Team 006 (GPT6) Final Report

by Shenghan Gao, Ke Yao, Jiajun Gao, Yingcheng Deng

Changes regarding the original proposal:

The final project is pretty close to the original idea for an e-comm seller dashboard but we actually tweak a few things as we rolled it out:

The original proposal was supposed to have a Sales Trends chart, but we shortened that one to create space for Category Distribution and Top Products using those fancy advanced queries. It made us concentrate on the real metrics that are close to the real world application.

We changed exact to partial matching to make the search a little more forgiving which is not a game-changer but a better user experience nonetheless.

We stayed with the Olist E-Commerce dataset from Kaggle as we were planning and therefore nothing changed in this regard.

It was not mentioned in the proposal but to ensure we handle the dynamic updates correctly to match with real world applications, we added explicit "versioning", "updated at" fields, audit logs, and history tables such as **ProductVersions** and **OrderStatusHistory**.

Achievements:

We nailed all the CRUD operations for products and orders, providing sellers with a full blown management interface.

Rather than requiring the user to press refresh. we do delta polling to mirror changes automatically - we do not require any manual clicks.

We dropped the Top Selling Products query right onto the reports page, now you have an idea how database optimizations actually come in handy for class projects.

When two people attempt to edit the same thing simultaneously, we provide them with a clear dialog in which to select the appropriate version.

Multi-user edits are now safely processed by the system as version control ensures that even when many people are accessing the same record, the data stays the same.

Shortcomings:

1. Although there is no conflict because of optimistic locking, it would still be desirable to have a live ping when a record is being edited by another person, which would be solved by a WebSocket.

2. The product search is just a simple LIKE; it would be a lot better with a full-text search.

3. Sampling every 5 seconds would be a choked bandwidth with a lot of users - we would have to be smarter in detecting changes in production.

4. We have the answer to the question of responsive design, but the dashboard is very desktop oriented; mobile is a potential candidate for a face-lift.

5. Delta polling mechanism allows the dashboard to reflect changes without manual refresh, improving user experience.

Schema and Data Source Changes:

We used the same data source - still the Olist E-Commerce Dataset from Kaggle for Product, Orders and Order Items, Customers, Payments and Reviews.

The data was pulled using the kagglehub library and reshaped to a normalized schema.

Schema Changes:

Products Table Enhancement:
   Added: version INTEGER field to optimistic locking.
   Added: changed in-TIMESTAMP to track the change.
   Why: These fields allow us to identify version based conflicts and maintain an audit trail without corrupting the base product table.

2.Orders Table Enhancement:
   Added: updated at TIMESTAMP field.
   Why: Monitors order modifications, which can be found in delta polling as well as audit purposes.

Additional Tracking Tables:
   ProductVersions: Contains an entire history of each product
   OrderStatusHistory: Records all changes of status timely and with comments.
   AuditLog: This is a list of every change to any table.
   Why: These were not in the original ER diagram, but are required to lock, comply and debug.

ER Diagram Differences:

The basic relationships remained the same as the original design.

More Suitable Design:

The final design is the better one because:

Auditability: All the changes are recorded in new tracking tables.
Concurrency: Version fields enable us to safely edit at the same time.

Performance: Indexing (visited in our analysis) makes the queries (with the additional tables) fast.
Maintainability: It is easier to debug problems with orders using status history tables.

Tweaks that we would consider to be production-ready:
Reporting queries are separated into read replicas.
Materialized views of commonly used reports.
Separating large tables (Orders, OrderItems) by date.

Functionalities Added or removed:

Added Functionalities:

Optimistic Locking System:
Product version tracking.
Conflict detection & a res- selection UI
Reason: Important where there are several people who are making changes on the same data.

Delta Polling::
Automatic refresh
Efficient change detection
Why: Updates the UI automatically, without the need to reload.

Order Status History:
Status changes have full audit trail.
Notes and times of every transition.
Reason: Customer service and compliance key.

Advanced Query Integration:
Top Products report Pulls Query 1 from the advanced queries
Why: Demonstrates the techniques of database optimization, which we learned in class.

Broad-based Mishandling of errors:
Error messages that are user-friendly.
Detailed backend error info
Reason: Less difficult debugging and enhanced user experience.

Removed Functionalities:

1.Sales Trends Chart:
Dropped the time-based sales trend view
Why: it was redundant, the advanced queries provided us with more useful insights.

Time Period filtering is an option that should be used to filter reports.
Removed the time selector

Why: The advanced queries don't require time filtering and it was only used for the removed trend chart.

Reason: We used those cuts to improve focus on the core business value. We could re-add Sales Trends sometime in the future, but for the moment the reports provide sufficient insight for decision-making.

Advanced Database Programs Supplementing the Application:

Our advanced queries are the direct driver of the reporting features of the app:

Query 1 - Top Selling Products , Revenue Analysis:
Where: CS411finalproject/cs411finalproject.py (lines 463-481)
Integration: Used in Getting /api/reports/top-products endpoint
Purpose: Powers "Top Products Table" on the Reports page
Business Value: Assists sellers to identify the most selling products, set priorities on inventory, and make smart promotion choices.

Question 2 - Customer Lifetime Value:
Place: CS411finalproject/cs411finalproject.py (lines 493-537)
Potential Integration: Can become a "Top Customers" report
Value: Identify the high value customers for targeted marketing

Query3 - Sales Velocity Inventory Analysis:
Location: CS411 final project/cs411finalproject.py (lines 549-591)
Potential Integration: Potential to beef up Inventory section
Value: Predicts stockouts, and helps to maintain the optimal inventory amounts

Query 4 - Analysis of Payment Method:
Location: CS411finalproject/cs411finalproject.py (lines 603-643)
-Potential Integration: Potential Integration Could Be Shown in Reports or the Dashboard
Value: Understands which payment methods people prefer

The indexing analysis that we ran (runcompleteindexinganalysis) showed performance gain of 20-60% on several queries which was a proof that our database design is ramped up for the load of the app.

How They Complement:

The advanced queries aren't just academic exercises—they solve real business problems:
Performance: Indexed queries ensure reports load quickly even with large datasets
Business Intelligence: Complex aggregations provide actionable insights
Scalability: Proper indexing means the application can handle growth without performance degradation

Technical Challenges:

Jiajun Gao: At the development stage, we were unable to make hangouts with the Flask back-end on our front-end. Any requests to /api/health were always failed, so it was apparent that we were not even connecting to the server. This reason was a port conflict on macOS. The system process had already occupied TCP port 5000, and in attempting to bind Flask to port 5000, the server simply refused to boot up correctly, that is, the front-end was unable to connect.

The front-end API client was also somewhat silent. The lack of detailed logs meant that it was not possible to know whether the failures were because of network hiccups or CORS hoops or a bunged start-up. Such lack of transparency made debugging a real pain.

To solve the problem, we renamed the Flask back-end back to port 5001 and avoided the conflict with the macOS altogether.

Yingcheng Deng:

Another challenge met during the development was the Report page. We wanted the page to show the table of the Top products and a bar graph of the Top products. However, at the beginning, those two tables didn't show anything at the frontend.

To fix the page and make it shown as exactly as we want. At the backend, we examined and almost completely rewrote the ‘ GET /api/reports/top-products’ (get_top_products() at line 866 in ./backend/app.py) so that it will use the Advanced Query 1 from the cs411_final_project.py and show exactly what the Query filters. We changed the query to filter by status IN ('delivered', 'shipped') instead of != 'canceled', added HAVING clause: `total_revenue > 1000` to match advanced query requirements, changed field names to match the advanced query, removed time-based filtering (days parameter) as advanced query doesn't use it and defined the default limit to 15 to match the advanced query.

At the frontend, we updated the TopProduct interface to match new field names, removed the time period selector as it's no longer needed, and updated the chart and table so that they correctly display.(./frontend/src/pages/Reports.tsx) (./frontend/src/api/reports/ts)

Ke Yao:

One major challenge was the reliability of product deletion. Originally, if a delete was attempted when an **order_item** already existed for the product, it would fail silently.

The backend would give a foreign key constraint error, but the frontend simply reported an undefined failure.

The main reason is that the products referenced in orders cannot be deleted due to foreign key constraints. Thus, on the backend, before running the actual DELETE query, we added a check to see whether the product appeared in the `order_items` **table.** If it did, the backend now returns a clear and human-readable message explaining that the product cannot be deleted because it is part of existing orders. This prevents unnecessary SQL errors and gives users meaningful feedback.

We also improve the error handling in the frontend **(Products.tsx)** to display the informative backend messages to the user and to refresh the product list only upon a successful delete. These changes provide a cleaner, safer and more predictable feedback to the user.

Tiger Gao:

One of the key issues I addressed is “Order Status Display and Update Fixes”. The problem is caused by the mismatch between how the backend formatted order data and how the frontend expected to receive it. The API response structure was slightly different from the TypeScript interfaces expected by the frontend code. As a result, several elements on the order detail page were inconsistent, with fields failed to load, dates being displayed incorrectly, and status updates appearing to not work with no clear indication to the user why.

To address this, I standardized the data structure on the backend order detail endpoint. The API now returns a clean, consistent, and predictable object that matches exactly what the frontend needs, rather than multiple nested objects that require additional parsing. This minimizes ambiguity and prevents undefined values from appearing in the UI.

On the frontend side, I added error handling logic to guarantee that all unexpected or invalid data is caught early and communicated clearly. In addition, I also made more checks for missing and malformed dates so they are displayed in a more readable manner.

Other Modifications of Original Proposal:

Error Handling Improvement: The extensive error management has been implemented throughout the application which was not present in the original proposal. This enhancement has greatly increased the debugging and user experience.

Customer Search Enhancement: Customer identifier search has been adjusted to allow partial matching as opposed to exact matching and this has enhanced usability.

Date Handling: There is also solid validation and null handling of a date to ensure that the display of an invalid date is avoided.

Empty State Handling: Empty-state indicators have been provided in the appropriate tables and lists, such as showing No items found, or No products found.

Foreign Key Constraint Checks: Pre-deletion validation has been used to check the foreign key constraints before any deletion attempts are undertaken and hence produce user friendly error messages.

Formatting API Response: To make API responses easier to consume by the frontend, they have been made more frontend-friendly, including the flattening of nested structures where necessary.

Future Work:

Advanced Search and Filtering:
- SQLite FTS or Elasticsearch full-text search of products.
- Multi-criteria filtering with preset filter settings.
- Sophisticated date-range filtering with relative time specification, e.g. last week, last month, etc.

Bulk Operations:
- Massive changes in product characteristics (price changes, category changes, etc.).
- Bulk updates to order status.
- Product data may be imported using CSV.
- Massive changes in inventory levels.

Security Enhancements:
- User authentication and authorization.
- Role based access control specifying such roles as an admin, seller, and viewer.
- API rate limiting.
-Sanitization and input validation.
- SQL injection prevention, which is currently being done by using parameterized queries, which may be enhanced by an ORM.

Scalability:
- Database replication or database sharding in large-scale deployment.
- Adoption of a microservices architecture that separates components including products, orders and reports.
- Implementation of a message queue that will support the asynchronous processing of bulk operations.
- Implementation of a content delivery network on static assets.

Monitoring and Logging:
- Application performance monitoring (APM).
- Tracking and notifying of errors via Sentry and Rollbar.
- Usage pattern and user behavior analytics.
- Database query performance monitoring.

Mobile Optimization:
- Progressive Web App (PWA) functionality.
- 10 point tablet friendly interface.
- Mobile-specific primary operations.
- Mobile users push notification.

Last Division of Labor and Teamwork:

Backend Development:
- Ke Yao: Flask API development, database schema, and advanced query development.
- Jiajun Gao: API endpoint implementation, error management, and request/response formatting.

Frontend Development:
- Jiajun Gao: React/TypeScript components and UI/UX design development.
- Ke Yao: API integration, state management, and front-end error management.

Database & Analytics:
- Tiger Gao: SQL query, analysis and query optimization.
- Tiger Gao: Data population, database schema design, and testing.

Integration & Testing:
- Yingcheng Deng: Bug fixing, frontend, and backend integration, and testing.
- Yingcheng Deng: API testing, edge-cases, and documentation.

Teamwork Assessment:

Strengths:

1. The separation of frontend and backend duties was clearly defined, which made it possible to develop them simultaneously.
2. The alignment was maintained through regular communication through meetings and code reviews.
3. Git version control made it possible to collaborate on development without conflict.
4. All the architectural decisions were kept in shared documentation (README) to inform all the team members.

Challenges:

1. The first discrepancies in API response formats and frontend requirements required coordination.
2. Differences in development environments also led to the problems of works on my machine that needed to be standardized.