

# **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.

```
mongodb | {"t":{"$date":"2023-06-19T16:34:42.675+00:00"},"s":"I", "c":"NETWORK", "id":51800, "ctx":"conn7", "msg":"client metadata", "attr":{"remote":"192.168.144.4:46244", "client":"conn7", "doc":{"driver":{"name":"PyMongo", "version":"4.3.3"}, "os":{"type":"Linux", "name":"Linux", "architecture":"x86_64", "version":"5.15.49-linuxkit"}, "platform":"CPython 3.10.12.final.0"}}}
frontend | * Serving Flask app 'main'
frontend | * Debug mode: on
frontend | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
frontend | * Running on all addresses (0.0.0.0)
frontend | * Running on https://127.0.0.1:5000
frontend | * Running on https://192.168.144.4:5000
frontend | Press CTRL+C to quit
frontend | * Restarting with stat
mongodb | {"t":{"$date":"2023-06-19T16:34:44.023+00:00"},"s":"I", "c":"NETWORK", "id":22943, "ctx":"listener", "msg":"Connection accepted", "attr":{"remote":"192.168.144.4:46248", "uid":"b73e0bf7-6883-4987-9139-a176ffd2cb76", "connectionId":8, "connectionCount":3}}
mongodb | {"t":{"$date":"2023-06-19T16:34:44.030+00:00"},"s":"I", "c":"NETWORK", "id":51800, "ctx":"conn8", "msg":"client metadata", "attr":{"remote":"192.168.144.4:46248",
```

### 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)
);

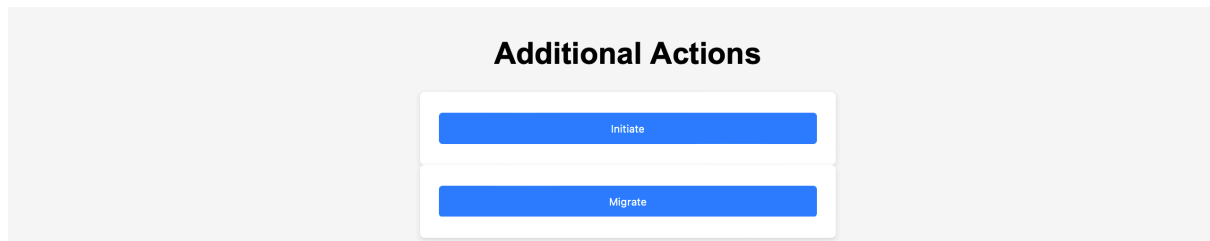
CREATE TABLE orderItem (
    order_id INT,
    item_id INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (item_id) REFERENCES item (item_id)
);

```

### 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.

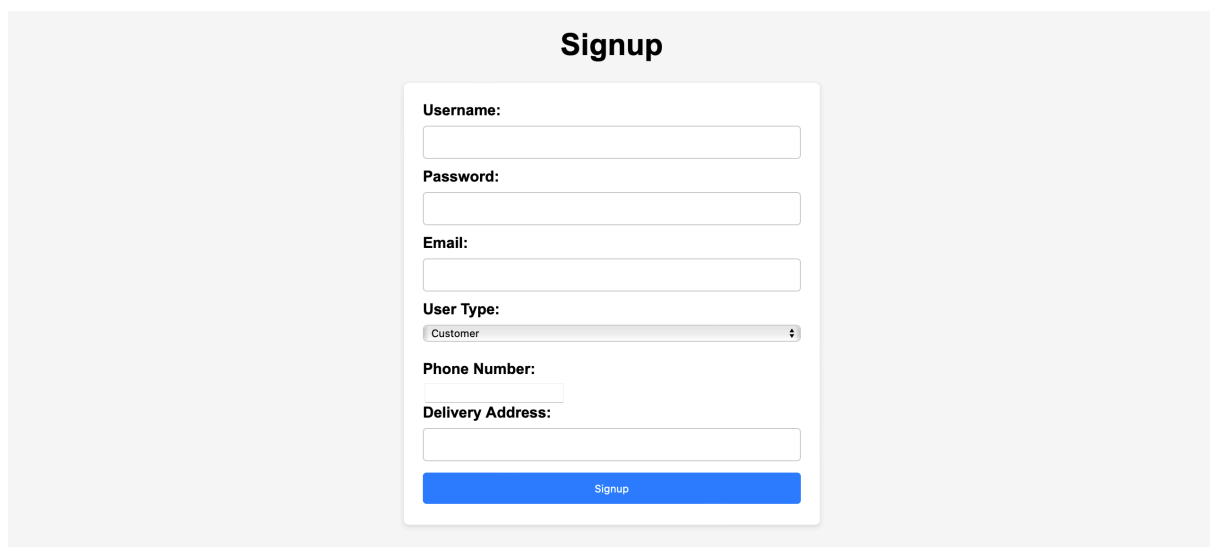


The image shows a light gray rectangular container with the title "Additional Actions" centered at the top in bold black text. Below the title, there are two white rectangular boxes stacked vertically. Each box contains a single blue button with white text. The top button is labeled "Initiate" and the bottom button is labeled "Migrate".

#### 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.



The image shows a light gray rectangular container with the title "Signup" centered at the top in bold black text. Below the title, there is a white rectangular form. The form contains the following fields: "Username:" with a text input field, "Password:" with a text input field, "Email:" with a text input field, "User Type:" with a dropdown menu showing "Customer" and a downward arrow, "Phone Number:" with a text input field, and "Delivery Address:" with a text input field. At the bottom of the form is a blue button with white text labeled "Signup".

##### 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.

## Available Products

ID	Item	Price	Category	Action		
0	Cargos	8.5 €	Men	<a href="#">Add to Order</a>	<a href="#">Add Review</a>	<a href="#">View Reviews</a>
1	Shoes	11.5 €	Men	<a href="#">Add to Order</a>	<a href="#">Add Review</a>	<a href="#">View Reviews</a>
2	Heals	8.5 €	Women	<a href="#">Add to Order</a>	<a href="#">Add Review</a>	<a href="#">View Reviews</a>
3	Poncho	23.49 €	Sale	<a href="#">Add to Order</a>	<a href="#">Add Review</a>	<a href="#">View Reviews</a>

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

## Orders

Order Id	Product	quantity	total price	delivery date
100	Cargos	2	17.0	2023-06-19

### 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.

## List of top 10 items by reviews

Select Category:  [Submit](#)

item_id	Name	Category	Count of Reviews
0	Cargos	Men	1
1	Shoes	Men	1
2	Heals	Women	1
3	Poncho	Sale	1

## 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.

age Products Orders Top-reviewed Items Top-bonus-points Customers

### List of top users by total purchases

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 :

```
586 @app.route('/use_case1', methods=['POST'])
587 def use_case1():
588
589     username_signup = request.form['username']
590     password_signup = request.form['password']
591     if len(password_signup) < 6:
592         return "weak password"
593     email = request.form['email']
594     user_type = request.form['user_type']
595
596     if is_migrated == 0:
597         cur = db.cursor()
598         cur.execute('SELECT * FROM user where username=%s', (username_signup,))
599         result = cur.fetchall()
600         print(len(result))
601         if len(result) > 0:
602             return 'duplicate user name '
603         else:
604             cur.execute('SELECT * FROM user where email=%s', (email,))
605             result_email = cur.fetchall()
606             print(len(result_email))
607             if len(result_email) != 0:
608                 return 'duplicate mail'
609         else:
610             cur.execute("INSERT INTO user (username,email, password) VALUES (%s, %s, %s)",
611                         (username_signup, email, password_signup))
612
613             if user_type == 'merchant':
614                 website = request.form['website']
615                 cur.execute(
616                     "INSERT INTO merchant (merchant_name,website) VALUES (%s, %s)", (username_signup, website))
617
618             elif user_type == 'customer':
619                 phone_number = request.form['phone_number']
620                 delivery_address = request.form['delivery_address']
621                 bonus = 0
622                 cur.execute("INSERT INTO customer (phone_number,delivery_address,bonus_points) VALUES (%s, %s,%s)", (
623                     phone_number, delivery_address, bonus))
624                 cur.close()
625     else:
```

## NoSQL :

```
625     else:
626         collection = mongo_db.user
627         last_document = collection.find().sort('user_id', -1).limit(1)
628         last_user_id = last_document[0]['user_id']
629         result_1 = collection.find_one({'username': username_signup})
630         if result_1:
631             return 'duplicate user name'
632         else:
633             result_2 = collection.find_one({'email': email})
634             if result_2:
635                 return 'duplicate email'
636             else:
637                 new_user = {
638                     'username': username_signup,
639                     'email': email,
640                     'password_signup': password_signup,
641                     'user_id': last_user_id+1
642                 }
643                 result = collection.insert_one(new_user)
644
645                 if user_type == 'merchant':
646                     website = request.form['website']
647                     new_merchant = {
648                         'merchant_name': username_signup, 'website': website
649                     }
650                     collection = mongo_db.merchant
651                     result = collection.insert_one(new_merchant)
652
653                 elif user_type == 'customer':
654                     phone_number = request.form['phone_number']
655                     delivery_address = request.form['delivery_address']
656                     bonus = 0
657                     new_customer = {
658                         'phone_number': phone_number,
659                         'delivery_address': delivery_address,
660                         'bonus_points': bonus
661                     }
662                     collection = mongo_db.customer
663                     result = collection.insert_one(new_customer)
664
665     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 :

```
691 @app.route('/use_case2', methods=['POST'])
692 def use_case2():
693     global is_migrated
694     id = request.form['id']
695     quantity = request.form['anzahl']
696     current_date = date.today()
697     formatted_date = current_date.strftime("%Y-%m-%d")
698     if is_migrated == 0:
699         cur = db.cursor(buffered=True)
700         cur.execute('SELECT user_id FROM user where username=%s', (username,))
701         user_id = cur.fetchone()[0]
702         cur.execute("select price from item where item_id=%s", (id,))
703         price = cur.fetchall()[0][0]
704         price = float(price)
705         print(type(quantity), type(price))
706
707         cur.execute("INSERT INTO orders (quantity,total_price,delivery_date,user_id) VALUES (%s, %s, %s,%s)",
708                     (quantity, int(quantity)*price, formatted_date, user_id))
709         cur.execute("select max(order_id) from orders")
710         order_id_needed = cur.fetchall()
711         order_id_needed = str(order_id_needed[0][0])
712         cur.execute(
713             "INSERT INTO orderItem (order_id,item_id) VALUES (%s, %s)", (order_id_needed, id))
714         db.commit()
715         totalprices = int(quantity)*price
716     else:
```

## NoSQL :

```
716     else:
717         collection_user = mongo_db.user
718         user = collection_user.find_one({'username': username}, {'user_id': 1})
719         if user:
720             user_id = user['user_id']
721             collection_item = mongo_db.item
722             item_col = collection_item.find_one(
723                 {'item_id': int(id)}, {'description': 1, 'price': 1})
724             if item_col:
725                 item_desc = item_col['description']
726                 item_price = item_col['price']
727                 print(item_desc, item_price, quantity)
728                 totalprices = item_price*int(quantity)
729             collection_order = mongo_db.orders
730             last_document = collection_order.find().sort('order_id', -1).limit(1)
731             last_order_id = last_document[0]['order_id']
732             new_order = {
733                 'order_id': last_order_id+1,
734                 'item_id': id,
735                 'description': item_desc,
736                 'quantity': quantity,
737                 'user_id': int(user_id),
738                 'delivery_date': formatted_date,
739                 'total_price': float(totalprices)
740             }
741             result = collection_order.insert_one(new_order)
742             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 :

```
792 @app.route('/report1', methods=['GET', 'POST'])
793 def report1():
794     if request.method == 'POST':
795         selected_category = request.form.get('category')
796         if selected_category == 'All':
797             if is_migrated == 0:
798                 cursor = db.cursor()
799                 cursor.execute("""SELECT top_items.item_id, item.description, item.category, top_items.count AS item_count
800                                FROM (
801                                    SELECT item_id, COUNT(item_id) AS count
802                                    FROM review
803                                    GROUP BY item_id
804                                    ORDER BY count DESC
805                                    LIMIT 10
806                                ) AS top_items
807                                JOIN item ON top_items.item_id = item.item_id
808                                LIMIT 10;""")
809                 order = cursor.fetchall()
810             else:
811                 review_collection = mongo_db.review
812                 pipeline = [
813                     {'$group': {
814                         '_id': '$item_id',
815                         'count': {'$sum': 1}
816                     }},
817                     {'$lookup': {
818                         'from': 'item',
819                         'localField': '_id',
820                         'foreignField': 'item_id',
821                         'as': 'item'
822                     }},
823                     {'$unwind': '$item'},
824                     {'$project': {
825                         '_id': 0,
826                         'item_id': '$_id',
827                         'description': '$item.description',
828                         'category': '$item.category',
829                         'count': 1
830                     }},
831                     {'$sort': {'count': -1}},
832                     {'$limit': 10}
833                 ]
834                 result = review_collection.aggregate(pipeline)
835                 order = [[item['item_id'], item['description'],
836                           item['category'], item['count']] for item in result]
```

## NoSQL :

```
837     else:
838         if is_migrated == 0:
839             cursor = db.cursor()
840             cursor.execute("""SELECT top_items.item_id, item.description, item.category, top_items.count AS item_count
841                             FROM (
842                                 SELECT item_id, COUNT(item_id) AS count
843                                 FROM review
844                                 GROUP BY item_id
845                                 ORDER BY count DESC
846                             ) AS top_items
847                             JOIN item ON top_items.item_id = item.item_id
848                             WHERE item.category = %s
849                             LIMIT 10;""", (selected_category,))
850             order = cursor.fetchall()
851         else:
852             review_collection = mongo_db.review
853             pipeline = [
854                 {'$group': {
855                     '_id': '$item_id',
856                     'count': {'$sum': 1}
857                 }},
858                 {'$lookup': {
859                     'from': 'item',
860                     'localField': '_id',
861                     'foreignField': 'item_id',
862                     'as': 'item'
863                 }},
864                 {'$unwind': '$item'},
865                 {'$project': {
866                     '_id': 0,
867                     'item_id': '$_id',
868                     'description': '$item.description',
869                     'category': '$item.category',
870                     'count': 1
871                 }},
872                 {'$match': {'category': selected_category}},
873                 {'$sort': {'count': -1}},
874                 {'$limit': 10}
875             ]
876             result = review_collection.aggregate(pipeline)
877             order = [[item['item_id'], item['description'],
878                     item['category'], item['count']] for item in result]
879             return render_template('report1.html', employees=order)
880         else:
881             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 :

```
884 @app.route('/report2', methods=['GET', 'POST'])
885 def report2():
886     global user_id, username
887     if is_migrated == 0:
888         cursor = db.cursor()
889         cursor.execute("""
890             SELECT u.user_id, u.username, MAX(c.bonus_points), COUNT(DISTINCT r.review_id)
891             FROM user u
892             JOIN customer c ON c.user_id = u.user_id
893             JOIN review r ON c.user_id = r.user_id
894             WHERE c.bonus_points > 100
895             GROUP BY u.user_id, u.username
896             ORDER BY MAX(c.bonus_points) DESC
897         """)
898         order = cursor.fetchall()
899     else:
```

## NoSQL :

```
899     else:
900         customer_collection = mongo_db.user
901         pipeline = [
902             {'$lookup': {
903                 'from': 'user',
904                 'localField': 'user_id',
905                 'foreignField': 'user_id',
906                 'as': 'user_info'
907             }},
908             {'$lookup': {
909                 'from': 'customer',
910                 'localField': 'user_id',
911                 'foreignField': 'user_id',
912                 'as': 'customer_info'
913             }},
914             {'$lookup': {
915                 'from': 'review',
916                 'localField': 'user_id',
917                 'foreignField': 'user_id',
918                 'as': 'reviews'
919             }},
920             {'$group': {
921                 '_id': '$user_id',
922                 'review_count': {'$sum': {'$size': '$reviews'}},
923                 'username': {'$first': {'$arrayElemAt': ['$user_info.username', 0]}},
924                 'bonus_points': {'$first': {'$arrayElemAt': ['$customer_info.bonus_points', 0]}}
925             }},
926             {'$match': {'bonus_points': {'$gt': 100}}},
927             {'$sort': {'bonus_points': -1}},
928             {'$project': {
929                 'user_id': '$_id',
930                 'username': 1,
931                 'review_count': 1,
932                 'bonus_points': 1,
933                 '_id': 0
934             }}
935         ]
936         result = customer_collection.aggregate(pipeline)
937         order = [[item['user_id'], item['username'],
938                 item['bonus_points'], item['review_count']] for item in result]
939         return render_template('report2.html', users=order)
940
```

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.

### Additional Actions

Initiate

Migrate

### 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**