DSE 203

Group 5: Tasty Recipes Knowledge Graph

Vaaruni, Zufeshan, Chris

Abstract

Food is a major part of our daily lives, it speaks to our culture, fuels our bodies, and serves as a central necessity for building community. Food is also a source of our differences, those following certain diets out of either personal choice or necessity often must buy specialized products or search high and low to find recipes and restaurants that fit their dietary requirements. We created a knowledge graph with nodes of recipes, ingredients, nutrient information, common diseases, and nutrients correlated with disease flare-ups to allow users to quickly query for recipes within a central database that will fit their needs, or to replace ingredients with recipes to help them make replacements with their health in mind.

Data Sources and Roles

The goal of the project is to combine unstructured data, semi-structured data, and structured data to produce a queryable knowledge graph that serves a purpose. We have 8 total datasets, broken down as follows:

1) Kaggle Datasets - (Source: https://www.kaggle.com/datasets/zeeenb/recipes-from-tasty | Tasty's API)
    a) dishes.csv [semi-structured and unstructured data] ~5k rows
    b) Ingredients_and_instructions.json [semi-structured and unstructured data] ~5k rows
2) USDA Datasets [relational tables] (Source: SDSC Nourish Database)
    a) usda_2022_food_branded_experimental: ~1M rows
    b) usda_2022_branded_food_nutrients ~21M rows
    c) usda_2022_branded_food_product ~1M rows

d) usda_2022_nutrient_master ~500 rows

3) Internal Datasets:

a) Vegan Substitute [semi-structured, dummy data] - 10 rows

b) Disease Nutrient-Avoidance [semi-structured, dummy data] - 10 rows

Our unstructured data is derived from both the Tasty JSON Dataset in item instructions, from the Tasty CSV Dataset in item name, and in the branded product dataset from item name.

Methods

Data Loading

For our Tasty CSV data, we were able to load directly into a Pandas Dataframe utilizing the read_csv function. For our Tasty JSON data, we had to design a custom function to unnest the data and generate a structured table from it. For this, we utilized the jsonpath library to query through the data. Once this table was unnested, we were able to join it with the CSV data based on a primary key from the CSV data to a foreign key in the JSON-derived data in the 'ID' column. Once the Tasty data was joined into one dataframe, we ran a method of entity resolution using dedupe clustering to categorize the products based on product name. This method of materialized processing was chosen to reduce the overall computation time when it came to querying the data. Instead of incurring each cost to fuzzy match through all entities, it was decided that with product matching being the main component of our query, it would be more beneficial to fuzzy match to a stemmed cluster keyword and then pull all filtered entries than fuzzy match to matches within the larger dataset. Keywords were then extracted from the entity resolution to form the cluster keywords, consisting of the two most common keyword occurrences.

To join our USDA datasets, we attempted to utilize distributed processing to load and query on our local machines. Our distributed processing tool of choice for this was Dask, with a partition number of 10. We set-up our initial connection and had to use SQLAlchemy queries, as Dask's read_from_query only supports this type. To optimize this processing, we pre-processed the data by using a computationally expensive sorting operation, and joined the subsequent tables on their indexes. We were then able to load the data into a Pandas DataFrame to utilize a larger variety of available libraries, and because of lazy loading, we were able to harvest a larger portion of data without memory issues.

$$\bowtie a = b$$

$$\bowtie b = c$$

distributed join processes

However, we then decided that we would not need to materialize our data in this fashion, and opted instead to query the products we were interested in joining from the stemmed Tasty clusters, resulting in a use of centralized, virtualized processing with sql query and loading the result directly into a Pandas Dataframe to be joined at user query.

Joining Unstructured Data

Our next challenge was to combine our USDA data source to the tasty data, which required a join based on unstructured data, as we did not have a shared 'ID' field. We accomplished this by preprocessing the text to relevant shared product stems and joining with the unstructured field as an ID.

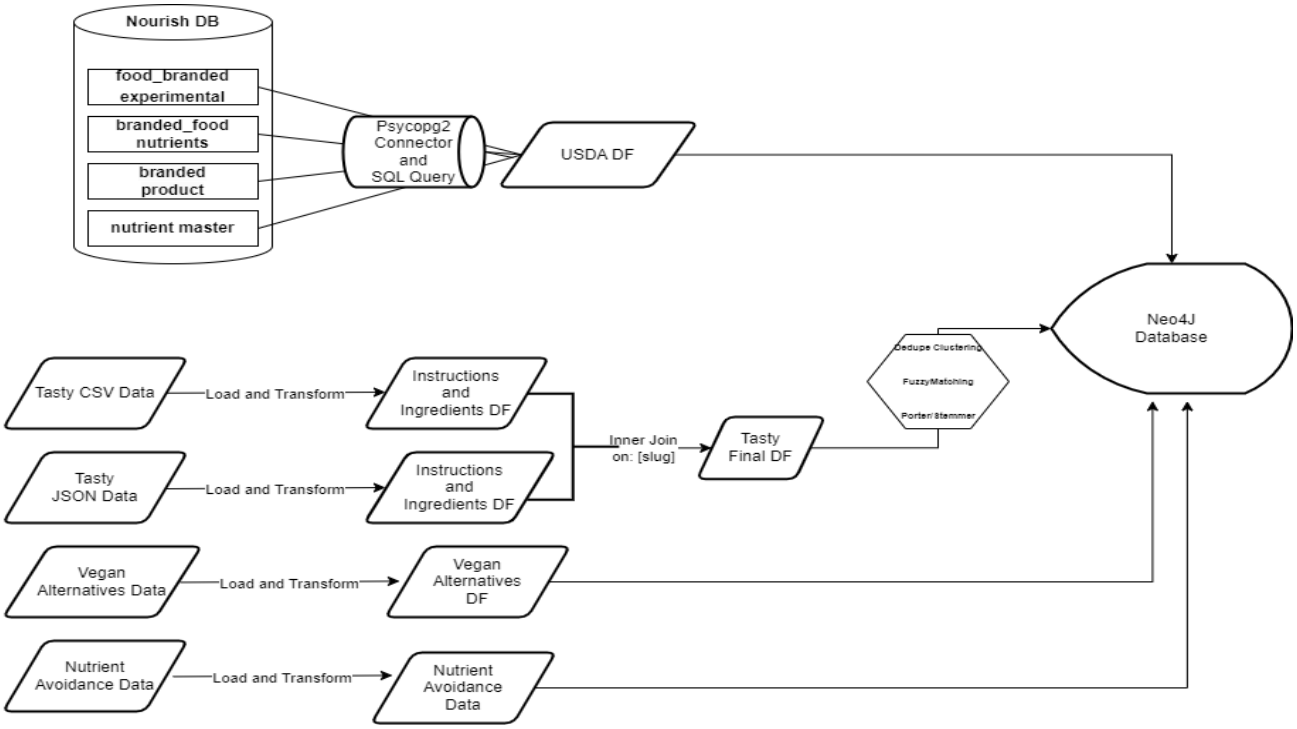

Figure 1. Data Schematic depicting different data sets and load procedures

A knowledge graph was then created from the datasets with Neo4J in python, with the following node and edge structure.

Processing and Constructing Graph Database:

<u>Rows to Nodes</u>

We have the following four nodes:

1. Product: Node consisting of the properties [<id>, calories, carbohydrates, fat, fiber, name, protein, and sugar], derived from the tasty dataset, and the usda dataset.
2. Ingredient: Node consisting of the properties [<id>, name], derived from both the Tasty dataset, the USDA datasets, and the internal disease nutrient avoidance and vegan alternatives datasets.
3. Cluster: Node consisting of the properties [<id>, keywords, name], derived from the dedupe clustering that occurred in the data process.
4. Instructions: Node consisting of the properties [<id>, instructions], derived from the Tasty Data.

Joins to relationships

We then established 3 relationships to serve as edges between our nodes:

1. Has_Products: Relationship directing from the Cluster node to the Product node, this relationship is a 1:N mapping as one cluster can contain multiple products.

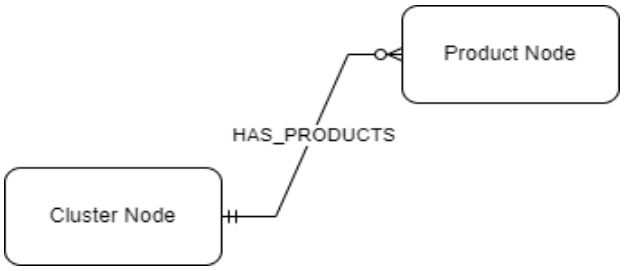

Figure 2a. Schema depicting HAS_PRODUCTS relationship

2. CONTAINS: Relationship directing from the Product node to the Ingredient Node, this relationship is an N:M mapping as multiple products can share multiple ingredients.

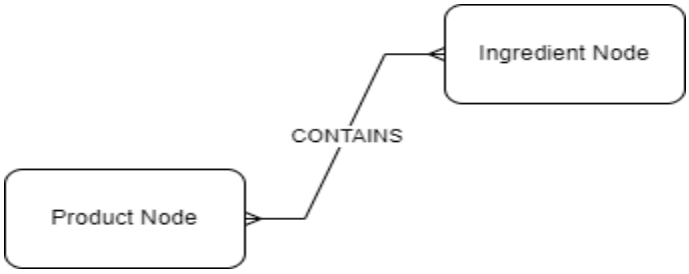

Figure 2b. Schema depicting CONTAINS relationship

3. HAS_INSTRUCTIONS: Relationship directing from the Product node to the Instruction node, this relationship is a 1:1 mapping as one product can only have one set of instructions.

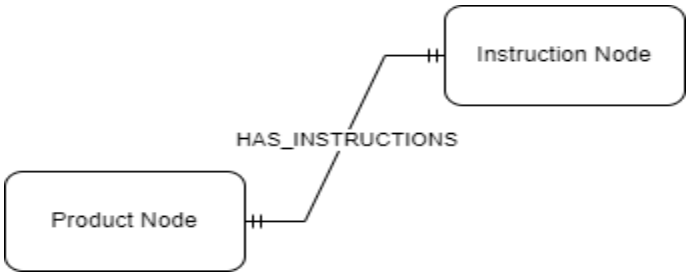

Figure 3b. Schema depicting HAS_INSTRUCTIONS relationship

Queries

We came up with 5 complex, analytical queries that showcase the versatile capabilities of our application. Each query is presented, shows the process within the query, and then exemplified by the cypher query itself.

1. *The user provides a list of ingredients, and asks for a suggestion of quick recipes to make.*

For this, our process is to first search for products that contain all of the ingredients provided by the user. Then, to sort the products based on the number of sentences in the instructions, and finally to return the recipe that contains the least number of sentences, which indicates quickness and reduced complexity.

Cypher Query 1:

```
MATCH (p:Product)-[:CONTAINS]->(i:Ingredient)
        WITH p, COLLECT(i.name) AS productIngredients
        WHERE ALL(i IN {list_of_ingredients} WHERE i IN
productIngredients)
        MATCH (p)-[:HAS_INSTRUCTIONS]->(instr)
        RETURN p.name AS productName, instr.instructions,
size(apoc.text.split(instr.instructions, "\.")) AS numsentences
        ORDER BY numsentences ASC LIMIT 1
```

---

2. *The user asks for a recipe for cheesecake, and asks that the query return results that are diabetes friendly.*

For this, our process is to first search for clusters with keywords that contain the stemmed version of the product (*cheesecake:cheesecak*), then for all products, remove the ingredients that the user needs to avoid based on the given condition. For example, in the case of diabetes, the result will remove sugar and alcohol. Then, the product will be sorted based on calories, and lastly the user is given an option to choose between the 3 healthiest products. Once the user chooses, they are provided with the ingredients and instructions for the recipe of choice.

Cypher Query 2:

```
WITH {search_keywords} AS subs
        MATCH (n:Cluster)
        WHERE n.keywords CONTAINS '{prod_to_make}'
        WITH n as cluster, subs
        MATCH (cluster)-[:HAS_PRODUCTS]->(p:Product)
        MATCH (p)-[:HAS_INSTRUCTIONS]->(instr:Instructions)
        MATCH (p)-[:CONTAINS]->(i:Ingredient)
        WHERE p.name CONTAINS '{prod_to_make}'
        AND NOT ANY(word IN subs WHERE i.name CONTAINS word)
        RETURN p.name as Product, instr.instructions AS instructions,
COLLECT(i.name) as ingredients, p.calories AS calories
        ORDER BY calories ASC
        LIMIT 3
```

---

3. *The user asks for a recipe for healthy pancakes.*

For this, our process is to first search for cluster keywords that contain the product provided by the user. Then, the returned results are sorted in ascending order based on caloric count, and the top 3 are returned. Lastly, OpenAI prompts are used to summarize the three returned recipes into a final novel recipe, equipped with ingredients and instructions to be returned to the user in a desirable format.

Cypher Query 3:

```
MATCH (n:Cluster)
        WHERE n.keywords CONTAINS '{search_substring}'
        WITH n as cluster
        MATCH (cluster)-[:HAS_PRODUCTS]->(p:Product)
```

```
MATCH (p)-[:HAS_INSTRUCTIONS]->(instr:Instructions)
MATCH (p)-[:CONTAINS]->(i:Ingredient)
WHERE p.name CONTAINS '{search_substring}'
RETURN p.name as Product, instr.instructions AS instructions,
COLLECT(i.name) as ingredients, p.calories AS calories
ORDER BY calories ASC
LIMIT 3
```

OpenAI Prompt:

```
prompt = """ Given the following instructions, ingredients and product to make, PLEASE summarize the instructions while STRICTLY following these rules
  1. Please get me a summary of the quickest of the recipes, where the instructions do NOT repeat any steps.
  2. Please include as many ingredients as possible in the instructions.
  3. DO NOT create fictitious data.
  4. The output content should be in text format.
  5. If you will be unable to output within the token limit, please DO NOT include that entry in the response. """

  #call gpt-4-turbo to extract relations
  response = openai.chat.completions.create(
    model="gpt-4",
    messages=[
    {"role": "system", "content": "You are a helpful assistant that gives the summarized instructions of the given recipes who does NOT repeat any steps"},
    {"role": "user", "content": f"{prompt}, instructions:{instructions}, ingredients: {ingredients}, product_to_make: {search_substring}"}
    ],
    temperature=0,
    max_tokens=1024 )
```

---

4. *The user asks for a recipe for a vegan version of stuffed meatball pie*

For this, we first achieve the base recipe for stuffed meatball pie, and then we check through our vegan substitutes data to create mapping from non-vegan original product to vegan replacement. We then replace the items both in ingredients and recipe, and return the instructions to the user for the new vegan product.

Further, this query aims to reduce overall data storage costs by introducing mapping from non-vegan variant to vegan variant, rather than storing two different versions of an already similar recipe.

Cypher Query 4:

```
MATCH(p:Product)-[:CONTAINS]->(i:Ingredient)
        MATCH (p)-[:HAS_INSTRUCTIONS]->(j:Instructions)
        WHERE p.name = '{non_veg_prod_name}'
        RETURN p.name, COLLECT(i.name) AS list_ingredients,
j.instructions AS recipe
```

Output

```
Ingredients required - ['olive oil', 'cashew cheese', 'vegan meatballs', 'bell peppers', 'low-moisture vegan mozzarella', 'dried basil', 'tomato puree', 'cara
melized onions', 'dried oregano', 'pizza dough', 'tomato paste', 'pepper', 'salt', 'garlic']

Recipe to make the vegan version of stuffed meatball pie is -

preheat oven to 450degf (230degc). cut a 1 pound ball of pizza dough in half and roll out two large discs of dough. spray the inside of a springform pan with
cooking spray and lay in one of the dough discs. press the dough into the corners. sprinkle the olive oil, 2 cloves of the minced garlic, salt, and pepper on
the dough and brush to cover the surface. press the bottom of the dough with a fork. bake until the dough is cooked and lightly golden brown, 15 minutes. mix
together the tomato puree, dried basil, 4 cloves of minced garlic, dried oregano, and tomato paste in a bowl. season to taste with salt and pepper. into the p
artially cooked bottom layer of pizza dough, shingle a layer of vegan mozzarella slices and then a layer of the homemade marinara sauce. spiral the vegan meat
balls in the dish as tightly as possible. cover with more marinara sauce. add a layer of caramelized onions, and more sliced vegan mozzarella. add the green,
yellow, and orange bell peppers on top of the vegan mozzarella. cut the second piece of dough into strips to form into a lattice pattern on top of the pie. br
ush with 2 tablespoons more of olive oil, 2 cloves minced garlic, and kosher salt. bake for 40 minutes to an hour, or until the dough on top is browned and th
e sauce, vegan meatballs, and cheese are heated through. slice the pie, serve with extra marinara sauce and grated cashew cheese. enjoy!
```

---

5. *The user requests for heart-disease friendly recipes, as they are not sure what they would like to make.*

For this, we pull the list of food items to be avoided by a user with heart-disease from our nutrient avoidance table. The query then unwinds the list for items, and checks for products that do not contain any of the items to avoid. The items are then avoided by duplicate records, and returns recipes where the number of duplicate records is equal to the length of the list of food items to avoid. The results are then ordered first by fiber, and then by calories, as our table of health conditions only contains conditions for which fiber is a beneficial nutrient. Finally, the top five recipes are returned to the user.

Cypher Query 5:

```
UNWIND {ingredients_to_remove} AS x
        WITH x, size({ingredients_to_remove}) AS no_of_items
        MATCH (p:Product)-[:CONTAINS]->(ing:Ingredient)
        MATCH (p:Product)-[:HAS_INSTRUCTIONS]->(ins:Instructions)
        WITH COLLECT(ing.name) as ingredients, ins,p,x,no_of_items
        WHERE NOT ANY(ingredient IN ingredients WHERE ingredient
CONTAINS x)
        WITH COUNT(p.name) AS pcount,p.name AS product,
ingredients, ins.instructions AS instructions,p.calories AS calories,
```

```
p.carbohydrates AS carbs, p.sugar AS total_sugar, p.fat AS fat, p.protein
AS protein, p.fiber AS fiber, no_of_items
        WHERE no_of_items-pcount=0
        RETURN
product,ingredients,instructions,calories,carbs,total_sugar,protein,fiber
        ORDER BY fiber DESC, calories ASC
        LIMIT 5
```

Challenges

We faced a number of challenges in arriving at the final knowledge graph.

- Dask processing required resetting of indexes on every run to allow us to compute with our local machine; and, thus, we went with a more efficient model that worked for our queries.
- Issues with integration of the extended APOC libraries with our Neo4J environment, where we were unable to utilize functions like text similarity and jaccard distance within the Neo4J environment itself. As a result of this, we were required to implement fuzzy matching, using the FuzzyWuzzy library (based on Levenshtein distance) on Python. This external processing would not have been required if we were able to utilize the libraries on Neo4J.
- Pandas Dedupe Clustering works only as well as data labeling, so trial and error was required to label the data for correct clusters. Even after active labeling for 25% of the data, the clusters returned grouped items like *strawberry cheesecake* and *strawberry cheesecake crescent rolls* or *japanese fluffy pancakes* and *japanese fluffy cheesecake* together into a single cluster rather than into the respective stemmed clusters with keywords *pancak, cheesecak,* and *roll*.

Conclusion

In conclusion, we were able to produce a knowledge graph that combined the information from a popular recipe searching site and USDA nutritional data to allow a user to query for replacements and suggestions based on their specific nutrient needs. We utilized pieces derived from each class and homework to accomplish our end-user product.

From class 1, we utilized data integration principles to combine structured and semi-structured data. From class 2, we utilized text processing to convert our external data source to a joinable format to our other relational source based on an unstructured data id. From class 3, we utilized the principles of distributed processing to allow us to load and query a large data source (>25,000,000 rows) with optimized processing time. And, from class 4, we were able to establish our entity relationships.

Utilizing these principles allowed us to generate a knowledge graph that will have great impact on those looking to cater their food searches to meet their nutritional needs.

Bibliography

---

[1] Title: "Real-world data in healthcare: Lessons learned from personalized medicine."
URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9921002/

[2] Title: "Considerations for distributed processing."
URL:
https://www.ibm.com/docs/en/zvm/7.2?topic=considerations-distributed-processing

[3] Title: "Dedupe: A python library for accurate and scalable fuzzy matching, record deduplication, and entity-resolution."
URL: https://docs.dedupe.io/en/latest/

[4] Title: "Dask documentation."
URL: https://docs.dask.org/en/stable/

[5] Title: "Fuzzy string matching in Python."
URL: https://github.com/seatgeek/fuzzywuzzy

[6] Title: "Neo4J Documentation."
URL: https://neo4j.com/docs/