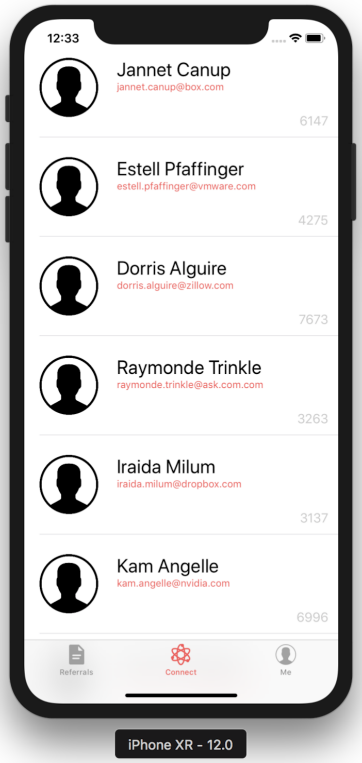
Fermi
COMPSCI 316 Project Milestone 2 - Progress Report
Davis Booth, Thara Veeramachaneni, Elliott Bolzan, Jamie Palka, Emily Mi

Note: All of the code we mention in this document is available on our Github repository (https://github.com/elliottbolzan/Fermi). Gradescope did not allow us to submit this code properly.

Note: In order for our front-end to function properly, our Google Cloud VM must be running. In order to preserve the limited credits that we have on our Google Cloud account, our VM has been stopped. We can easily start our VM instance back up at any time. Our app also currently only runs on the simulator in Xcode as we have not sent it up for approval by Apple to be hosted on the App Store. If you wish to see our app's front end functioning properly, feel free to shoot any of us an email and we will start the VM back up so that the app can be properly tested in the Xcode Simulator.

Proposal Updates



As we began to further implement Kondria, we realized that WebMD has a beta product whose functionality significantly overlaps with Kondria's. As a result, we have decided to move forward with a new iOS application called Fermi.

Fermi provides a sleek, simple interface to easily connect users seeking referrals to companies with other users who can potentially refer them to those companies. All users can both request and give referrals, and each user will have two scores: their success rate in giving referrals (the percentages of users who they referred that actually got offers) and their success rate as referral receivers (the percentage of referrals they got that actually turned into offers). As an app solely dedicated to the distribution of referrals, Fermi streamlines the process, making it quicker, easier, and less awkward. Users input their educational and professional experiences and can search for other potential referees by these characteristics. Our home page will also suggest users to reach out to based on similar educational background groups. With the simple click of a "refer me" button, an automatic message will be sent to the user asking for a referral. Once a referral is actually made, the referrer will signify this in the app. This referral will automatically be registered as "not converted to an offer" and the referee's "success rate as a referral receiver" will go down. If the referee receives an offer, they will register this into the app and their score will go up. The referrer's "success rate in giving referrals" will also go up.

We will answer questions 6 through 8 directly before addressing additional matters and examples pertaining to our progres report below.

6: Application Platform

Our back-end platform consists of a virtual machine run in Google Cloud. On this virtual machine, we run a PostgreSQL database and a Flask webserver. Our Flask server has an external IP address and can receive requests from the Internet at-large. Upon receiving these requests, the server queries our PostgreSQL database and serves a JSON response to the caller.

Our front-end consists of an iOS 12 application developed in Xcode using Swift 4. From this application, we make HTTP requests using the Alamofire framework to our Flask server. Upon receiving a response, our iOS application translates the JSON to local, Swift structs. These Swift structs are then displayed to the user using a variety of Swift components.

7: Production Dataset

Because our production dataset is based on users, we chose to randomly generate users and the relationships between them. To do this, we collected extensive data from the US Census (male first names, female first names, last names), Github (jobs in technology), and found a list of universities online.

Based on this source data, we wrote a Python script that generates an arbitrary number of users and referrals between them. Currently, we are creating 10,000 users and 50,000 referrals. This script creates an equal number of men and women; randomly pairs first names with last names; associates each user with a company and position; generates an email for each user based on their name and company; randomly generates an educational history for each user. This last detail is more complex: a user can have completed either:

- No college degree;
- A Bachelors degree;
- A Bachelors degree and a Master degree;
- A Bachelors degree, Masters degree, and PhD degree.

This script, after assembling these data points, produces a load.sql file which can insert this data into our database using the following command: psql fermi -af load.sql. Because the file inserts many, many records, it takes a few minutes to run.

Note: we have tested our script extensively, and it can convincingly create millions of random users and referrals. Because at this time, we are only testing our application, we have chosen to create fewer users to make for a speedier development process.

8: Testing SQL Statements

Our sample data consists of 10,000 users and 50,000 referrals between these users. Within the context of this data, our queries' performance is impressive.

Consider one specific query, which enables us to retrieve all the referrals a specific user is related to (whether they have referred or have been referred). In our timed PostgreSQL tests, our queries ran in 5.188 milliseconds on average. While this number is difficult to contextualize, we have come to the conclusion that our database schema and queries easily allow us to provide satisfactory performance to the user. Responses on the order of single-digit milliseconds are fast enough.

Assumptions Made

- People can only give referrals within the company they are currently working.
- It is acceptable for referral-giving to be streamlined in this manner in the real world.
- People seek referrals to the companies they are applying to work for.
- Working individuals are willing to give others referrals.
- People are incentivized not to make too many referrals because their score drops when they as for a referral and is only bumped when they get a job.
- People are incentivized to give more referrals because every time they do, it improves their score.
- The education and experience of a job-seeking individual will impact the decision of a working individual to refer them.
- Referrals positively impact recruitment process for a job-seeking individual.
- Referrers benefit from the person they referred being hired.
- There is a static set of companies and universities that are.
- For now, users can only filter their searches by user name, company name, and university (and joins of those attributes)

Platform and Progress

The platform we selected is an iOS app, written with Python for the backend API, Swift for the iOS development, and PostgreSQL for the queries. We have already created a new database schema, E/R diagram, and user-interface design for Fermi. We have also randomly created a large production dataset, written SQL queries for creating user profiles, calculating referral success rates, searching by education and company, finding suggested referrers, updating user profiles, and returning user profile attributes. We have tested our queries (with results displayed in the test-production.out file), and created Data Access Object (DAO) classes, Data Transfer

Object (DTO) classes, and a “Hello World” Flask webserver that provides an array of users from the Fermi database to callers in JSON.

Going forward, we still need to:

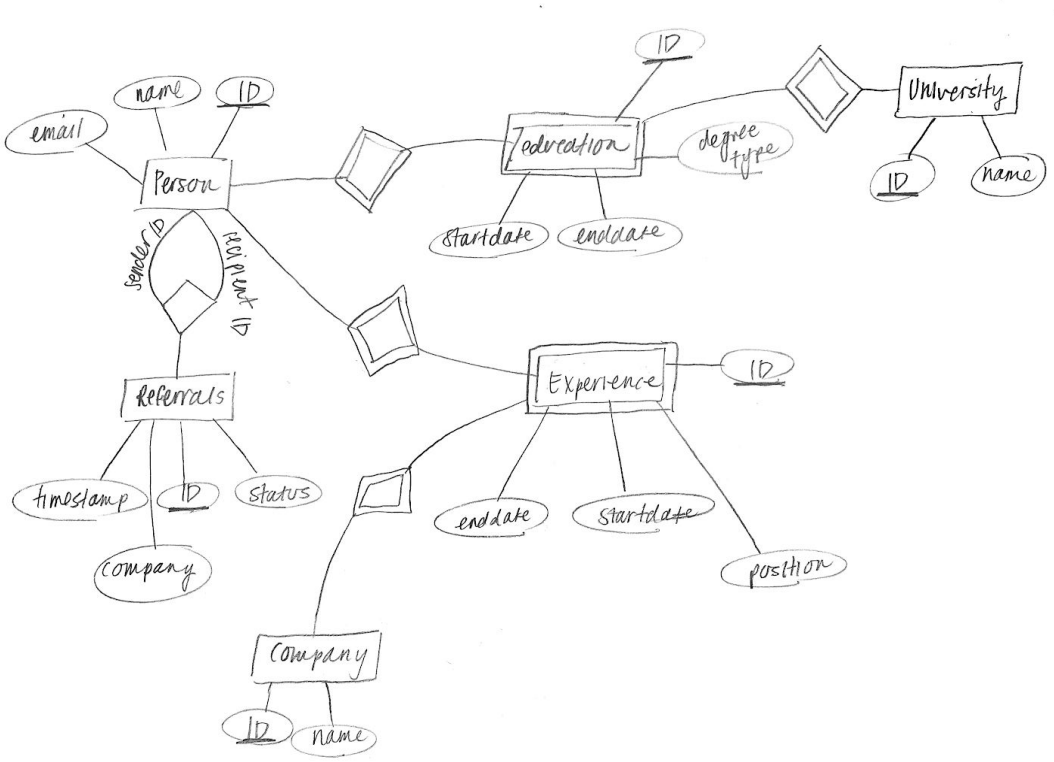
- Build out our front-end’s views. We have prepared a number of these, but have not yet implemented them;
- Create specific endpoints in the back-end for each query that we have developed. So far, only one of our queries can be called from the front-end, and we need to extend this to all of our queries.

Performance

See our answer to question 8 above. In our sample database, we have 10,000 users and 50,000 referrals between these users. Running timed tests on our PostgreSQL database, we found our performance to be impressive. One of our queries, for instance, enables us to retrieve all the referrals a specific user is related to (whether they have referred or have been referred) in 5.188 milliseconds on average.

Because we found this performance to be satisfactory from the user’s perspective, we chose, at this point, to not fine-tune our databases performance using indexes. That being said, as our application continues to develop, we will test it on millions of randomly-generated users, to determine whether additional tuning is necessary.

Updated E/R Diagram



Tables

Person(<u>id</u>, name, email)
Referrals(<u>id</u>, timestamp, company, status)
Experience(<u>id</u>, <u>company_id</u>, startdate, enddate, position)
Education(<u>id</u>, <u>university_id</u>, startdate, enddate, degreetype)
University(<u>id</u>, name)
Company(<u>id</u>, name)

User Interface

SQL Queries

So far, we have written queries to provide the following information:

- Create a user profile;
- Calculate referral success rate;
- Search by education and company;
- Find suggested referrers based on education;
- Update the attributes in a user profile;
- Return attributes in a user profile .

Anatomy of our Data: from Creation to Formatting

The data originates in a PostgreSQL database located on our Google Cloud virtual machine (external IP address: 35.196.36.216). A Flask webserver on this machine serves JSON responses to port 5000 when requests are made to certain endpoints (currently, only /example is active). To do so, our Python code models the contents of our database using Data Access Objects (DAO) and Data Transfer Objects (DTO), before transforming collections of DAOs to JSON.

Requests are made in the iOS front-end using the Alamofire framework in Swift 4. Requests are made to an endpoint, like "http://35.196.36.216:5000/example". In our "Hello World" example, the response is converted from JSON to an array of Swift structs called Persons. In turn, this array of Persons is passed to a UITableView to be displayed. Once the array is populated with all of the Person objects, the TableView in the Homepage is triggered to reload. Upon reload request, the methods heightForRowAt, numberOfRowsInSection, and cellForRowAt execute.

HeightForRowAt determines the height for each of the table cells to be populated. Since we want a static height of each of the table cells, we decided to make each table cell to be of height 130.

From here, the numberOfRowsInSection function runs and returns the length of the data array, which contains all of the Person objects that need to be imported. It creates this many cells.

Next, the cellForRowAt function is called for every cell to be created. This method grabs the Person object in the data array at the current index and creates a new InitialTableViewCell and then calls the commonInit function on each of the table cells to populate them with the name, id, and email of the user.

The commonInit function of the InitialTableViewCell class sets the text of the respect UILabel outlets to be equal to the proper components of the Person object (name, id, and email). Finally, this information is formatted by the InitialTableViewCell.xib file.

This is repeated for however many table cells are necessary to be created (which is based on however many people are returned by our server).