Cady Stringer & James Kistner

CPSC 408 Final Project

May 22nd, 2021

Cookies Database

The Cookies Database and Python application are based on a franchised cookie company. This cookie company collects data across all franchises and needs to track store performance, analyze trends, determine what stores and cookie flavors are the most successful, and use these metrics to determine future directions for the company. This company, like all franchised businesses, has corporate employees to support their franchise partners, but these individuals don’t necessarily have coding or analytics experience. This means they may not have the skills to access the data directly using MySQL. Beyond that, the company may not necessarily want them to have direct access to the database even if they did have the necessary skills and background. This could be for a variety of reasons, from confidentiality to protection against accidental modifications. This project addresses all these potential problems with a simple database that tracks store performance, and a matching Python application designed to help internal employees access the data and performance insights at a high level. Although this specific database and application uses a cookie company as a model, this is a challenge that all franchised businesses face. The framework outlined in this project could be modified to fit any franchised storefront business.

This application certainly isn’t the first data analytics application for internal employees; many companies are focusing on the importance of internal data literacy so all employees have access to analytics. Although not all companies are making this shift, many forward-looking businesses are cultivating data culture and encouraging individuals in all branches of the business to make data-driven decisions. In data applications like this one, the information can be presented in a variety of different ways. Some apps display already-analyzed data in dashboards with graphs and charts, while others create reports that users can export to run the analytics themselves. The Cookies Database and application does not include visualizations or a front-end UI because the project goal was to allow users to download raw or filtered data and perform the analytics themselves. The only function that provides analytics insight is the built-in profit queries that present results directly. The choice not to pre-analyze in charts and tables allows users to decide exactly how to use the data, and they can tailor their analysis to answer almost any question they have. Although dashboards with pre-determined charts are easier for the user, a major disadvantage is this loss of flexibility.

These internal data applications are growing in popularity, but because they are confidential and only used by a company’s employees, we were unable to find examples of related public applications online. There are many services that create data analytics products for businesses that can’t or don’t want to do it themselves. One of these is called GoodData. GoodData offers data apps that other companies can buy for their employees or customers to use. They offer a full UI tailored for non-technical users. GoodData and similar companies address the growing demand for easy and inexpensive methods to provide data to non-technical employees.

The primary goal of the Cookie Database’s normalized schema was to break each element of the company and franchises into separate tables that can be used together to calculate overall profit for the company, and drill down to display profits by store, state, date, weekday, store zip code, and cookie flavor. The schema diagram is included in the appendix. The Cookie, Customer, and Store tables are fairly simple. They store information about an entity and can connect to other tables via their respective Primary Key IDs. The Cookie table contains an ID, the cookie’s flavor name, how much the cookie costs to make, and a deleted boolean. The Customer table contains a customer’s ID, first name, last name, sex, age, and a deleted boolean. The Store table contains a store’s ID, name, state, phone number, zip code, and a deleted boolean. The Customer Order and Order Details tables are a bit more complicated. We began with a single customer order table and soon realized that if a customer had multiple cookie flavors in one order, we could not record this without duplicating rows or violating the atomic value rule of normalized tables. To address this issue, we created the Order Details table that connects back the Customer Order table via an Order ID and provides the cookie and the quantity per order. This way the customer, store, and date aren’t duplicated within the Customer Order table if a customer buys more than one cookie flavor in an order. Each order has its own row in Customer Order, and each cookie within each order has its own row in Order Details that connects back to the Customer Order table.

The Cookies Database application has a main menu with the following options: print and display records, add record, delete record, update record, query with filters, and display profits. Any time records are printed, including the first menu option, the user is given the option to export the data to a file. The first option allows users to print tables and query the Customer Metrics view, which displays the gender and age of all customers in the database. In option two, users can add a cookie, customer, store, or order. Adding an order inserts data into both the Customer Order and Order Details tables. The user is prompted for all the appropriate information depending on their selection. Users can soft delete a record in any table by ID number using the third menu option. The fourth option allows users to update cookie cost, customer age, and store name. Users can query any table by ID number, or query the Cookie, Customer, or Store tables by name using the fifth menu option. The sixth option allows users to display total profits for the company, or profits broken down by store, state, date, day of the week, zip code, and cookie flavor. Profit is calculated by multiplying the number of cookies sold by the selling price, $5, and subtracting the cost to make each cookie, which is stored in the Cookie table. This functionality allows users to identify ways the business can maximize profits by location, week day, or cookie flavor.

The framework of this solution includes a combination of Python and SQL techniques to ensure that the database remains in a stable state. Robust error checking prevents users from entering incorrect data types. Proper phone number and extension input is verified using RegEx, and integer and float values are ensured by isInt and isFloat functions. When a user enters invalid input at any point, they are prompted continuously until valid input is entered. Delete, update, add cookie, add customer, and add store all include rollback if the commit fails to ensure the database remains in a stable state and prevent any user errors that slipped through the input error checking. The Cookies Database also includes mechanisms to improve performance and efficiency: indexes were created for each table’s ID column to improve query speed. The pyfiglet and colorama packages were used to make the console application more visually appealing, along with a function that prints a box around menu options. There are stored procedures for frequently-used selection queries, and a Customer Metrics view is queried to display the customer metrics.

The Cookies Database schema lends itself well to many simple queries, but there were a few complex queries that required more thought and planning. For example, the query that produced the total profits by state. The design process for this query was as follows: since each cookie costs a different amount to make, first gross profit needed to be broken down by cookie and by state. So, there was a record for each cookie for each state that had the gross profit for that cookie within that state. Next, to attain gross profit for each state, distinct states were selected, gross profit was calculated with the aggregate sum, and the results were grouped by state. Some joins were performed in this query, but the database schema lends itself well to fairly straightforward joins. After creating these longer queries, it became clear that the database structure could be modified in the future to reduce the number of joins and improve efficiency. As it is now, the schema is almost too normalized, because finding simple information like total sales requires multiple joins. As the amount of data stored in the database increases, join performance will degrade, which could be an issue. In the future, a column could be added that contains the gross profit from each Customer Order so profits could be calculated as rows are added to avoid performing these joins all at once.

**Appendix**

1. Schema Diagram

