Final Project Report

Team Members

Zachary Thomas (thomasza@oregonstate.edu)
Daniel Garlock (garlockd@oregonstate.edu)
Peter Dorich (dorichp@oregonstate.edu)
Alex Edwards (edwarda3@oregonstate.edu)
Ehmar Khan (khaneh@oregonstate.edu)

Database: https://tools.engr.oregonstate.edu/phpMyAdmin/index.php
Username: cs440_thomasza, Password: 2NuJSk7k3hYtMiZ, Server Choice: classmysql

Website: http://web.engr.oregonstate.edu/~thomasza/CS440/index.html
This website has pages that run some SQL queries over our database and displays the results in tables. You can access these pages from the tabs at the top of the site.

The problem

With so many sources to find restaurant reviews, that have different rating systems, formats, and biases, it can be difficult to determine the ideal restaurant to go to. One website may rate an establishment as a five-star restaurant, while another site may rate it on a point scale, or some different system. The main problem is that inconsistency between websites does very little to help people decide where to eat.

In addition, there are multiple options for restaurant rating websites out there. Now a consumer is forced to first decide which website they'd like to use, and then use that to decide which restaurant to eat at. What if restaurant "A" only has reviews on Tripadvisor, and restaurant "B" only has reviews on Eater? If an individual is trying to select between restaurants "A" and "B" this requires them to understand the different formats of the sites and differing grading criteria.

The Solution:

We designed a site that uses our database to display standardized reviews of restaurants from multiple sources. Our database has standardized ratings and also has tables for menu items and cuisine categories. To standardize the review scores per site, we converted the site's score into our own scale. All values are converted to a 0-100 rating scale.

The database is split into the following tables and attributes:

Rests: Stores basic restaurant information.

- **rid:** An incremented ID to easily identify restaurants.
- **rname:** The restaurant name.
- **raddress:** The address of the restaurant.

Rating: Stores information about restaurant reviews. (1 to 1 relationship)

- **rid:** Foreign key that helps match reviews to the Rests table.
- **roverall:** This is the overall rating of the restaurant. The rating is from 0-100.
- **rfood:** This is the rating of food quality. The rating is from 0-100.
- **rservice:** This is the rating for the service. The rating is from 0-100.
- **rvalue:** This is the rating related to price of the food. The rating is from 0-100.
- **ratmo:** This is the rating of the atmosphere. The rating is from 0-100.

**Not all sites may have all aspects of the rating. Allow NULL.*

Meals: Stores information about specific meals.

- **mid:** An incremented ID to easily identify meals
- **mname:** Name of the meal.

Rests_Meals: M-to-M Relationship of restaurants to meals.

- **rid:** Foreign key that helps match to the Rests table.
- **mid:** Foreign key that helps match to the Meals table.

Cuisine: Stores information about specific cuisine (e.g. Mexican, Chinese).

- **cname:** The name of the type of cuisine.
- **cdesc:** A short description of the cuisine.

Rests_Cuisine: M-to-M Relationship of restaurants to cuisine.

- **rid:** Foreign key that helps match to the Rests table.
- **cname:** Foreign key that helps match to the Cuisine table.

Data Sources:

- TripAdvisor, Yelp, Zomato, Eater.

Collecting the Data:

API access:

Some data sources allow us to retrieve restaurant information from an API. With API access, we can easily iterate through the results and extract the information we need from their data without downloading or writing unnecessary data that will be useless for our schema.

Sites like Zomato have easy API access for free, but it comes with the problem of daily access limits. To get around this, we try to get as much information as possible from a single request. Zomato's API returns a maximum of twenty restaurants for a single city search request, with an offset option. So to get a city with 1000 restaurants for example, getting 20 restaurants per request, we can get all of them within 50 requests. With a daily access limit of 1000 requests, we can get 20,000 restaurants in a day, which is repeatable over more days.

This allows very rapid data collection. The primary benefit of using an API is the ease of data cleaning. Because we get information through JSON, we can simply keep the elements we want and ignore everything else with a few simple lines of code.

Direct Dataset:

Some datasets out there are available for direct download. Yelp, for instance, has a subset of their business reviews and user data available for download. They allow free use of their dataset for any personal or academic purpose. Yelp even encourages people to use their data to look for any helpful trends as part of a competition.

Although this is by far the most convenient way to acquire restaurant data, there are two notable issues with it. First, since the dataset is only a subset of their entire database, we have no idea what they omitted and how much data they are supplying us. Second, since Yelp doesn't only review restaurants, but businesses too, the dataset may have to be trimmed down significantly in order to limit our search to relevant establishments.

Web scraping:

Web scraping offers the ability to customize the data that we collect, at the cost of having to learn and build sitemaps to collect that data.

What ultimately made web scraping the most appealing data collection option is that it allowed us to store rows that preserved the relationship structure of our tables. For example we wanted a many-to-many relationship with our restaurants and menu items, for example many restaurants may serve hamburgers, and cheeseburgers. With data scraping we can have a row for each menu item of a restaurant that also stores the restaurant name and address on each row. While we are inserting it into the database we may have a lot of redundant data, but after it is integrated all of the redundant data is removed.

Cleaning the Data:

We used excel to clean the data. Sometimes cleaning data was as easy as multiplying a 0 to 5 integer rating column by 20 to get our 0 to 100 rating. Other times we had to strip characters or combine information from multiple columns. One additional challenge to cleaning data was that since we collected data from multiple sources, each separate source required different methods to clean.

Integrating the Data:

This was the greatest challenge. Ultimately we choose Web Scraping as our prefered data collection method as it helped make integration easier. Also cleaning the data was key in making integration possible. The first step after cleaning the data was importing the new data into the database as a single table.

These tables look something like this:

rname	radd	roverall	rfood	rserv	rvalue	ratmo	cuisine	meal
9er's Grill	5870 New Territory...	90	80	75	90	90		Chicken Tenders
9er's Grill	5870 New Territory...	90	80	75	90	90		Carrot Cake
9er's Grill	5870 New Territory...	90	80	75	90	90	American	
A Pizza Mart	2525 6th...	60	70	50	60	60		Pizza Mart Calzone

Note that some of the data is redundant. But this helps make integration easier, and the redundant data will be removed once we have completed integration.

Once we have the table inserted we created two procedures to distribute the data across the database. The first one we named expand_row. Expand_row takes a single row from our new table and inserts data into the appropriate tables while also saving the id returned from insertion. Once it has inserted into rests, meals, and cuisine it will have new values for rid, mid, and cid. It will then use these values to insert into Rests_Meal, and Rests_Cuisine. This is the key to preserving the many-to-many relationship that these meals and cuisine have with the restaurants. The next procedure we named unload_table. Unload_table simply calls expand_row for each row in our new table. So once the new table is in the database and ready to be distributed we simply call unload_table once, and then we drop the new table to prevent from storing redundant data.

Some illustrations to help show the integration process: