The Bookbaggregator - Revamping Bookbagging at Duke

Matthew O'Boyle, Christina Li, Suchir Bhatt, Salil Mitra, Feroze Mohideen

GitHub Repository: https://github.com/christinali/sqlproject

1. Introduction

For a component of Duke's academics as essential as bookbagging, its inconvenience is highly surprising. Each and every semester when students attempt to plan out their academic schedules, the same complaints arise without fail. Whether it is the difficulty in finding out what classes satisfy what requirements, the complexity of having to use many different systems to find out key information about classes, or simply the general inefficiencies of Dukehub, these complaints are as valid as they are arduous to resolve. In light of that, our project was motivated by a desire to simplify and ease the process of bookbagging at Duke. The Bookbaggregator is our attempt to create a system that merges everything one would need when trying to select classes into a single, smooth platform - one that we hope will expedite the process of bookbagging and make it more enjoyable for students to come.

Before discussing our solution to this problem, we first focused on developing a coherent understanding of how students actually bookbag in the present. For Pratt students, there are often very comprehensive 4-year plans that make class selection a fairly simple process, whereas Trinity students often struggle through Dukehub navigating complex requirements (Areas of Knowledge, Modes of Inquiry, Research, Writing, and many more) with little help. To correct for this, most of these students use Dukehub's existing search functionality and then rely on external sources such as RateMyProfessor or word-of-mouth reviews to actually identify which classes are good for them to take. The use of many different systems is not only inefficient, but also highly disadvantageous to students who lack these informal resources (such as independents who often have less access to word-of-mouth reviews), and this directly motivated our desire to synthesize all of this information into a single system. Another flaw with the current process is that it essentially leads to students "guessing-and-checking" their way into classes, which inspired us to try and streamline this process through some sort of recommendation system.

At a very high level, then, our project is a platform where students can go to receive recommendations about what classes are best for them to take in the future, as well as easily view information about those classes and their instructors (or any class/professor in general). Use of our application begins with the sign-up phase, where prospective users create an account and then input information about their year, planned major, and past classes they have taken. This information is then inputted into two recommendation algorithms, the first of which outputs the top 3 classes we recommend they take in order to satisfy their non-major requirements, and the second which outputs the top 3 classes we recommend they take in order to satisfy their major requirements. Moreover, our application provides wide-scale search functionality over a vast database of class and professor history; students can use our app to find out which professors are best at teaching a class, or which semester a class is most frequently offered, or even simple ratings/reviews of a professor or class they are interested in. Details on how all of these components work will follow in the later sections of this report, but ultimately, we believe our app corrects for the largest problems with bookbagging and synthesizes these solutions into a single, holistic system, making it a highly useful tool for students.

The rest of this report will be structured as follows. Section 2 will discuss how we implemented our project, breaking this discussion down into theoretical design decisions and then explaining our actual process of implementation. Section 3 will discuss our data collection process, as well as our created algorithms and their performance. Finally, Section 4 will conclude with an examination of some limitations of our project and propose areas for future work that could correct for or even improve those issues.

2.1 System Design - Theoretical

Beginning with the initial design of our database, there are several decisions we made that are worth explaining. To see this, we can start with our E/R diagram and list of tables, attached in Appendix 1. Most of the tables in here are fairly intuitives - Classes stores all class info we need, and Professors, Departments, etc. are much the same, and any additional details are discussed in the descriptions below each table. There are a couple assumptions, however, as well as 2 major design choices (the first to avoid redundancy and the second for optimization) we made that warrant some additional discussion.

In thinking about how we wanted our data to be formatted, we tried to minimize the number of assumptions we made. For example, we accounted for almost all possibilities where data would be null or missing, as well as edge cases such as classes that were taught by two professors. However, one such edge case assumption we made was that the combination of Dept+ClassNum (i.e. Compsci316) would uniquely identify a class. While this seems reasonable, there are classes with lab and discussion sections, where the numbers diverge in the form of 316L vs. 316D, and we fail to treat these as the same class. Secondarily, we make some limiting assumptions on the structure of our text data, most notably that the names are in English. While this is again necessary for our parsing, many of the foreign language classes or international studies classes offered utilize characters from different languages, which led to certain odd results being displayed on some searches. Other than that, we made very few assumptions about the form of our data, and in fact much of our testing focused on seeing how our app responded to unexpected data inputs.

With regards to design decisions, the first one to talk about is that we keep the Comments table with only a reference to Student_ID, and have Professor and Class reference Comments rather than including Class_ID and Professor_ID as attributes of Comments. This was an intentional design we made to avoid redundancy, and it works because there is no situation in which Comments would be referenced outside of the Professor/Class pages on our front-end. Notably, we discovered and corrected this redundancy by evaluating our schema using Boyce-Codd Normal Form criteria, as we did with several smaller redundancies. Second, we decided early on that we would store User_Email in our front-end implementation and then call all our endpoints using User_Email, a design made foremost for optimization, particularly the optimization of the most common use case of our website. Because User_Email never switches after login, we can store it upon sign-in, and then retain that info across all our pages. With these preliminary considerations of redundancy and optimization in mind, we can now move forward to discussing our implementation of this design.

2.2 - System Design - Practical Implementation

Moving from the lowest to the highest level of our tech stack, the tools we used in implementation are: PostgreSQL (PSQL), SQLAlchemy, Flask, and React.js (with a bit of Node.js integrated in here). We will discuss the motivations behind each of these, as well as how we utilize them in sequential order.

To store our data relations, we decided to use PSQL as opposed to other alternatives such as MySQL, simply because of the fact that all of us had more prior familiarity with PSQL. One level up, we decided to use SQLAlchemy, and this was for a couple important reasons. First, we needed some way to query our PSQL tables and get the results to our server, and SQLAlchemy provided an excellent interface that let us issue SQL queries through Python code (once again Python was chosen for familiarity reasons). Second, SQLAlchemy has a very important built-in security feature that provides protection against SQL injection attacks, which was highly important to our data security. Specifically, SQLAlchemy automatically detects any special characters (such as semicolons or apostrophes) and quotes them, preventing simple SQL injection attacks that would have been a breach otherwise. Together, we felt these reasons warranted us using SQLAlchemy, and so we moved forward using it as our primary means to query our SQL data. One level higher, we chose to use Flask to host our data on a server. Similarly to PSQL, there was nothing too particularly exciting about this decision; Flask is a familiar and common tool used to host a server, and provides a nice REST API for us to send our Get/Post requests to.

Together these 3 tools make up the bulk of our back-end implementation. Important to note here is that these choices allow our front-end to run completely agnostic to our back-end implementation. In other words, all of the final calls that we make from front-end to our Flask server use very simple endpoints, such as getAllClasses() or getProfInfo(), and are then executed however we chose. Beyond just being good practice, this was very important in our practical implementation, as it allowed us to be flexible and adaptive with our data structures and algorithms without having to redo our front-end design every time. We found this to be a particularly important feature of our design, as throughout the project we made several data structure changes in keeping with instructor feedback, and were able to keep much of our front-end design the same despite those changes.

With regards to front-end, we chose to use React.js as a framework to present our data. There are many known benefits of React.js, but chief among those was that react presented us with the ability to utilize reusable components, which prevents duplication and allowed for easy creation of our app. For example, rather than having to create our own star ratings system or dropdown menus, we could use previously created components such as Select and achieve the same outcomes in a much simpler fashion. Coupled with React.js, we also used a NPM module called Axios, which was just a tool we had worked with in the past that simplifies Get and Post requests. Finally, in order to ensure security of our login information for each user, we used Firebase as an authentication tool as per instructor recommendation. Together, these components made up our tech stack and worked cohesively to create the app we envisioned.

Section 3 - Data Collection and Algorithms

To fill our database, we pulled from two main sources of data - the open Dukehub API, and a script to generate sample reviews and ratings. For the former, we wrote several scripts that essentially pulled data on all the classes the API had available, as well as getting all the course info on it, such as which semester it was offered, the professor teaching it, the Trinity requirements it satisfied, and more. We stored this data in CSV files and then used it to populate several of our relations and their attributes. The only missing component was student reviews of those classes, which we had to artificially generate. In theory, this data would be generated upon students signing up for our app, because upon sign-in we ask users to input the classes they have taken as well as their ratings/reviews of the class. However, because our app has obviously not been deployed yet, we wrote a script to generate this data, creating around 50,000 sample ratings and reviews for each class (roughly 20/class/semester).

The reason we wanted to artificially generate this data was so that we could test the performance of our website at scale - with tens of thousands of tuples in our dataset, we could ensure efficient performance of our queries. As expected, when we initially switched to using this vast amount of data, we found some of our queries to be too slow, which we were then able to reevaluate and optimize moving forward. Specifically, our queries to get recommended classes became too slow, because they were synthesizing data from across many of our relations and thus were severely hampered when the size of those tables ballooned. This was primarily due to the fact that, for every single class in the database, we had to iterate over our Taken table (see Appendix A for details) to aggregate the class' overall rating and difficulty, which was extremely inefficient. To solve this, we turned to what we had learnt this semester about triggers, and began to utilize a trigger on the Taken table to keep an updated list of ratings and difficulty for each class within the Teachers table. By using this trigger, we were able to optimize the most common and important queries in our application, which was highly important to the effectiveness of our project.

The only algorithms we needed to create for our app were the two recommendation algorithms, one to identify the best classes for satisfying Trinity requirements and one to identify the best classes for satisfying major requirements. As this was a focal point of our app and a large selling point for potential users, we obviously wanted to ensure this algorithm was highly effective and useful to students. To accomplish that goal, our algorithm combines several important factors to recommend classes to a student, including: 1) the overall rating, 2) the overall difficulty, 3) the student's remaining requirements, and 4) course selections of similar students. The first two are relatively self-explanatory as they simply sort based on score (higher overall and lower difficulty are better), so we will focus on a discussion of the latter here. To account for Trinity requirements, we keep track of which requirements a student has completed (which we have from their class input), and then weight our recommendation based on what requirements a student has not fulfilled. For example, if a student has taken 1 NS requirement but 0 ALP requirements, then, holding all else equal, we would prioritize recommending a class that satisfies an ALP over a class that satisfies an NS. Lastly, to incorporate similar students, we select classes that the current user has taken and compare their ratings of those classes to ratings provided by other students who have also taken those same classes. This allows us to create a pool of the top 10 most similar students to the current user, and we then provide an additional weighting in our algorithm for the reviews provided by those 10 students. This is based off the idea of revealed preference, as we assume that these similar students will presumably be most useful in shaping recommendations for a given student's course path.

Ultimately, in creating our algorithm, we knew we wanted to go beyond simple sorting by good overall ratings and low difficulty. As a result, we chose to expand beyond that by fully utilizing all the data at our disposal to match preferences and recommend based on those, and by personalizing class recommendations down to the very preferences displayed by the users themselves in their ratings themselves. We believe this will be highly important to students seeking to identify which classes they should be taking in upcoming semesters, particularly those who struggle with that process currently.

Section 4 - Limitations and Future Work

A few limitations on the scope of our application warrant discussion. An obvious, previously mentioned weakness is simply that we had to generate artificial data for student reviews, which is inferior to real data, but this could only be fixed through long-term use of our app. Besides that, three other major problems stand out.

First, as mentioned in the introduction, this project is focused solely on Trinity students and expediting their bookbagging process, and we do not account for Pratt requirements. This was primarily because of the fact that many Pratt schedules are already fairly rigid and defined, and so the need for this type of solution was far less, but also a problem we could easily fix. We would actually only need to gather some additional data on what classes satisfy Pratt requirements (in the same way we did for Trinity requirements), and then we could use our same algorithms to produce results.

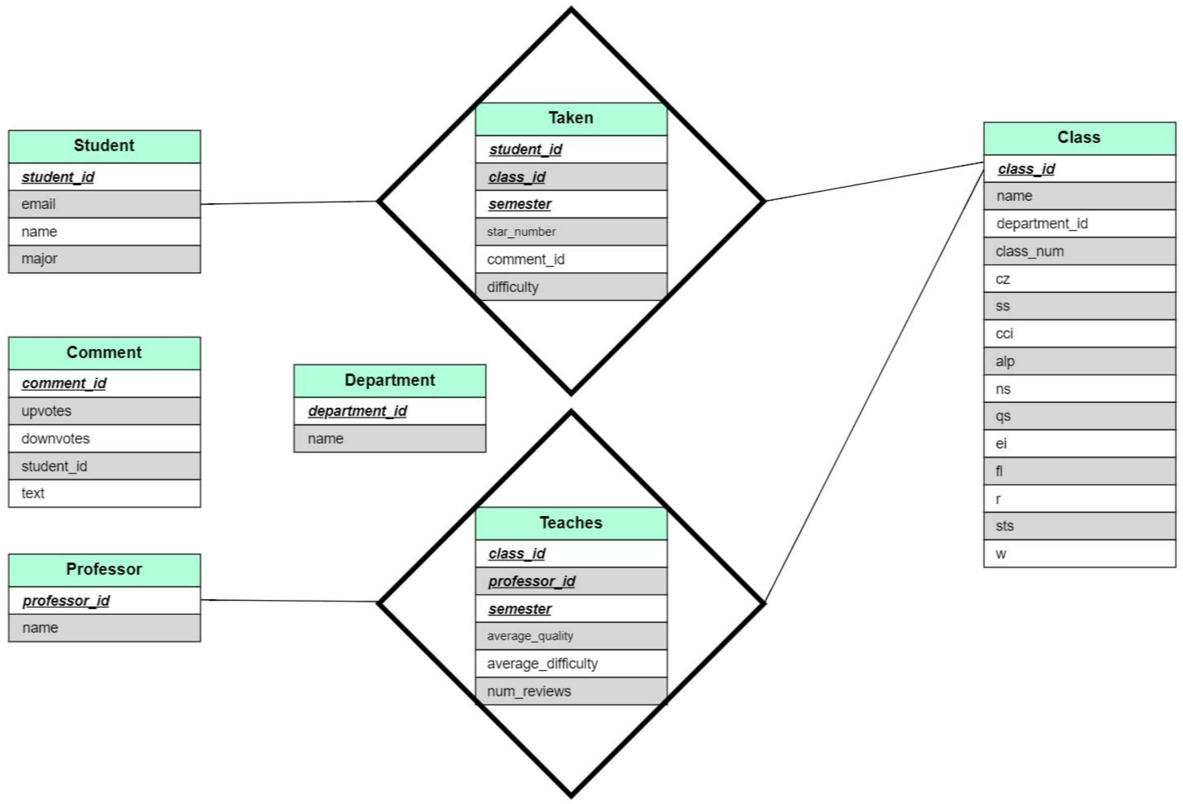
Secondarily, the Dukehub API does not have any mention of prerequisites in class info, meaning that our algorithm runs the risk of recommending classes that a student is ineligible to take. This is mitigated by the fact that our algorithm utilizes the course paths of similar students to generate recommendations, which tends to mean that students are recommended classes in the order of required classes (i.e. every student has to take CS330 after CS230, so our algorithm will very rarely recommend CS330 to a student who has not yet taken CS230). However, as before, we could also work in the future on adding a section for prerequisites that could prevent this possibility from ever occurring.

Finally, and most importantly, our application is not linked to Dukehub. This is a fairly obvious limitation, but has several important ramifications. First, this forces us to require users to input their own class history - if we could integrate with Dukehub, we could directly pull that information from their course history and make the signup process far less cumbersome. Second, were we able to access Dukehub, we could access the prior course evaluations for each class, which would not only fix our data problem, but also make it so that users would not have to input any data whatsoever on login, which would definitely increase the rates of our app usage. Lastly, the fact still remains that students must still ultimately register using Dukehub, meaning that our app can only go so far in easing the registration process.

At the end of the day, The Bookbaggregator is an initial attempt at rectifying the many problems that exist with registration at Duke. To fully accomplish this goal would certainly require future work, but we believe that our application is a strong step in the right direction. Finding engaging classes is essential to a positive college experience, and if our application can help any student with that, we are happy that it exists.

Appendix 1:

E/R Diagram:



Complete list of relations: *(italics indicates keys)*

1. Department(*department_id*, name)
    a. A list of department names keyed by their unique ID (pulled from Dukehub)
2. Professor(*professor_id*, name)
    a. A list of professor names keyed by their unique ID (pulled from Dukehub)
3. Student(*student_id,* major*,* email, name)
    a. Holds all relevant information about a user, including their ID (which we generate for them), major they are studying, email, and name
4. Comment(*comment_id,* student_id, upvotes, downvotes)
    a. Holds all information about comments for class and professor page reviews, with upvotes and downvotes being used to sort comments on each page
5. Teaches(*class_id, professor_id, semester,* average_quality, average_difficulty, num_reviews)

    a. Relates professor to classes, hold all information about what professor teaches which class and for which semester, with average quality and difficulty simply being generated as an average of all the individual quality/difficulty scores in each review

6. Class(*class_id,* department_id, name, class_num, cz, ss, cci, alp, ns, qs, ei, fl, r, sts, w)
    a. Holds information about each class. The last 11 attributes are requirement codes the class fulfills, with values either being 1 if the requirement is fulfilled or 0 if not
7. Taken(*student_id, class_id, semester,* star_number, comment_id, difficulty)
    a. Holds information about each class a student took and during what semester, as well as their feedback on the class