Andy Mateo

Project 1 (HetioNet)

Design Diagram:

# MongoDB Query (Question 1)

The MongoDB query uses the aggregation pipeline to retrieve all the required information about a disease in a single query. Here's a breakdown of the pipeline stages:

1. `$match`: Filters documents based on the provided disease ID
    a. Filters based on the initial disease ID.
2. `$lookup`: Joins the edges collection to find treating and palliative drugs.
    a. Joins edges to find treating and palliative **drugs**
3. `$lookup`: Joins the edges collection to find genes that cause the disease.
    a. Joins edges to find genes that **cause disease**
4. `$lookup`: Joins the edges collection to find locations where the disease occurs.
    a. Joins edges to find **where** in the body the disease occurs (anatomy)
5. `$project`: Shapes the final output, including filtering drugs by their relationship type (treating or palliative).

Overall, After being given the provided disease ID, the aggregation pipeline searches for information in 3 different lookups and then compiles it all in the project.

# Aggregating Data Items in MongoDB Compass

The provided hetionet.py file in the def query_disease(self, disease_id): has the full pipeline on aggregating into the compass if you wish to do it there rather than in the original python editor. It's straightforward to apply as in the Compass interface you navigate to nodes > Aggregations > Add Stage > and add the steps above as shown in the code. First stage is a match, followed by the lookups and ending with the project.
In the file this would start on line 38.

# Advantages of Using MongoDB for Question 1

- We used MongoDB's document model because it allows us to not only represent a lot of different TYPES of nodes without a rigid scheme but it also allows us to compile that information which was gathered from different lookups into one output quickly.

- The nature of how documents and lookups work mean that MongoDB should have fast read performance. Additionally it can be Scaled indefinitely which is a characteristic that most modern databases desire.

## Compared to Other Approaches:

- Relational Databases: While they could be used, they might require more complex JOIN operations and dont have the flexibility that documents provide.
- Graph Databases: Graph Databases would perform well for this task and are a great alternative to the document model. However, since we are prioritizing just getting data from a particular node,disease,etc it’s not as strong since we’re not looking for relationships between nodes ( where graphs excel)
- Key-Value Stores and Column-Family Stores have limited query flexibility and aren't well suited for this type of data. Key-value stores doesn't have good access to secondary indexes and representing relationships, column-family stores has a similar issue for complex data aggregations

## Potential Improvements for Question 1 MongoDB Query:

- The original implementation to solve question 1 was actually solved using neo4j and then with the intention to add MongoDB as a cache in order to improve performance. As a result frequently accessed diseases information can be retrieved quicker. This would provide a behemoth of benefits such as
    - Reduces the need to query Neo4j for related data, further improving response times.
    - Ensuring that commonly requested data is always available in cache.
- However, if we stay with our current implementation, I believe that adding optimized indexes that can assist with common query patterns would speed up queries by reducing the data MongoDB needs to scan through. Additionally increasing the # of bulk operations would improve the write performance as many of us had to do in order to import the data originally.

# Neo4j Query (Question 2)

The Neo4j query uses cypher queries in order to retrieve the necessary information to find all the compounds that can treat a new disease.

## Building the Query

Two queries were used to build the database. One to create the nodes and the other to create the relationships between nodes.

### 1. Query to create nodes

```
LOAD CSV WITH HEADERS FROM 'file:///nodes.tsv' AS csvLine FIELDTERMINATOR '\t'
CALL apoc.create.node([csvLine.kind], {id: csvLine.id, name: csvLine.name}) YIELD node
RETURN node
```

For this query the Awesome Procedures on Cypher (APOC) library is used to create nodes with dynamic labels. The labels of each node is given by the node's kind (disease, drugs, genes, and compounds).

### 2. Query to create relationships

```
:auto LOAD CSV WITH HEADERS FROM 'file:///edges.tsv' AS csvLine
FIELDTERMINATOR '\t'
CALL{
   WITH csvLine
   MATCH (source {id: csvLine.source}), (target {id: csvLine.target})
   CALL apoc.create.relationship(source, csvLine.metaedge, {metaedge: csvLine.metaedge},
target)
   YIELD rel
   RETURN rel
} IN TRANSACTIONS OF 500 ROWS
RETURN count(rel) AS relationshipsCreated
```

For this query, APOC is once again used to create relationships with dynamic labels. The labels of each node is given by the node's meta edge. In order to speedup the creation of the database, relationships are formed in batches of 500 operations.

# Query to Find All Compounds that Can Treat a New Disease Excluding Existing Durgs

```
MATCH (c:Compound)-[r1]->(g:Gene)
MATCH (d:Disease {id: $diseaseID})-[:DlA]->(a:Anatomy)-[r2]->(g)
WHERE NOT (c)-[:CtD]->(d)
AND (
   (r1.metaedge = 'CuG' AND r2.metaedge = 'AdG')
   OR
   (r1.metaedge = 'CdG' AND r2.metaedge = 'AuG')
)
RETURN DISTINCT c.name AS compound
```

This query prints out all compounds that can treat a new disease excluding existing drugs by looking at all compounds that either up-regulate or down-regulate a gene and that also have a location that down-regulates or up-regulates that same gene in the opposite direction of where the gene occurs.

# Advantages of Using Neo4j for Question 2

- We used Neo4j's graph model because our data involves looking up various relationship types such as CuG, AdG, CdG, and AuG between nodes. Using a graph database, we can easily represent these complex relations
- Graph databases are also optimized to traverse these multi-level relationships, providing us with fast lookup that other database types wouldn't be able to offer

# Compared to Other Approaches:

- Relational Databases: Relational databases are structured around predefined schemas. They don't have the flexibility required to handle queries that span across many different relationships.
- Document Databases: Similar to relational databases, document databases provide for limited flexibility, making it difficult to efficiently model and query the complex relationships required to retrieve the correct data.
- Key-Value Stores and Column-Family Stores: These types of databases inherently do not support complex relationships. These types of databases are not meant to be used for lookup between interconnected nodes.