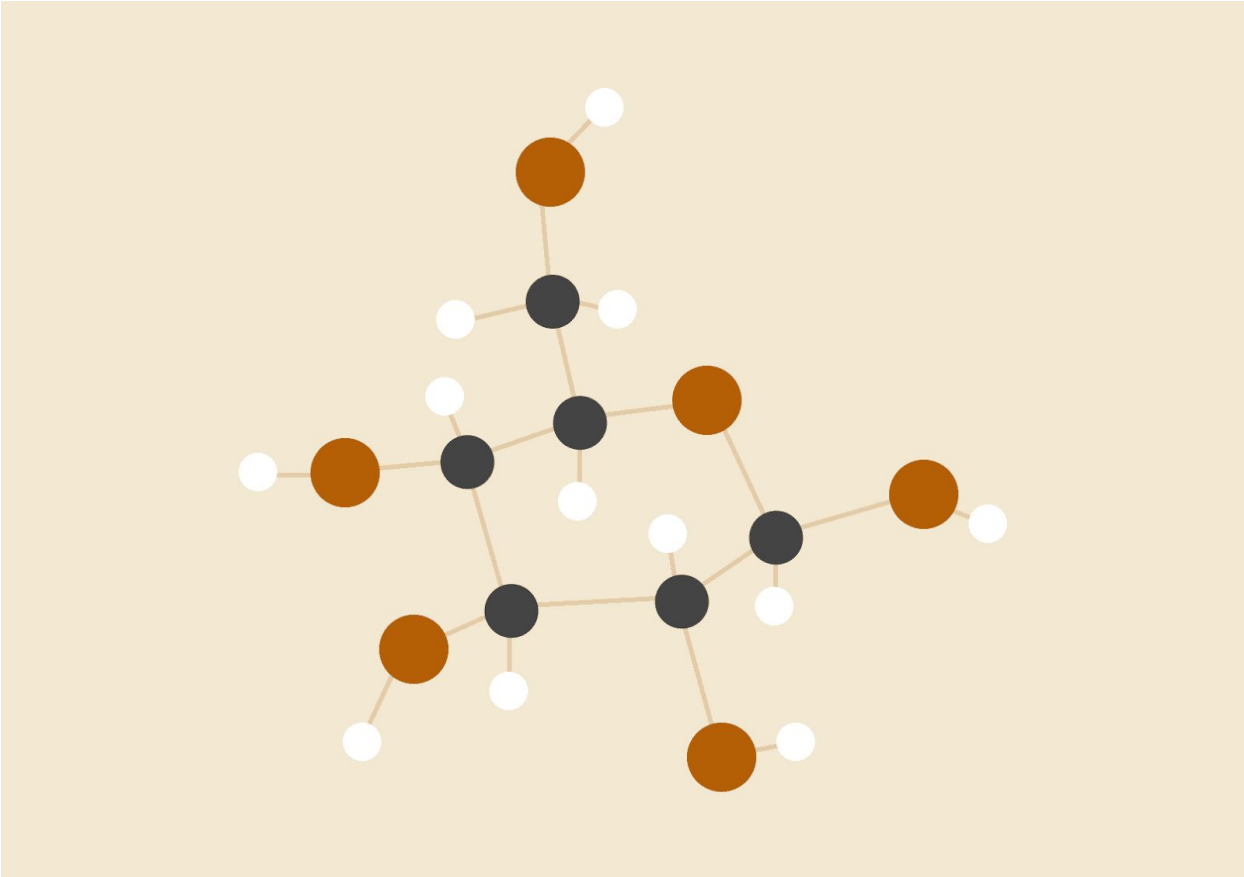
Competitive Analysis in the Online Retail Community

CPSC-408 Final Project



Alex Jones, Jin Jung

05.23.2020

Problem Statement

Helping startup retail companies compete with other larger retail companies by providing them with an application that will streamline business operations and also provide analysis tools to continually improve upon processes. Ideally, the end product will help bridge the gap between small startups and larger established corporations by providing an easily accessible, intuitive user interface for all facets of business operations.

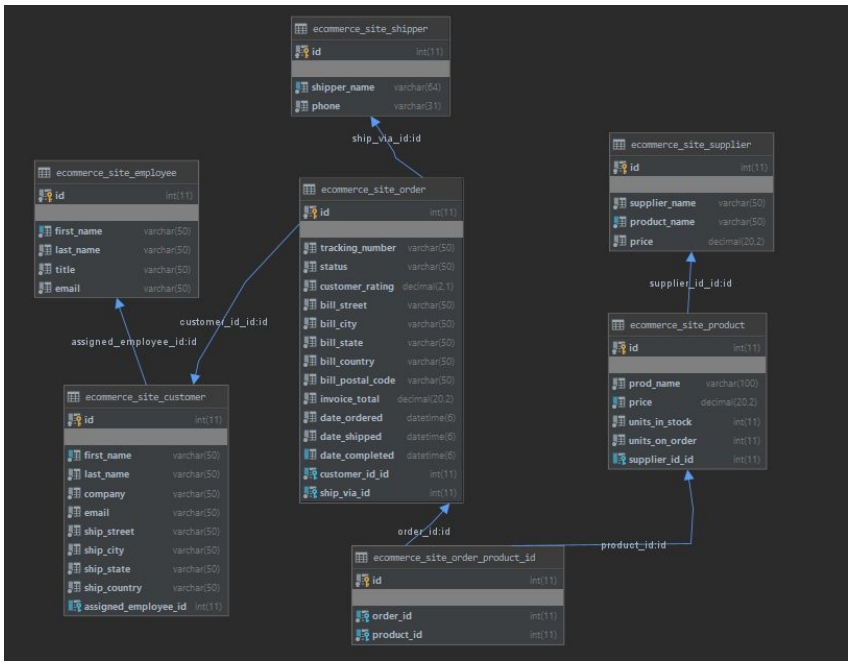
Proposed Solution

1. Create a Web Application connected to a database that can be used by businesses to help marketing, customer service, and shopping trends
2. Use Django to take advantage of database resources to provide a web-front end for smaller retail companies to use.
3. Analyze market trends, inventory needs, and customer feedback.

Technologies Used

- Front-End: HTML, CSS, BOOTSTRAP, Javascript, Chart.js
- Back-End: Python, Django, Datagrip (testing)

Schema



- Tables: employee, customer, shipper, order, order-to-product id, product, supplier
- Consists of one-to-one, one-to-many, and many-to-many relationships
- Orders can have many different products, as well as one product could be in many different orders. The order_product_id links a product to an order and an order to a product through keys.
- As evident from the schema diagram, every customer has an employee who helped them and an order. Every order has a shipper and one or multiple products. Every product has a supplier.

Trials and Tribulations

Challenges	Response	Status
Slow Page Load times when large data sets queried	Separate interfacing with each table to individual web pages.	Tables with large amounts of data are still a bit slow. Possible solution is to paginate the results and query a subsection of results at a time (i.e. Top 100) See planned future progress.
Data transmission issues when querying data required to populate the analytic charts.	Implemented Django rest framework library and used API views to achieve proper data flow.	Visualization page is functional, but still researching a way to streamline the process, and reducing the workload of adding additional insights.
Learning how to use Django	After poring over many tutorials and documentation, concluded django is actually really well documented and has no shortage of features	This allowed us to implement some interesting features, and leaves open the possibility of extending the site with more robust functionality.

Results

Many of the features implemented were using existing add-ons that are part of django to speed up the development process of the site and avoid repeating work that others have worked on before.

1. **The data visualization page**

   Implementing the data visualization page required the use of django's rest framework library as well as restructuring some views from function-based views to class-based views. When the user first logs in, this is the page that they will be redirected to. Once the user is authenticated in the log-in page, the url path that is called is 'api/chart/data', which will call the class based ChartData view which inherits from (APIView). Within ChartData view, a connection will be set up with our database hosted on the Google

Cloud Platform to make the necessary queries. The query results are then stored in a set of dictionaries and included in the response to the html template (main.html). The main.html template runs index.js script that uses the response data and Chart.js library to create the visualization. The response data was retrieved by specifying the endpoint as '/api/chart/data'. The remaining task involved parsing, formatting, and using the data on appropriate parts of Chart.js templates. The last step was to specify where on the html file to display the resulting charts.

2. **Report Generation**

   The Report Generation feature is also located on the data visualization page. Clicking on the button will allow users to download a flat file in csv format of all the query data that were required to generate the charts. As far as data flow goes, once the user clicks on the download button, the url path 'generate_report' is called, which then calls the generate_report function-based view. The generate_report view makes a connection to the database and makes the appropriate queries. The queried data is written to csv using python's csv library. It was a bit challenging to work with the data to format it as a flat file. Then, using Django's HttpResponse, the generated csv file is downloaded to the user's local Downloads directory.

3. **Create/Read/Update/Delete Data Management Functionality**

   Although Django has a convenient, built-in CRUD feature, we decided to implement our own. Initially, we had a web page that queried and listed every single table in our database, but the page load times were unbearably slow. The first idea to improve upon this was to implement caching so that previously queried data would not need to be performed again. Implementing the cache did speed up the page load times, but came with an unfortunate side effect of negating the responsiveness of the site. For example, if an update were to be made to a record, the changes would reflect on the actual database, but the webpage would continue displaying the unaltered, cached data as if no updates had been made. This caused an unreasonable amount of frustration and led to our removing the cache altogether and separating the CRUD functionality for each table to their own individual pages. Implementing separate pages required 3 views to be created for each table (create view, update view, delete view). In the respective views, the form inputs were designed to only be saved if the correct method was used ('POST') and if the form input is deemed valid. In order to display the forms correctly, Form classes were

created in forms.py for each model by making use of Django's ModelForm package. Additionally, separate sections had to be created in html templates for the create, update, and delete functionality.

4. **User Access Control**

   The target user base for our side are the employees of the retail company, so we restricted the ability to create users and manage user access to the administrator. Regular employees are not allowed to access the Django administration page and modify user privileges. In addition to this, various paths and views make use of login authentication features baked into Django to ensure unauthorized visitors are disallowed from directly accessing subdirectories of our website.

5. **Parameterized Queries**

   Implementing the parameterized queries relied heavily on the use of Django's filters package. A filters.py file was created to specify the custom FilterSet desired for each model. So a ProductFilter class can be created by inheriting from django_filters.FilterSet, and we can specify what parameters we wish to provide filter options for:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')

    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

6. **Test data generation using Faker**

   We created a fakerecords.py script which relies on python's faker, csv, sys, and random packages to generate correctly formatted data for each of our tables. This was a critical step for populating our database, since resorting to manual entry of each record would have been a nightmare.

7. **Importing data from existing csv files**

We implemented import.py to take the pre-existing csv files that were generated by our fakerecords.py and populate our database. Import.py relies on mysql.connector, faker, and pandas packages to correctly run. The csv are imported using pandas' read_csv method. The script first creates all required tables for the database if they do not already exist. If they do exist, all prior data are flushed. Afterwards, the data imported from csv files are inserted into the database. Then, an index is created for each table to optimize query times. Lastly, some stored procedures used on our site are created and inserted into the database.

Planned Future Progress

1. **User tracking to monitor user activity on retail web applications.**

   This will likely require the use of logging the current user's changes to the database and normal queries. Django has a HistoryRequestMiddleware that allows the tracking of users using history_user_id. For example using _history_user, we can see which user change a model by referencing the changed_by field:

```
from django.db import models
from simple_history.models import HistoricalRecords

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    changed_by = models.ForeignKey('auth.User')


def get_poll_user(instance, **kwargs):
    return instance.changed_by

register(Poll, get_user=get_poll_user)
```

2. **Improved parameterized queries and accessibility features.**

   Currently, the formatting and available parameters for filter options are not as extensive as it should be. Additional work is required in this section to improve the user experience standards for this feature. Another planned improvement is to sort the displayed results

by different attributes.

3. **Optimize page loading and query response time.**

   Currently, some pages with larger data being queried still take several seconds to load, which takes away from the user experience. Possible solution being considered is to paginate the results and load subsections at a time.

4. **Expand on the Data Visualization page to provide additional useful insights**

   There is still a lot of work that can be done to expand on the business insights displayed on the main page. Currently we have some charts that offer insight on who our top sales employees are, who the most customer friendly employees are, and what our monthly revenue trend is. Some additional insights we can offer are: Average Order completion time, Profit margin, suppliers of our best selling products, geo plotting of customer base, customers with lowest satisfaction, and much more.

REFERENCES

1. https://acquire.io/blog/problems-solutions-ecommerce-faces/
2. https://retailnext.net/en/blog/the-influence-of-database-in-the-retail-industry/
3. http://archive.ics.uci.edu/ml/machine-learning-databases/00352/
4. https://docs.djangoproject.com/en/3.0/topics/db/sql/
5. https://django-filter.readthedocs.io/en/stable/
6. https://data-flair.training/blogs/django-caching/
7. https://docs.djangoproject.com/en/3.0/topics/db/sql/
8. https://mdbootstrap.com/docs/jquery/javascript/charts/
9. https://django-simple-history.readthedocs.io/en/2.7.0/user_tracking.html