MIDWEEK PROGRESS REPORT

Travel Genie
-A Personalized Travel Recommendation System

Darshil Shah (NUID:002814137)
Ishan Joshi (NUID: 002836254)
Mitesh Singh (NUID: 002726596)
Prarthana Ganesh Shetty (NUID: 00283314)

Table of Contents

1. INTRODUCTION

1.1 Problem Statement

In today's fast-paced world, travelers are faced with a complex web of information scattered across multiple platforms. Most tourism websites offer generalized recommendations, often failing to consider specific individual preferences like budget constraints, unique travel interests, or desired travel style. This lack of personalization results in an overwhelming experience for users who must piece together relevant information from different sources. Thus, Travel Genie addresses a fundamental challenge: **How can we streamline the travel planning process while ensuring that recommendations align with the user's profile in a single, user-friendly platform?**

1.2 Objective

Our team's objective with Travel Genie is to create a platform that leverages advanced technologies to personalize and streamline travel recommendations. The growing popularity of travel, both for leisure and professional purposes, **underscores the significant need for effective and personalized travel planning solutions**. Traditionally, travelers have been burdened with navigating multiple platforms to coordinate their journey details, a process often tedious and time-intensive. By **consolidating information from various sources** into a single, cohesive system, we aim to offer users an all-encompassing travel planning solution tailored to individual preferences, budgets, travel styles, and available time frames. The core of Travel Genie's functionality lies in its ability to dynamically process user data and preferences, enabling recommendations, and introducing a seamless and user-centered approach.

1.3 Key features

Key to Travel Genie's success is its **dynamic query-handling** ability, enabling it to process even incomplete user inputs intelligently. Additionally, Travel Genie emphasizes **user personalization** by integrating recommendation systems alongside Retrieval-Augmented Generation (RAG) methods, making it an innovative hybrid solution that prioritizes user experience. From understanding budgetary needs to accommodating travel duration and location preferences, the app is designed to refine each aspect of travel planning to ensure that user-specific details are accurately addressed.

Technologies and Tools

To develop the Travel Genie platform, our team selected a blend of tools and technologies tailored to facilitate efficient data handling, personalization, and dynamic user interactions:

- **Apify**: Used as an API for collecting data on hotels, attractions, and restaurants within selected cities, providing reliable, real-time data for tailored travel recommendations.

- **Python**: The primary programming language for handling web scraping tasks and data profiling, facilitating data transformation and storage processes.
- **Machine Learning and Natural Language Processing Libraries**: Libraries like **OpenAI, HuggingFace, PyTorch, and LangChain** power the recommendation engine, enabling advanced ML models and natural language understanding to refine travel suggestions.
- **Neo4j Graph Database**: We selected Neo4j to model and store relationships between various travel entities. This graph database allows Travel Genie to query and retrieve data based on connections between entities like hotels, attractions, and user preferences, enabling highly contextual and personalized recommendations that reflect real-world relationships.

These tools form the technical backbone of Travel Genie, each playing a critical role in delivering a user-centric and responsive travel planning experience.

2. DEFINING DATA REQUIREMENTS

At the foundation of Travel Genie's personalized travel recommendation system is a comprehensive dataset that caters to diverse user needs. This document outlines the data requirements necessary to support our platform's functionality and enhance user experience. By identifying and organizing data requirements across various dimensions—such as user profiles, destination characteristics, and external factors like weather—we ensure that Travel Genie meets its objectives of delivering accurate, tailored travel recommendations.

2.1 Destination and Accommodation Data

Data on destinations and accommodations is crucial for offering users a comprehensive range of options tailored to their preferences and location choices.

1) **Data Fields Required**
    a) **Attraction Details**: Location attributes (city, location, description, price, etc.), popular attractions, cultural landmarks, transportation options, and points of interest.
    b) **Accommodation Information**: Hotel names, types (e.g., budget, luxury), amenities, pricing, ratings, and reviews from users.
    c) **Dining and Restaurant Options**: Restaurant names, cuisine types, dietary options, ratings, pricing, and operating hours if available.
2) **Source**
    a. Apify served as the primary API for collecting data on accommodations, attractions, and dining options.

    B. Web scraping from Tripadvisor to collect data for attractions, hotels, and events reviews.

3) **Purpose**
    This data is fundamental for generating accurate recommendations for accommodation and dining that fit within user-specified budgets and location preferences. It also helps in creating a comprehensive and coherent itinerary for users who are unfamiliar with the destination.

2.2 Weather Data

Weather data provides contextual information on conditions at various destinations, enhancing the practicality of the recommendations provided by Travel Genie.

- **Data Fields Required**
    - **Weather Conditions**: Average temperature history, weather conditions, precipitation, wind speed, etc. for the travel.
- **Source**
    We used External weather APIs, such as OpenWeatherMap and VisualCrossing API

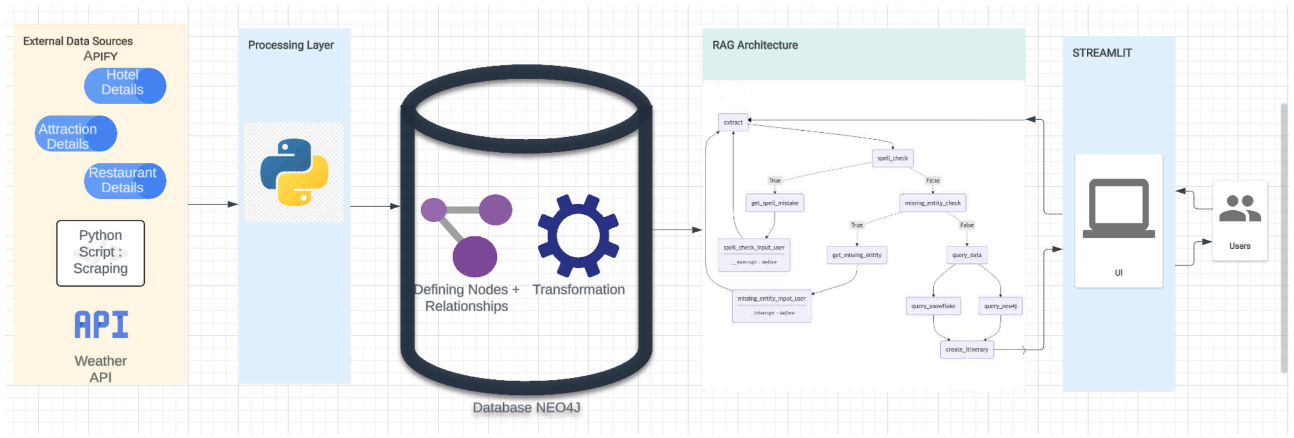
- **Purpose**
  By integrating weather data, we will enable Travel Genie to advise users on the best travel periods and offer location-based recommendations based on anticipated weather conditions. For example, if a user plans to travel during the rainy season, the system may recommend indoor activities or provide alternative destination options.

2.3 User-Generated Review Data

User reviews provide insight into traveler experiences and are invaluable in recommending accommodations, attractions, and dining options.

- **Data Fields Required**
  - **Ratings and Review Text**: User ratings, reviews, and summaries for accommodations, restaurants, and attractions.
  - **Review Metadata**: Date of visit, review date, and user profile information (if available).
- **Source**
  Developed Python scripts, focusing on data scraping from popular review sites like TripAdvisor to ensure comprehensive feedback.
- **Purpose**
  User-generated reviews add a layer of credibility and social proof to recommendations, helping users make decisions based on real experiences and aggregated feedback. For instance, Travel Genie's recommendations can prioritize highly rated attractions or accommodations.

3. ARCHITECTURE DIAGRAM



3.1 Data Collection and Processing:

- **External Data Sources:** We begin with gathering information from various external data sources. This includes hotel, attraction, and restaurant details from platforms like APIFY, and weather data from a Weather API. Data is obtained through APIs and Python scripts that include scraping functionalities.
- **Processing Layer:** Once data is collected, we use a processing layer powered by Python. Here, the data is merged and structured. This transformation prepares the data for efficient storage and retrieval in the database, optimizing it for later use in generating travel itineraries.

3.2 Database (NEO4J):

- We have used a Neo4j database, a graph database suitable for handling relationships between entities. The data model here is structured with nodes (like hotels, attractions, etc.) and relationships (such as proximity, similarity, etc.), providing a highly interconnected data framework. This enables quick, complex queries, ideal for generating recommendations and itineraries based on user preferences.

3.3 RAG (Retrieval-Augmented Generation) Architecture:

- This module forms the core of user interaction with the data. It handles the following processes:
  - **Extract**: Initiates the extraction of relevant data based on user input.
  - **Spell Check**: Ensures that user inputs are error-free. If a mistake is detected, it routes the input for correction through `get_spell_mistake` and prompts the user to re-input data if needed.
  - **Missing Entity Check**: If required information is missing, the system identifies the missing elements and prompts the user to provide them. This

process ensures the input data is complete for generating a comprehensive itinerary.
    - **Query Data**: Once inputs are verified, the system queries data from Neo4j which houses processed data ready for itinerary generation.
    - **Create Itinerary**: Based on user preferences and available data, the system generates a personalized travel itinerary.

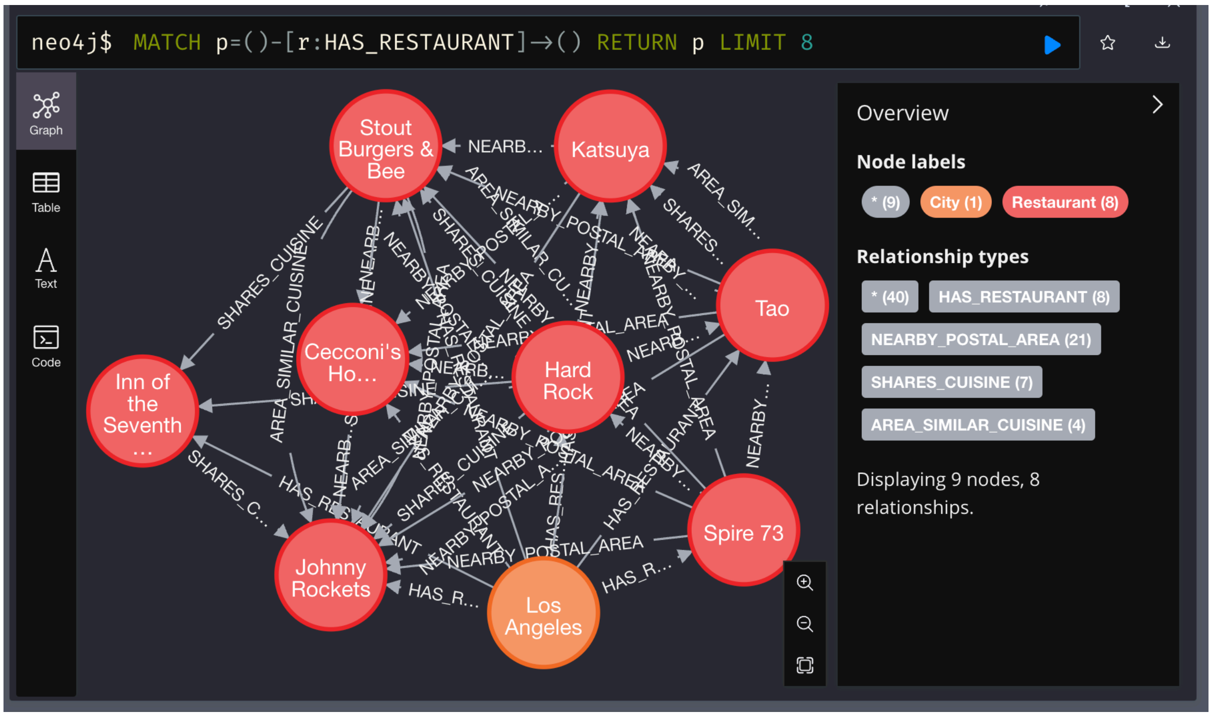
3.4 User Interface (Streamlit):

- We plan on building the project's frontend interface using Streamlit, a Python-based framework designed for creating interactive web applications. This interface allows users to input their preferences, receive itinerary recommendations, and visualize them directly within the application.
- Users can interact with the system, adjust parameters as needed, and receive a dynamic, user-friendly experience.

Summary:

This architecture combines a rich data layer with advanced NLP techniques (RAG) and an interactive interface. The integration of Neo4j enables efficient data retrieval, while the RAG component manages input validation, ensuring that users receive accurate, customized recommendations. The Streamlit interface completes the solution by providing an intuitive, responsive platform for users to explore their travel plans. This system is well-suited for generating detailed, personalized travel itineraries based on current data sources and user inputs, providing an end-to-end solution for travel planning.

4. DATABASE AND ONTOLOGY



```
neo4j$ MATCH p=()-[r:HAS_RESTAURANT]→() RETURN p LIMIT 8
```

Overview

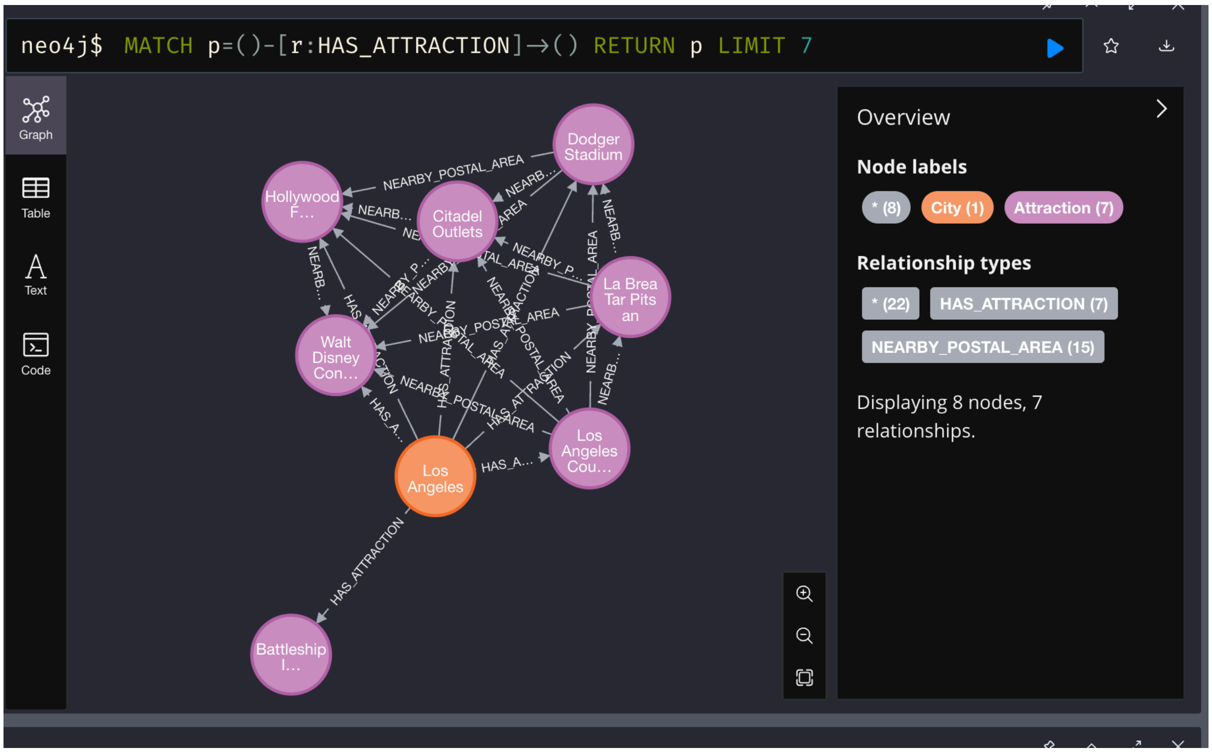
Node labels

* (9) City (1) Restaurant (8)

Relationship types

* (40) HAS_RESTAURANT (8) NEARBY_POSTAL_AREA (21) SHARES_CUISINE (7) AREA_SIMILAR_CUISINE (4)

Displaying 9 nodes, 8 relationships.



```
neo4j$ MATCH p=()-[r:HAS_ATTRACTION]→() RETURN p LIMIT 7
```

Overview

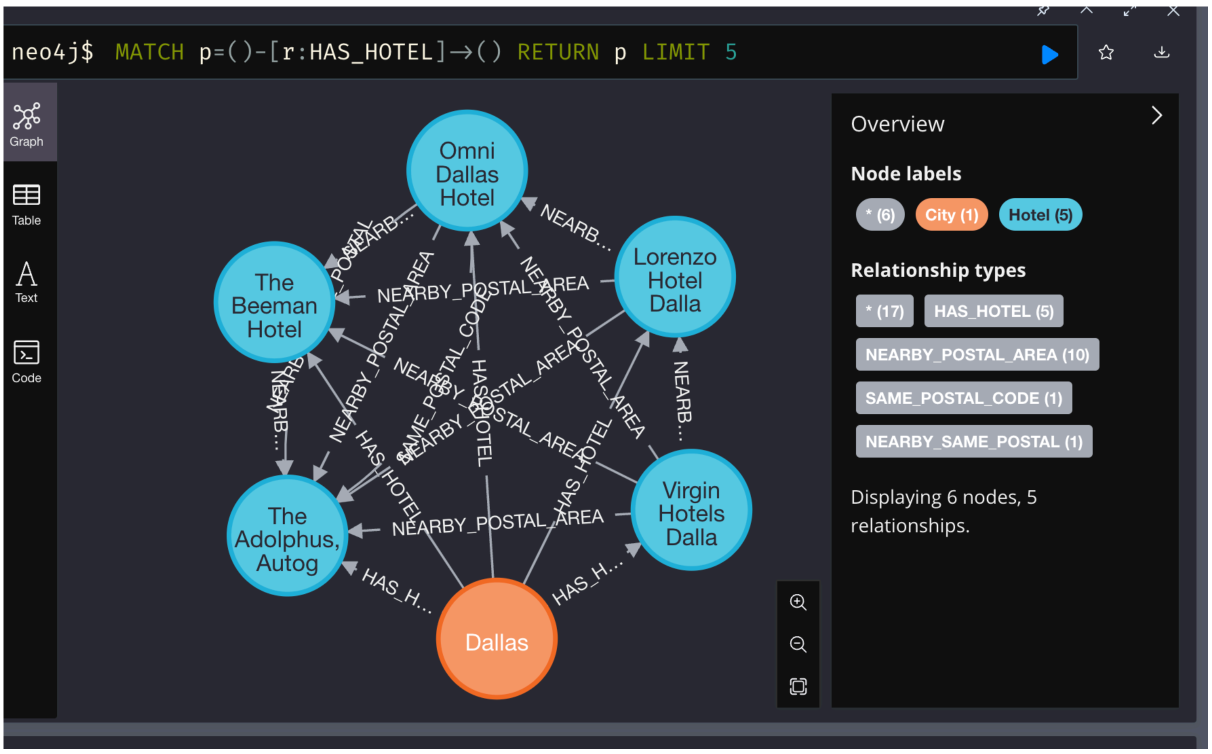
Node labels

* (8) City (1) Attraction (7)

Relationship types

* (22) HAS_ATTRACTION (7) NEARBY_POSTAL_AREA (15)

Displaying 8 nodes, 7 relationships.



```
neo4j$ MATCH p=()-[r:HAS_HOTEL]→() RETURN p LIMIT 5
```

Overview

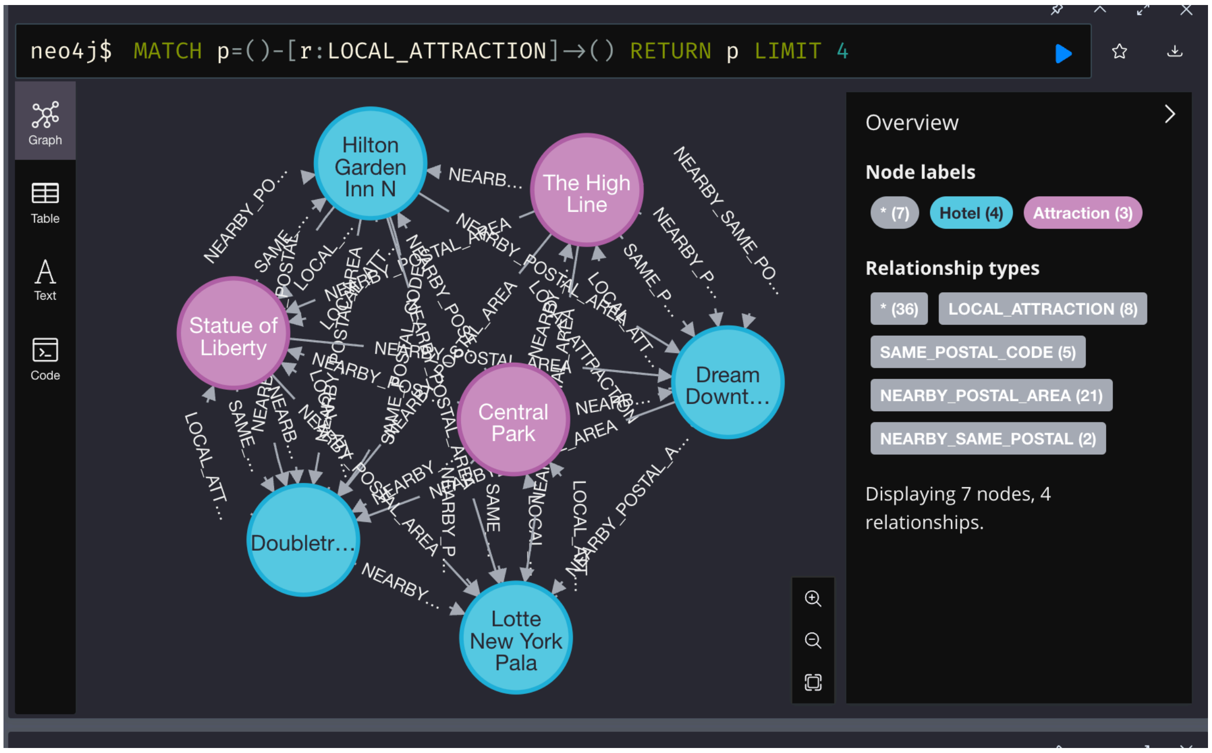
Node labels

* (6) City (1) Hotel (5)

Relationship types

* (17) HAS_HOTEL (5) NEARBY_POSTAL_AREA (10) SAME_POSTAL_CODE (1) NEARBY_SAME_POSTAL (1)

Displaying 6 nodes, 5 relationships.



```
neo4j$ MATCH p=()-[r:LOCAL_ATTRACTION]→() RETURN p LIMIT 4
```

Overview

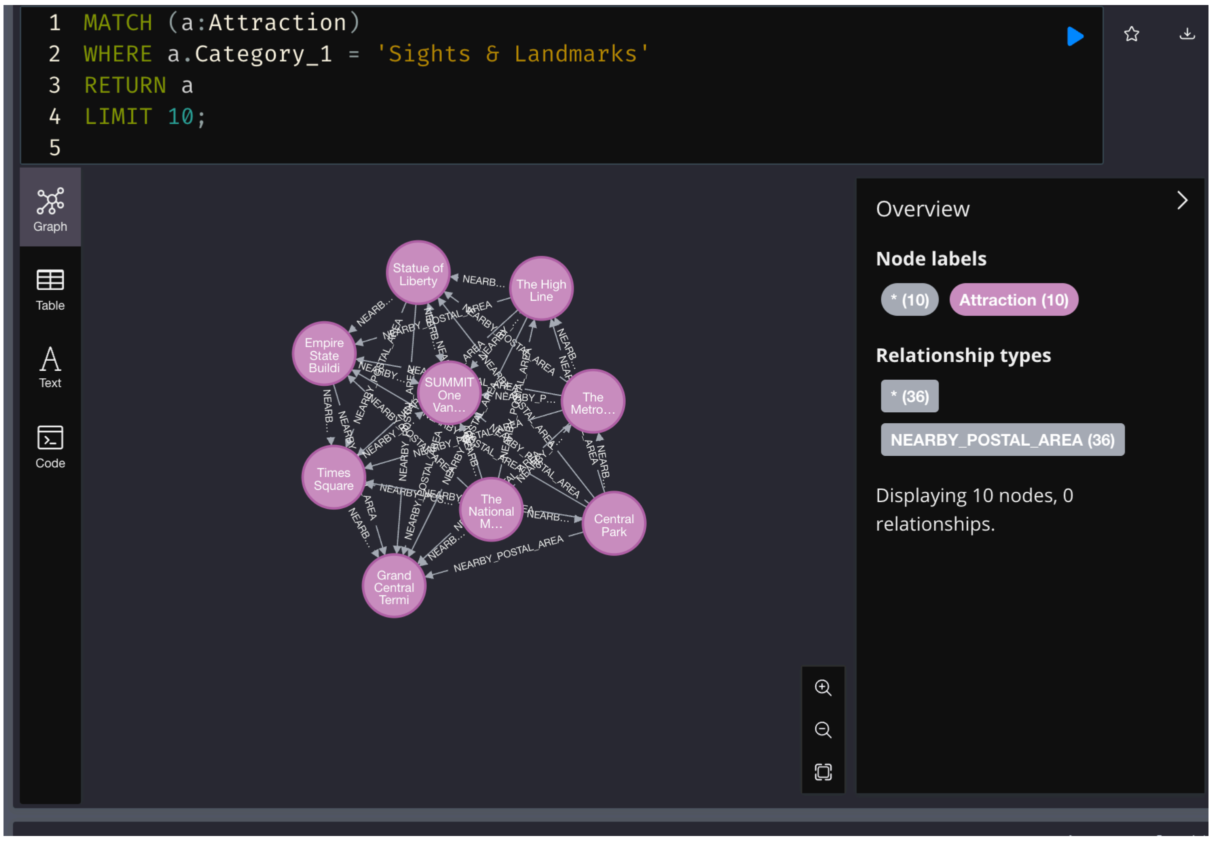
Node labels

* (7) Hotel (4) Attraction (3)

Relationship types

* (36) LOCAL_ATTRACTION (8) SAME_POSTAL_CODE (5) NEARBY_POSTAL_AREA (21) NEARBY_SAME_POSTAL (2)

Displaying 7 nodes, 4 relationships.



```
MATCH (a:Attraction)
WHERE a.Category_1 = 'Sights & Landmarks'
RETURN a
LIMIT 10;
```

Overview

Node labels

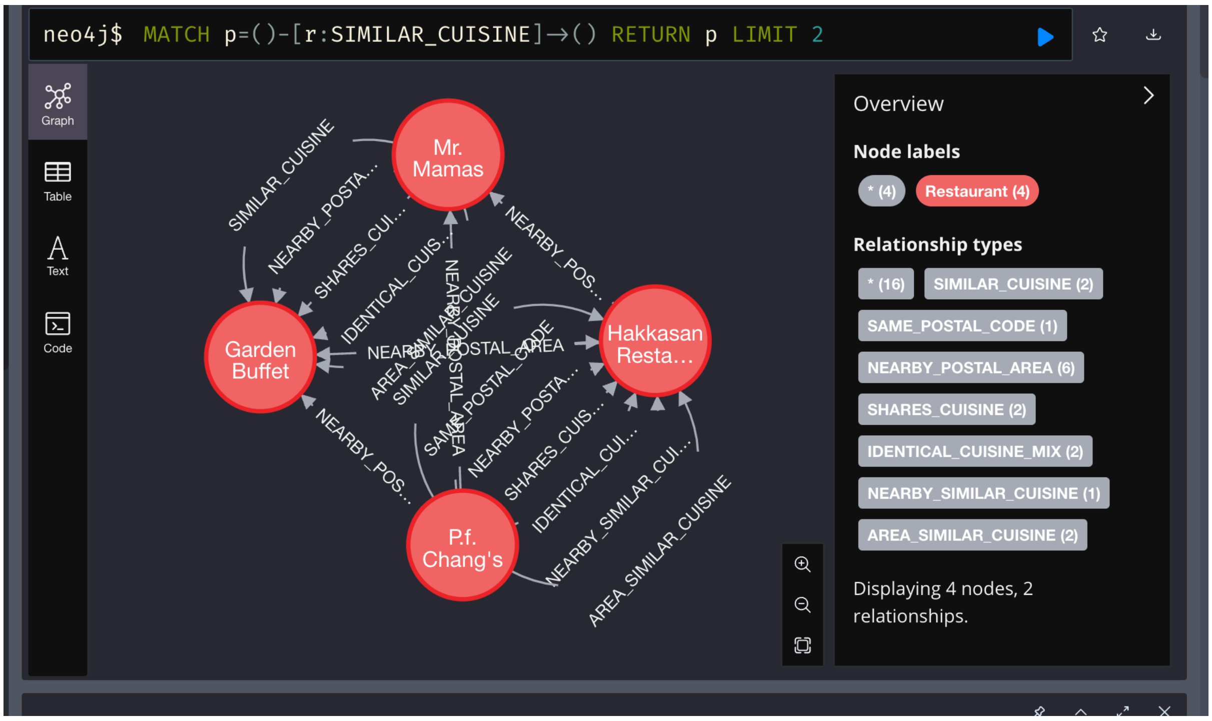
* (10) Attraction (10)

Relationship types

* (36)

NEARBY_POSTAL_AREA (36)

Displaying 10 nodes, 0 relationships.



```
neo4j$ MATCH p=()-[r:SIMILAR_CUISINE]→() RETURN p LIMIT 2
```

Overview

Node labels

* (4) Restaurant (4)

Relationship types

* (16) SIMILAR_CUISINE (2)

SAME_POSTAL_CODE (1)

NEARBY_POSTAL_AREA (6)

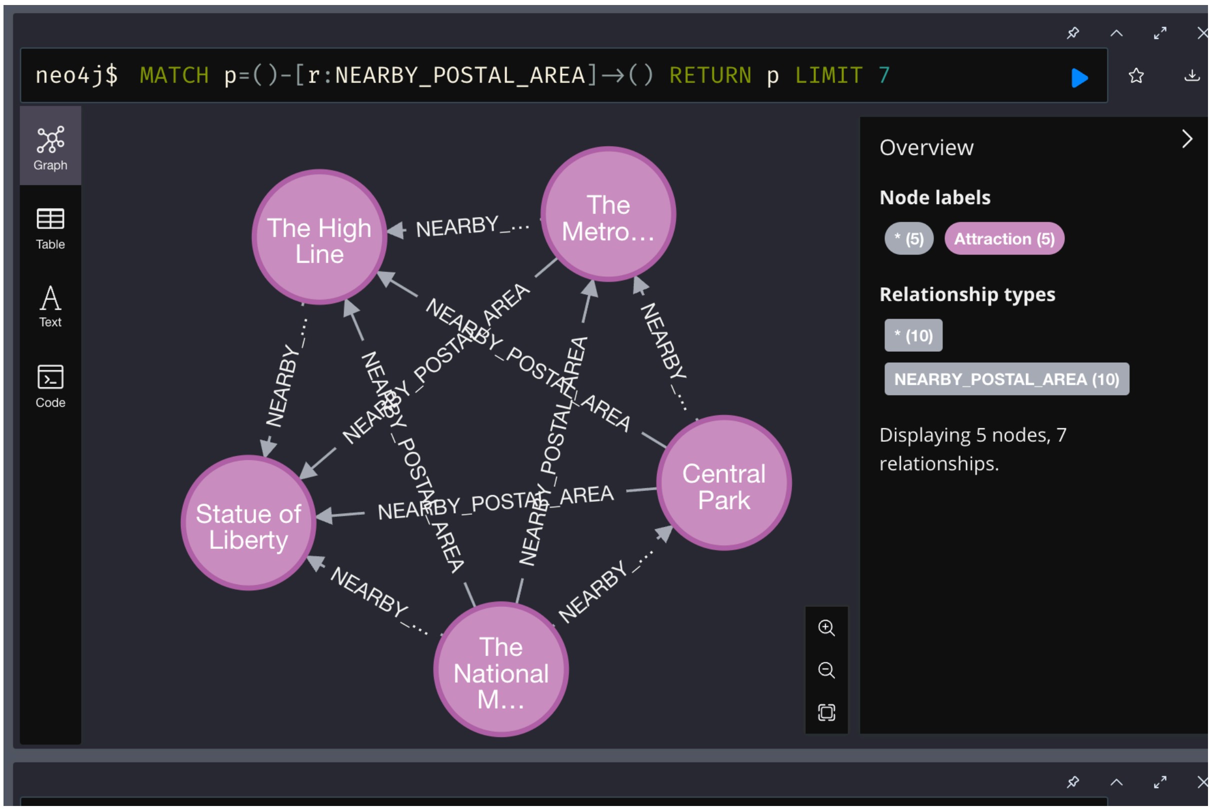
SHARES_CUISINE (2)

IDENTICAL_CUISINE_MIX (2)

NEARBY_SIMILAR_CUISINE (1)

AREA_SIMILAR_CUISINE (2)

Displaying 4 nodes, 2 relationships.



```
neo4j$ MATCH p=()-[r:NEARBY_POSTAL_AREA]→() RETURN p LIMIT 7
```

Overview

Node labels

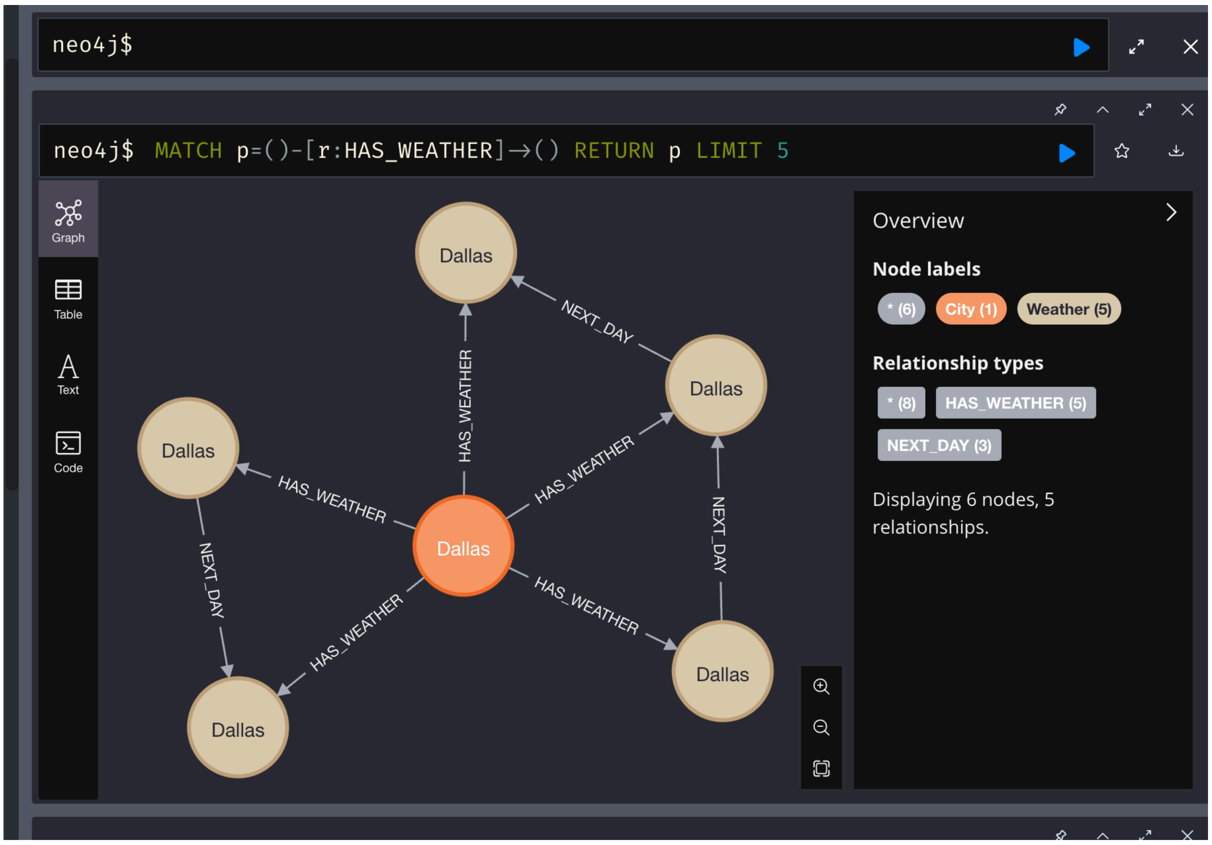
* (5) Attraction (5)

Relationship types

* (10)

NEARBY_POSTAL_AREA (10)

Displaying 5 nodes, 7 relationships.



```
neo4j$
```

```
neo4j$ MATCH p=()-[r:HAS_WEATHER]→() RETURN p LIMIT 5
```

Overview

Node labels

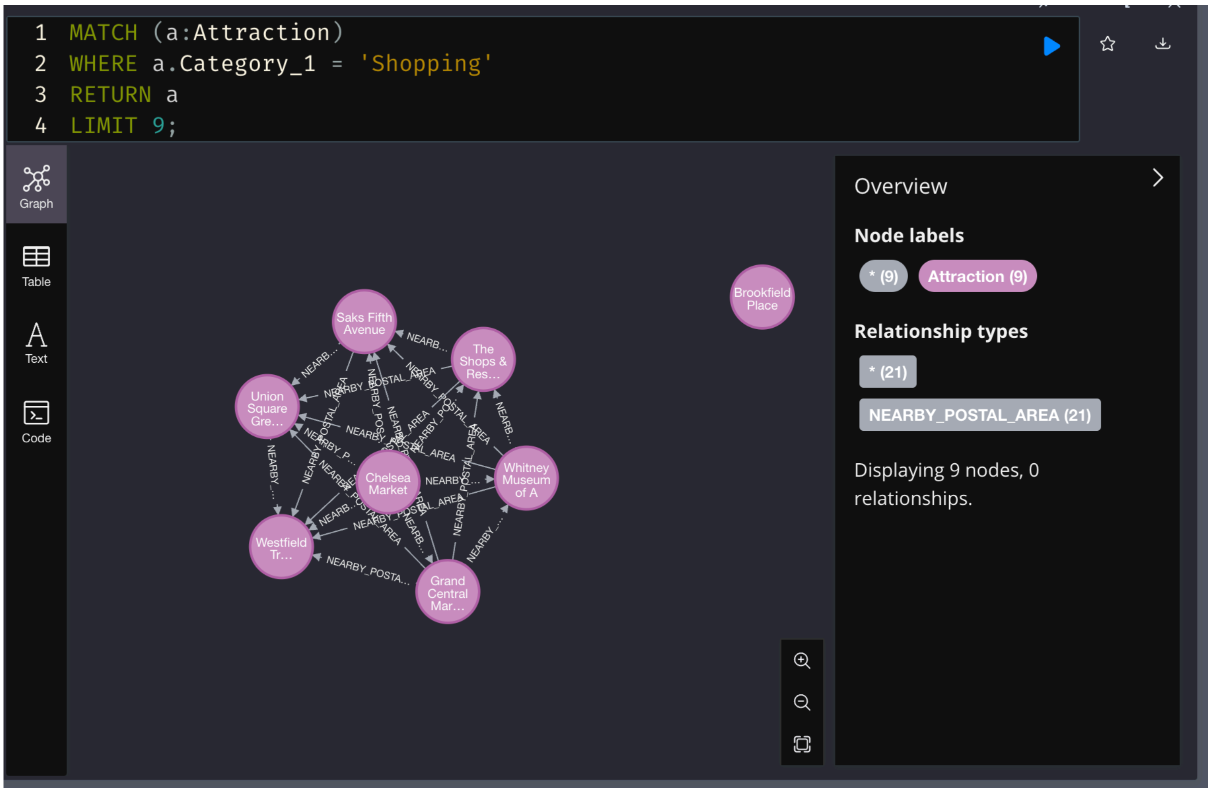
* (6) City (1) Weather (5)

Relationship types

* (8) HAS_WEATHER (5)

NEXT_DAY (3)

Displaying 6 nodes, 5 relationships.



```
MATCH (a:Attraction)
WHERE a.Category_1 = 'Shopping'
RETURN a
LIMIT 9;
```

Overview

Node labels

* (9) Attraction (9)

Relationship types

* (21)

NEARBY_POSTAL_AREA (21)

Displaying 9 nodes, 0 relationships.

A graph database like Neo4j is ideal for a travel recommendation system for the following reasons:

1. **Efficient Relationship Management**: Graph databases quickly handle complex relationships, making it easy to explore connections like proximity and shared attributes without slow JOINs.
2. **Flexible Schema**: They adapt easily to evolving data models, accommodating new entities and relationships as the system scales.
3. **Geospatial Queries**: Built-in spatial indexing efficiently supports proximity-based recommendations, like finding nearby attractions or hotels.
4. **Multi-hop Relationship Handling**: Graphs excel in multi-hop queries, enabling insights from complex, interconnected data for better recommendations.
5. **Real-time Performance**: Graphs allow for rapid traversal, delivering fast, real-time results that improve user experience.
6. **Personalization**: Graph databases enable easy personalization by linking user preferences to relevant recommendations.
7. **Data Visualization**: Graph structures align with visualization tools, allowing easy exploration of travel data for users and administrators.
8. **Multi-layered Data Support**: They manage layered data (e.g., geographic, thematic) naturally, offering richer, more nuanced recommendations.

Nodes and Labels

1. **City** (Label: City)
   Description: Represents a central geographical hub for related entities like restaurants, attractions, and hotels.
   Properties:
   name: Name of the city (e.g., "Los Angeles", "Dallas").
   postal_code (optional): A postal code identifier for the city.
   Examples:
   Los Angeles
   Dallas

2. **Restaurant** (Label: Restaurant)
   Description: Represents a dining establishment associated with a city.
   Properties:
   name: Name of the restaurant (e.g., "Stout Burgers & Bee").
   cuisine_type: Type of cuisine offered (e.g., "American", "Japanese").
   postal_code: Postal code area where the restaurant is located.
   Examples:
   Stout Burgers & Bee
   Hard Rock Cafe

3. **Attraction** (Label: Attraction)
   Description: Represents a tourist spot or point of interest associated with a city.
   Properties:
   name: Name of the attraction (e.g., "Hollywood Walk of Fame").
   attraction_type: Type of attraction (e.g., "Museum", "Park").
   postal_code: Postal code area where the attraction is located.
   Examples:
   Hollywood Walk of Fame
   Walt Disney Concert Hall

4. **Hotel** (Label: Hotel)
   Description: Represents an accommodation option within or near a city.
   Properties:
   name: Name of the hotel (e.g., "Omni Dallas Hotel").
   postal_code: Postal code area where the hotel is located.
   Examples:
   The Beeman Hotel
   Lorenzo Hotel Dallas

Relationships and Types

1. HAS_RESTAURANT
   Type: HAS_RESTAURANT
   Description: Connects a city to its restaurants, indicating that these restaurants are located within the city.
   Example: (Los Angeles) -[:HAS_RESTAURANT]-> (Stout Burgers & Bee)

2. HAS_ATTRACTION
   Type: HAS_ATTRACTION
   Description: Connects a city to its attractions, showing that these attractions are located in or near the city.
   Example: (Los Angeles) -[:HAS_ATTRACTION]-> (Hollywood Walk of Fame)

3. HAS_HOTEL
   Type: HAS_HOTEL
   Description: Connects a city to its hotels, indicating that these hotels are located within the city or are associated with it.
   Example: (Dallas) -[:HAS_HOTEL]-> (Omni Dallas Hotel)

4. NEARBY_POSTAL_AREA
   Type: NEARBY_POSTAL_AREA
   Description: Links nodes that are within the same or adjacent postal areas, allowing for recommendations based on geographical proximity.
   Example: (Stout Burgers & Bee) -[:NEARBY_POSTAL_AREA]-> (Hard Rock Cafe)

5. SHARES_CUISINE
   Type: SHARES_CUISINE
   Description: Connects restaurants that offer similar types of cuisine, enabling cuisine-based recommendations.
   Example: (Stout Burgers & Bee) -[:SHARES_CUISINE]-> (Hard Rock Cafe)

6. AREA_SIMILAR_CUISINE
   Type: AREA_SIMILAR_CUISINE
   Description: Connects areas with similar cuisine types, allowing for exploration of culinary areas.
   Example: (Katsuya) -[:AREA_SIMILAR_CUISINE]-> (Tao)

7. SAME_POSTAL_CODE
   Type: SAME_POSTAL_CODE
   Description: Links nodes that are located within the same postal code, helping to group entities by their exact postal area.
   Example: (The Beeman Hotel) -[:SAME_POSTAL_CODE]-> (The Adolphus, Autograph Collection)

5. LLM and Agentic Workflow Development for Travel Recommendations

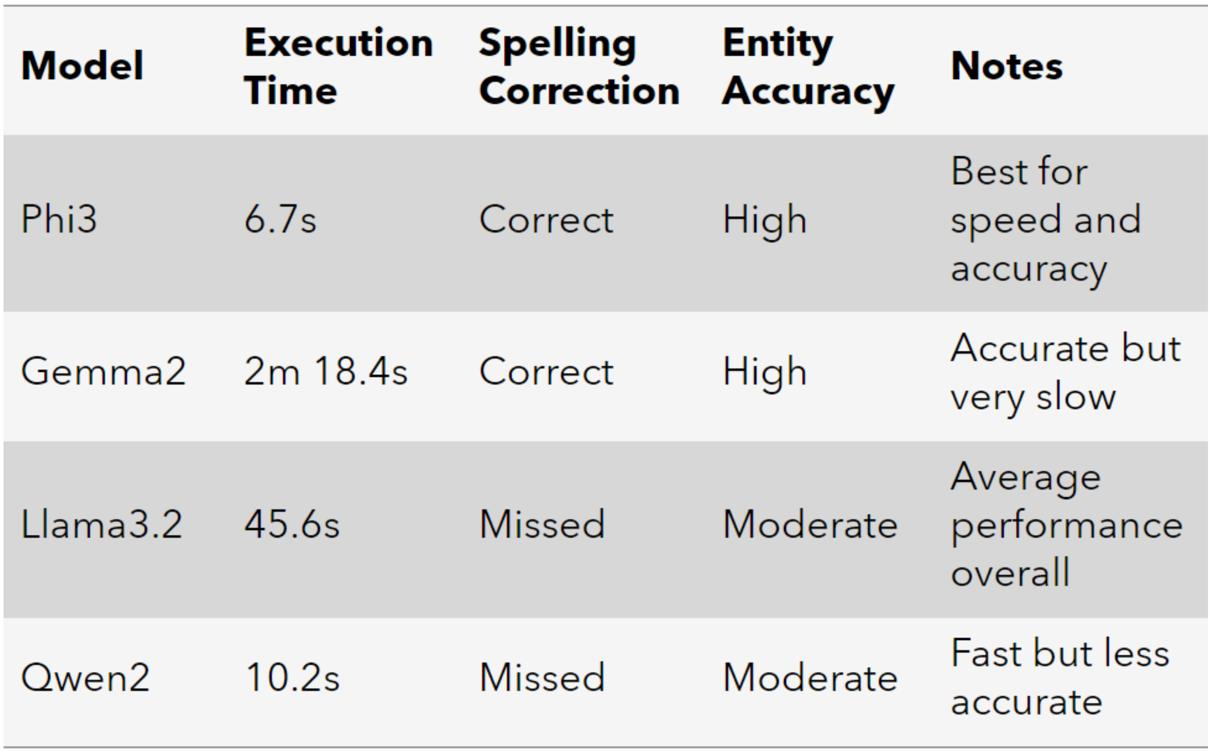
5.1 Objective: Establish a responsive agentic workflow that leverages Large Language Models (LLMs) for travel recommendation systems to deliver personalized, real-time itineraries.

Requirements for Agentic Workflow:
- Employ LLMs for precise entity extraction and to map relationships within user prompts.
- Integrate function calling capabilities and conditionally trigger human feedback based on contextual needs.
- Enable real-time data retrieval, validation, and tailored recommendations based on user preferences.
- Optimize data retrieval with a knowledge graph tailored to extracted entities.

5.2 Research and Progress:

5.2.1 LLM Model Benchmarking: Conducted a detailed benchmark assessment of various LLMs for entity extraction:
- **Phi3** (Microsoft/Phi-3-mini-128k-instruct)
- **Gemma2** (google/gemma-2-2b-it)
- **Llama3.2** (meta-llama/Llama-3.2-3B-Instruct)
- **Qwen2** (Qwen/Qwen2-1.5B-Instruct)

Performance Overview:



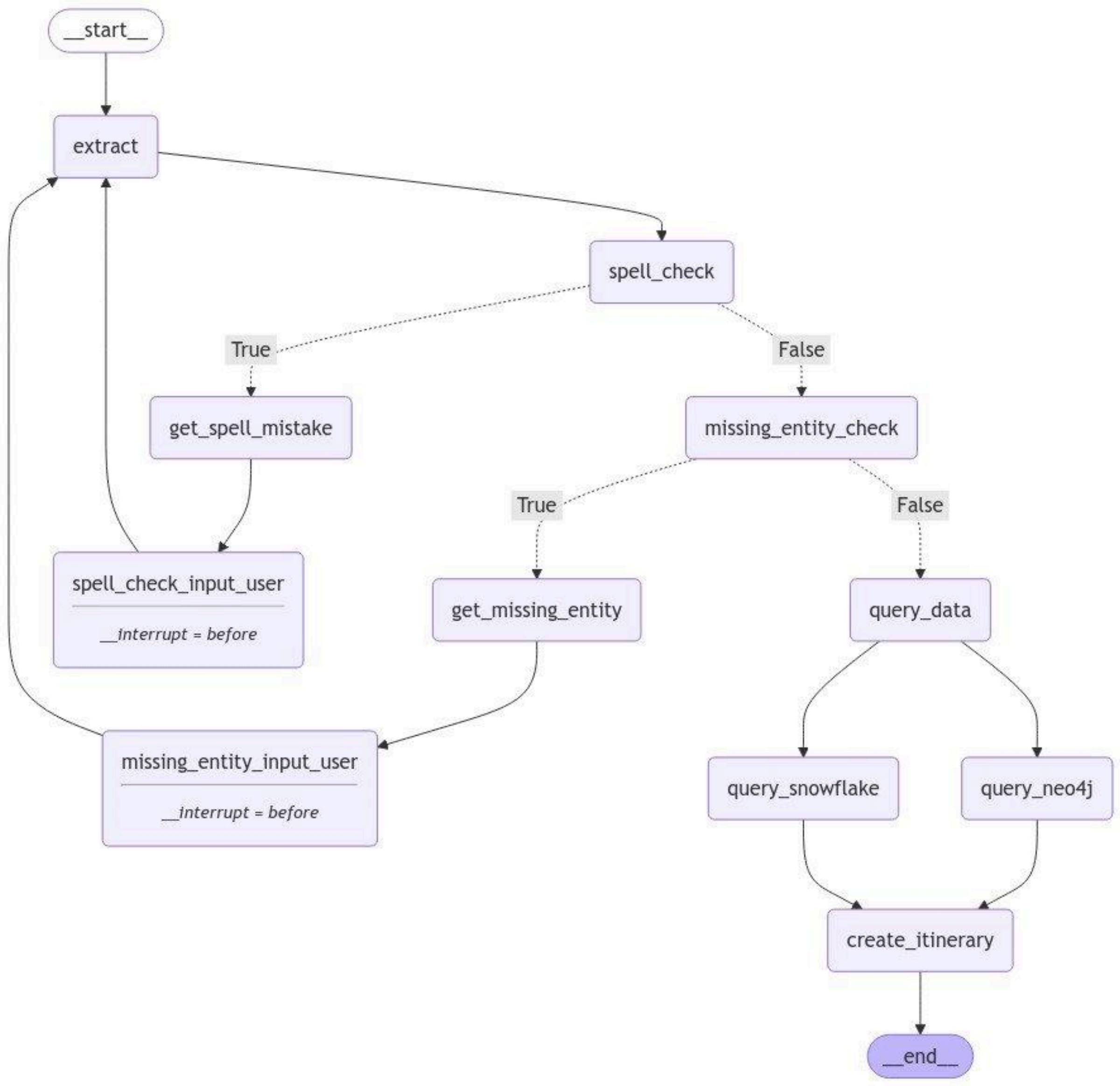
Model	Execution Time	Spelling Correction	Entity Accuracy	Notes
Phi3	6.7s	Correct	High	Best for speed and accuracy
Gemma2	2m 18.4s	Correct	High	Accurate but very slow
Llama3.2	45.6s	Missed	Moderate	Average performance overall
Qwen2	10.2s	Missed	Moderate	Fast but less accurate

Selection of Agentic Workflow Framework:

- **Framework Comparison:** Evaluated frameworks like LangGraph, AutoGen, CrewAI, and LlamaIndex.
- **Framework Selection:** Choose LangGraph for its flexible, node-based structure, facilitating complex workflows with validation checkpoints and human-in-the-loop mechanisms.
- **Fine-Grained Control:** Configures workflows to initiate human intervention based on specified conditions, such as identifying spelling errors.
- **Modularity:** Supports modular integration for components like entity extraction and spell-checking.
- **Conditional Routing:** Incorporates nodes for conditional branching, prompting human input only when necessary.

Prompt Engineering and Refinement:
Recognizing the importance of prompt quality in guiding LLM responses, we experimented with different prompt formats and refined them to improve entity recognition and extraction accuracy. Our work here includes testing varied structures to optimize model comprehension and response, allowing the system to accurately interpret user intentions, even when inputs are ambiguous. The iterative refinement of prompts also contributes to reducing latency, as more efficient prompts lead to faster response times.

Breakpoint Integration:



Workflow Node Definitions:
extract: Extracts entities from user prompts.
spell_check: Triggers spell-check conditions.
get_spell_mistake: Generates prompts to clarify user spelling errors.
spell_check_input_user: Processes corrected inputs for accurate extraction.
missing_entity_check: Checks for missing entities in user input.
get_missing_entity: Prompts clarification on missing entities.
missing_entity_input_user: Processes user-provided corrections for enhanced entity extraction.
query_data: Passes data to querying nodes (e.g., Snowflake, Neo4j).
create_itinerary: Generates an itinerary using retrieved data based on user specifications.

5.3 Future Work in Agentic Workflow:

- **Research into GraphRAG:** Explore GraphRAG (Graph Retrieval-Augmented Generation) for more advanced data retrieval methods and enhanced context maintenance. Integrating GraphRAG can help structure the knowledge graph to capture complex user intent, enabling more precise and contextually accurate responses in the recommendation system.

- **Neo4j Query Generation and Retrieval:** Use LLM function calls to generate and execute Neo4j queries dynamically, tailored to user prompts. By enabling function calling, the workflow can autonomously retrieve relevant data from the Neo4j knowledge graph, improving the adaptability of itinerary recommendations based on user inputs.

- **Optimization of Graph Structure:** Optimize LangGraph's node structure to streamline the workflow, reduce response times, and minimize latency. Focus on modular node dependencies and conditional flows to make the graph more efficient and adaptable for future expansions.

- **Development of Auto-Evaluation Node:** Implement an auto-evaluation node using an LLM to assess response quality and ensure accuracy in real time. This node will act as a "judge," scoring the quality of extracted entities, response relevance, and adherence to user specifications, enhancing the overall reliability of the workflow.

6. CODE DOCUMENTATION

6.1 Python Scripts for Web Scraping

Selenium script to scrape the attractions review:

1. Extract Location Name

```
def extract_location_name(url):
    # Extract location name from the URL (between the last two dashes)
    match = re.search(r'-Reviews-(.*?)-', url)
    if match:
        location_name = match.group(1).replace('_', ' ')
        return location_name.strip()
    return 'Unknown_Location'
```

- **Purpose**: Extracts the location name from the provided URL.
- **Explanation**: Uses regex to find the part of the URL that represents the location name, replacing underscores with spaces. If no match is found, returns `"Unknown_Location"`.

2. Extract Location ID

```
def extract_location_id(url):
    # Extract location name from the URL (between the last two dashes)
    match = re.search(r'-d(\d+)-', url)
    if match:
        location_id = match.group(1)
        return location_id
    return 'Unknown_Location'
```

- **Purpose**: Extracts the location ID from the URL.
- **Explanation**: Uses regex to find the unique location identifier (`d<number>`), which can be used as a unique key in the dataset. If no match is found, returns `"Unknown_Location"`.

3. Save Reviews

```
def save_reviews_to_csv(reviews, location_name, location_id):
    # Create 'reviews' folder if it doesn't exist
    if not os.path.exists('dmg7374/reviews'):
        os.makedirs('dmg7374/reviews')

    # Define the path to save the CSV file (with the location name)
    file_path = os.path.join('dmg7374/reviews', f'{location_name}.csv')

    # Define the CSV headers
    headers = ['Review ID', 'Review Link', 'Review Title', 'Ratings
Score', 'Review Date', 'Review Body', 'Username',
               'Username Link']

    # Write reviews to CSV
    with open(file_path, 'w', newline='', encoding='utf-8') as file:
        writer = csv.DictWriter(file, fieldnames=headers)
        writer.writeheader()
        for review in reviews:
            writer.writerow({
                'Attraction ID': location_id,
                'Attraction Name': location_name,
                'Review ID': review.get('review_id'),
                'Review Link': review['review_link_href'],
                'Review Title': review['review_title'],
                'Ratings Score': review['rating_score'],
                'Review Date': review['review_date'],
                'Review Body': review['review_body'],
                'Username': review['username'],
                'Username Link': review['username_link'],
            })
    print(f"Reviews saved to {file_path}")
```

- **Purpose**: Saves all extracted reviews to a CSV file.
- **Explanation**: Creates a directory if it doesn't exist, defines the CSV headers, then writes each review as a row. Each row includes details such as `review_id`, `review_link`, `review_title`, `rating_score`, and other relevant information.

4. Extract Reviews Function

```
def extract_reviews(url, rating=1, limit=100):
    reviews_data = []
    count = 0
    page_number = 1  # Track the current page number

    filter_button = 'span:contains("Filters")'
    rating_button = f".qgcDG > div:nth-child(2) > div >
button:nth-of-type({rating})"
    apply_button = 'span:contains("Apply")'

    with SB(uc=True, demo=True, incognito=True, locale_code="en") as
sb:
        sb.uc_open_with_reconnect(url, 4)
        sb.uc_click(filter_button, reconnect_time=2)
        sb.uc_click(rating_button, reconnect_time=2)
        sb.uc_click(apply_button, reconnect_time=2)
        time.sleep(random.uniform(3, 6))
        while count < limit:
            get_reviews = sb.find_element(By.CLASS_NAME, "LbPSX")
            get_each_review_child =
get_reviews.find_elements(By.CLASS_NAME, "_c")

            for element in get_each_review_child:
                review_data = {}

                try:
                    review_data['rating_score'] = rating
                    review_link = element.find_element(By.CSS_SELECTOR,
".biGQs._P.fiohW.qWPrE.ncFvv.fOtGX")
                    review_data['review_link_href'] =
review_link.find_element(By.XPATH, "./*").get_attribute("href")
                    review_data['review_title'] =
review_link.find_element(By.XPATH, "./*").text
                except NoSuchElementException:
                    review_data['review_link_href'] = "N/A"
                    review_data['review_title'] = "N/A"

                # Extract review_id from the review URL
                try:
                    review_id_match = re.search(r"-r(\d+)-",
review_data.get('review_link_href', ''))
                    review_data['review_id'] = review_id_match.group(1)
if review_id_match else "N/A"
                except Exception:
                    review_data['review_id'] = "N/A"

                try:
                    review_data['review_date'] =
element.find_element(By.CLASS_NAME, "RpeCd").text.strip().split('•')[
                        0].strip()
                except NoSuchElementException:
                    review_data['review_date'] = "N/A"

                try:
                    username_entities =
element.find_element(By.CSS_SELECTOR, ".biGQs._P.fiohW.fOtGX")
                    review_data['username'] = username_entities.text
                    review_data['username_link'] =
username_entities.find_element(By.CSS_SELECTOR,
".BMQDV._F.Gv.wSSLS.SwZTJ.FGwzt.ukgoS").get_attribute(
                        "href")
                except NoSuchElementException:
                    review_data['username'] = "N/A"
                    review_data['username_link'] = "N/A"

                try:
                    review_data['review_body'] =
element.find_element(By.CLASS_NAME, "JguWG").text.strip()
                except NoSuchElementException:
                    review_data['review_body'] = "N/A"

                reviews_data.append(review_data)
                count += 1

                if count >= limit:
                    break

            if count >= limit:
                break

            try:
                # Pagination Logic
                next_button_divs = sb.find_elements(By.CLASS_NAME,
"xkSty")
                next_button =
next_button_divs[0].find_element(By.XPATH, ".//a[@aria-label='Next
page']")

                # Check if the button is enabled before clicking
                if next_button.is_enabled():
                    next_button.click()  # Click the next button
                    time.sleep(random.uniform(3, 6))  # Add a delay to
prevent detection
                else:
                    print("Next button is disabled or not clickable.")
                    break

            except NoSuchElementException:
                print("Next button not found.")
                break
```

```
    return reviews_data
```

```
# Usage

link = input("Enter the attraction URL\n")

ratings_to_extract = [1, 2, 3, 4, 5]

all_reviews = []

location_name = extract_location_name(link)  # Extract location name
from URL

location_id = extract_location_id(link)

for rating in ratings_to_extract:

    try:

        reviews = extract_reviews(link, rating, limit=400)  # Extract
reviews with a specified limit for each rating

        all_reviews.extend(reviews)  # Add the extracted reviews to the
main list

    except Exception as e:

        print(f"Error extracting reviews for rating {rating}: {e}")



# Save all reviews to a single CSV regardless of errors

save_reviews_to_csv(all_reviews,location_name,location_id)  # Save all
reviews to a single C
```

This script automates the extraction of reviews from a specified attraction URL, applying different rating filters (1–5) and saving the reviews to a CSV file. It begins by setting up the extraction process, defining CSS selectors for filtering and using incognito mode with reconnect options to avoid detection. The script then sequentially applies each rating filter, opening the URL and simulating human-like delays to extract review data such as title, link, rating score, date, username, and review body. Missing elements default to "N/A" values. To gather more reviews, the script includes pagination logic that clicks the "Next page" button if available, allowing it to scrape reviews across multiple pages. Once all reviews are collected, they are saved to a CSV file named after the attraction location. The main execution block coordinates the entire process: it prompts the user for the attraction URL, extracts the location name and ID, and iterates over each rating level, handling errors to continue if one rating fails. After compiling all reviews, it saves them in a single CSV file for easy

6.2 Cypher Query Code

Creating Nodes for Hotels, Attractions, Restaurants and Weathers

1) Attractions

```
LOAD CSV WITH HEADERS FROM 'file:///attractions.csv' AS row

MERGE (a:Attraction {name: row.name})  // MERGE on name as unique identifier

ON CREATE SET

    a.name = row.name,

    a.city = row['addressObj/city'],

    a.postal_code = row['addressObj/postalcode'],

    a.rating = toFloat(row.rating),

    a.latitude = toFloat(row.latitude),

    a.longitude = toFloat(row.longitude),

    a.description = row.description,

    a.No_of_Reviews = row.numberOfReviews,

    a.Ranking_Denominator = toInteger(row.rankingDenominator),

    a.Ranking_Position = toInteger(row.rankingPosition),  // Fixed typo in 'Position'

    a.Ranking_String = row.rankingString,

    a.Raw_Ranking = toFloat(row.rawRanking),

    a.Category_1 = row['subcategories/0'],

    a.Category_2 = row['subcategories/1'],

    a.Sub_Type_1 = row['subtype/0'],

    a.Sub_Type_2 = row['subtype/1']

ON MATCH SET

    a.city = row['addressObj/city'],
    a.postal_code = row['addressObj/postalcode'],

    a.rating = toFloat(row.rating),

    a.latitude = toFloat(row.latitude),

    a.longitude = toFloat(row.longitude),

    a.description = row.description,

    a.No_of_Reviews = row.numberOfReviews,

    a.Ranking_Denominator = toInteger(row.rankingDenominator),

    a.Ranking_Position = toInteger(row.rankingPosition),  // Fixed typo in 'Position'

    a.Ranking_String = row.rankingString,

    a.Raw_Ranking = toFloat(row.rawRanking),

    a.Category_1 = row['subcategories/0'],

    a.Category_2 = row['subcategories/1'],

    a.Sub_Type_1 = row['subtype/0'],

    a.Sub_Type_2 = row['subtype/1'];
```

2) Hotel

```
LOAD CSV WITH HEADERS FROM 'file:///hotels.csv' AS row

WITH row WHERE row['addressObj/city'] <> '' AND row.name IS NOT NULL

MERGE (h:Hotel {name: row.name})

ON CREATE SET

    h.name = row.name,

    h.street1 = CASE WHEN row['addressObj/street1'] = '' THEN null ELSE row['addressObj/street1'] END,

    h.street2 = CASE WHEN row['addressObj/street2'] = '' THEN null ELSE row['addressObj/street2'] END,

    h.city = CASE WHEN row['addressObj/city'] = '' THEN null ELSE row['addressObj/city'] END,

    h.postal_code = CASE WHEN row['addressObj/postalcode'] = '' THEN null ELSE row['addressObj/postalcode'] END,

    h.state = CASE WHEN row['addressObj/state'] = '' THEN null ELSE row['addressObj/state'] END,

    h.amenities = [amenity IN [

        row['amenities/0'],

        row['amenities/1'],

        row['amenities/2'],
        row['amenities/3'],

        row['amenities/4'],

        row['amenities/5'],

        row['amenities/6'],

        row['amenities/7'],

        row['amenities/8'],

        row['amenities/9'],

        row['amenities/10'],

        row['amenities/11'],

        row['amenities/12'],

        row['amenities/13'],

        row['amenities/14']

    ] WHERE amenity IS NOT NULL AND amenity <> ''],

    h.description = CASE WHEN row.description = '' THEN null ELSE row.description END,

    h.hotel_class = CASE WHEN row.hotelClass = '' THEN null ELSE toFloat(row.hotelClass) END,

    h.latitude = CASE WHEN row.latitude = '' THEN null ELSE toFloat(row.latitude) END,

    h.longitude = CASE WHEN row.longitude = '' THEN null ELSE toFloat(row.longitude) END,

    h.number_of_rooms = CASE WHEN row.numberOfRooms = '' THEN null ELSE toInteger(row.numberOfRooms)
END,

    h.price_level = CASE WHEN row.priceLevel = '' THEN null ELSE row.priceLevel END,

    h.price_range = CASE WHEN row.priceRange = '' THEN null ELSE row.priceRange END,

    h.ranking_denominator = CASE WHEN row.rankingDenominator = '' THEN null ELSE
toInteger(row.rankingDenominator) END,

    h.ranking_position = CASE WHEN row.rankingPosition = '' THEN null ELSE toInteger(row.rankingPosition) END,

    h.ranking_string = CASE WHEN row.rankingString = '' THEN null ELSE row.rankingString END,

    h.rating = CASE WHEN row.rating = '' THEN null ELSE toFloat(row.rating) END
```

3) Restaurants

```
LOAD CSV WITH HEADERS FROM 'file:///restaurants.csv' AS row

WHERE row['addressObj/city'] <> 'San Jose'

MERGE (r:Restaurant {name: row.name})
```

```
ON CREATE SET
    r.name = row.name,
    r.address = row['addressObj/street1'],
    r.city = row['addressObj/city'],
    r.state = row['addressObj/state'],
    r.postal_code = row['addressObj/postalcode'],
    // Filter out null values from cuisines array
    r.cuisines = [cuisine IN [row['cuisines/0'], row['cuisines/1'], row['cuisines/2']]
                WHERE cuisine IS NOT NULL],
    r.description = row.description,
    // Filter out null values from dietary restrictions array
    r.dietary_restrictions = [restriction IN [row['dietaryRestrictions/0'],
                         row['dietaryRestrictions/1'],
                         row['dietaryRestrictions/2'],
                         row['dietaryRestrictions/3']]
                         WHERE restriction IS NOT NULL],
    r.open_hours = toString(toInteger(row['hours/weekRanges/0/0/openHours'])/60) + ':' +
                toString(toInteger(row['hours/weekRanges/0/0/openHours'])%60),
    r.close_hours = toString(toInteger(row['hours/weekRanges/0/0/closeHours'])/60) + ':' +
                 toString(toInteger(row['hours/weekRanges/0/0/closeHours'])%60),
    r.price_level = row.priceLevel,
    r.price_range = row.priceRange,
    r.ranking_denominator = toInteger(row.rankingDenominator),
    r.ranking_position = toInteger(row.rankingPosition),
    r.ranking_string = row.rankingString,
    r.rating = toFloat(row.rating)
```

4) Weather

```
LOAD CSV WITH HEADERS FROM 'file:///weather_data.csv' AS line
WITH line,
CASE
  // Extract and pad day, month, year components
  WHEN line.datetime IS NOT NULL
  THEN apoc.date.parseDefault(
    line.datetime,
    'dd-MM-yyyy',
    null
  )
  ELSE null
END as proper_date
WHERE proper_date IS NOT NULL
CREATE (w:Weather {
  name: line.name,
  datetime: datetime({ epochmillis: proper_date }),
  tempmax: toFloat(COALESCE(line.tempmax, '0')),
  tempmin: toFloat(COALESCE(line.tempmin, '0')),
  temp: toFloat(COALESCE(line.temp, '0')),
  feelslikemax: toFloat(COALESCE(line.feelslikemax, '0')),
  feelslikemin: toFloat(COALESCE(line.feelslikemin, '0')),
  feelslike: toFloat(COALESCE(line.feelslike, '0')),
  dew: toFloat(COALESCE(line.dew, '0')),
  humidity: toFloat(COALESCE(line.humidity, '0')),
  precip: toFloat(COALESCE(line.precip, '0')),
  precipprob: toFloat(COALESCE(line.precipprob, '0')),
  precipcover: toFloat(COALESCE(line.precipcover, '0')),
  preciptype: CASE
    WHEN line.preciptype IS NULL OR line.preciptype = ''
    THEN null
    ELSE split(line.preciptype, ',')
  END,
  snow: toFloat(COALESCE(line.snow, '0')),
  snowdepth: toFloat(COALESCE(line.snowdepth, '0')),
  windgust: toFloat(COALESCE(line.windgust, '0')),
  windspeed: toFloat(COALESCE(line.windspeed, '0')),
  winddir: toFloat(COALESCE(line.winddir, '0')),
  sealevelpressure: toFloat(COALESCE(line.sealevelpressure, '0')),
  cloudcover: toFloat(COALESCE(line.cloudcover, '0')),
  visibility: toFloat(COALESCE(line.visibility, '0')),
  solarradiation: toFloat(COALESCE(line.solarradiation, '0')),
  solarenergy: toFloat(COALESCE(line.solarenergy, '0')),
  uvindex: toInteger(COALESCE(line.uvindex, '0')),
  severerisk: toInteger(COALESCE(line.severerisk, '0')),
  sunrise: line.sunrise,
  sunset: line.sunset,
  moonphase: toFloat(COALESCE(line.moonphase, '0')),
  conditions: line.conditions,
  description: line.description,
  icon: line.icon
})
```

Some examples of creating relationships

1)

```
MATCH (c:City)

WITH c

MATCH (r:Restaurant)

WHERE c.name = r.city_name

CREATE (c)-[:HAS_RESTAURANT]->(r)
```

Purpose: Creates a HAS_RESTAURANT relationship between a city and its restaurants.

Explanation: Finds all City nodes and iterates over each city. For each city, it matches Restaurant nodes where the restaurant's city_name property matches the name property of the city, then creates a relationship from the city node to the restaurant node.

2)

```
MATCH (c:City)

WITH c

MATCH (a:Attraction)

WHERE c.name = a.city_name

CREATE (c)-[:HAS_ATTRACTION]->(a)
```

Purpose: Establishes a HAS_ATTRACTION relationship between a city and its attractions.

Explanation: For each City node, this code finds Attraction nodes with a matching city_name, then creates a HAS_ATTRACTION relationship from the city to the attraction

3)

```
MATCH (c:City)

WITH c

MATCH (h:Hotel)

WHERE c.name = h.city_name

CREATE (c)-[:HAS_HOTEL]->(h)
```

Purpose: Connects cities to hotels with a HAS_HOTEL relationship.

Explanation: Similar to the previous queries, it matches Hotel nodes where the city_name matches the city name, then creates a relationship from the city to the hotel.

4)

```
MATCH (p1), (p2)

WHERE (p1:Hotel OR p1:Restaurant OR p1:Attraction)

AND (p2:Hotel OR p2:Restaurant OR p2:Attraction)

AND id(p1) < id(p2)

AND p1.city_name = p2.city_name

AND point.distance(

    point({latitude: p1.latitude, longitude: p1.longitude}),

    point({latitude: p2.latitude, longitude: p2.longitude})

) <= 2000  // 2000 meters = 2km

CREATE (p1)-[:NEARBY {

    distance_meters: toInteger(point.distance(

        point({latitude: p1.latitude, longitude: p1.longitude}),

        point({latitude: p2.latitude, longitude: p2.longitude})

    ))

}]->(p2)
```

Purpose: Creates NEARBY relationships between hotels, restaurants, and attractions within 2 km.

Explanation: Matches two nodes (p1 and p2) if they are both either a Hotel, Restaurant, or Attraction in the same city. To avoid duplicate relationships, it ensures id(p1) < id(p2). If the geographical distance between them is within 2 km (2000 meters), a NEARBY relationship is created with the exact distance stored as a property distance_meters.

5)

```
MATCH (p1), (p2)

WHERE (p1:Hotel OR p1:Restaurant OR p1:Attraction)

AND (p2:Hotel OR p2:Restaurant OR p2:Attraction)

AND id(p1) < id(p2)

AND p1.postal_code = p2.postal_code

CREATE (p1)-[:SAME_POSTAL_CODE]->(p2)
```

Purpose: Links places that share the exact same postal code with a SAME_POSTAL_CODE relationship.

Explanation: Finds pairs of nodes that are either Hotel, Restaurant, or Attraction with the same postal_code. Ensures uniqueness by only creating the relationship when id(p1) < id(p2). If they share the same postal code, it creates a SAME_POSTAL_CODE relationship.

6)

```
MATCH (p1), (p2)

WHERE (p1:Hotel OR p1:Restaurant OR p1:Attraction)

AND (p2:Hotel OR p2:Restaurant OR p2:Attraction)

AND id(p1) < id(p2)

AND left(p1.postal_code, 3) = left(p2.postal_code, 3)  // First 3 digits often indicate same area

CREATE (p1)-[:NEARBY_POSTAL_AREA]->(p2)
```

Purpose: Creates a NEARBY_POSTAL_AREA relationship between locations within the same larger postal area.

Explanation: Matches two nodes if they are hotels, restaurants, or attractions and have the same first three digits in their postal codes (indicating proximity). The NEARBY_POSTAL_AREA relationship is created for these nodes.

7)

```
MATCH (r1:Restaurant), (r2:Restaurant)

WHERE r1.city_name = r2.city_name

AND id(r1) < id(r2)

AND any(x IN r1.cuisines WHERE x IN r2.cuisines)

AND point.distance(

   point({latitude: r1.latitude, longitude: r1.longitude}),

   point({latitude: r2.latitude, longitude: r2.longitude})

) <= 1000

CREATE (r1)-[:NEARBY_SIMILAR_CUISINE {

   common_cuisines: [x IN r1.cuisines WHERE x IN r2.cuisines],

   distance_meters: toInteger(point.distance(

      point({latitude: r1.latitude, longitude: r1.longitude}),

      point({latitude: r2.latitude, longitude: r2.longitude})

   ))

}]->(r2)
```

Purpose: Connects nearby restaurants that serve similar cuisines and are within 1 km.
Explanation: Matches pairs of Restaurant nodes in the same city with overlapping cuisines and within 1 km of each other. The relationship NEARBY_SIMILAR_CUISINE includes properties for the common_cuisines and distance_meters

8)

```
MATCH (w:Weather)
WHERE NOT EXISTS {
   MATCH (:City)-[:HAS_WEATHER]->(w)
}
RETURN w.city_name;
```

Purpose: Creates a relationship between City and Weather

6.3 LLM Code Samples

```python
from langchain.llms import HuggingFacePipeline
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

phi3 = "microsoft/Phi-3-mini-128k-instruct"
phi3_model = AutoModelForCausalLM.from_pretrained(
    phi3,
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
    )

tokenizer = AutoTokenizer.from_pretrained(phi3)

phi3_pipe = pipeline(
        "text-generation",
        model=phi3_model,
        tokenizer=tokenizer,

    )
generation_args = {
    "max_new_tokens": 500,
    "return_full_text": False,
    "temperature": 0.2,
    "do_sample": True,
}

local_llms = HuggingFacePipeline(
    pipeline=phi3_pipe,
    pipeline_kwargs=generation_args,
)
```

```python
from langgraph.graph import StateGraph, END
import json
from typing import TypedDict, Annotated, List, Dict
import operator
from langgraph.checkpoint.memory import MemorySaver
from langchain_core.messages import AnyMessage, SystemMessage,
HumanMessage, AIMessage, ChatMessage

memory = MemorySaver()
```

```python
class AgentState(TypedDict):
    user_prompt: str
    entity: str | Dict[str, List[str]]
    plan: str
    interupt_system_prompt: str
    get_spell_error: str | Dict[str, List[str]]
    missing_entity_query: str | Dict[str, List[str]]
    spell_fix_prompt: str
    was_missing_entities: bool
    topk: List[str]
    err: str
```

```python
builder = StateGraph(AgentState)
builder.add_node("extract", extract_node)
builder.add_node("get_spell_mistake", get_spell_mistake_node)
builder.add_node("get_missing_entity", get_missing_entity_node)
builder.add_node("spell_check", just_passer)
builder.add_node("spell_check_input_user", spell_fix)
builder.add_node("missing_entity_check", just_passer)
builder.add_node("missing_entity_input_user", just_passer)
builder.add_node("query_data", just_passer)
builder.add_node("query_snowflake", just_passer)
builder.add_node("query_neo4j", just_passer)
builder.add_node("create_itinerary", just_passer)

# builder.add_edge("extract", "missing_entity")
builder.add_edge("extract", "spell_check")

builder.add_conditional_edges("spell_check", should_ask_human_spell,
{True: "get_spell_mistake", False: "missing_entity_check"})
```

```
builder.add_conditional_edges("missing_entity_check",
should_ask_human_missing, {True: "get_missing_entity", False:
"query_data"})

builder.add_edge("get_spell_mistake", "spell_check_input_user")
builder.add_edge("get_missing_entity", "missing_entity_input_user")

builder.add_edge("spell_check_input_user", "extract")
builder.add_edge("missing_entity_input_user", "extract")

builder.add_edge("query_data", "query_snowflake")
builder.add_edge("query_data", "query_neo4j")
builder.add_edge(["query_snowflake", "query_neo4j"],
"create_itinerary")
# builder.add_edge("query_neo4j", "create_itinerary")
builder.add_edge("create_itinerary", END)
```

```
builder.set_entry_point("extract")
```

```
graph = builder.compile(checkpointer=memory,
interrupt_before=["spell_check_input_user",
"missing_entity_input_user"])
```

```
thread = {"configurable" : {"thread_id": "1"}}

for s in graph.stream({
    'user_prompt': my_sentence,
},
thread,
):
    print(s)
```

```
while graph.get_state(thread).next:
    print("\n", graph.get_state(thread).next, "\n")
    state = graph.get_state(thread)
    if graph.get_state(thread).next == ('spell_check_input_user',):
        print("**"*50, "Get Spell Error", "**"*50)
        state_key = get_state_data_in_json(state, "get_spell_error")
        # print(state_key)
        print(state_key["PROMPT"][0])
        input_data = input("Enter the correct spelling: ")
        user_prompt = f"""
        Earlier my prompt
        # {graph.get_state(thread)[0]["user_prompt"]}

        My above user prompy has the given spelling error.

        Additional Spell Error:
        {state_key["PROMPT"]}

        My Response to Above Spell Error:
        {input_data}
        """
        graph.update_state(thread, {"spell_fix_prompt": user_prompt})

    if graph.get_state(thread).next == ("missing_entity_input_user",):
        print("**"*50, "Missing Entity", "**"*50)
        state_key = get_state_data_in_json(state,
"missing_entity_query")
        print(state_key["PROMPT"][0])
        input_data = input("Enter the missing entities: ")
        user_prompt = f"""
        Earlier my prompt
        {graph.get_state(thread)[0]["user_prompt"]}

        Additional Missing Entities:
        {state_key["PROMPT"]}

        User Response to Above Missing Entities:
        {input_data}
        """

        graph.update_state(thread, {"user_prompt": user_prompt})

    for s in graph.stream(None, thread):
        print(s)
```

7. FUTURE TIMELINE AND CONCLUSION

Task	TimeLine	Milestones
1.Automate Data Ingestion (Snowflake to Neo4j)	Oct 28 – Nov 18	- Week 1: Design UI layout and structure. - Week 2: Implement input/output functionalities. - Week 3: Test integration with backend services.
2.Streamlit UI Creation & Connection	Nov 19 – Dec 2	- Week 4: Set up Snowflake schema and data pipeline. - Week 5: Integrate Snowflake and Neo4j; automate ingestion
3. Neo4j Query Generation via LLM	Nov 11 – Dec 2	- Week 3: Prototype Neo4j query generation using LLM. - Weeks 4-5: Test adaptability with various user inputs; integrate with Streamlit backend
4. Research GraphRAG	Dec 2 – Dec 9	- Week 6: Explore GraphRAG for enhanced retrieval and context. - Outline potential implementation approaches.
5. Optimize LangGraph Structure	Dec 5 – Dec 11	- Week 6: Assess and streamline node dependencies in LangGraph for better performance.

Conclusion and Future Considerations:

To date, we have successfully defined and gathered the essential data sources, including accommodation, attractions, and weather information. This data has been integrated into Neo4j, with entities modeled to reflect meaningful relationships that support user-specific recommendations. Additionally, our implementation of ML and NLP tools, combined with a RAG framework, prepares Travel Genie to dynamically process and respond to diverse user inputs, enhancing the relevance and accuracy of travel suggestions.

Looking ahead, our next steps are organized into targeted milestones to ensure timely progress and cohesion across all components. Beginning with automating the Data Ingestion from Snowflake to Neo4j to the creation and integration of Streamlit UI, we will establish a responsive interface for user interactions that is user centric. Concurrently, we will establish a prototype LLM-driven Neo4j query generation. Toward the end of the project timeline, we will look into researching more about GraphRAG and will explore new avenues for optimized retrieval, while adjustments to the LangGraph structure will address performance efficiency.

With these future milestones in view, we remain on track to complete a robust and user-centered travel planning platform that harnesses cutting-edge technologies to meet the needs of modern travelers.