CS 121 Final Project Reflection

Due: Saturday March 16th, 11:59PM PST

Note: This specification is currently for 23wi, subject to minor changes for 24wi.

This reflection document will include written portions of the Final Project. Submit this as **`reflection.pdf`** with the rest of the required files for your submission on CodePost.

For ease, we have highlighted any parts which require answers in **blue**.

Student name(s): jonathan@caltech.edu

Student email(s): eluk@caltech.edu

Part L0. Introduction

Answer the following questions to introduce us to your database and application; you may pull anything relevant from your Project Proposal and/or README.

Database/Application Overview

What application did you design and implement? What was the motivation for your application? What was the dataset and rationale behind finding your dataset?

Database and Application Overview Answer (3-4 sentences) :

We designed and implemented a supermarket data analysis and order management system that helps store managers and analysts track sales trends, optimize stock levels, optimize store layout, and optimize the suppliers for their stores. The motivation behind this application was to provide data-driven insights to improve decision-making in supermarkets, such as identifying high-demand products and evaluating supplier efficiency. The system features a command-line interface for clients to retrieve sales analytics and for admins to update product and order data. The dataset we used was sourced from Instacart's Market Basket Analysis dataset on Kaggle, which provides real-world transactional data that aligns well with our goal of analyzing shopping patterns.

Data set (general or specific) Answer:

We used the Instacart Market Basket Analysis dataset from Kaggle and performed extensive cleaning and preprocessing. To improve clarity and scalability, we renamed the order_products table to products_in_order and removed many columns from our tables, such as order day of week, hour of day, if the product was reordered, and arrangement of orders for a given customer, since they can all be derived from timestamps. This approach makes the dataset easier to maintain, more scalable, and less prone to corruption/inconsistencies. To enhance flexibility, we modified the dataset to allow order_timestamp to be NULL, accommodating cases

where handwritten or manually entered data lacks precise timestamps. Furthermore, we introduced a store_id field to support multi-location supermarkets, making the analysis more applicable to real-world store chains. We artificially generated the stores and suppliers tables with python scripts. A given order can only be applied to a particular store, and a given product in an order can be applied to a supplier. If someone wants to order across multiple stores, they can just split up their original order into multiple orders. Also, we assume all online orders are routed to the supermarket closest to the user. We put the supplier_id in the products_in_order table instead of the products table because we thought it would be more realistic if we allowed product suppliers to be able to change over time, instead of assuming we always have the same supplier for a given product .These modifications significantly improved the dataset's usability, making it more efficient, scalable, resistant to data corruption, and realistic, ensuring that it serves as a robust foundation for our supermarket management system.

***Client user(s) Answer*:**

The client users are lower-level supermarket managers or analysts who primarily use the system for read-only queries. They can analyze the supplier efficiency of all the stores, analyze product sales trends, and view the most popular aisles. These insights help them make data-driven stocking and marketing decisions, but they do not have the ability to modify inventory data.

Admin user(s) Answer:

The admin users are higher-level store managers or regional supervisors who have full control over the inventory system. They can add new orders and change product names. Admins ensure that the system remains up to date and accurate, preventing discrepancies between store stock levels and customer demand trends.

Part A. ER Diagrams

As we've practiced these past few weeks, the ER model is essential for designing your database application, and we expect you to iterate upon your design as you work through the ER and implementation steps. In this answer, you should provide a full ER diagram of your system. Your grade will be based on correct representation of the ER model as well as readability, consistency, and organization.

Notes: For this section **only**, we will allow (and encourage) students to share their diagrams on Discord (**#er-diagram-feedback**) to get feedback from other students on their ER diagrams given a brief summary of your dataset and domain requirements. This is offered as an opportunity to test your ER diagrams for accuracy and robustness, as another pair of eyes can sometimes catch constraints that are not satisfied or which are inconsistent with your specified domain requirements.

Requirements:

- Entity sets, relationship sets, and weak entity sets should be properly represented (also, do not use ER symbols not taught in class)
- Mapping cardinalities should be appropriate for your database schema, and in sync with your DDL
- Participation constraints should be appropriate and in sync with your DDL (total, partial, numeric)
- Use specialization where appropriate (e.g. *purchasers* and *travelers* inheriting from a *customers* specialization in A6)
- Do not use degrees greater than 3 in your relationships, do not use more than one arrow in ternary relationships.
- Use descriptive attributes appropriately
- Underline primary keys and dotted-underline discriminators
- Expectations from A6 still apply here
- Note: You do not need ER diagrams for views

ER Diagrams:

The ER diagram for our supermarket data analysis and inventory management system consists of five main entity sets: Products, Orders, Stores, Aisles, Suppliers, and Departments, with a products_in_order relationship linking products to orders. products_in_order is a relationship set where it connects products, orders, and suppliers. The supplier for a given product can change over time, that's why we didn't associate the suppliers directly with the products table. Our tertiary relationship is saying that for each relationship between a product and order, it can have 1 supplier. But there are no arrows for products or orders because multiple arrows on a 3+ relationship is ambiguous, and it doesn't make sense in our situation. Products can be part of different orders, and orders can have different products. But each product in an order can only have a single supplier; so, suppliers can't "work together" to create a product. Also, we have total participation in an order, because we assume that all orders have at least 1 product in it. We have a python script that

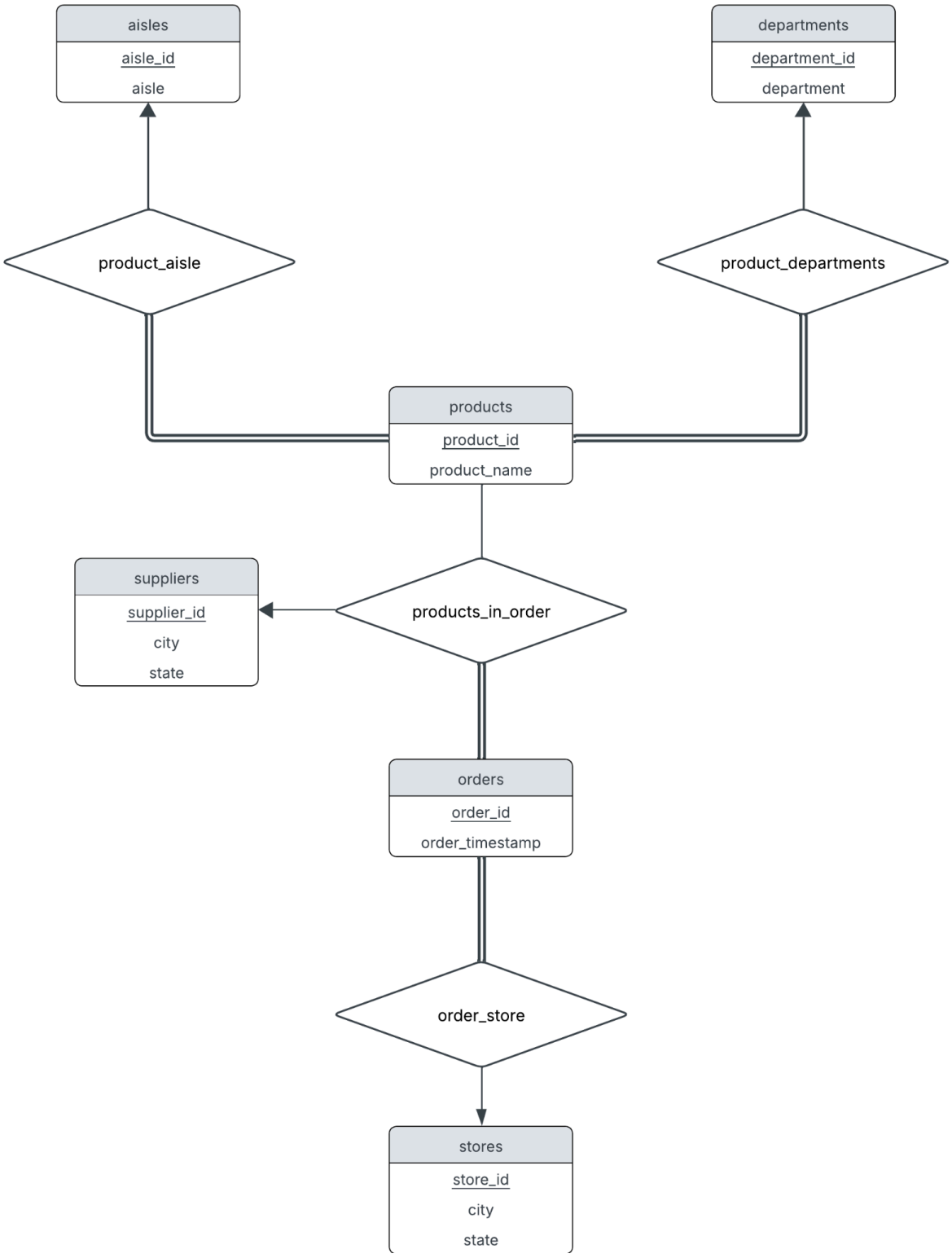
removes all orders from the original dataset that don't match this constraint. We also have a script that removes all rows in products_in_order where the order doesn't exist in the orders table.

The orders entity, uniquely identified by order_id, records transactions made by users and includes a timestamp that may be NULL for manually entered data. We allowed null values for timestamp data because grocery stores are usually not very technologically advanced (like Walmart still not having touch to pay even though it's been around for decades). Thus, it is very possible that handwritten logs do not have timestamps on every order.

The stores entity represents physical supermarket locations, identified by store_id, and includes details about the city and state. The same goes with the suppliers entity set. Each aisle and department organizes products, with each product assigned to a single aisle and department, creating a many-to-one relationship between products and aisles/departments.

Participation constraints ensure total participation for products in the aisles and departments relationships, as every product must be categorized appropriately. However, orders have partial participation in the products_in_order relationship, as not every product must be part of an order. A good store manager would want to minimize the number of products in their store that haven't been ordered. Additionally, derived attributes such as is_reordered and days_since_prior_order have been removed from storage and can instead be dynamically computed when needed (as described in the intro section of this document).

(see next page)


aisles
aisle_id
aisle
departments
department_id
department
product_aisle
product_departments
products
product_id
product_name
suppliers
supplier_id
city
state
products_in_order
orders
order_id
order_timestamp
order_store
stores
store_id
city
state

Part B. DDL (Indexes)

As mentioned in Part B, you will need to add at least one index at the bottom of your **`setup.sql`** and show that it makes a performance benefit for some query(s).

Here, describe your process for choosing your index(es) and show that it is used by at least one query, which speeds up the performance of the same query on a version of the same table without that index. You may find lec14-analysis.sql and Lecture 14 slides on indexes useful for strategies to choose and test your indexes. **Remember that indexes are already created in MySQL for PKs and FKs, so you should not be recreating these.**

Index(es):

We created this index:
CREATE INDEX idx_suppliers_city ON suppliers (city);

Justification and Performance Testing Results:

This helped with the efficiency calculations, specifically the store_efficiency UDF. When comparing if suppliers are in the same city as the store, the index makes this comparison faster. It made our function that calculates store_efficiency 15% faster.

We also made the following changes to our DDL after our final project proposal meeting:

- Removed ON DELETE CASCADE from products table for aisle_id and department_id. Did this because we still want the product info data to exist for historical data analysis, even if an aisle or department is deleted
- Removed NOT NULL requirements from order_id and product_id in order_products table because they're both primary keys, so it's redundant to require them to be not null

---

Part G. Relational Algebra

Requirements (from Final Project specification, Part G):

- Minimum of 3 non-trivial queries (e.g. no queries simply in the form **`SELECT <x> FROM <y>`**)
- At least 1 group by with aggregation
- At least 3 joins (across a minimum of 2 queries)
- At least 1 update, insert, and/or delete
    - This may be equivalent to said SQL statements elsewhere (e.g. queries or procedural code), but are not required to be; in other words, you can write these independent of other sections
- Appropriate projection/extended projection use
- Computed attributes should be renamed appropriately
- Part of your grade will come from overall demonstration of relational algebra in the context of your schemas; obviously minimal effort will be ineligible for full credit; it is difficult to formally define "obviously minimal", but refer to A1 and the midterm for examples of what we're looking for
- Above each query, briefly describe what it is computing; we will use this to grade for correctness based on what the query is supposed to compute; lack of descriptions will result in deductions, since we have no idea otherwise of what the query is intended to do.

Below, provide each of your RA queries following their respective description.

We weren't sure how to do the sql LIMIT function in RA, so we used python convention for a sublist. We also couldn't figure out from the notes how to do ascending or descending, so we used \tau to indicate order.

```
SELECT product_name, COUNT(*) AS total_orders
FROM orders o
NATURAL JOIN products_in_order
NATURAL JOIN products
GROUP BY product_id
ORDER BY total_orders DESC
LIMIT 15;
```

$$\Pi_{product\_name,total\_orders}(\tau_{total\_orders DESC}({}_{product\_id}G_{COUNT(*) \text{ as } total\_orders}(orders \bowtie products\_in\_order \bowtie products)))[0:15]$$

```sql
SELECT aisle, COUNT(*) AS order_count
FROM orders o
NATURAL JOIN products_in_order
NATURAL JOIN products
NATURAL JOIN aisles
GROUP BY aisle
ORDER BY order_count DESC
LIMIT 10;
```

$$\Pi_{aisle\_name,total\_orders}(\tau_{total\_orders DESC}(_{aisle\_id}G_{COUNT(*)\text{ as }\rightarrow total\_orders}(orders \bowtie products\_in\_order \bowtie products \bowtie aisles)))[0:15]$$

Note: for the following query, our python logic checks that the product_id exists in the product table. If not, it does not let the user rename and asks the user for a valid product_id again.

```sql
UPDATE products
SET product_name = 'Uncle Irohs Jasmine Dragon Tea'
WHERE product_id = 3;
```

$$products' \leftarrow \sigma_{product\_id \neq 3}(products) \cup \rho_{product\_name \rightarrow \text{ 'Uncle Irohs Jasmine Dragon Tea'}}(\sigma_{product\_id=3}(products))$$

Part L1. Written Reflection Responses

Challenges and Limitations

List any problems (at least one) that came up in the design and implementation of your database/application (minimum 2-3 sentences)

Answer:

One of the biggest problems for the implementation of our database was that the original database we downloaded our data from had a lot of redundant data. For example, there was a flag that indicated whether a product was reordered or not. This can easily be derived by just checking in the data if the user has already ordered the product in a previous order. To solve this problem of almost half the attributes able to be derived, we created a python script that just generated artificial timestamp objects. Then using the timestamps, a user could derive the attributes we removed. The referential integrity was also broken. There were orders with no products, and products within orders where the order_id didn't exist. Thus, we had to write about 10 data cleaning scripts (they can be found in our repo at data_cleaning/ directory) to ensure referential integrity and a host of other functions. Finally, we added the concepts of different stores and suppliers. These two tables added the problem of logistics that a user could analyze (whether shipping for a given store was efficient or not)

Future work

If you are particularly eager for a certain application (have your own start-up in mind?), it is easy to over-scope a final project, especially one that isn't a term-long project. You can list any stretch goals you might have "if you had the time" which staff can help give feedback on prioritizing (2-3 sentences).

Answer:

- One major improvement we would like to make is to allow stores to have different layouts. Right now, we assume that all stores have the exact same configuration (which products are in which aisles and departments). Once a chain grows relatively large, it is unreasonable to assume that each store can have the exact same layout due to physical constraints. Thus, a next improvement could be creating a weak entity set that defines which products belong to which aisles in a specific store. Since some stores will have almost identical layouts, we can create layout versions and assign each store a layout version.
- Another functionality we could have achieved would be predictive ordering based on historical customer behavior. By analyzing past orders and user preferences, we could suggest frequently purchased items to streamline shopping experiences. Additionally, we would consider implementing a REST API for web-based interaction, making the system

more scalable. Creating an API would allow other developers to interact with our database, without the need to use the command-line interface.
- We could also add quantity to each product sold. Right now, we only care about if a product is being sold. This is okay for predictive modeling purposes for suggesting products, since the vendor likely just cares if they buy the product, and less about how many they buy. But the quantity of products purchased would offer more analysis options, such as revenue flow.

Self-Evaluation

What is your expected grade for the Final Project (out of 100)? Justify what you think are the strongest points, especially pointing to demonstrated improvement in areas that may have had lower scores in assignments throughout the term. Also provide any notes here on areas that could be improved (and what you would do differently next time).

Answer:

We expect a grade of 95+ for the Final Project. The strongest aspects of our project include:

A well-designed relational schema that maintains referential integrity while allowing efficient queries. We spent several hours really thinking this through and transforming the Kaggle dataset.

In particular, we spent a lot of time doing data preprocessing, since the dataset we originally downloaded was meant for machine learning models and predictive tasks, rather than direct data analysis. In particular, we created a script that ensured all orders had products, and all products that were associated with an order_id actually had a row in the orders table with that order_id (some products belonged to “ghost” orders - orders that we didn’t have data for). We also created artificial timestamps for the orders, removing many rows: order_dow, order_hour_of_day, reorder, customer_order_number, days_since_prior_order. Since we felt that the dataset was lacking, we also created the concept of separate stores and suppliers. Each order is associated with a store, and each product within an order is associated with a supplier. For the stores and suppliers table, the city and state are attributes. Our initial dataset was too large, so we removed all orders associated with a user whose user_id was greater than a set constant. Without “trimming” our dataset, it would take a very long time to test our SQL scripts, especially the initial loading of the data.

We also went above and beyond for our admin function that adds new orders. Jonathan talked with Hovik during a meeting on Thursday, and Professor Hovik introduced material related to rollback() and commit(). Jonathan implemented these functions into the python program. They’re used to ensure transaction integrity. In other words, if one operation fails when we are trying to insert the new order, then we want the entire transaction to fail and to notify the user. We call the rollback() if any number of errors happen, like the defined list of products is empty. The python code then does conn.commit() if everything goes correctly.

One area for improvement is indexing strategies - we could have experimented more with performance tuning for high-volume queries, given that our index only improved performance by 15%. Given more time, we would also add unit tests for stored procedures to catch edge cases more effectively. We also do not have that many command line, but we believe that the amount of effort we put into the existing ones, namely the adding of new orders, that used concepts that we didn't learn in class (such as passing around lists, transaction guarantees, and commit/rollback) prove that we went above and beyond than the bare minimum for the specs, canceling out any silly mistake(s) we made. This also shows that it would be relatively easy to implement more features that were within the scope of the class, since we implemented functions that went beyond the scope.

Collaboration (Required for Partners)

This section is required for projects which involved partner work. Each partner should include 2-3 sentences identifying the amount of time they spent working on the project, as well as their specific responsibilities and overall experience working with a partner in this project.

Though each partner had their focuses, we still consulted with each other for each part of the project.

Partner 1 Name: Jonathan Lin
Partner 1 Responsibilities and Reflection:
I primarily focused on the sql and sql-python integration side of the project. For example, I wrote the python app.py scripts/variants and a lot of the sql that was used, such as the procedural sql. I also went to office hours and asked many questions about overarching design questions that do not technically have a best solution - each option has tradeoffs.

Partner 2 Name: Enoch Luk
Partner 2 Responsibilities and Reflection:
I mainly focused on the data preprocessing, writing the majority of the python scripts (in data_cleaning/) that cleaned up the data and created artificial data, and ER to DDL conversion: how to design the DDL based on the ER and our initial csv files. I also helped discuss implementation design choices with Jonathan on a high level, allowing me to write most of the reflection document, RA, etc - the task beyond writing actual sql code, though I did help with some of the procedural sql syntax and application testing.

Other comments

This is the first time CS 121 has had a Final Project, and we would appreciate your feedback on whether you would recommend this in future terms, as well as what you found most helpful, and what you might find helpful to change.

Answer:

We appreciated that the final project was spread across the entire term, but it added a lot of course load to the existing sets. The fire taking a week off our term also compressed the time given by a noticeable amount. Thus, we feel that for future years, the set lengths should decrease by a bit to accommodate to the relatively beefy final project.