Smart Doctor & Hospital Finder: Insurance Coverage & Specialty-Based Search

Arya Atulkumar Gaikwad (A69035152)
University of California San Diego

Devana Perupurayil (A69034326)
University of California San Diego

Harini Gurusankar (A16264685)
University of California San Diego

Shruti Prasad Sawant (A69033524)
University of California San Diego

Abstract

In this project, we created an application to consolidate healthcare information for Medi-Cal, a low-income state insurance program in California. To create this application we utilized a relational DBMS (Postgres) and a graph DBMS (Neo4j) and used Python as a connection between the systems. The data was collected from various Medi-Cal datasets found online, ranging from provider information to insurance data plan models and their costs. The data was preprocessed to remove duplicates and unwanted columns and linked together using primary and foreign keys in Postgres to query information. Networks were created in Neo4j using hospital, provider, and taxonomy datasets. After the relationships and data were set up, we created use-case queries to demonstrate the capabilities of our application, which is including but not limited to: finding the relationship between a hospital and a specialty and giving the user the desired hospital, selecting the right insurance plan based on healthcare needs and budget, and finding the capabilities of a hospital to suit the user's needs.

1 Introduction

In California, Medi-Cal is the health insurance option for low-income individuals and their dependents. Healthcare providers, insurance companies, and government agencies often struggle to manage large-scale datasets that span multiple domains, including hospitals, providers, insurance plans, and specializations. This fragmented data leads to inefficiencies in provider management, and difficulty for patients to access the healthcare information needed. The idea of providing universal access to healthcare means we need to provide equitable access to healthcare, which includes making the access of information easier for everyone. In order to solve this issue, we decided to centralize the information regarding Medi-Cal into one application, so that users can access the information they need with ease. The application will facilitate efficient data retrieval, improve network transparency, and enhance healthcare decision-making for all stakeholders.

In this paper we will discuss data sources, go through the methodology of creating our application, discuss results of some queries, and expand upon potential future applications for this project.

2 Data Sources

2.1 Provider Information

The first dataset under analysis is the Medi-Cal Managed Care Provider Listing [1], which contains comprehensive information on healthcare providers participating in the Medi-Cal program. With over 3.2 million records and 33 data attributes, this dataset provides details such as provider specialization, licensing, and whether they serve children. Additionally, it includes hospital affiliations, location coordinates, and service types, making it a valuable resource for understanding provider distribution across different regions of California. Given that Medi-Cal serves a significant portion of the state's population, this dataset is crucial for evaluating the accessibility of healthcare services for low-income and vulnerable communities.

2.2 Hospital Information

The second dataset, the Current California Healthcare Facility Listing [2], offers an extensive inventory of licensed healthcare facilities across the state. Comprising 10,908 records and 18 attributes, this dataset includes essential information such as facility type, location, emergency service availability, and bed capacity. By analyzing this dataset, researchers can assess the geographical distribution of hospitals and healthcare centers, identifying areas with limited access to emergency services or specialized care. The dataset is particularly useful for evaluating the adequacy of healthcare infrastructure in both urban and rural regions, ensuring that facilities are appropriately distributed to meet the needs of the population.

2.3 Hospital Ratings

The third dataset, California Hospital Performance Ratings [3], provides insights into hospital quality and risk-adjusted patient outcomes. With 25,975 records spanning 13 attributes, this dataset includes hospital names, event rates, and performance indicators based on adverse events. The dataset enables a deeper understanding of hospital safety and service quality by highlighting variations in performance across different institutions. By examining risk-adjusted event rates, policymakers can identify hospitals that may require quality improvement initiatives and allocate resources to enhance patient outcomes.

2.4 Taxonomy

The Medicare Provider and Supplier Taxonomy Crosswalk [4] dataset consists of 561 rows and 4 columns, providing a comprehensive mapping of taxonomy codes to their respective medical provider and supplier types. This dataset is essential for understanding the unique taxonomy codes assigned to different healthcare providers and their specializations, facilitating better data organization and connection to providers. The taxonomy crosswalk is widely used for regulatory, administrative, and analytical purposes in healthcare, ensuring that providers are categorized accurately within medical systems. The dataset is publicly available through the Centers for Medicare & Medicaid Services (CMS) and can serve as a foundational resource for studies related to healthcare classification and provider enrollment.

2.5 Insurance Plan

The CY 2023 Two-Plan Model Rates [5] dataset contains 644 rows and 10 columns, offering detailed cost information on various health plans for 2023. It includes key attributes such as the category of aid, geographic county distribution, and a cost breakdown using lower, midpoint, and upper bound values. This dataset is particularly valuable for analyzing the variations in healthcare costs across different plans and beneficiary categories, helping policymakers, insurers, and researchers understand pricing structures and financial implications of managed care plans. Sourced from the California Department of Health Care Services, this dataset provides transparency into Medi-Cal managed care rates, supporting research on healthcare affordability and accessibility.

3 Methodology

3.1 Data Pre-processing

Data pre-processing is necessary to ensure that we can conduct a worthwhile analysis. The datasets we currently have contain many unwanted attributes and are not consistent. For instance, the facility name in the Current California Healthcare Facility Listing and the Medi-Cal Managed Care Provider Listing are not exactly the same, which may cause issues when performing analyses that depend on data from these two datasets. Therefore, we performed the following processing using Python.

3.1.1 Current California Healthcare Facility Listing

From this dataset, we extracted the columns:

- OSHPD_ID
- FacilityName
- FacilityType
- Address, Address2, City, State, ZIP, County
- FACILITY_LEVEL_DESC, TOTAL_NUMBER_BEDS, ER_SERVICE_LEVEL_DESC
- FACILITY_STATUS_DESC, LICENSE_TYPE_DESC, LICENSE_CATEGORY_DESC

OSHPD_ID serves as the unique identification number for each hospital and is used as the primary key when creating the table. We ensured that OSHPD_ID is unique, no duplicate rows are present, and any rows with a null FacilityName are excluded. After pre-processing, we saved the data as `final_hospitals_data.csv`, consisting of 548 records.

3.1.2 Medi-Cal Managed Care Provider Listing

We first compared the similarity between the facility name in this file and the facility name in `final_hospitals_data.csv`. If the similarity was above 90%, we added the OSHPD_ID from `final_hospitals_data.csv` and removed other rows. Then, we extracted the following columns:

- providerid, firstname, lastname, managedcareplan, subnetwork, npi, taxonomy
- mcnaprovidergroup, mcnaprovidertype, licensuretype, primarycare, specialist
- seeschildren, telehealth, bhindicator, oshpd_id

The values in the dataset for **seeschildren** and **telehealth** were (B, O, N), which can be unclear, so we changed them to (Both, Only, No). Finally, we ensured that providerid values were unique, removed any duplicate rows, and excluded rows where the first name and last name were empty or null. Also, we selected only representatives working at at most 10 locations. After pre-processing, we saved the data as `providernetwork.csv`, consisting of 27,131 records.

3.1.3 California Hospital Performance Ratings

From this dataset, we extracted the columns:

- OSHPDID, Performance Measure, # of Adverse Events, # of Cases
- Risk-adjusted Rate, Hospital Ratings

We changed the column name **OSHPDID** to **OSHPD_ID** for consistency across all datasets. We followed the same procedure as before, removing any duplicates or unwanted data. Then we saved the processed data as `final_hospital_ratings_data.csv`, consisting of 7,267 records.

3.1.4 Medicare Provider and Supplier Taxonomy Crosswalk

From this dataset, we extracted the columns:

- PROVIDER TAXONOMY CODE
- MEDICARE PROVIDER/SUPPLIER TYPE DESCRIPTION
- PROVIDER TAXONOMY DESCRIPTION: TYPE, CLASSIFICATION, SPECIALIZATION
- MEDICARE SPECIALTY CODE

We ensured that the PROVIDER TAXONOMY CODE values were unique and that no duplicates were present. The cleaned data was saved as `final_taxonomy_data.csv`, consisting of 463 records.

3.1.5 CY 2023 Two-Plan Model Rates

From this dataset, we extracted the columns:

- _id, County, Health Plan Name, Category of Aid
- Lower Bound, Midpoint, Upper Bound

We did not consider the **Year** and **Model** columns, as their values were identical throughout the dataset for the 2023 Two-Plan Model. We ensured there were no duplicate rows, deleted rows without values for the lower bound, midpoint, or upper bound, and saved the cleaned data as `final_insurance_data.csv`, consisting of 645 records.

3.2 Database Schema Design

Our project utilizes two database technologies : Neo4j and PostgreSQL

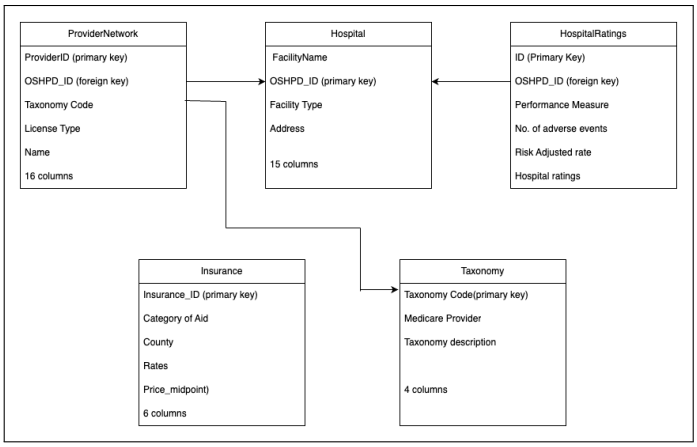

Figure 1: Relational Schema Diagram

3.2.1 PostgreSQL

This is a relational database that has structured tables to store hospitals, hospital ratings, providers, insurance plans and taxonomy codes.

1. **Hospitals**: This table stores the hospital related information like facility names, types, address, license type, total number of beds, license category etc. OSHPD_ID is the primary key.
2. **Hospital Ratings**: This table holds the ratings for each hospital per performance measure. The attributes include performance measure, number of adverse events, number of cases, risk adjusted rate and ratings. ID is the primary key and OSHPD_ID is a foreign key that refers to the OSHPD_ID present in the hospitals table. The risk adjusted rate refers to:

$$\text{Risk-Adjusted Rate} = \left(\frac{\text{Observed Events}}{\text{Expected Events}}\right) \times \text{Overall Average Rate}$$

   Where:

   - **Observed Events**: The actual number of adverse events recorded at a hospital.
   - **Expected Events**: The predicted number of adverse events based on patient risk factors such as age, underlying health conditions, and severity of illness.
   - **Overall Average Rate**: The benchmark rate calculated across all hospitals for the specific adverse event.

3. **ProviderNetwork**: This table stores information about the providers and their affiliated hospitals along with their specialization taxonomy details. It includes attributes like name, managed care plan, sub-network,

taxonomy code, licensure type, sees children, specialist, telehealth, BHIndicator. ProviderID is the primary key and OSHPD_ID is a foreign key that refers to the OSHPD_ID present in the hospitals table.

4. **Insurance Plan**: It contains information about the Medical plans and cost. Several attributes include ID, county, health plan, category of aid, cost. ID is the primary key.

5. **Taxonomy Codes**: It stores information about taxonomy codes and their specializations. The attributes include provider taxonomy code, provider taxonomy description, medicare specialty code and medicare provider supplier type description. Provider taxonomy code is the primary key.

3.2.2 Neo4j

This is a graph database that represents the relationships between various nodes. All the relationships used here are directed edges. It is used to model and query healthcare provider networks, insurance coverage and hospital relationships. We have 75,220 edges and 28,539 nodes. Therefore the edge to node ratio is 2.63.

Graph Nodes:

1. **Hospital**: Represents healthcare facilities.

2. **Provider**: Represents individual healthcare providers.

3. **SubNetwork**: Represents a network within a managed care plan.

4. **ManagedCarePlan**: Represents healthcare insurance plans covering various providers.

5. **Taxonomy**: Represents the specialization categories of healthcare providers.

Graph Relationships:

1. **WORKS_AT**: It connects a provider to a hospital.

2. **PART_OF**: Represents individual healthcare providers.

3. **UNDER**: Represents a network within a managed care plan.

4. **SPECIALIZES_IN**: Represents healthcare insurance plans covering various providers.

To run queries on PostgreSQL and Neo4j, we first established connections to both databases using the `psycopg2` and `neo4j` Python libraries. For Neo4j, we connected using the `GraphDatabase` module and defined a function `execute_query(query, params)` to execute Cypher queries. This function starts a session, runs the query, and returns the results in a structured format. For PostgreSQL, we used `psycopg2.connect()` to establish a connection and created a cursor (`pg_cur`) to execute SQL queries. A function `get_connection()` was implemented to handle connection setup and potential failures. To display query results in a structured manner, we used the `tabulate` library to format output as a readable table. This setup ensures efficient querying and retrieval of data from both databases within Python.

4 Results

4.1 Application Queries

Query 1:
Get the shortest path from a hospital to a specialty the user needs and return the shortest path of relationships. This result is then used in the Postgres query to return provider and hospital information regarding that specialty.
In this query, the user begins with at Holt Medical Center and is looking for the closest related hospital to the specialty 'Physician Assistants & Advanced Practice Nursing Providers/Nurse Practitioner, Acute Care'. Since Holt Medical Center does not have that specialty, it uses WORKS AT AND SPECIALIZES IN relations to find the path to the specialty. Then, in Python, we use list processing to get the hospital that provides that specialty, which is Anza Community Health. This hospital is linked through two providers with the same specialization working at each respective hospital. Then, the name of this hospital is fed into the Postgres query to get information about the providers that work there and their license type, if they see children, and if they are a primary care physician or specialist.

```
query = """MATCH p=shortestPath((h1:Hospital {
FacilityName: 'HOLT MEDICAL CENTER'})
-[:WORKS_AT|SPECIALIZES_IN*]-
(t:Taxonomy {Description:
'Physician Assistants & Advanced Practice
Nursing Providers/Nurse Practitioner, Acute Care'}))
RETURN p;"""
result = execute_query(query)

def extract_facility_names(data):
   facility_names = []
   for entry in data:
       if 'p' in entry:
           for item in entry['p']:
               if isinstance(item, dict) and
               'FacilityName' in item:
                   facility_names
                   .append(item['FacilityName'])
   return facility_names
```

```
# Get facility names
facility_names = extract_facility_names(result)
target_facility = facility_names[-1]
print("Facilities in path:", target_facility)
print('\n')

postgres_query1 =
f"SELECT DISTINCT h.FacilityName,
CONCAT(LOWER(p.FirstName), ' ',
LOWER(p.LastName)) AS provider_name,
p.MCNAProviderType AS provider_type,
p.PrimaryCare, p.Specialist,
p.SeesChildren, p.BHIndicator,
p.LicensureType FROM hospitals h
INNER JOIN
providernetwork p ON p.oshpd_id = h.oshpd_id
WHERE h.FacilityName = '{target_facility}';"
pg_cur.execute(f'''{postgres_query1}''')
details = pg_cur.fetchall()
```

Query 2

Get the best/worst rated hospital (using average of risk-adjusted ratings) and use facility names in Neo4j to find specialties provided.

In this query, we start off with a Postgres query that finds the average risk-adjusted rate for each hospital and selects the best and worst hospital in terms of these ratings. The best hospital has the lowest rate, while the worst hospital has the highest rate. Then, these names are selected from the list returned by Postgres and fed into the Neo4j query that then gets a list of the specialties of each hospital, which is found through the providers by utilizing the WORKS AT and SPECIALIZES IN relationships.

```
pg_cur.execute("""WITH HospitalRatings AS (SELECT
        h.FacilityName,
        AVG(hr.Risk_adjusted_rate) AS
        Avg_Risk_Adjusted_Rate
    FROM hospital_ratings hr
    JOIN Hospitals h ON hr.OSHPD_ID = h.OSHPD_ID
    GROUP BY h.FacilityName
)
(
    -- Best-rated hospital
    (Lowest risk-adjusted rate)
    SELECT FacilityName, Avg_Risk_Adjusted_Rate
    FROM HospitalRatings
    ORDER BY Avg_Risk_Adjusted_Rate ASC
    LIMIT 1
)
UNION ALL
(
    -- Worst-rated hospital
    (Highest risk-adjusted rate)
    SELECT FacilityName, Avg_Risk_Adjusted_Rate
    FROM HospitalRatings
    ORDER BY Avg_Risk_Adjusted_Rate DESC
    LIMIT 1
);""")
results = pg_cur.fetchall()
print('Display the worst and the best rated
hospital with their ratings')
print_table(results, [desc[0] for
desc in pg_cur.description])


best_rating_hospital = results[0][0]
worst_rating_hospital=results[1][0]

query1 = """
MATCH (h:Hospital)<-[:WORKS_AT]-
(p:Provider)-
[:SPECIALIZES_IN]->(t:Taxonomy)
WHERE h.FacilityName = $hospital_name
RETURN h.FacilityName AS Hospital,
COLLECT(DISTINCT t.Description)
AS Specialties;
"""

result1 = execute_query(query1,
{"hospital_name": worst_rating_hospital})
result2 = execute_query(query1,
{"hospital_name": best_rating_hospital})

print('\n\nDisplay the specialities for
the worst and the best rated hospital:')
print_table(result1+result2)
```

The combined query then results in the following table:

Display the worst and the best rated hospital with their ratings

facilityname	avg_risk_adjusted_rate
WEST COVINA MEDICAL CENTER	0
VO MEDICAL CENTER	12.9667

Display the specialities for the worst and the best rated hospital:

Hospital	Specialties
VO MEDICAL CENTER	['Eye and Vision Service Providers/Optometrist', 'Allopathic & Osteopathic Physicians/Internal Medicine, Rheumatology', 'Allopathic & Osteopathic Physicians/Psychiatry ']
WEST COVINA MEDICAL CENTER	['Allopathic & Osteopathic Physicians/Orthopedic Surgery', 'Allopathic & Osteopathic Physicians/Psychiatry ', 'Allopathic & Osteopathic Physicians/Colon & Rectal Surgery']

Figure 2: Query 2 Result

4.2 Addendum: Care Plan Within a Zip Code using Graph Topology

This section is an extra algorithm we chose to experiment with on the Neo4j side. It creates new relationships that we chose not to add as part of our schema design, but rather just to make this query. This query outlines the process of constructing a healthcare network graph in Neo4j, where ZIP codes, hospitals, and managed care plans are connected through relationships. The goal is to estimate the probability of a ManagedCarePlan being available in a given ZIP code based on its connectivity within the graph. This is achieved using shortest-path algorithms and graph data science (GDS) techniques.

Step 1: Creating ZIP Nodes
To establish a structured network, we begin by creating ZIP nodes representing unique ZIP codes found in the dataset. Each hospital has an associated ZIP code, and we extract these distinct ZIP values to form individual ZIP nodes in the graph.

```
MATCH (h:Hospital)
WITH DISTINCT h.zip AS zip_code
CREATE (:ZIP {code: zip_code});
```

Step 2: Establishing the LOCATED_IN Relationship
Hospitals are assigned to specific ZIP codes through the LOCATED_IN relationship. This provides a direct way to determine which hospitals are available within a specific ZIP code. This connection is established using the following query:

```
MATCH (h:Hospital)
MERGE (z:ZIP {code: h.ZIP})
MERGE (h)-[:LOCATED_IN]->(z);
```

Step 3: Mapping ACCEPTS Relationship Between Hospitals and ManagedCarePlans
The ACCEPTS relationship is used to connect hospitals to ManagedCarePlans that they accept. The relationship is created only when a hospital has at least one provider accepting a specific managed care plan.

```
MATCH (p:Provider)-[:WORKS_AT]->(h:Hospital),
(m:ManagedCarePlan)
WHERE p.managedcareplan = m.name
MERGE (h)-[:ACCEPTS]->(m);
```

Step 4: Defining IS_NEAR Relationship Between ZIP Codes
The 5 closest ZIP codes to each ZIP node are determined based on ZIP code number difference. This enables the shortest path algorithm to explore alternative routes through nearby ZIP codes when a direct connection to a ManagedCarePlan is unavailable.

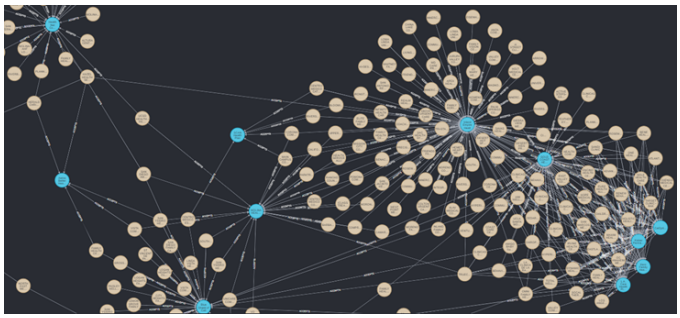

Figure 3: ACCEPTS Relationship in Neo4j

```
MATCH (z1:ZIP), (z2:ZIP)
WHERE z1 <> z2  // Avoid self-connections
WITH z1, z2,
abs(toInteger(z1.code) - toInteger(z2.code))
AS zip_difference
ORDER BY zip_difference ASC
// Sort by numeric proximity
WITH z1,collect(z2)[..5]  AS  nearest_neighbors
// Take the 5 closest ZIP codes
UNWIND nearest_neighbors AS neighbor
MERGE (z1)-[:IS_NEAR]->(neighbor);
```

Step 5: Creating a Graph for Analysis
This process prepares the graph for running advanced algorithms by storing the relationships in an optimized format. To perform shortest path calculations efficiently, we project the graph into the Graph Data Science (GDS) Library:

```
CALL gds.graph.project(
'zipPlanGraph',
['ZIP', 'Hospital', 'ManagedCarePlan'],
{
LOCATED_IN: {orientation: 'UNDIRECTED'},
ACCEPTS: {orientation: 'UNDIRECTED'},
IS_NEAR: {orientation: 'UNDIRECTED'}
}
);
```

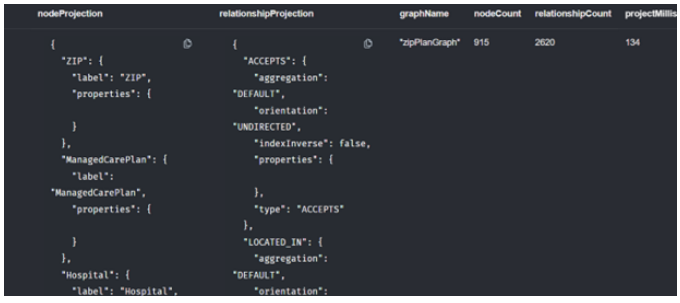

Figure 4: Graph Structure in Neo4j

Step 6: Computing Shortest Path and Similarity Score
We calculate the probability of a ManagedCarePlan being available in a ZIP code using Dijkstra's shortest path algorithm:

```
MATCH (z:ZIP), (m:ManagedCarePlan)
CALL
gds.shortestPath.dijkstra.stream(
'zipPlanGraph',
{
sourceNode: z,
targetNode: m,
relationshipWeightProperty: NULL
})
YIELD totalCost
RETURN
  z.code AS zip_code,
  m.name AS plan_name,
  CASE
    WHEN totalCost IS NOT NULL
    THEN 1.0 / (totalCost + 1)
    // Convert distance to probability
    ELSE 0
  END AS probability_score
ORDER BY zip_code, plan_name;
```

A lower distance means a higher probability. If no path exists, the probability score is 0, indicating no connection between the ZIP code and the ManagedCarePlan.

zip_code	plan_name	similarity_score
"90001"	"AIDS Healthcare Foundation"	0.14285714285714285
"90001"	"Aetna Better Health"	0.2
"90001"	"Alameda Alliance for Health"	0.1111111111111111
"90001"	"Blue Shield of California Promise"	0.2
"90001"	"CENTRAL CALIFORNIA ALLIANCE FOR HEALTH"	0.14285714285714285
"90001"	"CalOptima"	0.14285714285714285

Figure 5: Likelihood score of ManagedCarePlan for ZipCode

5 Conclusion

Our project aimed to create a centralized application to help users access Medi-Cal healthcare information efficiently. The application integrates data from multiple sources and uses relational databases (PostgreSQL) and graph databases (Neo4j). Key components include:

- Data Preprocessing: Data cleaning and standardization to ensure consistency and usability.
- Database Design: Creating relational tables in PostgreSQL and graph nodes/relationships in Neo4j to model the healthcare network.
- Querying and Analysis: Demonstrating the capabilities of the application through relational and graph-based queries, such as finding the shortest path between hospitals, ranking hospitals by primary care physicians, and identifying healthcare providers within specific cost ranges.

Lessons Learned:

- Defining the foundation for an application is critical and requires explicit definitions.
- We learned how to combine data sets from different sources, which required significant pre-processing to ensure consistency and remove duplicates. We also handled missing or inconsistent data that required careful cleaning and standardization.
- We gained experience designing relational tables, which involved careful consideration of primary and foreign keys to maintain data integrity in PostgreSQL, as well as modeling relationships between nodes for efficient querying and pathfinding in Neo4j.
- Additionally, we learned about query optimization by writing efficient SQL queries for relational data, which required an understanding of aggregation functions and filtering. In Neo4j, we leveraged graph algorithms like Dijkstra's shortest path for finding connections between hospitals and insurance plans.

Future Work:
With more time, this project could be expanded into a recommender system based on proximity. As of right now, we can find the hospital, specialty, or provider by using relationships. By adding more datasets regarding location and creating more relationships, we could give users recommendations of hospitals based on the proximity to their location as well.

References

[1] Medi-cal managed care provider listing - dataset - california health and human services open data portal. (n.d.). Link.

[2] Licensed healthcare facility listing - dataset - california health and human services open data portal. (n.d.-a). Link.

[3] California hospital performance ratings. Link.

[4] Medicare provider and supplier taxonomy crosswalk. Link.

[5] Cy 2023 two-plan model rates. Link.