%" + keyword + "%' OR "
            # sqlWhereClause = sqlWhereClause+  "keyword2 LIKE '%"+ keyword + "%' OR "
            # sqlWhereClause = sqlWhereClause+  "keyword3 LIKE '%" + keyword + "%'"
            # sqlWhereClause = sqlWhereClause+  ")"
            #
            # sql = "SELECT item_name, price, i.shop_id, shop_name FROM items_sell i, shops s
WHERE i.shop_id=s.shop_id AND " + sqlWhereClause
            # cursor.execute(sql)

            #*full matches only
            sql = "SELECT item_name, price, i.shop_id, shop_name FROM items_sell i, shops s
WHERE i.shop_id=s.shop_id AND(item_name = %s OR keyword1 = %s OR keyword2 = %s OR
keyword3 = %s)"
            cursor.execute(sql, (keyword, keyword, keyword, keyword,))

            rows = cursor.fetchall()
            result = []
            for row in rows:
                temp = []
                temp.append(row[0])
                temp.append(row[1])
                temp.append(row[2])
```

```
            temp.append(row[3])
            result.append(temp)
        for i in range(0,len(result)):
            print(tblFormat1.format((str(i+1)+":"), result[i][0], result[i][1], result[i][2],result[i][3]))
    except:
        print("Error fetching data!")
else:
    print("No keyword detected")
menu()

def showAllCustomers():
    print("Customers list:")
    sql = "SELECT customer_id, tel_no, addr FROM customers"
    try:
        cursor.execute(sql)
        rows = cursor.fetchall()
        if len(rows)>0:
            print(tblFormat2.format(" ","Customer Id", "Phone Number", "Address"))
            # for customer in rows:
            #     print(tblFormat2.format((customer[0]+":"), customer[1], customer[2]))
            val = getInput(rows,4)
            global userid
            userid = rows[val-1][0]
            print("SELECTED ",userid)
        else:
            print("No customers found.")

    except:
        print("Error fetching data from customers table")
    finally:
        menu()

def createCustomer():

    phNum = input("Enter Phone Number: ")
    address = input("Enter Address: ")

    #*Insert data
    try:
        num = cursor.execute("SELECT customer_id FROM customers") +1
        id = str(num).zfill(4)
        id = "C" + id
        print(id)

        sql = "INSERT INTO customers (customer_id, tel_no, addr) VALUES (%s, %s, %s)"
```

```
        cursor.execute(sql,(id, phNum,address))
        print("sql ran")
        db.commit()
        print("Customer with Phone Number: " + phNum + " successfully added!")
        print("")
        menu()
    except:
        print("Error fetching data!")

def showOrders():
    if userid == "NULL":
        print("Please select a customer first in the Show all customers function first!")
        menu()
    else:
        try:
            sql = "SELECT order_id FROM Orders_Purchase WHERE customer_id = %s"
            cursor.execute(sql, userid)
            rows = cursor.fetchall()
            if len(rows) == 0:
                print("No orders from selected user")
                menu()
            else:
                val = getInput(rows,5)
                orderid = rows[val-1][0]
                showOrderItem(orderid)
        except:
            print("Error fetching orders of selected users")

#*Show all items in a order
def showOrderItem(orderid):
    try:
        sql = "SELECT Items_Sell.item_name, Items_Sell.price, Transactions.item_id,
Transactions.order_qty FROM Items_Sell INNER JOIN Transactions ON Items_Sell.item_id =
Transactions.Item_id WHERE Transactions.order_id = %s"
        cursor.execute(sql, orderid)
        rows = cursor.fetchall()
        for row in rows:
            print("{:<50} {:<15} {:<7} {:<5}".format(row[0],row[2],row[1],row[3]))
        print("\n")

        #*Options provided to modify orders
        options = ["Back to main menu","Delete entrie order","Update some items"]
        val = getInput(options, 0)

        if val == 0:
```

```
            menu()
        elif val == 1:
            deleteAllTransactions(orderid)
        elif val == 2:
            updateOrderItemQuantity(orderid, rows)

    except Exception as e:
        print("Error fetching items from selected order")
        print(str(e))
```

#*Operations for editing orders deleted entire order or delete some items

```
def deleteOrder(orderid):
    try:
        sql = "DELETE FROM Orders_Purchase WHERE order_id = %s AND customer_id = %s"
        cursor.execute(sql, (orderid, userid))
        db.commit()
        print("Successfully deleted the order")
        menu()
    except Exception as e:
        print(str(e))

def deleteAllTransactions(orderid):
    try:
        sql = "DELETE FROM Transactions WHERE order_id = %s "
        cursor.execute(sql, (orderid))
        db.commit()
        deleteOrder(orderid)
        print("Successfully deleted the order")
        menu()
    except Exception as e:
        print(str(e))
```

#*Update items quantity or delete items (qty = 0) in orders

```
def updateOrderItemQuantity(orderid, rows):
    for i in range(0,len(rows)):
        print("{:<50} {:<15} {:<7} {:<5}".format(rows[i][0],rows[i][2],rows[i][1],rows[i][3]))
    print("\n0: Back to main menu")
    try:
        itemid, new_qty = input("Enter ItemID and New Quantity: ").split()
        new_qty = int(new_qty)

        print(new_qty, type(new_qty))

        if new_qty == 0:
            sql = "DELETE FROM Transactions WHERE order_id = %s AND item_id = %s"
```

```
            cursor.execute(sql, (orderid, itemid))
            db.commit()
            print("Successfully deleted",itemid,"from the order")
            sql = "SELECT * FROM Transactions WHERE order_id = %s"
            num = cursor.execute(sql, (orderid, ))
            if num == 0:
                deleteOrder(orderid)
        else:
            sql = "UPDATE Transactions SET order_qty = %s WHERE item_id = %s AND order_id =
%s"
            cursor.execute(sql, (new_qty, itemid, orderid))
            db.commit()
            print("\nSuccessfully updated quantity of",itemid,"to",new_qty)
            print("\n")
        showOrderItem(orderid)
    except ValueError:
        menu()
    except Exception as e:
        print("Error deleting items from selected order")
        print(str(e))
```

#*function for adding item to an order

```
def IteminOrders(id, userid):
    itemid, qty = input("Enter ItemID and Quantity: ").split()
    qty = int(qty)
    try:
        try:
            sql = "INSERT INTO transactions (order_id, item_id, order_qty) VALUES (%s, %s, %s)"
            cursor.execute(sql,(id, itemid, qty))
            db.commit()
            print("itemID " + itemid + " successfully added!")
            print("")
            continueorder(id, userid)
```

#*print SQL error and show to customer for an unsuccessful order entry

```
        except (pymysql.Error, pymysql.Warning) as e:
            print(e)
            CheckifOrderEmpty(id, userid)

    except:
        print("Error occured! Item does not exist/Same item already added to your order")
        CheckifOrderEmpty(id, userid)
```

#*delete the order created if no items in the order after “IteminOrders” function

```
def CheckifOrderEmpty(id, userid):
    sql = "SELECT order_id FROM transactions WHERE order_id = %s"
    cursor.execute(sql, (id))
    iteminorder = cursor.fetchall()
    if len(iteminorder) == False:
        print("Order "+id+" has no items, will be deleted.")
        deleteOrder(id)
    else:
        continueorder(id, userid)
```

#*function for adding more than one item in the same order

```
def continueorder(id, userid):
    option = input("Do you want to add more items into the order? [Y/N]: ")

    if option == "Y" or option =="y":
        IteminOrders(id, userid)
    else:
        print("Order "+id+" completed!")
        menu()
```

#*create a new order for the customer

```
def createOrders():
    if userid == "NULL":
        print("Please select a customer first in the Show all customers function first!")
        menu()
    else:
```

#*generate a new order_id

```
            try:
                cursor.execute("SELECT MAX(order_id) FROM orders_purchase")
                result = cursor.fetchone()
                result = result[0]
                result = int(result[1:5])
                id = result + 1
                id = str(id).zfill(4)
                id = str("O" + id)
                sql = "INSERT INTO orders_purchase (order_id, customer_id) VALUES (%s, %s)"
                cursor.execute(sql,(id, userid))
                db.commit()
                print("Order ID " + id + " successfully created for user "+ userid+"!")
                print("")
                IteminOrders(id, userid)
            except:
                print("Error fetching data!")
                CheckifOrderEmpty(id, userid)
```

```
#*Main menu function
def menu():
    print("")
    print("Main Menu - Please select an option:")
    option = ["Quit", "Show all shops", "Add new shop","Add new item","Search by item
keyword","Show all customers","Add new customer","Show Orders","Create Orders"]
    val = getInput(option, 0)

    #*Switch for different option
    if (val == 0):
        return
    elif (val == 1):
        allShops()
    elif (val == 2):
        newShop()
    elif (val ==3):
        newItem()
    elif (val ==4):
        searchItems()
    elif (val ==5):
        showAllCustomers()
    elif (val ==6):
        createCustomer()
    elif (val ==7):
        showOrders()
    elif (val ==8):
        createOrders()

    else:
        print("Error occured")

#*main
print("Welcome to our e-shopping platform")
menu()
```

SQL Triggers

Five triggers were created in this project. Two of them are used to check the stock before a customer adds or updates an order. “BEFORE INSERT” is implemented to check the stock before adding a new order and “BEFORE UPDATE” is applied to perform stock checking before

updating an order. Different error messages will be shown when an item is out of stock and when the order quantity is larger than the stock quantity.

After checking the stock quantity, another two triggers are used to update the stock quantity when a customer adds or updates an order. Similar to stock checking, “AFTER INSERT” is used to update the stock quantity after adding an order, while “AFTER UPDATE” is implemented to update the stock quantity after updating an order. After adding a new order, the stock quantity will be updated as the original quantity minus the order quantity. On the other hand, the new stock quantity will be the original quantity minus the difference between the new and old order quantity after updating an order. The “AFTER UPDATE” trigger will only be triggered if the new order quantity is different from the old quantity.

The last trigger is for updating stock quantity after an order is canceled. “AFTER DELETE” is implemented for this purpose. The stock quantity will be the total number after adding the order quantity to the original stock quantity.

Program readme

A local file “localpw.txt” could be used to store the database password if required. If not found, the default password is “password”. Our default database configurations are as follows:

host	localhost
user	root
password	password
database	Comp7640_Proj

Menus can be traversed by inputting the respective numbers listed in the menu.

Main Menu

1: Show all shops
2: Add new shop
3: Add new item
4: Search by item keyword
5: Show all customers
6: Add new customer

7: Show Orders
8: Create Orders
0: Quit

Sub-menu 1: Show all shops

Choosing this option would show all shops in the database. Users can further select the shop they want and all the items sold by that shop would be displayed.

Sub-menu 2: Add new shop

This function allows users to create new shops. Users are required to define a name for the shop and input the address for the shop. ShopID would be automatically defined by the system. Then, the new shop would be created and data would be stored in the database.

Sub-menu 3: Add new item

This function is similar to the “Add new shop function” while this function is used to add new items. Users are first required to specify which shop they want to add the new items to. Then, they are required to define the name, price and quantity of the new items. After that, they can define up to three keywords for the new item.

Sub-menu 4: Search by item keyword

The user is required to input a keyword. After entering the keyword and pressing Enter, the program will show all items that match the exact keyword, listing item mane, price, shop ID, and shop name.

Sub-menu 5: Show all customers

This is the function that shows all the customers’ information and it is also the function to “login”. After choosing this function, all customers together with their information would be shown, users can choose which user they are by typing the corresponding number and press “enter”. The user id would then be stored and used when editing and creating orders.

Sub-menu 6: Add new customer

This function allows users to create new customers. Users are required to enter the phone number for the customer and input the address for the customer. CustomerID would be

automatically defined by the system. Then, the new customer would be created and data would be stored in the database.

Sub-menu 7: Show Orders

This function shows all the orders related to the selected user chosen using the “Show all customers” (sub-menu 5). They can then select the order they want to modify by entering the corresponding number and press “enter”. Then, the users would have two options which are 1. Deleting the entire order and 2. Update some items. Choosing the first option would delete the whole record while choosing the second option allows users to update the quantity of an item within the order. Users are required to input the item id of the item they want to update and the new quantity. Whether the update process would be successful or not is depending on the stock of that item. Updating an item with quantity equal to 0 indicates the user wants to delete that ietm in the order.

Sub-menu 8: Create Orders

This function is available after the user has already specified the customer ID with “Show all customers” (sub-menu 5).

The program will create a new order and show the current order ID and user ID to the customer. The user is required to input the itemID and quantity. After successfully adding an item to the order, the program will prompt the user to choose if they want to continue adding items or not [Y/N]. If the user inputs Y, they can continue adding items to the order. If the user inputs N, the order will be completed.

If the user inputs a wrong itemID, or the item does not have enough stock for the order, an error message will be displayed, and the order for that item will not be input into the database. If it is the first item the user attempts to enter, the order will be deleted as no item has been successfully added to the order.