Information Management & Systems Engineering

Milestone 2: Online shop

Bryan Yi Jue Tan 11930138
Marwa Wahdan 00727689

2.1. The RDBMS Part (Phase 1)

2.1.1. Configuration of Infrastructure

We used **Python/ Flask** for our project. There are three services needed in the container: **frontend**, **db** (MySQL), and **mongodb**. The services are connected to the default-network network and share a common volume named default-volume. For HTTPS we used the **pyopenssl library** with a self-signed SSL certificate ('adhoc').
To start the program you need to first use the command '**docker-compose up —build**', after everything is ready you will see a link 'https://127.0.0.1:5000/' to our login page.

2.1.2. Logical/Physical database design

Logical database design:

user(user_id, username, email, password)
PK: user_id

customer(phone_number, delivery_address, bonus_points, user_id)
FK: customer.user_id <> user.user_id

normal_account (delivery_fee, point_limit, user_id)
FK: normal_account.user_id <>user.user_id

premium_account (invitation, discount, user_id)
FK:premium_account.user_id <> user.user_id

merchant (merchant_name, website, user_id)
FK: merchant.user_id <>user.user_id

item (item_id, description, price, category, user_id)
PK: item_id
FK: item.user_id <>merchant.user_id

order (order_id, quantity, total_price, delivery_date, user_id)
PK: order_id
FK: order.user_id <>customer.user_id

review (review_id, publish_timestamp, title, description, rating, user_id, item_id)
PK: review_id
FK: review.user_id <>user.user_id
FK: review.item_id <>item.item_id

comment (comment_id, publish_timestamp, content, review_id, user_id)

PK:comment_id
FK: comment.review_id <> review.review_id
FK: comment.user_id <>user.user_id

orderItem(item_id, user_id)
FK: orderItem.item_id<> item.item_id
FK: orderItem.user_id<> customer.user_id

Physical database design:

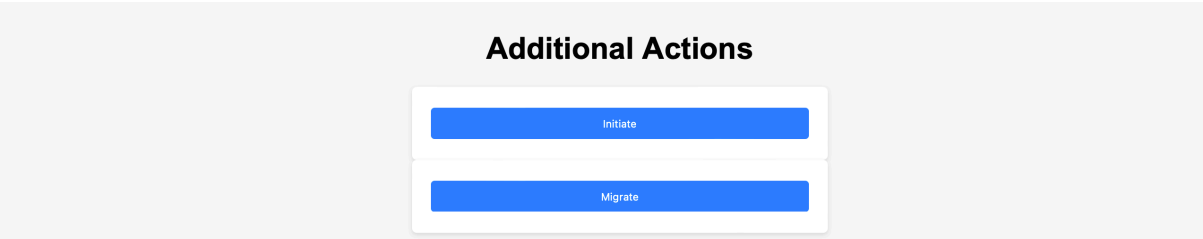
```
CREATE TABLE user (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(50) UNIQUE,
    password VARCHAR(50)
);

 CREATE TABLE customer (
    phone_number BIGINT,
    delivery_address VARCHAR(100),
    bonus_points INT,
    user_id INT AUTO_INCREMENT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);

CREATE TABLE normal_account (
    delivery_fee DECIMAL(10,2),
    point_limit INT,
    user_id INT AUTO_INCREMENT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);

CREATE TABLE premium_account (
    invitation VARCHAR(50),
    discount DECIMAL(10,2),
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);

CREATE TABLE merchant (
    merchant_name VARCHAR(100),
    website VARCHAR(50) ,
    user_id INT AUTO_INCREMENT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);

 CREATE TABLE item (
    item_id INT PRIMARY KEY,
    description VARCHAR(200),
    price DECIMAL(10,2),
    category VARCHAR(200)
```

```
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES customer(user_id)
);

CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    quantity INT,
    total_price INT,
    delivery_date DATE,
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES customer(user_id)

);

CREATE TABLE review (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    publish_timestamp TIMESTAMP,
    title VARCHAR(200),
    description VARCHAR(200),
    rating INT,
    user_id INT,
    item_id INT,
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (item_id) REFERENCES item(item_id)
);

CREATE TABLE comment (
    comment_id INT PRIMARY KEY,
    publish_timestamp TIMESTAMP,
    content VARCHAR(200),
    review_id INT,
    user_id INT,
    FOREIGN KEY (review_id) REFERENCES review(review_id),
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);

CREATE TABLE orderItem (
    order_id INT,
    item_id INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (item_id) REFERENCES item (item_id)
);
```

2.1.3. Data import

In main.py we use the function '**testing_insert_data()**' to complete the initiation. It will delete all data ('**drop_tables(db)**') first and fill it with randomised data ('**insert_data(db)**') again in new created tables ('**create_tables(db)**'). All the data text files are saved in the

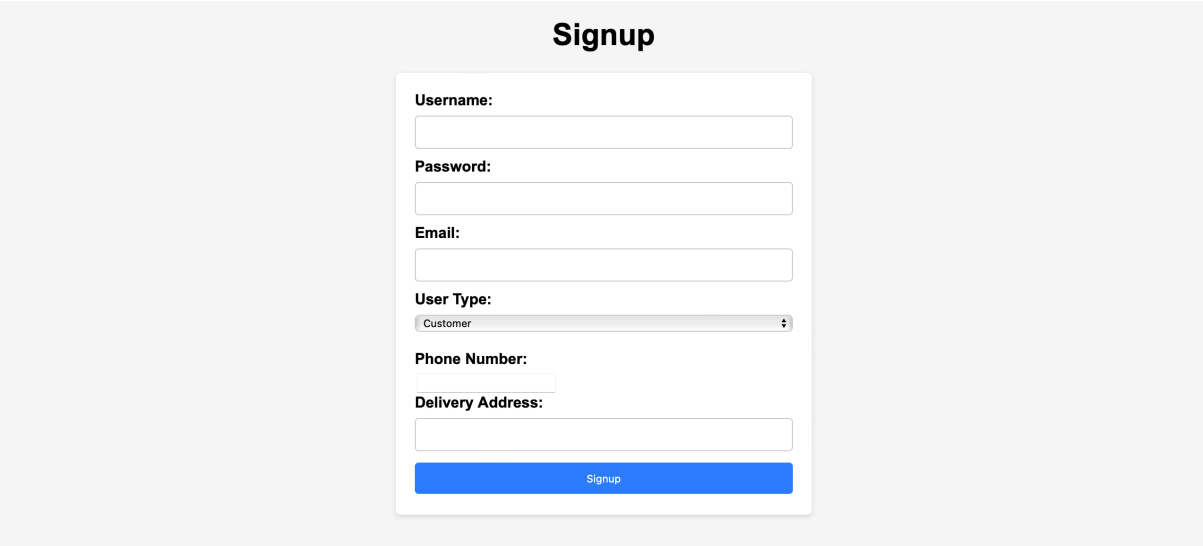
folder name ‘**data**’. A button ‘**Initiate**’ is implemented in the login page. You will be taken to a notification page if the initiation was successful.



2.1.4. Implementation of a Web system

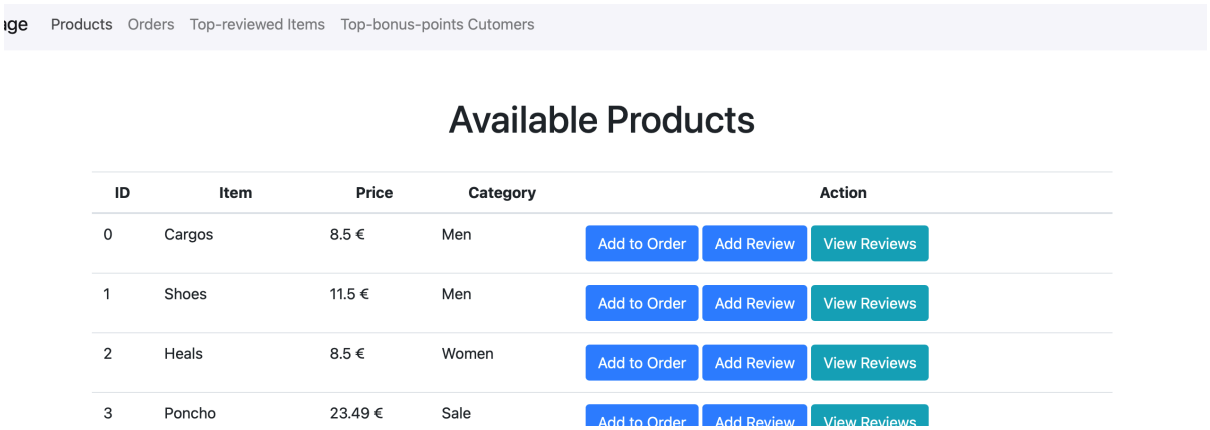
Main Use Case 1 (Marwa Wahdan): Register a new customer

The first use case starts from entering the login page. The user enters his basic information in '**Signup**', selects '**Customer**' in '**User Type**' and continues to enter the extended information of Customer, and finally clicks the 'Signup' button. If there is no problem with the entered information, he will be taken to the prompt interface of successful registration.

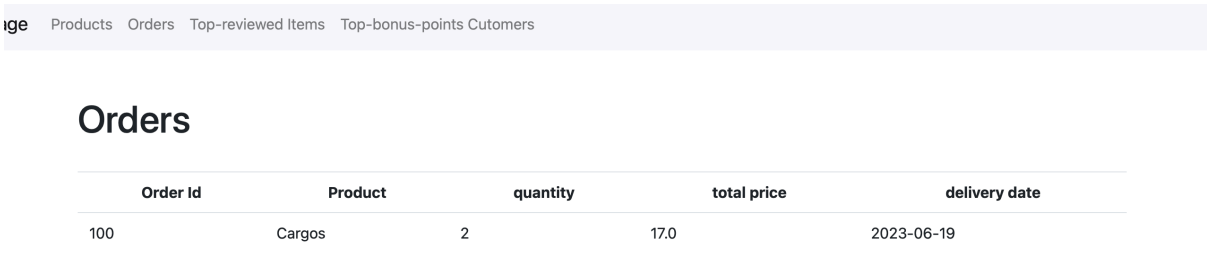


Main Use Case 2 (Bryan Yi Jue Tan): Add item to the order

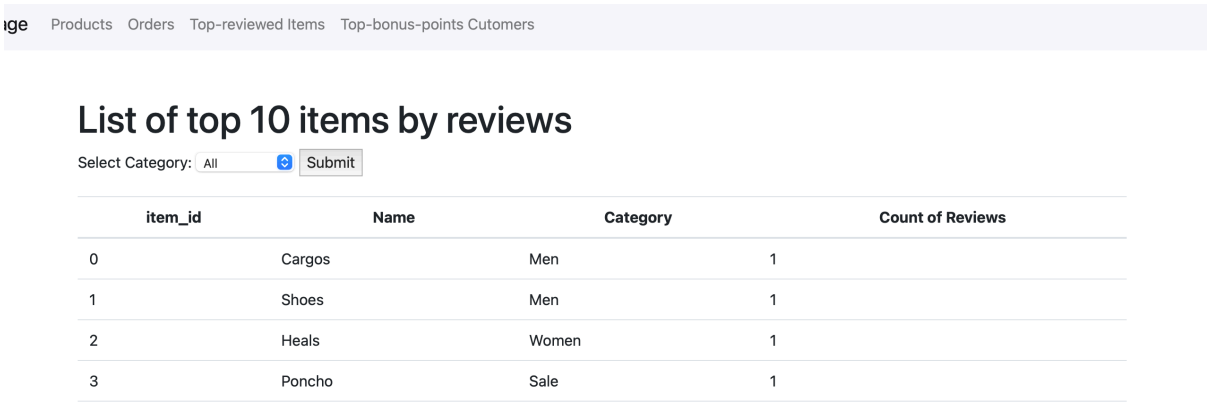
After logging into the account, the user can click the blue '**Shop Now**' button in the middle of the Customer page or the '**Products**' on the upper left to enter the single product page. The customer will add the purchased item to the order by clicking the '**Add to Order**' button. After filling in the quantity of the single item they want to buy, there will be a reminder that the transportation is on the way and the total amount.



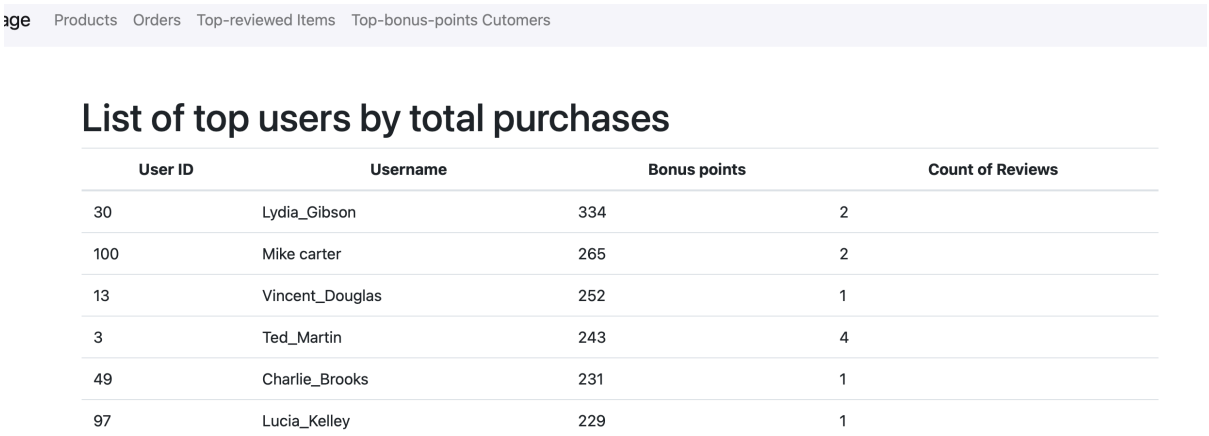
Users can also view their orders by clicking the '**Orders**' button above the site.



Report 1 (Marwa Wahdan):
The report 1 shows the top 10 items with the most written reviews by the customers in a specific category sorted by the amount of review of the item. You can find the report by clicking the 'Top-reviewed Items' button above the site.



Report 2 (Bryan Yi Jue Tan):
The report 2 shows the top 10 customers who have more than 100 bonus points, including the total number of reviews they have written before and sorted by their bonus points. You can find the report by clicking the 'Top-bonus-points Customers' button above the site.



User ID	Username	Bonus points	Count of Reviews
30	Lydia_Gibson	334	2
100	Mike carter	265	2
13	Vincent_Douglas	252	1
3	Ted_Martin	243	4
49	Charlie_Brooks	231	1
97	Lucia_Kelley	229	1

2.2. NoSQL Design (Phase 2)

2.2.1. Compare the design of the RDBS data model to the NoSQL design

In our NoSQL model, we basically stored all the information in a single **document** within a **collection** instead of in a **table** with columns like SQL model. For example:

SQL :

```
CREATE TABLE user (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(50) UNIQUE,
    password VARCHAR(50)
);
```

NoSQL:

```
example_user = {
    "user_id": 1,
    "username": "Julia Barrett",
    "email": "GreenApple88@gmail.com",
    "password": "Juliabar"
}
```

SQL:

```
CREATE TABLE customer (
    phone_number BIGINT,
    delivery_address VARCHAR(100),
    bonus_points INT,
    user_id INT AUTO_INCREMENT,
```

```
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

NoSQL:

```
example_customer= {
    "phone_number": 06817198109,
    "delivery_address": "Koloniestrasse 01a, 9.OG, 9812, Krumbach, NiederÖsterreich, Austria",
    "bonus_points": 334,
    "user_id": 1,
}
```

SQL:

```
CREATE TABLE normal_account (
    delivery_fee DECIMAL(10,2),
    point_limit INT,
    user_id INT AUTO_INCREMENT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

NoSQL:

```
example_normal_account = {
    "delivery_fee": 5,
    "point_limit": 100,
    "user_id": 2
}
```

SQL:

```
CREATE TABLE premium_account (
    invitation VARCHAR(50),
    discount DECIMAL(10,2),
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

NoSQL:

```
example_premium_account = {
    "invitation": "Invitationlink",
    "discount": 0.2,
    "user_id": 3
}
```

SQL:

```
CREATE TABLE merchant (
    merchant_name VARCHAR(100),
    website VARCHAR(50) ,
    user_id INT AUTO_INCREMENT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

NoSQL:

```
example_merchant = {
    "merchant_name": "Roy Sawyer",
    "website": "www.roysawyer.com",
    "user_id": 6
}
```

SQL:

```
CREATE TABLE item (
    item_id INT PRIMARY KEY,
    description VARCHAR(200),
    price DECIMAL(10,2),
    category VARCHAR(200)
);
```

NoSQL:

```
example_item = {
    "item_id": 8,
    "description": "Swimwear",
    "price": 23.49,
    "category": "Women"
}
```

SQL:

```
CREATE TABLE review (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    publish_timestamp TIMESTAMP,
    title VARCHAR(200),
    description VARCHAR(200),
    rating INT,
    user_id INT,
    item_id INT,
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (item_id) REFERENCES item(item_id)

);
```

NoSQL:

```
example_review= {
    "review_id":7,
    "publish_timestamp": (2023-03-13 04:53:39),
    "title": "very Big ",
    "description": "very nice but a bit big. ",
    "rating":4,
    "user_id":11,
    "item_id":43
}
```

SQL:

```
CREATE TABLE comment (
    comment_id INT PRIMARY KEY,
    publish_timestamp TIMESTAMP,
    content VARCHAR(200),
    review_id INT,
    user_id INT,
    FOREIGN KEY (review_id) REFERENCES review(review_id),
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

NoSQL:

```
example_comment= {
    "comment_id":7,
    "publish_timestamp": (2023-04-04 20:07:43),
    "content": "I took a size smaller ",
    "review_id": 7,
    "user_id":11,
}
```

SQL :

```
CREATE TABLE orderItem (
    order_id INT,
    item_id INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (item_id) REFERENCES item (item_id)
);
```

NoSQL :

```
example_orderItem= {
    "order_id":7,
    "item_id":43,
}
```

Since all the information is stored as a single document, retrieval of data can be **faster** compared to performing joins in an SQL database. NoSQL databases are often optimized for read-heavy workloads, which can result in faster read operations.

SQL:

```
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    quantity INT,
    total_price INT,
    delivery_date DATE,
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES user(user_id)

);
```

NoSQL :

```
example_order= {
    "order_id": "10",
    "quantity": 2,
    "total_price": 38.99,
    "delivery_date": 20.03.2023,
    "user_id": 9,
    "item":[
        {     "item_id": 9,
                "description": "Coat",
                "price": 27.99,
                "category": "Women",
        },
         {     "item_id": 8,
                "description": "Swimwear",
                "price": 23.49,
                "category": "Women",
        }
    ]
}
```

In the NoSQL data model for orders, we have a single document for each order, which includes information about the order itself and an array of embedded documents representing the items included in the order. With embedded documents, retrieving an entire order with its associated items can be done **in a single read operation**, eliminating the need for joins and reducing I/O operations. This design can result in **faster retrieval** of order information.

2.2.3. Show and compare the SQL statement and the according NoSQL query

Main Use Case 1 (Marwa Wahdan): Register a new customer

SQL :

```
@app.route('/use_case1', methods=['POST'])
def use_case1():

    username_signup = request.form['username']
    password_signup = request.form['password']
    if len(password_signup) < 6:
        return "weak password"
    email = request.form['email']
    user_type = request.form['user_type']

    if is_migrated == 0:
        cur = db.cursor()
        cur.execute('SELECT * FROM user where username=%s', (username_signup,))
        result = cur.fetchall()
        print(len(result))
        if len(result) > 0:
            return 'duplicate user name '
        else:
            cur.execute('SELECT * FROM user where email=%s', (email,))
            result_email = cur.fetchall()
            print(len(result_email))
            if len(result_email) != 0:
                return 'duplicate mail'
            else:
                cur.execute("INSERT INTO user (username,email, password) VALUES (%s, %s, %s)",
                            (username_signup, email, password_signup))

                if user_type == 'merchant':
                    website = request.form['website']
                    cur.execute(
                        "INSERT INTO merchant (merchant_name,website) VALUES (%s, %s)", (username_signup, website))

                elif user_type == 'customer':
                    phone_number = request.form['phone_number']
                    delivery_address = request.form['delivery_address']
                    bonus = 0
                    cur.execute("INSERT INTO customer (phone_number,delivery_address,bonus_points) VALUES (%s, %s,%s)", (
                        phone_number, delivery_address, bonus))
                cur.close()
    else:
```

NoSQL :

```
    else:
        collection = mongo_db.user
        last_document = collection.find().sort('user_id', -1).limit(1)
        last_user_id = last_document[0]['user_id']
        result_1 = collection.find_one({'username': username_signup})
        if result_1:
            return 'duplicate user name'
        else:
            result_2 = collection.find_one({'email': email})
            if result_2:
                return 'duplicate email'
            else:
                new_user = {
                    'username': username_signup,
                    'email': email,
                    'password': password_signup,
                    'user_id': last_user_id+1
                }
                result = collection.insert_one(new_user)

                if user_type == 'merchant':
                    website = request.form['website']
                    new_merchant = {
                        'merchant_name': username_signup, 'website': website}
                    collection = mongo_db.merchant
                    result = collection.insert_one(new_merchant)

                elif user_type == 'customer':
                    phone_number = request.form['phone_number']
                    delivery_address = request.form['delivery_address']
                    bonus = 0
                    new_customer = {
                        'phone_number': phone_number,
                        'delivery_address': delivery_address,
                        'bonus_points': bonus}
                    collection = mongo_db.customer
                    result = collection.insert_one(new_customer)

    return render_template('success.html', message="Signed up successfully")

```

The use case described in the code is a **user signup process**. Both the SQL and NoSQL approaches perform similar tasks of checking for duplicate usernames and emails before inserting user information.
The SQL statement performs a series of **SELECT** queries to check for duplicate username and email in the "user" table. If no duplicates are found, it inserts a new user into the **table**. Depending on the user type, it also inserts data into either the "merchant" or "customer" table.
The NoSQL query interacts with the "user" collection in MongoDB. It uses the **find_one** method to check for duplicate username and email. If no duplicates are found, it constructs a new user document and inserts it into the **collection**. Similar to the SQL statement, it also handles the insertion into the "merchant" or "customer" collections based on the user type.

Main Use Case 2 (Bryan Yi Jue Tan): Add item to the order
SQL :

```
@app.route('/use_case2', methods=['POST'])
def use_case2():
    global is_migrated
    id = request.form['id']
    quantity = request.form['anzahl']
    current_date = date.today()
    formatted_date = current_date.strftime("%Y-%m-%d")
    if is_migrated == 0:
        cur = db.cursor(buffered=True)
        cur.execute('SELECT user_id FROM user where username=%s', (username,))
        user_id = cur.fetchone()[0]
        cur.execute("select price from item where item_id=%s", (id,))
        price = cur.fetchall()[0][0]
        price = float(price)
        print(type(quantity), type(price))

        cur.execute("INSERT INTO orders (quantity,total_price,delivery_date,user_id) VALUES (%s, %s, %s,%s)",
                    (quantity, int(quantity)*price, formatted_date, user_id))
        cur.execute("select max(order_id) from orders")
        order_id_needed = cur.fetchall()
        order_id_needed = str(order_id_needed[0][0])
        cur.execute(
            "INSERT INTO orderItem (order_id,item_id) VALUES (%s, %s)", (order_id_needed, id))
        db.commit()
        totalprices = int(quantity)*price
    else:
```

NoSQL :

```
    else:
        collection_user = mongo_db.user
        user = collection_user.find_one({'username': username}, {'user_id': 1})
        if user:
            user_id = user['user_id']
        collection_item = mongo_db.item
        item_col = collection_item.find_one(
            {'item_id': int(id)}, {'description': 1, 'price': 1})
        if item_col:
            item_desc = item_col['description']
            item_price = item_col['price']
            print(item_desc, item_price, quantity)
            totalprices = item_price*int(quantity)
        collection_order = mongo_db.orders
        last_document = collection_order.find().sort('order_id', -1).limit(1)
        last_order_id = last_document[0]['order_id']
        new_order = {
            'order_id': last_order_id+1,
            'item_id': id,
            'description': item_desc,
            'quantity': quantity,
            'user_id': int(user_id),
            "delivery_date": formatted_date,
            "total_price": float(totalprices)
        }
        result = collection_order.insert_one(new_order)
    return "delivery is on your way and the total price is " + str(totalprices)
```

The use case described in the code is an **adding item to order process**. Both the SQL and NoSQL approaches achieve the goal of inserting data into the respective database systems. The SQL statement retrieves data from the "user" and "item" tables, performs necessary calculations, and inserts data into the "orders" and "orderItem" **tables**. It uses SQL statements such as **SELECT**, **INSERT**, and **UPDATE**.
The NoSQL query retrieves data from the "user" and "item" collections using the **find_one()** method. It performs necessary calculations and inserts data into the "orders" **collection**. It uses MongoDB's query syntax and methods to interact with the NoSQL database.

Report 1 (Marwa Wahdan):
SQL :

```
@app.route('/report1', methods=['GET', 'POST'])
def report1():
    if request.method == 'POST':
        selected_category = request.form.get('category')
        if selected_category == 'All':
            if is_migrated == 0:
                cursor = db.cursor()
                cursor.execute("""SELECT top_items.item_id, item.description, item.category, top_items.count AS item_count
                            FROM (
                                SELECT item_id, COUNT(item_id) AS count
                                FROM review
                                GROUP BY item_id
                                ORDER BY count DESC
                                LIMIT 10
                            ) AS top_items
                            JOIN item ON top_items.item_id = item.item_id
                            LIMIT 10;""")
                order = cursor.fetchall()
            else:
                review_collection = mongo_db.review
                pipeline = [
                    {'$group': {
                        '_id': '$item_id',
                        'count': {'$sum': 1}
                    }},
                    {'$lookup': {
                        'from': 'item',
                        'localField': '_id',
                        'foreignField': 'item_id',
                        'as': 'item'
                    }},
                    {'$unwind': '$item'},
                    {'$project': {
                        '_id': 0,
                        'item_id': '$_id',
                        'description': '$item.description',
                        'category': '$item.category',
                        'count': 1
                    }},
                    {'$sort': {'count': -1}},
                    {'$limit': 10}
                ]
                result = review_collection.aggregate(pipeline)
                order = [[item['item_id'], item['description'],
                          item['category'], item['count']] for item in result]
```

NoSQL :

```
    else:
        if is_migrated == 0:
            cursor = db.cursor()
            cursor.execute("""SELECT top_items.item_id, item.description, item.category, top_items.count AS item_count
                        FROM (
                            SELECT item_id, COUNT(item_id) AS count
                            FROM review
                            GROUP BY item_id
                            ORDER BY count DESC
                        ) AS top_items
                        JOIN item ON top_items.item_id = item.item_id
                        WHERE item.category = %s
                        LIMIT 10;""", (selected_category,))
            order = cursor.fetchall()
        else:
            review_collection = mongo_db.review
            pipeline = [
                {'$group': {
                    '_id': '$item_id',
                    'count': {'$sum': 1}
                }},
                {'$lookup': {
                    'from': 'item',
                    'localField': '_id',
                    'foreignField': 'item_id',
                    'as': 'item'
                }},
                {'$unwind': '$item'},
                {'$project': {
                    '_id': 0,
                    'item_id': '$_id',
                    'description': '$item.description',
                    'category': '$item.category',
                    'count': 1
                }},
                {'$match': {'category': selected_category}},
                {'$sort': {'count': -1}},
                {'$limit': 10}
            ]
            result = review_collection.aggregate(pipeline)
            order = [[item['item_id'], item['description'],
                      item['category'], item['count']] for item in result]
        return render_template('report1.html', employees=order)
    else:
        return render_template('report1.html', employees=[])
```

The SQL statement retrieves data from the "review" and "item" tables based on the selected category. It uses SQL statements such as **SELECT**, **JOIN**, and **GROUP BY** to perform aggregations and join operations.
The NoSQL query retrieves data from the "review" and "item" collections based on the selected category using the aggregation framework in MongoDB. It uses pipeline stages such as **$group**, **$lookup**, **$unwind**, **$project**, **$match**, **$sort**, and **$limit** to perform aggregations, joins, filtering, sorting, and limiting.
The **$group** stage in the NoSQL query is equivalent to the **GROUP BY** clause in SQL, while **$lookup** is equivalent to the **JOIN**, **$project** to **SELECT**, **$match** to **WHERE**, **$sort** to **ORDER BY** and **$limit** to **LIMIT** clause in SQL. The **$unwind** stage in the NoSQL aggregation framework is not directly equivalent to any specific clause in SQL.

Report 2 (Bryan Yi Jue Tan):
SQL :

```
@app.route('/report2', methods=['GET', 'POST'])
def report2():
    global user_id, username
    if is_migrated == 0:
        cursor = db.cursor()
        cursor.execute("""
            SELECT u.user_id, u.username, MAX(c.bonus_points), COUNT(DISTINCT r.review_id)
            FROM user u
            JOIN customer c ON c.user_id = u.user_id
            JOIN review r ON c.user_id = r.user_id
            WHERE c.bonus_points > 100
            GROUP BY u.user_id, u.username
            ORDER BY MAX(c.bonus_points) DESC
        """)
        order = cursor.fetchall()
    else:
```

NoSQL :

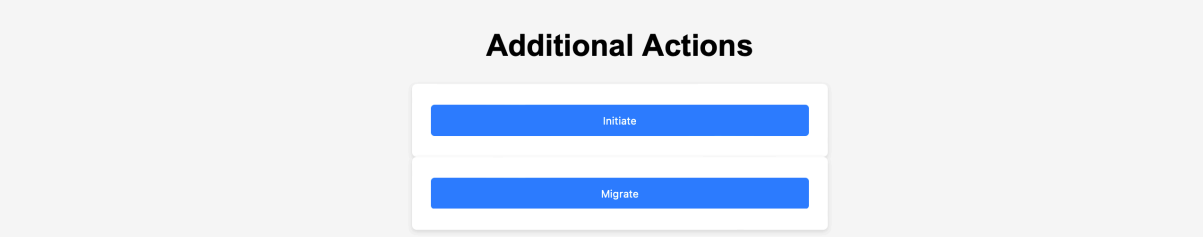
```
    else:
        customer_collection = mongo_db.user
        pipeline = [
            {'$lookup': {
                'from': 'user',
                'localField': 'user_id',
                'foreignField': 'user_id',
                'as': 'user_info'
            }},
            {'$lookup': {
                'from': 'customer',
                'localField': 'user_id',
                'foreignField': 'user_id',
                'as': 'customer_info'
            }},
            {'$lookup': {
                'from': 'review',
                'localField': 'user_id',
                'foreignField': 'user_id',
                'as': 'reviews'
            }},
            {'$group': {
                '_id': '$user_id',
                'review_count': {'$sum': {'$size': '$reviews'}},
                'username': {'$first': {'$arrayElemAt': ['$user_info.username', 0]}},
                'bonus_points': {'$first': {'$arrayElemAt': ['$customer_info.bonus_points', 0]}}
            }},
            {'$match': {'bonus_points': {'$gt': 100}}},
            {'$sort': {'bonus_points': -1
                       }},
            {'$project': {
                'user_id': '$_id',
                'username': 1,
                'review_count': 1,
                'bonus_points': 1,
                '_id': 0
            }}
        ]
        result = customer_collection.aggregate(pipeline)
        order = [[item['user_id'], item['username'],
                  item['bonus_points'], item['review_count']] for item in result]
    return render_template('report2.html', users=order)
```

The SQL statement retrieves user information for customers who have more than 100 bonus points. It performs **joins** between the "user," "customer," and "review" tables and **groups** the results by user ID and username. The results are ordered based on the maximum bonus points.
The NoSQL query interacts with the "user," "customer," and "review" collections in MongoDB. It uses the **$lookup** stage to join the collections and retrieve the necessary information. It then applies the **$group** stage to group the data by user ID, extracts the username and bonus points using $first and calculates the count of reviews using **$size**. The results are filtered using the **$match** stage to select users with bonus points greater than 100. Finally, the results are sorted based on bonus points and projected to include the desired fields.

2.3. NoSQL Implementation (Phase 3)

2.3.1. Data migration

In main.py we use the function '**migrate_data(db)**' to complete the migration. It will delete all collections ('**drop_mongodb_columns()**') first and create collections for different entities before migrate the data into collections. A button '**Migrate**' is implemented as well in the login page. You will be taken to a notification page if the migration was successful.



2.3.2. Implementation IS (NoSQL)

In order to facilitate subsequent editing of the code, we compile the NoSQL part under the SQL part. Specifically, we first define a global variable **is_migrated**. If the migration_state is 0, the SQL part will be executed, otherwise the NoSQL part will be executed.

3. Task assignment:

- Configuration of Infrastructure - **Bryan**
- Logical/Physical database design - **Marwa**
- Data import - **Marwa**
- NoSQL design decisions - **Bryan, Marwa**
- Data migration - **Bryan**
- Implementation Web system and Use Cases (relational DBMS):
  - Use Case 1, Report 1 - **Marwa**
  - Use Case 2, Report 2 - **Bryan**
- Implementation IS (NoSQL):
  - Use Case 1, Report 1 - **Marwa**
  - Use Case 2, Report 2 - **Bryan**