**College Data Visualizer**

Helen Zhang and Michelle Lin

*CS 3265: Database Management Systems*

April 2020

**I. Introduction**

*Overview*

College Data Visualizer is an analytical web application developed with Flask, HTML, CSS and JavaScript for the front-end and MySQL as the back-end data source. The application is intended to give users the ability to filter through colleges based certain attributes such as SAT scores and location.

*Application*

We choose to build a web application because it was the most suitable given our project experience and language knowledge. Moreover, we decided to build a filtering application with the college scorecard dataset due to the lack of resources available for students to be able to preview aggregate data during the college search. Our hope is that the application will allow students to not only gain a helpful snapshot of each college, but also the ability to filter based on a certain attribute with an intuitive interface.

*Dataset*

The application data is derived from a modified version of the most recently updated (March 30, 2020) College Scorecard distributed by the U.S. Department of Education. The College Scorecard contains data on all registered US Colleges, including tuition, cost, and undergraduate enrollment size from the years 1995 - 2018. The years 2012 and 2016 were excluded due to database insertion issues, which are to be discussed later in the report. This dataset was chosen for three reasons. First, the raw data was substantial at roughly 250 MB. Second, the raw data had at least a couple hundred attributes, which allowed us the flexibility to pick and choose those that we felt were relevant and would provide a helpful overview of each college. Third, we thought this data would be helpful for current high school students who are either researching colleges or making college decisions. This is especially important during this time, when most colleges have closed off their campus and visits have been cancelled.

**II. Final Implementation**

*Final Implementation*

The final implementation of the web app was written using Python Flask for the web framework, Flask-MySQLdb for the database connection, and a combination of HTML/CSS, Bootstrap, JavaScript, and jQuery for the front-end and visuals. This app is meant to be run on the web, although we are currently hosting it locally, and it can be used to selectively filter college based on desired data. A “quick-view” profile is also available for every single college. There are 4 primary filtering options available. They will be discussed later in this report.

*Final Database Design*

1. Normalization Process

The database US\_Colleges contains exactly 52 attributes. Here are a couple attribute shorthands to better understand the dependencies:

* UNITID: institutional unit id identified by US Department of Education
* INSTNM: name of institution
* DATA\_YEAR: year data refers to
* CITY/STABBR/ZIP: city/state/zip of school
* INSTURL: college website link
* NPCURL: college financial aid website link
* CONTROL: type of institution

For the sake of space and clarity the simplified function dependencies are listed below:

Violations in **BOLD:**

UNITID 🡪 all other attributes

INSTNM 🡪 all other attributes

**UNITD 🡪 INSTNM**

Thus, a separate basic\_info table was created with only UNITID and INSTNM and UNITID is the main key for all other attributes in the table.

CITY, STABBR, ZIP -> UNITID, INSTNM, INSTURL (and etc)

**UNITID -> CITY, STABBR, ZIP**

Thus a separate geographical\_data was created with UNITID as primary key and CITY, STABBR and ZIP as the only other attributes.

UNITID -> INSTURL

**INSTURL, NPCURL, CONTROL -> all other attributes**

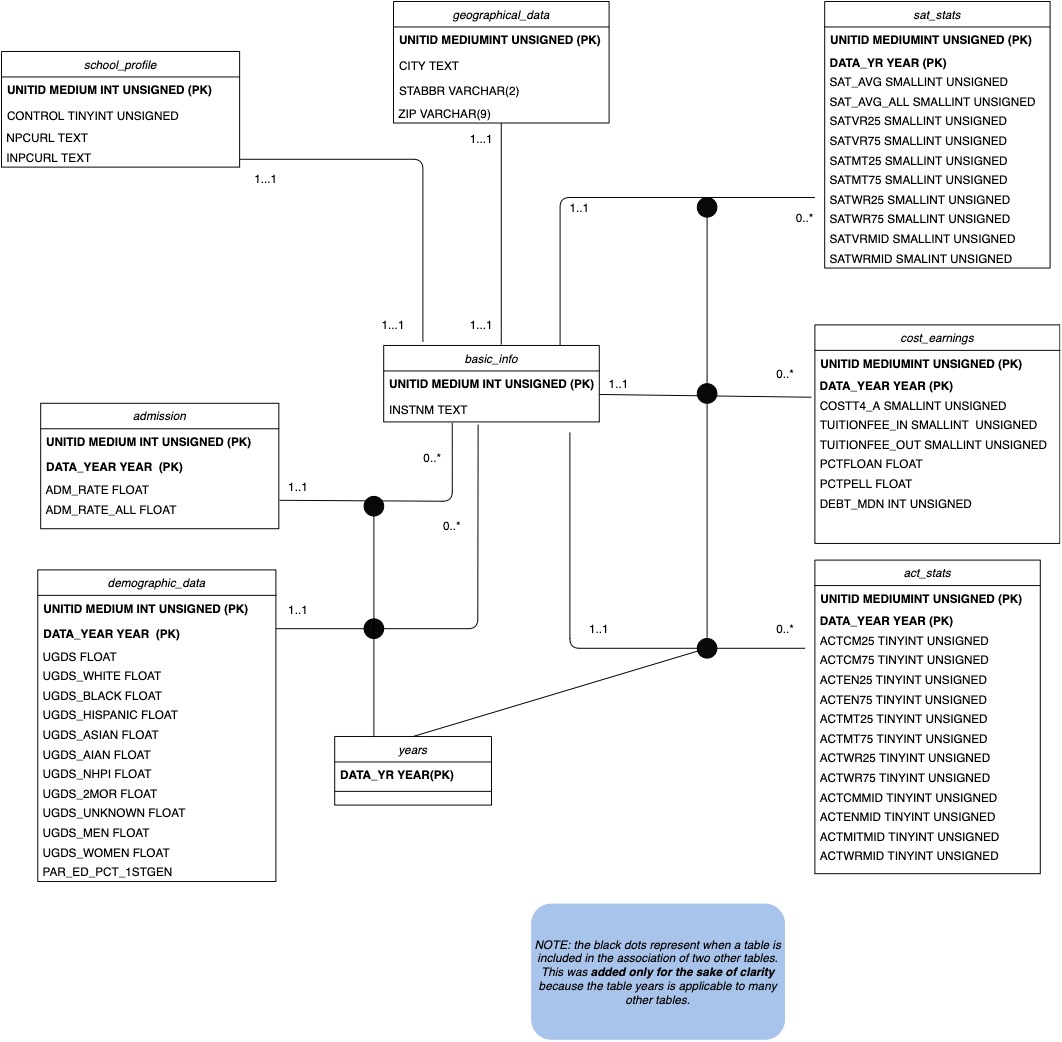
Thus, a separate school\_profile table was created with the four attributes above with UNITID as primary key.

While the attributes in cost\_earnings, sat\_stats, act\_stats and demographic\_data were not directly found to create violations, these tables were created to ease querying and aggregation of relevant data for future data insertions.

2. Final Design

The main attribute for each college is UNITID which is the unit id for each institution. The core table in the database is basic\_info which maps UNITID to the (INSTNM) institution name. Only 2018 data was inserted into this table to avoid discrepancies between INSTNM and UNITID in previous years. This is the case for school\_profile as well as geographical\_data. The other tables sat\_stats, act\_stats, demographic\_data and admission describes data for insitutions for all years listed in the years table. Finding information in these tables requires UNITD and year as there are multiple data points. Allocating a separate table for year is meant to allow for the insertion of multiple years and future additional data.

3. UML with cardinality

**

*Description of Data*

The raw data is disturbed and maintained by the U.S. Department of Education. The data set can be found at this link: <https://collegescorecard.ed.gov/data/>. Since the original data contained a significant amount of attributes (up to hundreds), the data files, each describing college data based on year, was cleaned to produce 20 individual files with only application relevant attributes using python pandas. Each file consisted of approximately 7200 rows of information with each row concerning only one college. The data used for testing consisted only of the 2018 data which had 7243 rows or colleges. This data was used to determine the insertion process from education\_mega (mega table with no normalization) to the normalized tables and determined if there were any dependencies violation between attributes as well as figuring out the semantic of loading the file correctly.

*Summary of Implemented Use Cases*

1. **Querying by Name:**

Users can search for a college profile directly by inputting the name of the college into the search bar. The submission of the form results in a POST request, where the name of the college is sent to the server, and the server then executes a query that returns a JSON object containing the result of the query. The attributes returned are actually a part of a View created in MySQL called “Profile”. The profile page is then dynamically populated using the data returned from the query on the Profile view.

* Query performed: *SELECT \* FROM profile WHERE NAME={{input\_name}}*
* Underlying Database functionality: *VIEW*

2. **Filtering by Location:**

Users can filter a list of colleges by any combination of City, State, and Zip (although they must input at least **one** of the three filters, otherwise the database will not execute a query). On submission of the form, an AJAX POST request is sent to the server containing city, state, and zip. A query is performed using the data sent and the result is then loaded onto the page without having to reload the entire page. The underlying query is actually a procedure that filters colleges based on city, state, and/or zip.

* QUERY performed*: CALL filter\_location(city, state, zip)*
* Underlying Database functionality: *Stored Procedure*

3. **Filtering by Test Score:**

Users can also filter a list of colleges by any combination of ACT Score, SAT score, or both. Users can input desired minimum and maximum values for ACT Score and/or SAT score. They can then select whether to filter based on ACT, SAT, or both. The default values are 0, 36 and 0, 1600 respectively. On submission of the form, an AJAX POST request is sent to the server containing act\_min, act\_max, (bool) act, sat\_min, sat\_max, (bool) sat, and (bool) both. A query is then performed, calling the appropriate procedure, and the results are then loaded onto the page.

* Query performed: *CALL filter\_act(act\_min, act\_max) or CALL filter\_sat(sat\_min, sat\_max) or CALL filter\_both(sat\_min, sat\_max, act\_min, act\_max)*
* Underlying database functionality: *Stored Procedure*

4, **Filtering by Cost:**

Users can filter by cost and limit the number of rows desired. The result set is different in that it contains the demographic/cost data of the college, rather than score range and location. Users can move a slider to indicate minimum and maximum cost, and then specify the number of rows to return (50, 100, 150, 200). An AJAX POST request is then sent to the server containing the values in the form and a call to the stored procedure is then performed, returning a view. The results are then loaded onto the page.

* Query performed*: CALL filter\_cost(min\_cost, max\_cost, limit)*
* Underlying database functionality: *Stored Procedure, View*

**III. Illustration of Functionality**

**To set up:**

Since the web app is hosted locally, you will need a working Python environment in which you need to install Flask and flask-mysqldb if you haven’t already. Once those are installed, configure the database connection settings in app.py to match those of your machine. Start the database server. Navigate to the root directory, run “flask run” in the terminal, and go to localhost:5000. The web-app should be running there.

**1. Find College By Name:**

Input the name of the college that you want to view in the search bar and press enter. The input is case-insensitive but space-sensitive. Also, the input must match the name of the college **exactly**. This means you cannot type “Vanderbilt” – you have to type “Vanderbilt University”. If the college is not found, then it will return a 404 Error and you will have to try again.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

2. **Filter based on location**:

Enter any one, two, or all of City, State, and Zip and press Enter. If you do not enter anything, then no results will be shown. The city name must match the City directly and is case and space sensitive. Press the search button. Due to the nature of the query, only the first 100 results will be returned.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Filter by city only Filter by city and state

3. **Filter by Score**:

You may specify any combination of minimum ACT, maximum ACT, minimum SAT, and maximum SAT. If you do not enter anything then the default values will be 0, 36 and 0, 1600 respectively. From the drop-down menu, select the type of filtering that you want to do. Press the search button. You must select a filtration type. If min and max are equal, then nothing will be returned. Due to the nature of the query, only the first 100 queries will be returned.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

ACT only SAT and ACT

A screenshot of a cell phone

Description automatically generated

Left: ACT and SAT

4. **Filter by Cost:**

Drag the two ends of the sliders to specify the minimum and maximum cost that you want to query for. In the dropdown, select the number of results that you would like to see. Press the search button.

A screenshot of a cell phone

Description automatically generated

Cost range $0-$1095. I wish Vanderbilt were included in this list!

**IV. Summary Discussion**

*Challenges*

On the database side, the biggest challenges were cleaning the data, loading and inserting the data. Since each individual files had hundreds of attributes with thousands of rows, it was difficult to determine relevant attributes as well as produce completely new csv files for each containing the desired attributes. To resolve this, attribute inclusion was decided based on personal experience and the file cleaning was automated using python pandas. The test file 2018 data some anomalies such as non UT-8 encodings and duplicate keys that made it difficult to load the file into the education\_mega table. This was resolved by deleting some rows of data that caused error messages. The insertion process was relatively smooth for just the 2018 data. However, inserting into multiple tables for 20 files was tedious as the insertion statements were needed to be written for each file and each file needed to be check for duplicates keys. To mitigate, atom was utilized to replace each instance of the year data and replicated the insert statements. However, each insert still needed to be manually tested.

On the front-end side, the biggest challenge was configuring the API endpoints as well as validating the input data to make sure that it was in a suitable form to send to the database. Since this was my first time building an app using Flask, I had to figure out how to use Flask in general. This was done by reading the documentation. In terms of making sure that the endpoints worked, I used Talend API Tester to put in dummy data and test the endpoints. I also used plenty of print statements and stackoverflow. Overall, it just required a lot of trial and error, starting and restarting the server, refreshing the webpage over and over again, and multiple print statements.

*Division of Work*

Michelle was responsible for the database side of the project this included: normalization, database design, cleaning raw data, database analysis, UML diagram and the majority of the report. Helen was responsible for the front-end and back-end including: environment setup, connection between all three interfaces, front-end design, retrieving data from the database at the back-end, error-handling on the front and back end, database analysis/modification, video walkthrough and some of this report. This was a fair and equal split because setting up the database and the front-end both required significant amounts of time. Since Helen was responsible connecting all three interfaces together as well as the video, Michelle created most of the project report. Contributions outside of one’s delegated responsibilities were also made.