CIS 550 Final Project Report
Group 26

• Group Member:

Boyi He, Yizhou Luo, Yan Gao, Yinjun Wu

• Abstract

The objective of this project is to build an integrated databases related to cities, including geographic, education and corporation information. The application provides students, researchers and companies with comprehensive information in a specific region in order to make optimized decisions.

• Modules and Architecture

The whole application is designed with model-view-controller pattern.

How it works in general: We applied the MEAN stack to develop the whole project. We have servers running on our PCs answering requests from clients and fetching data from the databases we populated. When a client selects an option on the webpage, the server would send a query to the back-end database. When the results are received from the database, the JavaScript code, acting as the controller, would fill the jade files with the requested data, which would then render the resulting pages to the users.

The back-end database: We populated our relational SQL database on AWS, and NoSQL database (mongoDB) on mLab. There are more than 5 relations which can provide very detailed information on the topic we choose (cities, schools, and companies). We populated them according to the principals such as BCNFs.

The servers: The server was developed with MEAN stack as described. The NodeJS code works perfectly with the database and the jade files in a coherent way. It also has great scalability and is able to cope with large scale of users.

The view: For the front-end, we uses jade files to render pages and present the data in a clean and precise manner. Since the queries and the data of our topic are highly related to geographic information, we presented the results with the Google Maps JS API.

• Data Instance Use

Data instances in SQL (MySQL):

For the SQL database system, we built several tables that represent the companies, schools and cities, etc. They provide detailed information on the corresponding relations. In addition, we separated some tables,

for example the table for Colleges, in order to reduce data entries with duplicated attributes and achieve a clean structure.

Data instance in NoSQL (MongoDB):
In terms of unstructured dataset, the DBpedia (https://http://wiki.dbpedia.org/) was used for our project. DBpedia contains all the information from wikipedia and is updated regularly. In our project, we extracted some useful information, such as the types and the introduction of certain organizations. The introduction is especially useful for our application since there exists nearly no other single data source can cover such comprehensive information for various organizations.

• Data Cleaning and Import Mechanism

Data cleaning and importing for NoSQL:
The original dataset in DBpedia is in RDF form, i.e. a triple (subject, predicate, object). It is impractical to download the whole DBpedia dataset since the size is up to 100 GB. We only extract small parts of it for our application, which is mainly the information about various organization appears in other data sources.

In terms of RDF format, it can be imagined that triples with the same Subject (i.e. in our application, it is organization) can compose a table in SQL and Predicate is often corresponding to the attributes in SQL while Object is often corresponding to the values of that attribute. However, in DBpedia, the same predicates can appear multiple times and be associated with different objects for the same subjects. Besides, some predicates may exist in some subjects but disappear in others. So we determined to deal with DBpedia data in NoSQL way, i.e. using MongoDB. So the triples with the same Subject in DBpedia are stored in the same document in our MongoDB instance and the predicate and the object are treated as key-value pairs for each subject.

In our application, since we only care about some information of the Top 1000 companies in America, for each company (suppose the name is c_name), we use a simple sparql query to extract the information from DBpedia:

```
SELECT ?property ?hasValue
WHERE {
  { <http://dbpedia.org/resource/c_name> ?property ?hasValue }
}
```

This query means that it will return all the predicate (?property) and object (?hasValue) for a specified company. In order to answer the question “Divide the companies to different commercial areas like finance and technology.” and “Provide companies’ information of relevant job market, and competent companies”,

we are only interested in the “name”, “type” and “introduction” predicates in the query result, which will be inserted into our MongoDB instance. Since each company can have multiple types, which leads to an array of values for the attribute “type”.

Data cleaning and importing for relational SQL:
The original data we obtained are messy with different format such as RDF, text file, CSV and so on. However, using some C++ scripts we wrote ourselves, we are able to deal with those data and convert them into queries that can be executed. The C++ scripts are able to detect any format violations in CSV files, correct them and convert them back to the SQL query language that can be directly used to create tables and insert data.

• Uses Cases

User Login:
We enable user login and registration by using an authentication middleware-Passport. User are able to login, register and sign out. They can check their info in the setting page.

- Regular login

We create a Mongodb model-user on our configured Mongodb to store user information. Add a folder called passport to configure passport. We use login strategy and registration strategy to handle users and create their entries in Mongodb. Each strategy created using passport.use() is an instance of the Local Authentication Strategy of Passport. We use connect-flash node module to help us providing flash error messages.

- Social networks login

We also enable Facebook and twitter login for users, which are implemented through passport-facebook and passport-twitter models (strategies). We created a facebook app and a twitter app on their developer website. Using the requested appID, appSecret and defined callbackurl as configuration information, we are authenticated to connect to Facebook and twitter.

Cities Display:
In the homepage of our system, all the related cities (or towns, in which companies or universities are located) are displayed (default display center is New York) using Google map API, which can provide a high-level picture for users. In this UI, the name of cities or towns will be shown in the infobox once users click on these cities in the map.

Correlation Search:
Correlation search is the option to find the top cities that have the most schools and companies at the same time within a given state. It is designed to help users to analyze the correlation between the geographic distributions of schools and companies, and potentially select an ideal location for residence. The user is

required to give a state abbreviation, and click “Show Top Cities”. The resulting cities would then be displayed as pins on the customized Google Map. If the user clicks on a resulting pin, a message box would be shown to provide the name of the city, the number of schools and companies within the city. The “Return” button allows the user to go back and start a new query.

College Choice:
College choice enables the users to search for companies within a certain region. Users are able to choose a state in the web page. After their choice is submitted, the database will return the colleges and relative information inside that state. The information includes the name, the tuition, the jobs, the dormitory number. This can greatly help them in search for the ideal college.
There is also a Google search bar in the web page. They can search for more information with the search engine.

Closest Companies:
Closest Companies is an option for users to choose fitable colleges based on their geographical locations, which refer to the surrounded companies. Given a college name from the list provided in this page, we reply 20 closest companies around this college calculated from relative latitude and longitude.

MongoDB Queries:
MongoDB query is used to filter the types of different companies. Given a certain company type, the system can provide the relevant companies that match users’ interests. Related introduction information about those companies will also be returned to help user have a better knowledge of those companies.

• Optimization techniques employed (indexing, optimization)

View and Query Optimization
We created views and stored that in memory in order for frequent access and query. We also did some optimization in the database structure itself.
For example, since we are striving to achieve BCNF in the the College relational database, we have separated some information and created two different tables. That greatly saves the disk storage as well as access efficiency.
On the other hand, some information across these two tables are constantly being queried. So we created a view to store that. In this way, the efficiency problem within the table College itself and the outside queries have been perfectly solved.
As a great result, although some of our dataset are very large (which originally should have consumed a lot of time), we are able to reduce the query return time to a very small interval. This also gives a better user experience.

Primary Key Selection for Better Scalability

We cautiously selected and assigned the Primary key while constructing the tables in our relational database. This actually improves the scalability. Take the *ranking of company* as an example, originally the ranking was the primary key since as normally thoughts, it would be okay to use auto-increment. However, we aim at better scalability and hope the function of ranking can be independent and is able to change in real-time with the existing data.

Indexes:

city (local_name)

- An index on city name

company (City)

college1 (CITY, STABBR)

- An index on the location (city name, state) for each college.

• Technical specifications

MySQL Queries:

For correlation search, given a state abbreviation, such as 'PA', the query is to take the intersection between the top 30 cities with most schools and the top 30 cities with the most companies. Since MySQL does not support intersection, a join operation is taken instead. For each city, the name, geographic location, number of schools and number of companies are returned, and passed to the jade file to displayed through the Google Map API.

```
SELECT TC.local_name, T3.geo_lat, T3.geo_lng, TC.c_count, TU.u_count
FROM
        (SELECT T1.local_name, count(*) AS c_count
        FROM city T1, company C
        WHERE C.City = T1.local_name AND T1.State = 'PA'
        GROUP BY T1.local_name
        ORDER BY count(C.id) DESC
        LIMIT 30
) TC JOIN
        (SELECT T2.local_name, count(*) AS u_count
        FROM city T2, college1 U
        WHERE U.CITY = T2.local_name AND U.STABBR = 'PA'
        GROUP BY T2.local_name
        ORDER BY count(U.ID) DESC
        LIMIT 30
```

) TU ON TC.local_name = TU.local_name, city T3
WHERE TC.local_name = T3.local_name AND T3.State = 'PA';

For college choices, users first choose the state they want to search. After receiving the state, the web page render the corresponding colleges in that state and all the related information about that college. In the NodeJS file, there are two functions that render the result web pages. There is a variable marking the state of the current user, ie. if the user enter the state already then render the colleges with respect to the state. Otherwise, render the original page that allows them to select the state.

For the search of closest companies, given a college name, we join the tables of Company and College, sort the entries by the distance between the companies and the given college, using their latitudes and longitudes, limit the result by top 20.

For NoSQL data:
Since mentioned before, we extract non-structured data from DBpedia, which is stored in MongoDB and used for answering two questions:
"Divide the companies to different commercial areas like finance and technology." and "Provide companies' information of relevant job market, and competent companies."

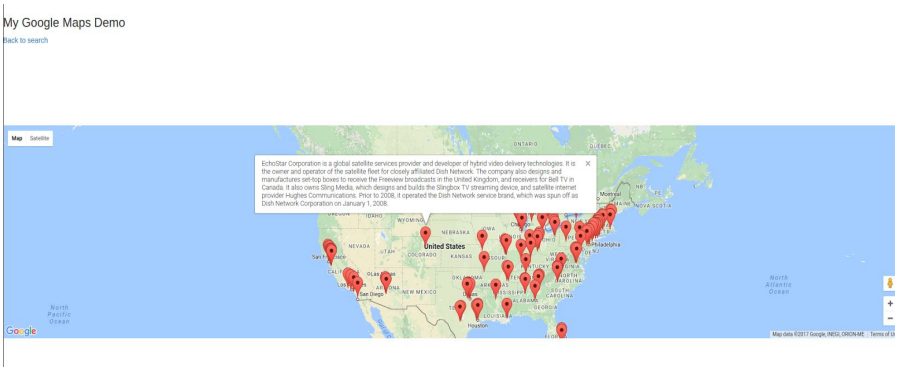
The corresponding MongoDB query is:

Company_type = (user_input);
db.user.createIndex({ type: "text" }); db.user.find({},{_id:0, name:1},{ $text: { $search:Company_type } })
And
key_word= (user_input); db.user.createIndex({ introduction: "text" }); db.user.find({},{_id:0, name:1, introduction:1},{ $text: { $search:key_word } })

In our implementations, since the input of the two users are the same, we combine the two queries together and present the result in the same web page. Given a user input, i.e. the type of companies that the user is interested in, we use the text search query in MongoDB, which is building index for the attribute (type, introduction) first and then launch text search within the values of that attribute.

We give a query example here. Suppose a user is interested in companies which is about business, he or she can go to the site http://localhost:3000/mongodb_query and the query results will be shown directly on the Google map:

Here, the location of each company is shown as marks while the introduction of it is shown in the infobox of each location.



• Special Features that you'd like to highlight

We used Google map API to show the locations of different cities while we render the results. When the mouse click on a pin, detailed information will appear in the message box.

The application allows users to create their own accounts and log in. For future updates, we could allow authenticated users to view some specific data provided in detail, to make better decisions. In addition, their search history could be stored on our server, with which we would be able to build recommendation systems based on their preferences.

We also authenticate users to login with Facebook, Twitter accounts, which allows them to have multiple choices instead of a single account.

We integrated a Google search API that allows the user to search for information online in case they do not get exactly what they are looking for from our database.

• Division of work between group members

Yizhou Luo: Wrote C++ scripts to generate queries to create and insert into database, built the college_choice web page search for college info, and created a simple google API to search online data.

Boyi He: Data cleaning, SQL query design, indexing, optimization, and the feature of correlation search.

Yinjun Wu: Data extraction and cleaning from DBpedia, Google map API basic features and NoSql query implementations.

Yan Gao: Data cleaning of the city and college database. Implement regular and social user login. Implement the closest_company feature.