Team 003 CS 411 Project Report

TA:Aryan Bhardwaj

Team Member: Jack Zhang, Chris Huang, Xingyu Zhu, Yiheng Zhou

1. Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

Initially, we had planned to develop a general inventory-management system—tracking real-time product inventories and alerting when products needed to be restocked—but because our data did not include storage levels per product and timestamp information, we shifted our focus to order management. By concentrating on inserting, updating, querying, and deleting orders (and their corresponding OrderItems), we were able to utilize the complete order details to flesh out and demonstrate our core features—CRUD operations, keyword search, transactions, triggers, and stored procedures—against a real-world domain.

2. Discuss what you think your application achieved or failed to achieve regarding its usefulness.

While our original proposal focused heavily on predictive modeling and inventory optimization, our final application evolved to emphasize strong foundational database management features, customer/order/product tracking, and real-time updates through stored procedures and triggers. Initially, we intended to prioritize machine learning components like predictive sales forecasting early, but due to time constraints and limited data source, we first implemented a comprehensive database-backed management system. Compared to the original plan, the current system now includes functioning stored procedures (for customer order summaries), transaction-safe order insertions, automatic updates using database triggers, and user-friendly CRUD operations for products and orders. Additionally, the customer management section was expanded to allow both viewing and updating customer

order summaries directly through the interface. Although advanced machine learning components like the sales forecasting model and anomaly detection were planned, they were not yet fully incorporated. We shifted focus toward making the system fully functional, stable, and ready for data ingestion, which sets a strong foundation for future work like predictive analytics, inventory auto-adjustments based on orders, CSV upload automation, and market trend API integrations.

3. Discuss if you changed the schema or source of the data for your application

We did not make major changes to the original data sources. Our project still relies heavily on the Olist E-Commerce dataset (structured in CSV format) as originally planned. The source datasets, including information on orders, order items, customers, sellers, and products, were kept intact to preserve the integrity and real-world relevance of our application. Overall, the core data source remained unchanged, and the only modifications were extensions to the schema — not alterations of the original datasets. These extensions were necessary to enable functionalities like dynamic summary updates and better application performance.

4. Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?

We had a redundant customer_unique_id in addition to customer_id in our initial ER model. Because customer_id already ensures uniqueness, retaining both added unnecessary storage and join logic. We removed customer_unique_id, streamlining the Customers table and all queries involving it.

For OrderItems, we'd implemented a surrogate order_item_id per row, but that hid the reality that OrderItems is simply the many-to-many join of Orders and Products. Substituting the

surrogate with a composite (order_id, product_id) primary key better enforces "one entry per product per order," prohibits duplicate line items, and gets rid of the additional column.

Lastly, we initially had seller_id on OrderItems—i.e., each order line had its own seller reference—but a seller is actually an inherent characteristic of a product. Putting seller_id on the Products table keeps all product metadata in one place, prevents duplicating the same seller reference on each order line, and foreign-key constraints are simpler and faster.

5. Discuss what functionalities you added or removed. Why?

Functionality Changes

- Added:
    1. We added stored procedures and triggers to automatically update the customer_summary table whenever an order or order item is inserted or updated.
    2. We added a feature that fetches and displays detailed customer orders along with their average order value using an advanced stored procedure.
    3. We implemented a "Submit Advanced Order" function that inserts both orders and order items within a transaction block.

- Removed:
    1. External Market Trend API Integration:
       Initially, we planned to integrate external APIs (such as Google Trends or Amazon APIs) to pull market trend data. Due to the complexity and time constraints, we decided to postpone this feature for future work.

    2. Machine Learning Model for stock management
       While originally proposed, we did not find appropriate sources for our machine model training process. Additionally, after implementing the advanced query, we do not have enough time to train the model.

Reasons for Changes:

- Technical Complexity: Some features like API integrations require additional backend processing pipelines that would have extended the timeline significantly.

- Focus on Core Functionality: We decided it was more important to perfect essential database operations (insert, update, delete, advanced queries) first to meet our application's core purpose: reliable inventory management and sales prediction.
- Additionally, we did not find appropriate data sources to train the machine learning model, we were afraid that using the current datasets we found will lead to inaccurate prediction therefore we eventually decided to remove the machine learning part of our project.

6. Explain how you think your advanced database programs complement your application.

Our advanced database programs, including stored procedures, triggers, and transactions, constraints enhance the functionality of our application. Stored procedures like update_customer_order_stats and GetCustomerOrders allow complex calculations such as computing customer total spending, order counts, and average order values to be handled efficiently, reducing the workload on the backend server and improving response times for users. Triggers, such as the automatic update of customer_summary after an order insertion or the deletion of order items when an order is canceled.. This ensures that the database remains consistent even when users perform operations like inserting, updating, or deleting orders through the CRUD functions.. By embedding logic directly into the database, we minimized potential errors, improved database query performance, and created a much more robust and responsive user experience.

7. Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.

Problem 1:

When implementing product insertion, the operation was not working properly.

Solution:

We updated the insertion logic to ensure that when a new product is inserted, a corresponding entry is also created in the order_item table with the correct primary key linkage.

Problem 2:

React™ was initially not running correctly during the development environment setup.

Solution:

We switched the development environment to Windows, which fully supports React™, ensuring that the application can compile and run without system-related errors.

Problem 3:

During product insertion, the available product category options shown to the user did not match the values stored in the backend category dataset.

Solution:

We implemented a dropdown box populated with valid product categories pulled from the backend dataset. This restricts user input to only valid, recognized category values, ensuring data consistency and preventing insertion errors

Porblem 4:
Originally, when inserting new order items during a transaction (especially in the /api/insert-order-advanced backend route), you forgot that order_id is part of the composite primary key for the olist_order_items_dataset table. Because order_id wasn't being handled properly during the insert, the database rejected the transaction with a primary key constraint error (because order_id + order_item_id together define uniqueness).

Solution:In the backend transaction logic, we explicitly added order_id into the SQL insert queries for the order items. We also updated the frontend forms (React side) to ensure that order_id is properly collected from the user and sent to the backend whenever submitting a new order.

Problem 5:
While implementing the customer summary update feature, we misunderstood how stored procedures return multiple results. When calling the stored procedure from the backend, the application expected a single result, but instead received multiple result sets because the procedure returned customer orders, average order value, etc. This mismatch caused runtime errors in the frontend and backend parsing.

Solution:
We modified the backend to correctly process all result sets returned by the stored procedure using stored_results().In the frontend, we updated the React logic to safely check if fields like average_order_value are present before trying to format them.

There are many additional bugs that we faced but these are the ones we can currently think of,

8. Are there other things that changed comparing the final application with the original proposal?

Overall, our final application stayed close to the original proposal in terms of its core idea: building an E-Commerce Inventory Optimization and Order Management System. However, a few small changes occurred during implementation:

More Focus on Database Operations:

In the final version, we spent significantly more time strengthening the database backend, including creating advanced SQL stored procedures, triggers, and transactions. We added two complex stored procedures (customer order details + summary update) and a trigger to automatically update inventory after orders are placed.

Frontend Focus Shift:

The original proposal emphasized complex machine learning visualization dashboards. However, during implementation, we first prioritized building a clean, robust frontend to handle product and order management reliably. Features like customer summary fetching and updating, and transaction-based order insertions, were implemented before diving into predictive analytics modules. Additionally, we did not find appropriate data sources to useful our machine learning model trainning so we have to give up on this part.

Prediction Model and External APIs Deferred:

Due to time and scope management, the integration of external APIs (e.g., Google Trends, Amazon API) and sales forecasting models was postponed for future work. We focused on completing a solid CRUD + transactional + analytical system first, leaving prediction modules (like Linear Regression for best-seller prediction) for future expansion.

9. Describe future work that you think, other than the interface, that the application can improve on

Beyond enhancing the user interface, there are several directions for improving the Online Shop Management System. First, we plan to implement a stock management system, as we are the only system that processes order and customer information. For the stock part, we are considering implementing the feature such that whenever a customer places an order, the system would automatically decrease the available product quantity, ensuring accurate inventory tracking and preventing overselling. Second, we want to add a CSV file converter that allows users to bulk upload data into the database. This feature would enable integration of large datasets, improving efficiency for shop managers who handle numerous products or orders. Lastly, we propose developing a predictive analytics feature using a linear regression model. By analyzing historical sales data, the system could predict the best-selling products, helping shop owners make smarter stocking and marketing decisions. These improvements would make the application more intelligent, scalable, and user-friendly, pushing it closer toward a full-scale commercial e-commerce management solution.

10. Describe the final division of labor and how well you managed teamwork.

Jack Zhang:

1. Advanced Database Feature: including transaction, trigger, stored procedure, constraints and its integration to front/back end.
2. Frontend optimization
3. Connection between databases and backend.
4. Final Report Part 2 5 6 8 9
5. Part of the video
6. Most of stage3

Xingyu Zhu:

1. Final Report Part 1 3 4
2. Redesigned the UML diagram and refined table implementations for a more normalized, efficient schema
3. Updated database DDL scripts and documentation to reflect those schema changes

Yiheng Zhou:

1. Final report part 3
2. Designed the ER diagram
3. Worked on frontend (committed under teammates device because I never owned a windows machine)

Chris Huang:

1. Final report part 7
2. Frontend paging building and products page functionality
3. backend implementation for products querying and optimization for orders
4. designed for 3ML normalization
5. part of the video

Final Project Video

https://illinois.zoom.us/rec/share/xDhJ7RsrpPP-NQLP_CSyRfcRIK5ZnN1L48wjCJVYQND6Vj6fjQrgZ1yhfYn7X7NT.V9brgC0-pR18NMfv?startTime=1745895523000

Implementation detail of our Advanced Database Feature:

Trigger
Trigger 1

```
CREATE TRIGGER trg_auto_delete_canceled_orders
AFTER UPDATE ON olist_orders_dataset
FOR EACH ROW
BEGIN
   -- If an order is canceled, delete its associated order items
   IF NEW.order_status = 'canceled' AND OLD.order_status != 'canceled' THEN
      DELETE FROM olist_order_items_dataset
      WHERE order_id = NEW.order_id;
   END IF;
END
```

Purpose :This trigger automatically deletes all related order items if an order gets canceled.

Trigger 2

```
CREATE TRIGGER update_customer_summary_after_item_insert
AFTER INSERT ON olist_order_items_dataset
FOR EACH ROW
BEGIN
   DECLARE cust_id VARCHAR(50);
   -- Get the customer_id linked to this order
   SELECT customer_id INTO cust_id FROM olist_orders_dataset WHERE order_id =
NEW.order_id;

   INSERT INTO customer_summary (customer_id, total_orders, total_spent)
   VALUES (
      cust_id,
      1,
      NEW.price + NEW.freight_value
   )
   ON DUPLICATE KEY UPDATE
      total_orders = total_orders + 1,
      total_spent = total_spent + NEW.price + NEW.freight_value;
END
```

Purpose:This trigger updates or inserts a customer's summary (number of orders and total spending) whenever a new order item is inserted.

Transaction 1

```
START TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
INSERT INTO olist_orders_dataset (order_id, customer_id, order_status)
VALUES ('ORD12345', 'CUST67890', 'processing');

-- 2. Insert items for that order
INSERT INTO olist_order_items_dataset (order_id, product_id, seller_id, price,
freight_value)
VALUES
('ORD12345', 'PROD1', 'SELLER1', 100.00, 10.00),
('ORD12345', 'PROD2', 'SELLER2', 200.00, 15.00);

-- 3. Update status of a different order if customer has more than 5 orders
UPDATE olist_orders_dataset
SET order_status = 'priority'
WHERE order_id = (
   SELECT order_id FROM (
      SELECT order_id
      FROM olist_orders_dataset
      WHERE customer_id = 'CUST67890'
      ORDER BY order_id DESC
      LIMIT 1
   ) AS latest_order
)
AND (
   SELECT COUNT(*)
   FROM olist_orders_dataset
   WHERE customer_id = 'CUST67890'
) > 5;

-- 4. Insert or Update customer total spending in summary table
INSERT INTO customer_order_summary (customer_id, total_spent)
VALUES (
   'CUST67890',
   (
      SELECT SUM(price + freight_value)
      FROM olist_order_items_dataset oi
      JOIN olist_orders_dataset o ON oi.order_id = o.order_id
      WHERE o.customer_id = 'CUST67890'
   )
)
ON DUPLICATE KEY UPDATE
   total_spent = VALUES(total_spent),
   last_updated = CURRENT_TIMESTAMP;

COMMIT;
```

Stored Procedure 1

This stored procedure calculates and updates summary statistics for a specific customer in our Online Shop Management System.

```
CREATE PROCEDURE update_customer_order_stats(IN cust_id VARCHAR(50))
BEGIN
   DECLARE total_spent DECIMAL(10,2);
   DECLARE total_orders INT;

Query 1
(Advanced – Aggregation):
   -- Calculate total orders
   SELECT COUNT(DISTINCT o.order_id)
   INTO total_orders
   FROM olist_orders_dataset o
   WHERE o.customer_id = cust_id;

(Advanced – JOIN + Aggregation):
   -- Calculate total spent
   SELECT COALESCE(SUM(oi.price + oi.freight_value), 0)
   INTO total_spent
   FROM olist_orders_dataset o
   JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
   WHERE o.customer_id = cust_id;

   --  Update customer_summary table
   IF EXISTS (SELECT 1 FROM customer_summary WHERE customer_id = cust_id) THEN
      UPDATE customer_summary
      SET total_orders = total_orders,
         total_spent = total_spent,
         last_updated = NOW()
      WHERE customer_id = cust_id;
   ELSE
      INSERT INTO customer_summary (customer_id, total_orders, total_spent,
last_updated)
      VALUES (cust_id, total_orders, total_spent, NOW());
   END IF;
```

Stored Procedure 2

This stored procedure retrieves detailed order information for a specific customer, including aggregated values and an alert if no orders exist.

```
CREATE PROCEDURE GetCustomerOrders(IN cust_id VARCHAR(50))
BEGIN
   DECLARE total_orders INT;

   -- Advanced Query: Get all orders with their total price and freight (JOIN + GROUP BY)
   SELECT
      o.order_id,
      o.order_status,
      o.order_purchase_timestamp,
      SUM(oi.price + oi.freight_value) AS total_amount
   FROM olist_orders_dataset o
   JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
   WHERE o.customer_id = cust_id
   GROUP BY o.order_id, o.order_status, o.order_purchase_timestamp;

   -- Advanced Query: Get average order amount for this customer (Subquery +
Aggregation)
   SELECT
      AVG(order_totals.total_amount) AS avg_order_value
   FROM (
      SELECT
         o.order_id,
         SUM(oi.price + oi.freight_value) AS total_amount
      FROM olist_orders_dataset o
      JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
      WHERE o.customer_id = cust_id
      GROUP BY o.order_id
   ) AS order_totals;

   SELECT COUNT(*) INTO total_orders
   FROM olist_orders_dataset
   WHERE customer_id = cust_id;

   IF total_orders = 0 THEN
      SELECT 'This customer has no orders' AS message;
   END IF;

END
```

Attribute-Level Constraints:
Non-negative price and freight_value:

```
ALTER TABLE olist_order_items_dataset
MODIFY price DECIMAL(10,2) NOT NULL CHECK (price >= 0),
MODIFY freight_value DECIMAL(10,2) NOT NULL CHECK (freight_value >= 0);
```

Enforce order_status is not empty or null

```
ALTER TABLE olist_orders_dataset
MODIFY order_status VARCHAR(50) NOT NULL CHECK (order_status <> '');
```

Ensure product_photos_qty is non-negative

```
ALTER TABLE olist_products_dataset
MODIFY product_photos_qty INT NOT NULL CHECK (product_photos_qty >= 0);
```

Tuple-Level Constraints (Multi-column checks):

freight_value cannot exceed price:

```
ALTER TABLE olist_order_items_dataset
ADD CONSTRAINT chk_freight_vs_price CHECK (freight_value <= price);
```