ElectroBase Management System

EBMS - *an Online Electronics Retail Store*, built as a course project for
CSE202: Fundamentals of Database Management Systems

Project Report and Documentation

]Divyajeet Singh (2021529) Mehar Khurana (2021541)

April 27, 2023

Deadline 1

Defining Project Scope and Requirements

January 25, 2023

Project Scope

Electronics has always been a booming industry. With the advent of the internet, the industry has seen a massive shift towards online retail. However, it is difficult to keep track of the technical requirements of a store. It is difficult to keep all stakeholders in the loop and keep them involved and updated.

This is where we come in with **EBMS**, i.e. the **ElectroBase Management System**. EBMS is an online retail platform for electronics. It aims to provide a common platform for suppliers, store managers, customers, and delivery agents.

- It is an easy solution for the **customers**, as it aims to provide a diverse catalogue of products to customers. The customers get to choose from a wide range of categories, make changes to their cart, and make secure payments with a method of their choice.
- The **suppliers** get to keep track of their products and change their description, price, etc. They can also keep track of their sales statistics and make changes to their products as and when required.
- The database managers (**admins**) get assisted in monitoring the transactions and managing the inventory. Based on their requirements, they can add deals or combos on the available products or remove categories from their store.
- EBMS provides a platform for **delivery agents** to keep track of all orders that have been assigned to them. They can set their activity/inactivity status and view the feedback given to them.

The primary focus of the project is to design an efficient backend. We aim to create a system that is smooth and easy to use for the customers and easy to manage for the suppliers. The system should support efficient searching through the catalogue and should be able to handle a large number of transactions.

The backend will be built using MySQL, along with Python and Django, and will be hosted on a server. The frontend will be built using HTML, CSS, and ReactJS. By the end of the semester, we plan to host this project on a public server and make it accessible publicly.

TL; DR

The aim of this project is to bring to life an integrated online retail store for electronics. The project will bring all stakeholders on a common platform and will ensure a smooth and easy-to-use experience for the customers.

Technical Requirements

Tech Stack

We plan for EBMS to be a full-stack project with a backend and a frontend. According to the requirements, we plan to use the following tools and technologies:

- **MySQL Database**
- **Python-3**
- **Flask Micro-Framework**
- **HTML**
- **CSS**
- **ReactJS**

Entities, Relations, & Constraints

Entities

The following entities are identified and will be used in the project:

1. **Admin:** Admins are the store managers. They are the stakeholders responsible for managing the inventory and maintaining the store (database).
2. **Customers:** Customers are the primary stakeholders interacting with the system. They must create an account and log in to use the app.
3. **Suppliers:** Suppliers are responsible for supplying products to the store. Different suppliers are allowed to supply the same products to the store.
4. **Delivery Agents:** A delivery agent will be assigned to each order. They will be responsible for delivering orders.
5. **Products:** As the name suggests, these are the products supplied to the store by suppliers. Products are weak entities since several sellers can sell the same product, ie, we can have several records for the same productID.
6. **Orders:** An order entity is used to keep track of the orders placed by different customers. Each order also has one associated delivery agent.
7. **Product Reviews:** Product reviews are the feedback given by customers on products. A customer can only review a product they have previously purchased.
8. **Delivery Agent Reviews:** Delivery agent reviews are the feedback given by customers to delivery agents. A customer can only post a review on a delivery agent they have received an order from.

9. **Wallet:** A wallet is a belonging of a customer that stores attributes like the current balance of the customer, their UPI-ID etc. Users' wallets are hidden from the admin's view. Wallets are weak entities since several people can be using the same UPI-ID, and we require the customerID to uniquely identify a wallet.

10. **Description:** The description weak entity is used to store the description of different products. This is used to avoid redundancy of data storage (in the case where multiple suppliers are selling the same products with the same descriptions).

Relationships & Cardinality Constraints

To effectively manage the database, we will be using the following (non-exhaustive list of) relationships among the data:

1. ***Cart*: Cart** is a relationship between **Customer** and **Product**. One customer can add multiple products to the cart (and choose a specific supplier). The same products may be added to more than one cart. When a customer checks out, the cart is used to generate an order entity, and a delivery agent is assigned to it.

2. ***Sells*: Supplier** *sells* **Product**. This many-to-many relationship is used to keep track of what products one supplier sells. Each supplier can supply multiple products. The same product may be sold by different suppliers.

3. ***Sold*: Supplier** has *sold* **Product**. This many-to-many relationship covers the products sold by a supplier and will be used to generate their sales statistics.

4. ***Delivered*: Delivery Agent** has *delivered* to **Customer**. This is a ternary relationship involving **Delivery Agent Review** as well. There can be, at most, one (editable) review from one customer for a delivery agent.

5. ***Purchased*: Customer** has *purchased* **Product**. This is a ternary relationship involving **Product Review** as well. There can be, at most, one (editable) review from one customer for a product.

6. ***Describes*: Description** *describes* **Product**. This is a one-to-one relationship.

7. ***Belongs To*: Wallet** *belongs to* **Customer**. This is a one-to-one relationship.

8. ***Consists Of*: Order** *consists* **Products**. This is a one-to-many relationship, as one order may consist of multiple products. The product quantities are moved from the cart to the order on checkout.

9. ***Manages*: Admin** *manages* the app. This relationship is used to hide entities like wallets from the admin's view.

Most cardinality constraints have been mentioned above in the description of relationships. Some other constraints are as follows:

1. **Existential:**
   (a) **Product - Supplier:** There can be no product without a supplier.
   (b) **Order - Customer:** An order cannot exist without a customer.
   (c) **Order - Delivery Agent:** An order cannot exist without a delivery agent.
   (d) **Product Review - Customer *has purchased* Product:** To ensure that a customer can review only the products they have purchased.
   (e) **Delivery Agent Review - Delivery Agent *delivered to* Customer:** to ensure that a customer can review only the delivery agents they have received orders from.
2. **One-to-One: Delivery Agent - Order:** One delivery agent can be assigned at most one order at a time. We may remove this constraint later and implement an algorithm to find the best agent to assign an order to, based on the current delivery addresses, with a cap on the number of orders per agent.
3. **One-to-Many: Customer - Phone Number** One customer can have multiple phone numbers. It will be implemented as a multi-valued attribute of Customer.

Access Constraints

Since some data must remain private while other data must be accessible to all, the following access-control constraints will be implemented:

1. **Admin:** All data except for Customer passwords and Wallets
2. **Customer:** Personal records, past and current Orders, and all Products and Reviews
3. **Supplier:** Their Product catalogue, personal records, sales statistics, and Customer Reviews
4. **Delivery Agent:** Personal records, current Orders, and Reviews from past Orders

Functional Requirements

All stakeholders except the **Admins** will need to create an account and log in. The following (non-exhaustive list of) features are some of the functional requirements of the project:

1. As a **Customer:**

- Add balance to wallet
- Browse and search/sort/filter for products
- Manage (add/remove) items in their cart
- Place an order (checkout cart)
- Confirm/Authenticate transaction
- View and search for previous orders

2. As an **Admin:**
    - Add/Delete categories
    - Delete products
    - Add/Modify discounts
    - Create deals/combos
    - View transaction history
    - Appoint other Admins
3. As a **Supplier:**
    - View products currently on sale
    - Add/Discontinue a product
    - Change price of a product
    - Change quantity of a product
    - View sales statistics
4. As a **Delivery Agent:**
    - Confirm a delivery
    - View the address of the order
    - View the ETA of the order
    - View reviews
    - Choose current activity status

Deadline 2

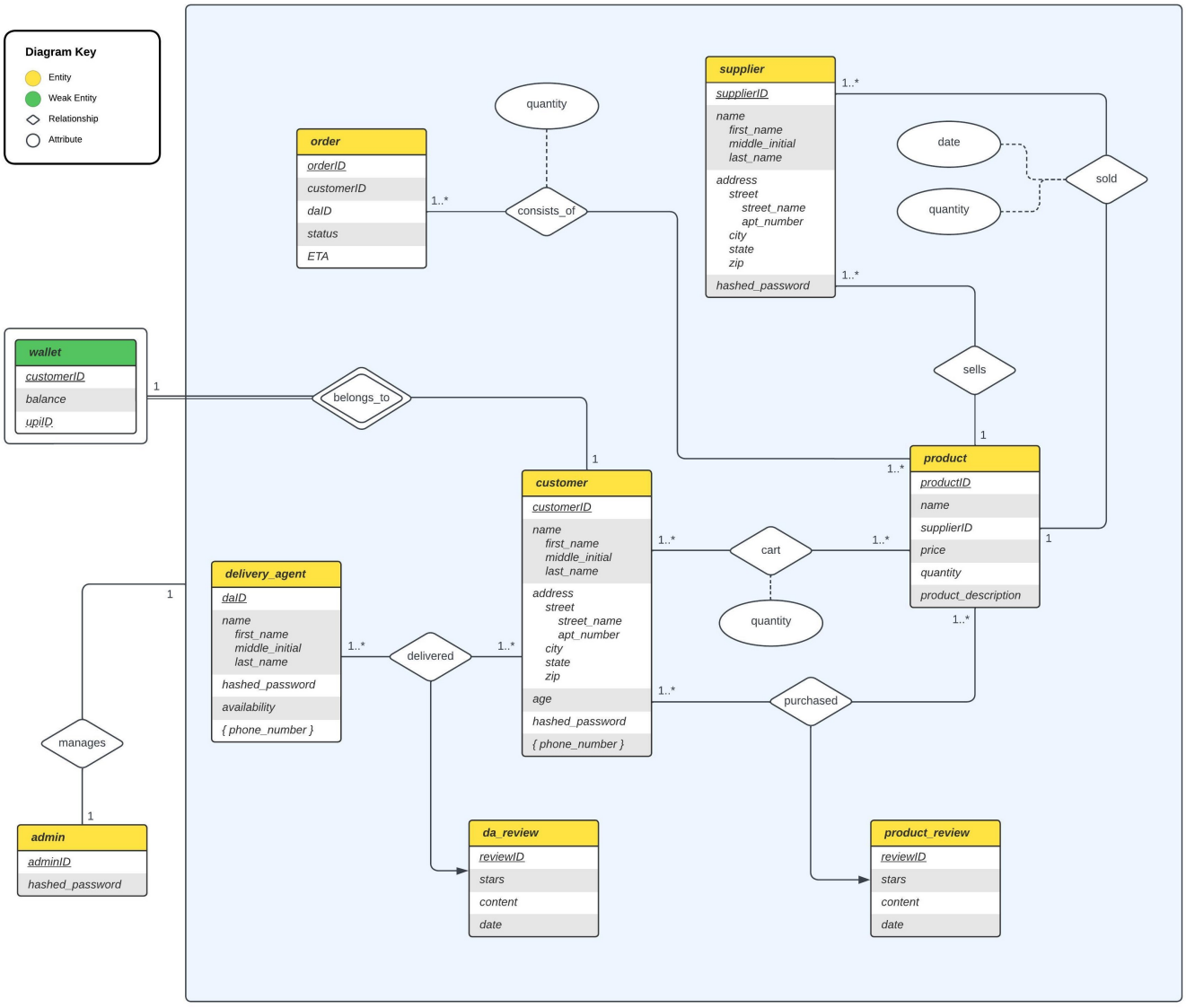
Designing the ER Model and converting it to a Relational Model

February 3, 2023

Entity-Relationship Model

Entity-Relationship (ER) Models are used to plan how different entities in a project interact with each other.

Our ER Model captures the nature of the relationships and entities planned to be used in the project. The ER Model is designed in accordance with the assumptions and constraints as mentioned in the document above. Hence, we plan to build our system on the basis of the following Entity-Relationship Model:



Ternary Relationships

The following ternary relationships have been identified:

1. **Customer - Product - Product Review:** A customer can review multiple products, and a product can be reviewed by multiple customers. A customer can give at most one review per product. This ternary relationship will be decomposed into the following binary relationships at the time of implentation:

   (a) **Customer - Product:** (Many-to-Many) To keep track of which customers have purchased which products.

   (b) **Product - Product Review:** (One-to-Many) To keep track of all reviews given to a product.

   (c) **Customer - Product Review:** (One-to-Many) To keep track of all reviews given by a customer. This relationship may not be needed and may be removed in the future.

2. **Delivery Agent - Customer - Delivery Agent Review:** A customer can review multiple delivery agents, and a delivery agent can be reviewed by multiple customers. A customer can give at most one review per delivery agent. This ternary relationship will be decomposed into the following binary relationships at the time of implentation:

   (a) **Customer - Delivery Agent:** (Many-to-Many) To keep track of which customers have received orders from which delivery agents.

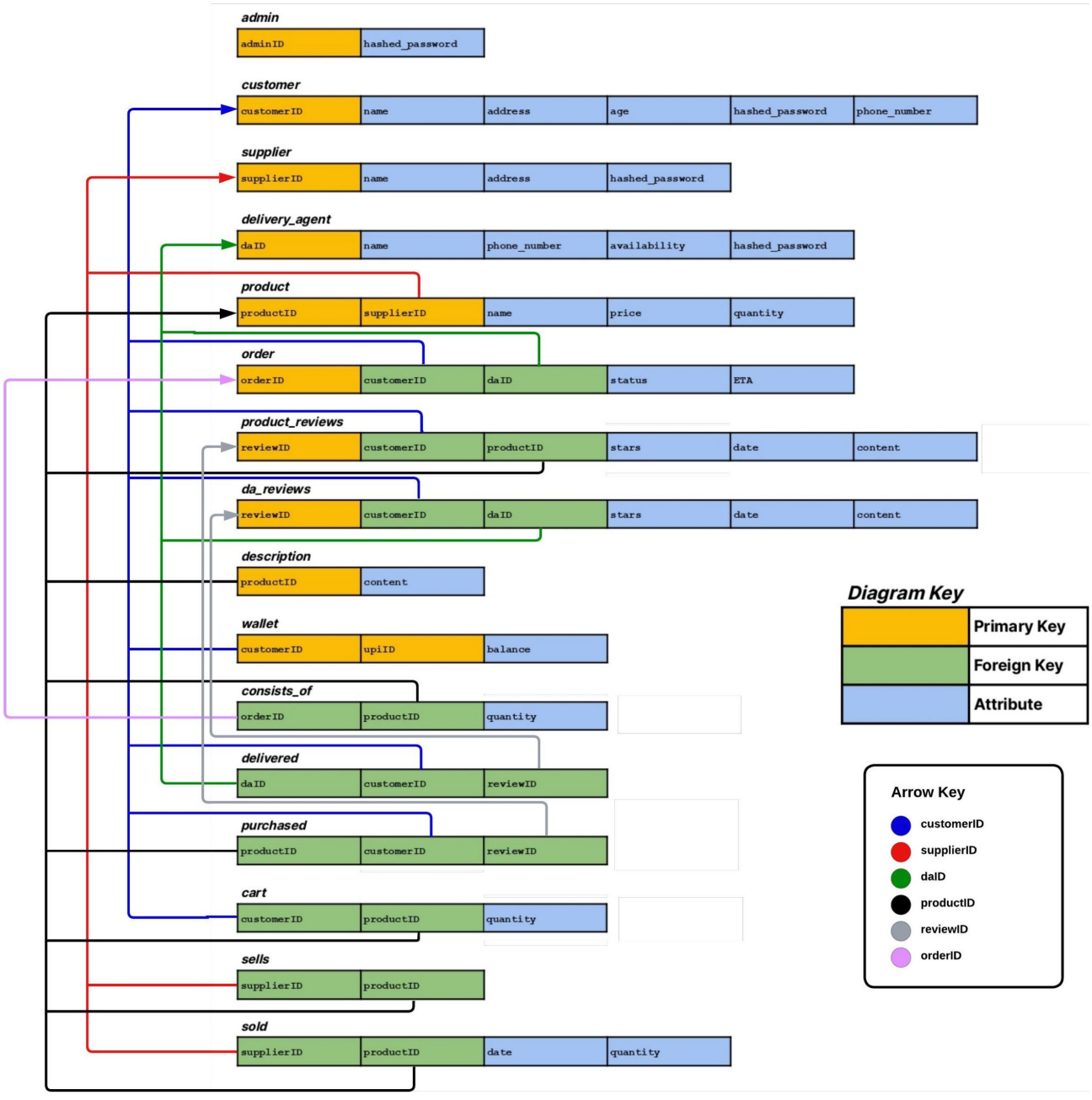
   (b) **Delivery Agent - Delivery Agent Review:** (One-to-Many) To keep track of all reviews given to a delivery agent.

   (c) **Customer - Delivery Agent Review:** (One-to-Many) To keep track of all reviews given by a customer. This relationship may not be needed and may be removed in the future.

Relational Model

Relationship Models are used to represent how data will be stored in the database, along with the attributes of each entity and relationship. The Relational Model is designed in accordance with the assumptions and constraints as mentioned in the document above.

Note: The arrows represent that a field is *derived* from another. For example, `productID` in `description` will contain values of `productID` from table `product`.


admin
adminID
hashed_password
customer
customerID
name
address
age
hashed_password
phone_number
supplier
supplierID
name
address
hashed_password
delivery_agent
daID
name
phone_number
availability
hashed_password
product
productID
supplierID
name
price
quantity
order
orderID
customerID
daID
status
ETA
product_reviews
reviewID
customerID
productID
stars
date
content
da_reviews
reviewID
customerID
daID
stars
date
content
description
productID
content
wallet
customerID
upiID
balance
consists_of
orderID
productID
quantity
delivered
daID
customerID
reviewID
purchased
productID
customerID
reviewID
cart
customerID
productID
quantity
sells
supplierID
productID
sold
supplierID
productID
date
quantity
Diagram Key
Primary Key
Foreign Key
Attribute
Arrow Key
customerID
supplierID
daID
productID
reviewID
orderID

Deadline 3

Implementing the Database Schema and Integrity Constraints, and Populating simulated data satisfying them

February 10, 2023

Iterations on the Database Schema

Tables added or renamed:

1. Since **address** is a composite attribute, we store it in a separate table since we would rarely need to search for the address. The search/join/union query efficiency would get affected if we store all the address data in the same stakeholder table.

2. The **consists_of** relationship table has been renamed to **order_product** to make it more intuitive.

3. Entity table **description** has been renamed to **product_description**, since `description` is a reserved keyword in MySQL. Its field **description** has also been renamed to **content** for the same reason.

4. Entity table **order** has also been renamed to **orders** for the same reason.

Columns/Fields added, removed, or renamed:

1. All fields called **hashed_password** have been renamed to **pwd** for simplicity.

2. Added field **email** to **customer**, **supplier**, and **delivery_agent** tables.

3. Added field **quantity** to table **product**.

4. Removed fields **reviewID** from the **product_review** and **da_review** tables, so as to decompose the ternary relationships **delivered** and **purchased**.

5. Added **order_date** and **delivery_date** to the **order** table.

Handling Different Attributes

Composite Attributes:

1. Addresses of all stakeholders are stored in a single table with a unique **addressID** assigned to each address. This **addressID** is then stored along with other stakeholder data in their tables.

2. Added sub-attribute **country** to the **address** composite attribute.

3. The **name** composite attribute is stored in the same table, since we might need to search for the names of the stakeholders. Each sub-attribute is kept as a column (**first_name**, **middle_initial**, **last_name**) in the table.

Multi-Valued Attributes:

1. Since **phone_number** is a multivalued attribute, we store it in a separate table, with the **phoneID** attribute from this table being stored in the **delivery_agent** and **customer** tables. This table does not have a primary key since each **phoneID** associates to one or more **num**'s (phone numbers).

Derived Attributes:

1. Changed **ETA** and **delivered** (earlier **status**) to derived attributes. **ETA** will be calculated as **order_date** + 15 days when required.

2. The attribute **delivered** will be |True— if **delivery_date** is not null, else |False—.

Since these constraints/relations will be implemented outside the database, and hence these fields are not present in the tables.

Assumptions:

1. The field **delivery_date** of **orders** is allowed to be `NULL`. This is because we do not know the **delivery_date** when the order is placed, and it is only updated when the order is delivered. We derive the attribute **delivered** from **delivery_date** to check if the order has been delivered or not.

2. Since **phone numbers** are implemented as multi-valued attributes, we assume that phone numbers are not unique to a customer. This means that multiple users may have the same phone number.

3. All primary keys are defined with the `AUTO_INCREMENT` constraints so that we do not need to insert ID values ourselves, and duplicity errors on primary keys are avoided.

4. **Product descriptions** and **review contents** are being stored using `TEXT` datatype in MySQL, which is not stored in the server memory and does not hamper query times.

5. All attributes that can not have null values have been specified as `NOT NULL`. Attributes like **last_name**, **middle_initial**, **content** in the **review** tables, and **delivery_date** in the **orders** table can have null values.

6. **Availability** of a delivery agent has been given a `DEFAULT` value of true, ie, when a delivery agent is added to the database, they are available to deliver an order by default.

7. **Balance** in a customer wallet, on account creation, has been given a `DEFAULT` value of 0.

8. **pwd**'s in all tables are stored as SHA1 hashes.

9. Some other integrity constrains have also been added, for example, **rating** must be between 0 and 5, and **quantity** and **price** must be positive.

10. Based on a rough idea of the types of queries we plan to use, some indices have also been added on the fields of the tables. These may be updated in the future.

Data Generation & Population

Data Generation

Most of the simulated data for the stakeholder and main entity tables was generated using https://www.mockaroo.com.

We utilised the (Ruby) code functionality to implement viable constraints on the data while data generation. The data for each table was downloaded as a CSV-file. We then used python scripts to generate the MySQL insertion queries and to populate relations. Data for some tables (like **cart** and **orders**) was mainly done through python scripts so as to make sure the existential constraints were not violated.

Data Population

All the tables of the database were pre-populated with data with integrity-constrains maintained to start querying. The database was populated with the following number of rows of data:

- `address`: 400 rows
- `phone_number`: 400 rows
- `admin`: 2 rows
- `supplier`: 200 rows
- `customer`: 200 rows
- `delivery_agent`: 200 rows
- `product`: 200 rows
- `orders`: 1000 rows
- `wallet`: 200 rows
- `product_review`: 200 rows
- `da_review`: 200 rows
- `cart`: 1073 rows
- `order_product`: 5552 rows

Deadline 4

SQL Queries and Relational Algebraic Operations

February 17, 2023

SQL Queries

We attempted to implement the most relevant queries for the application using the database. These queries are utilized by different stakeholders to perform their tasks. The following is a list of SQL Queries and their use cases:

1. The following queries are used while placing an order.

```sql
-- Find out an available delivery agent
SELECT daID FROM delivery_agent
WHERE availability = 1 ORDER BY daID ASC LIMIT 1;

-- Add the order to the table orders and assign the order to the selected
    delivery agent
INSERT INTO orders (customerID, daID, order_date)
VALUES (50, (SELECT daID FROM delivery_agent
WHERE availability = 1 ORDER BY daID ASC LIMIT 1), '2021-04-01');

-- Add the ordered products in the table order_product
INSERT INTO order_product (orderID, productID, quantity)
SELECT MAX(o.orderID), c.productID, c.quantity
FROM orders o INNER JOIN cart c ON o.customerID = c.customerID
WHERE o.customerID = 50
GROUP BY c.productID;

-- Reduce the quantity of products ordered from table product
UPDATE product p INNER JOIN cart c ON p.productID = c.productID
SET p.quantity = p.quantity - c.quantity
WHERE c.customerID = 50;

-- Delete the products from the cart
DELETE FROM cart WHERE customerID = 50;

-- Update the wallet balance of the customer
UPDATE wallet w
INNER JOIN orders o ON w.customerID = o.customerID
SET w.balance = w.balance - (
    SELECT SUM(p.price * op.quantity)
    FROM product p, order_product op
    WHERE p.productID = op.productID AND op.orderID = o.orderID
);

-- Update the availability of the delivery agent
UPDATE delivery_agent
SET availability = 0
WHERE daID = (
    SELECT daID FROM orders WHERE customerID = 50 AND orderID = (
        SELECT MAX(orderID) FROM orders WHERE customerID = 50
    )
);
```

2. This query is used to get a list of all suppliers who have all their products with average rating above 3.

```
SELECT
    s.supplierID,
    CONCAT(s.first_name, ' ', s.middle_initial, ' ', s.last_name) AS name
FROM supplier s
WHERE (
    SELECT AVG(pr.rating) FROM product_review pr, product p
    WHERE p.productID = pr.productID AND p.supplierID = s.supplierID
    GROUP BY p.supplierID
) > 3;
```

3. This query lists the suppliers who do not sell any products.

```
SELECT
    s.supplierID,
    CONCAT(s.first_name, ' ', s.middle_initial, ' ', s.last_name) AS name
FROM supplier s
WHERE NOT EXISTS (
    SELECT * FROM product p WHERE p.supplierID = s.supplierID
);
```

4. This query displays all the products that have average rating above or equal to 3.5 ordered by average rating.

```
SELECT p.productID, p.name, AVG(pr.rating) AS avg_rating
FROM product_review pr, product p
WHERE p.productID = pr.productID
GROUP BY p.productID
HAVING avg_rating >= 3.5
ORDER BY avg_rating DESC;
```

5. This query is used to display the top 40 delivery agents with the highest average rating.

```
SELECT
    da.daID,
    CONCAT(da.first_name, ' ', da.middle_initial, ' ', da.last_name) AS name,
    da.email,
    AVG(dr.rating) AS avg_rating
FROM
    da_review dr,
    delivery_agent da
WHERE da.daID = dr.daID
GROUP BY da.daID
ORDER BY avg_rating DESC
LIMIT 40;
```

6. This query shows the undelivered orders of a delivery agent (where delivery date is NULL).

```
SELECT
    orderID,
    customerID,
    daID,
    order_date,
    DATE_FORMAT(ADDDATE(order_date, INTERVAL 15 DAY), '%Y-%m-%d') AS ETA
FROM orders
WHERE daID = 1 AND delivery_date IS NULL;
```

7. This query is used to properly display the entire record of a customer.

```
SELECT
    customerID,
    CONCAT(first_name, ' ', middle_initial, ' ', last_name) AS name,
    CONCAT(
        apt_number, ', ', street_name, ', ', city, ', ',
        state, ' - ', zip, ', ', country
    ) AS customer_address,
    age,
    ph.num as primary_phone_number,
    email
FROM
    customer,
    address,
    (
        SELECT num FROM phone_number, customer
        WHERE phone_number.phoneID = customer.phoneID
        AND customerID = 50 LIMIT 1
    ) AS ph
WHERE customer.addressID = address.addressID AND customer.customerID = 50;
```

8. This query shows the top 15 customers who have spent the most money on their orders.

```
SELECT
    customer.customerID,
    COUNT(orders.orderID) AS total_orders,
    SUM(product.price * order_product.quantity) AS total_spent
FROM customer
INNER JOIN orders ON customer.customerID = orders.customerID
INNER JOIN order_product ON orders.orderID = order_product.orderID
INNER JOIN product ON order_product.productID = product.productID
GROUP BY customer.customerID
ORDER BY total_spent DESC
LIMIT 15;
```

9. This query is used to view the order history of a customer.

```
SELECT
    orders.orderID,
    orders.order_date, orders.delivery_date,
    product.name, order_product.quantity, product.price,
    (order_product.quantity * product.price) AS total_price,
    CASE
        WHEN (orders.delivery_date IS NULL) = True THEN 'Not Delivered'
        ELSE 'Delivered'
    END AS delivered
FROM orders
INNER JOIN order_product ON orders.orderID = order_product.orderID
INNER JOIN product ON order_product.productID = product.productID
WHERE orders.customerID = 1
ORDER BY orders.order_date;
```

10. This query is used to find out the total price of an order.

```
SELECT orderID, SUM(product.price * order_product.quantity) AS order_price
FROM order_product
INNER JOIN product ON order_product.productID = product.productID
WHERE order_product.orderID = 1;
```

11. This query is used to show all underlivered orders with their ETA for a customer.

```
SELECT
    orderID,
    order_date,
    DATE_FORMAT(ADDDATE(order_date, INTERVAL 15 DAY), "%Y-%m-%d") AS ETA
FROM orders
INNER JOIN customer ON orders.customerID = customer.customerID
WHERE customer.customerID = 1 AND orders.delivery_date IS NULL
ORDER BY order_date;
```

12. This query find out the total revenue and total quantity sold per product for a supplier (sales statistics).

```
SELECT
    product.name,
    SUM(order_product.quantity) AS total_quantity_sold,
    SUM(order_product.quantity * product.price) AS total_revenue
FROM product
INNER JOIN order_product ON product.productID = order_product.productID
INNER JOIN orders ON order_product.orderID = orders.orderID
WHERE product.supplierID = 1
GROUP BY product.name;
```

13. This query is used to search through the product catalogue for names of the products using pattern matching.

```
SELECT name, AVG(rating)
FROM product
JOIN product_review ON product.productID = product_review.productID
WHERE name LIKE 'LED %'
GROUP BY name;
```

14. This query is used to add a product to a customer's cart.

```
INSERT INTO cart (customerID, productID, quantity) VALUES (50, 1, 1);
```

15. This query adds a new phone number into the table for a customer.

```
INSERT INTO phone_number
VALUES ((SELECT phoneID FROM customer WHERE customerID = 50), '1234567890');
```

16. This query adds more quantity of products for an existing product.

```
UPDATE product SET quantity = quantity + 100 WHERE productID = 1;
```

17. This query updates the address of a customer.

```
UPDATE address SET
    apt_number = '100',
    street_name = 'Thomspon St.',
    city = 'Albany',
    state = 'New York',
    zip = '12207',
    country = 'United States'
WHERE addressID = (SELECT addressID FROM customer WHERE customerID = 1);
```

18. This query deletes a product from a customer's cart.

```
DELETE FROM cart WHERE customerID = 99 AND productID = 14;
```

Deadline 5

Embedded SQL Queries, OLAP Queries, and Triggers

March 26, 2023

Frontend Development

Since this deadline now requires embedding SQL queries in a host programming language, the frontend development has also been started. As decided previously, the host langauge is Python-3, and the frontend is being developed using the Flask Web-Development Micro Framework.
The raw frontend is in the first stages of development and is hosted on a local server. This document contains references and bibliographies for all open-source code available on the internet for tasks like placing sortable tables on a page, CSS animations, and JavaScript functions for form validation.

Embedded SQL Queries

As of now, the only SQL queries which have been embedded in the Python code are those for which the frontend interaces have been developed. In the future, these will be refactored into small `.sql` files and imported into the Python code for security[1] and readability purposes.

The following is a (non-exhaustive) list of already embedded SQL queries:

1. **Feature:** This query is used to display the entire product catalogue.
   Location: In function `search()` in `front-end/__init__.py`

```
SELECT p.productID, p.name, p.price, AVG(pr.rating) AS avg_rating, p.quantity
FROM product p
LEFT JOIN product_review pr ON p.productID = pr.productID
GROUP BY p.productID
ORDER BY p.name ASC;
```

   A variation[2] of the same query (which uses pattern matching in SQL) is also used to search through the product catalogue.

```
SELECT p.productID, p.name, p.price, AVG(pr.rating) AS avg_rating, p.quantity
FROM product p
LEFT JOIN product_review pr ON p.productID = pr.productID
WHERE p.name LIKE '%{search}%'
GROUP BY p.productID
ORDER BY p.name ASC;
```

[1] The embedded SQL Queries are currently not protected against SQL Injection Attacks.
[2] In this variant, `search` is the key entered by the user for searching. It is substituted into the query using Python f-strings.

2. **Feature:** This query is used to get the login credentials of a customer.
   Location: In function `login()` in `front-end/__init__.py`

```
SELECT customerID, first_name, email, pwd FROM customer WHERE email = %s;
```

   A variation[3] of the same query is also used for supplier login.

```
SELECT supplierID, first_name, email, pwd FROM supplier WHERE email = %s;
```

   Another variant will also be added for delivery agent login. This variant can also be found in the function `login()` in `front-end/__init__.py`.

3. **Feature:** These queries are used to get an overview of statistics for the admin.
   Location: In function `admin()` in `front-end/__init__.py`

```
SELECT COUNT(*) AS customer_count FROM customer;
SELECT COUNT(*) AS supplier_count FROM supplier;
SELECT COUNT(*) AS da_count FROM delivery_agent;
SELECT COUNT(*) AS order_count FROM orders;
SELECT COUNT(*) AS product_count FROM order_product;
```

4. **Feature:** This query is used to display the top-10 highest spending customers.
   Location: In function `search()` in `front-end/__init__.py`

```
SELECT
    customer.customerID,
    CONCAT(customer.first_name, ' ', customer.last_name) AS name,
    COUNT(orders.orderID) AS total_orders,
    SUM(product.price * order_product.quantity) AS total_spent,
    AVG(product.price * order_product.quantity) AS avg_spent
FROM customer
INNER JOIN orders ON customer.customerID = orders.customerID
INNER JOIN order_product ON orders.orderID = order_product.orderID
INNER JOIN product ON order_product.productID = product.productID
GROUP BY customer.customerID
ORDER BY total_spent DESC
LIMIT 10;
```

5. **Feature:** This query is used to get the number of orders by status.
   Location: In function `search()` in `front-end/__init__.py`

```
SELECT (delivery_date IS NULL) as status, COUNT(*) AS n
FROM orders GROUP BY status;
```

[3] `%s` denotes a placeholder in Python, which will be substituted by the value entered by the user in the login form.

6. **Feature:** This query is used to display the customers having the least orders.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT
    customer.customerID,
    CONCAT(customer.first_name, ' ', customer.last_name) AS name,
    COUNT(orders.orderID) AS total_orders,
    SUM(product.price * order_product.quantity) AS total_spent
FROM customer
INNER JOIN orders ON customer.customerID = orders.customerID
INNER JOIN order_product ON orders.orderID = order_product.orderID
INNER JOIN product ON order_product.productID = product.productID
GROUP BY customer.customerID
HAVING total_orders <= 10
ORDER BY total_orders ASC
LIMIT 10;
```

7. **Feature:** This query is used to display the top-10 best selling products.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT p.productID, p.name, p.price, SUM(op.quantity) AS units_sold
FROM order_product op, product p
WHERE p.productID = op.productID
GROUP BY p.productID
ORDER BY units_sold DESC
LIMIT 10;
```

8. **Feature:** This query is used to display the top-10 highest rated products.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT p.productID, p.name, p.price, AVG(pr.rating) AS avg_rating
FROM product_review pr, product p
WHERE p.productID = pr.productID
GROUP BY p.productID
ORDER BY avg_rating DESC
LIMIT 10;
```

9. **Feature:** This query is used to display inactive suppliers.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT s.supplierID, CONCAT(s.first_name, ' ', s.last_name) AS name, email
FROM supplier s
WHERE NOT EXISTS (
    SELECT * FROM product p WHERE p.supplierID = s.supplierID
)
LIMIT 10;
```

10. **Feature:** This query is used to display the top-10 highest rated suppliers.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT
    s.supplierID,
    CONCAT(s.first_name, ' ', s.last_name) AS name,
    email,
    AVG(pr.rating) AS avg_rating
FROM supplier s, product_review pr, product prod
WHERE (
    SELECT AVG(pr.rating)
    FROM product_review pr, product p
    WHERE p.productID = pr.productID AND p.supplierID = s.supplierID
    GROUP BY p.supplierID
) > 3
AND s.supplierID = prod.supplierID AND pr.productID = prod.productID
GROUP BY s.supplierID
ORDER BY avg_rating DESC
LIMIT 10;
```

11. **Feature:** This query is used to display the top-10 highest rated delivery agents.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT
    da.daID,
    CONCAT(da.first_name, ' ', da.last_name) AS name,
    email,
    AVG(dr.rating) AS avg_rating
FROM da_review dr, delivery_agent da
WHERE da.daID = dr.daID
GROUP BY da.daID
ORDER BY avg_rating DESC
LIMIT 10;
```

12. **Feature:** This query is used to display the top-10 most active delivery agents.
Location: In function `search()` in `front-end/__init__.py`

```
SELECT
    da.daID,
    CONCAT(da.first_name, ' ', da.last_name) AS name,
    email,
    COUNT(o.orderID) AS total_orders
FROM delivery_agent da
LEFT JOIN orders o ON da.daID = o.daID
GROUP BY da.daID
ORDER BY total_orders DESC
LIMIT 10;
```

OLAP SQL Queries

OLAP queries are (mainly) utilised by the Admin in our use case to perform data analytics. The OLAP data is separated from the rolled-up form before rendering the HTML page. These queries are also embedded in Python.

The following is a (non-exhaustive) list of already implemented OLAP SQL queries:

1. **Feature:** This query is used to find out the trends in the number of orders on a monthly, quarterly, and yearly basis.
   Location: In function `admin_stats(page)` in `front-end/__init__.py`

   Usage: Non-empty month, quarter, and year values indicate monthly, querterly, and yearly trends in order. An overall total is also provided by the roll-up query.

```
SELECT
    YEAR(order_date) AS date_year,
    QUARTER(order_date) AS date_quarter,
    MONTH(order_date) AS date_month,
    COUNT(orders.orderID) AS order_count,
    SUM(price * order_product.quantity) AS revenue
FROM
    orders
    JOIN order_product ON orders.orderID = order_product.orderID
    JOIN product ON order_product.productID = product.productID
GROUP BY date_year, date_quarter, date_month WITH ROLLUP
ORDER BY date_year DESC, date_month DESC;
```

2. **Feature:** This query is used to find out the country-wise trends in the number of orders and revenue.
   Location: In function `admin_stats(page)` in `front-end/__init__.py`

   Usage: The query can be extended to rollup with states, cities, etc. by adding them to the `SELECT` and `GROUP BY` clause.

```
SELECT
    a.country,
    COUNT(DISTINCT o.orderID) AS order_count,
    SUM(op.quantity * p.price) AS revenue
FROM orders o
JOIN order_product op ON o.orderID = op.orderID
JOIN product p ON op.productID = p.productID
JOIN customer c ON o.customerID = c.customerID
JOIN address a ON c.addressID = a.addressID
GROUP BY a.country WITH ROLLUP
ORDER BY revenue DESC;
```

3. **Feature:** This query is used to retrieve the demographics of customer activity.
   Location: In function `admin_stats(page)` in `front-end/__init__.py`

   Usage: The OLAP data contains the number of customers that are registered on EBMS grouped by demographics. Empty values in the field `state` indicate that the data is an aggregate of country-wise data. The data record where country is empty gives the global aggregated data. The average spending per order along with total spending is also shown.

```
SELECT
    country, state,
    COUNT(DISTINCT customer.customerID) AS customer_count,
    AVG(product.price * order_product.quantity) AS avg_spent,
    SUM(product.price * order_product.quantity) AS total_spent
FROM customer
JOIN orders ON customer.customerID = orders.customerID
JOIN order_product ON orders.orderID = order_product.orderID
JOIN product ON order_product.productID = product.productID
JOIN address ON customer.addressID = address.addressID
GROUP BY country, state WITH ROLLUP
ORDER BY country ASC, total_spent DESC;
```

4. **Feature:** This query is used to retrieve the demographics of supplier activity.
   Location: In function `admin_stats(page)` in `front-end/__init__.py`

   Usage: The OLAP data contains the number of suppliers that are selling products on EBMS grouped by demographics. The average earning per order along with total earning is also shown. This query is analogous to the query retrieving customer demographics.

```
SELECT
    country, state,
    COUNT(DISTINCT supplier.supplierID) AS supplier_count,
    AVG(product.price * order_product.quantity) AS avg_earned,
    SUM(product.price * order_product.quantity) AS total_earned
FROM supplier
JOIN orders ON supplier.supplierID IN (
    SELECT supplierID FROM product WHERE productID IN (
        SELECT productID FROM order_product WHERE orderID = orders.orderID
    )
)
JOIN order_product ON orders.orderID = order_product.orderID
JOIN product ON order_product.productID = product.productID
JOIN address ON supplier.addressID = address.addressID
GROUP BY country, state WITH ROLLUP
ORDER BY country ASC, total_earned DESC;
```

5. **Feature:** This query is used to find out a supplier's statistics by date and region.

Location: Not been added yet, as it is a supplier-specific query and supplier-pages are not yet implemented.

Usage: A supplier's sales grouped by a particular month/year and country are returned by this query. We can separate the rolled up data into monthly, yearly, and country-wise trends. [4]

```
SELECT
    YEAR(order_date) AS year,
    MONTH(order_date) AS month,
    country,
    COUNT(DISTINCT o.orderID) AS total_quantity,
    SUM(op.quantity * p.price) AS total_sales
FROM orders o
JOIN order_product op ON o.orderID = op.orderID
JOIN product p ON op.productID = p.productID
JOIN customer c ON o.customerID = c.customerID
JOIN address a ON c.addressID = a.addressID
JOIN supplier s ON p.supplierID = s.supplierID
WHERE s.supplierID = {s_id}
GROUP BY year, month, a.country WITH ROLLUP;
```

Triggers

The following triggers are identified according to the use cases and implemented in the database:

1. **Trigger Name:** `create_wallet`
   This trigger is used to create a wallet for a customer when the record for the customer is created.

```
DROP TRIGGER IF EXISTS create_wallet;
DELIMITER $$
CREATE TRIGGER create_wallet AFTER INSERT ON customer
FOR EACH ROW
BEGIN
    IF NOT EXISTS (SELECT * FROM wallet WHERE customerID = NEW.customerID) THEN
        INSERT INTO wallet (customerID, balance) VALUES (NEW.customerID, 0);
    END IF;
END;
$$
DELIMITER ;
```

[4] Here, `s_id` is the supplier ID of the currently logged in supplier. The string is substituted into the query using an `f-string` in Python.

2. **Trigger Name:** `da_unavailable`
   This trigger is used to toggle a delivery agent's availability to False when they are assigned to an order

```
DROP TRIGGER IF EXISTS da_unavailable;
DELIMITER $$
CREATE TRIGGER da_unavailable AFTER INSERT ON orders
FOR EACH ROW
BEGIN
    UPDATE delivery_agent SET avalability = FALSE WHERE daID = NEW.daID;
END;
$$
DELIMITER ;
```

3. **Trigger Name:** `da_unavailable`
   This trigger is used to toggle a delivery agent's availability to to True when they complete an order

```
DROP TRIGGER IF EXISTS da_available;
DELIMITER $$
CREATE TRIGGER da_available AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    IF NOT EXISTS (
        SELECT * FROM orders WHERE daID = NEW.daID AND delivery_date IS NULL
    ) THEN
        UPDATE delivery_agent SET avalability = TRUE WHERE daID = NEW.daID;
    END IF;
END;
$$
DELIMITER ;
```

Deadline 6

Conflicting and Non-Conflicting Database Transactions

April 24, 2023

Concurrent Database Transactions

Assuming the application is hosted on a public server, multiple users can access and use it simultaneously. Hence, there may arise cases where multiple users access the database (through transactions) in a way that creates conflicts and inconsistency.

Transaction Pair 1

Consider the case of two customers buying the same product simultaneously. The following table of transactions sums up the reads and writes to the database needed to complete the required actions.
Here, Q represents the quantity (in stock) of Product 1. Similarly, B_1 and B_2 represent the balance of customers 1 and 2, whereas their orders are represented by O_1 and O_2, respectively.

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(product 1 quantity)	READ(product 1 quantity)
$Q := Q - q_1$	$Q := Q - q_2$
WRITE(product 1 quantity)	WRITE(product 1 quantity)
READ(balance user 1)	READ(balance user 2)
WRITE(new balance user 1)	WRITE(new balance user 2)
COMMIT	COMMIT

Conflict-Serializable Schedule

Transaction-1 (T_1)	**Transaction-2** (T_2)
	READ(Q)
	$Q := Q - q_2$
	WRITE(Q)
	READ(B_2)
	WRITE(B_2')
READ(Q)	
$Q := Q - q_1$	
WRITE(Q)	
	WRITE(ORDER O_2)
READ(B_1)	
CHECK(B_1)	
WRITE(B_1')	
WRITE(ORDER O_1)	

First, the quantity bought by B_2 is deducted from the stock quantity Q. B_2's balance is also updated. Then we switch to T_1 and deduct the quantity bought by B_1 from Q. Now, we switch back to T_2 and write the order details. Finally, we switch back to T_1 and execute the remaining commands and commit the transactions.

This schedule is conflict serializable since we can repeatedly switch the last command in T_2 up, such that both the transactions get executed serially (T_1 after T_2).

Conflict-Serializable Schedule with Locks

Transaction-1 (T_1)	**Transaction-2** (T_2)
	LOCK-X(Q)
	READ(Q)
	$Q := Q - q_2$
	WRITE(Q)
	UNLOCK(Q)
	LOCK-X(B_2)
	READ(B_2)
	WRITE(B_2')
	UNLOCK(B_2)
LOCK-X(Q)	
READ(Q)	
$Q := Q - q_1$	
WRITE(Q)	
UNLOCK(Q)	
	LOCK-X(ORDER O_2)
	WRITE(ORDER O_2)
	UNLOCK(ORDER O_2)
LOCK-X(B_1)	
READ(B_1)	
CHECK(B_1)	
WRITE(B_1')	
UNLOCK(B_1)	
LOCK-X(ORDER O_1)	
WRITE(ORDER O_1)	
UNLOCK(ORDER O_1)	

We add locks before each read, and unlock after each write. This ensures that no two transactions are accessing/altering the same data point at the same time, which could lead to conflicts.

Non-Conflict-Serializable Schedule

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(Q)	
	READ(Q)
$Q := Q - q_2$	
WRITE(Q)	
	$Q := Q - q_2$
	WRITE(Q)
	READ(B_2)
	WRITE(B_2')
	WRITE(ORDER O_2)
READ(B_1)	
CHECK(B_1)	
WRITE(B_1')	
WRITE(ORDER O_1)	

The above schedule is non-conflict-serializable as there does not exist any sequence of non-conflicting switches to make the schedule serial. This is because T_1 writes to Q before and after a read and write, respectively, which are executed by T_2.

Non-Conflict-Serializable Schedule with Locks

Transaction-1 (T_1)	**Transaction-2** (T_2)
LOCK-S(Q)	
READ(Q)	
UNLOCK(Q)	
	LOCK-S(Q)
	READ(Q)
	UNLOCK(Q)
LOCK-X(Q)	
$Q := Q - q_2$	
WRITE(Q)	
UNLOCK(Q)	
	LOCK-X(Q)
	$Q := Q - q_2$
	WRITE(Q)
	UNLOCK(Q)
	LOCK-X(B_2)
	READ(B_2)
	CHECK(B_2)
	WRITE(B_2')
	UNLOCK(B_2)
	LOCK-X(ORDER O_2)
	WRITE(ORDER O_2)
	UNLOCK(ORDER O_2)
LOCK-X(B_1)	
READ(B_1)	
CHECK(B_1)	
WRITE(B_1')	
UNLOCK(B_1)	
LOCK-X(ORDER O_1)	
WRITE(ORDER O_1)	
UNLOCK(ORDER O_1)	

We add shared locks before and after each solitary read, and exclusive locks around a read-write pair. This makes sure that no two transactions are accessing/altering the same data point at the same time, which could lead to conflicts.

Transaction Pair 2

Consider the case where a supplier adds products to stock (i.e., increases the quantity), and simultaneously, a customer buys the same product (i.e. decreases the quantity). Here, Q represents the quantity (in stock) of Product 1. B_2 represents the balance of the customer in T_2, while O_2 represents their order.

Transaction-1 (T_1)	**Transaction-2** (T_2)
READ(Q) $Q := Q + q_1$ WRITE(Q) COMMIT	READ(Q) $Q := Q - q_2$ WRITE(Q) READ(B_2) WRITE(B_2) WRITE(ORDER O_2) COMMIT

Conflict-Serializable Schedule

Transaction-1 (T_1)	**Transaction-2** (T_2)
	READ(Q_1) $Q := Q - q_2$ WRITE(Q)
READ(Q) $Q := Q + q_1$ WRITE(Q)	
	READ(B_2) CHECK(B_2) WRITE(B_2) WRITE(ORDER O_2)

Conflict-Serializable Schedule with Locks

Transaction-1 (T_1)	**Transaction-2** (T_2)
	LOCK-X(Q) READ(Q) $Q := Q - q_2$ WRITE(Q) UNLOCK(Q)
LOCK-X(Q) READ(Q) $Q := Q + q_1$ WRITE(Q) UNLOCK(Q)	
	LOCK-X(B_2) READ(B_2) CHECK(B_2) WRITE(B_2) UNLOCK(B_2) LOCK-X(ORDER O_2) WRITE(ORDER O_2) UNLOCK(ORDER O_2)

Non-Conflict-Serializable Schedule

Transaction-1 (T_1)	**Transaction-2** (T_2)
	READ(Q_1)
READ(Q) $Q := Q + q_1$ WRITE(Q)	
	$Q := Q - q_2$ WRITE(Q) READ(B_2) CHECK(B_2) WRITE(B_2) WRITE(ORDER O_2)

Non-Conflict-Serializable Schedule with Locks

Transaction-1 (T_1)	**Transaction-2** (T_2)
	LOCK-S(Q) READ(Q) UNLOCK(Q)
LOCK-X(Q) READ(Q) $Q := Q + q_1$ WRITE(Q) UNLOCK(Q)	
	LOCK-X(Q) $Q := Q - q_2$ WRITE(Q) UNLOCK(Q) LOCK-X(B_2) READ(B_2) CHECK(B_2) WRITE(B_2) UNLOCK(B_2) LOCK-X(ORDER O_2) WRITE(ORDER O_2) UNLOCK(ORDER O_2)

User Guide

How to use ElectroBase Management System

Frontend Development

The frontend development of the application has been completed within the timeframe of Deadline-6, the final deadline for the project. The website is currently only hosted on a developmental server, while the database is hosted on a local machine.

Navigation through the application

The frontend of the application is designed to be as intuitive as possible. Although there remain some minor bugs and unimplemented features, the application is fully functional and can be used by all stakeholders. The UI is very simple and easy to understand, and any user can navigate through the application with ease.

Usage as Customers

Without creating an account

The application can be used without creating an account, but the user will not be able to access the full functionality of the application. Without creating an account, the user can view limited data, such as the top-3 selling products, and the top-3 selling suppliers. They can also browse through the product calalogue.

Creating an account

To create an account, the user can follow one of two steps:

1. Click on the `Log In` button located at the top right of the navigation bar. From there, the user can click on the `Create an account` button to create an account.
2. While browsing the product catalogue, if the user clicks on the `Add to Cart` button, they will be prompted to log in first. From there, they can create an account.

Logging in

After logging in, the user can add products to their cart. They can use the `Cart` button on the navigation bar to proceed to the checkout page. They can review their order before placing it, and if confirmed, they are redirected to the payment gateway and the order will be placed.

On the profile page, the customer can view their personal information[5], their order history, and the current active (undelivered) orders.

[5]The feature to update personal information is not yet implemented.

Usage as Admins

Logging In

The admin[6] can log in to the admin panel by clicking on the `Log In as Admin` button shown in the footer. This button is displayed on almost all pages in the footer, even on the user-login page.

How to use

Once logged in, the admin can use the admin panel to perform various tasks[7], such as viewing the stocks through the catalogue, viewing orders, order statistics, and viewing the sales statistics[8].

Usage as Suppliers and Delivery Agents

Both suppliers and delivery agents have similar interfaces to users for logging in. They can log in by clicking on the `Log In` button located at the top right of the navigation bar. From there, they can perform the tasks specific to their role.

[6]Currently, there are only two admin accounts, both having the same functionality.

[7]The feature to add records through the admin panel is not yet implemented.

[8]Sales statistics are currently displayed in tables. However, we plan to display them through charts it the future.

References and Bibliography

- Sortable Tables in HTML
  The JavaScript code to implement sorting on tables in HTML is taken from this website.
- Button animations using CSS