**Project I: HetioNet Database System Documentation**

**1. Introduction**

This project models the HetioNet biomedical network using a NoSQL database. The system supports the following functionalities:

* **Data Ingestion:** Import nodes and edges from TSV files to construct a network of biomedical entities.
* **Query Support:** Answers two queries:
  1. Given a disease ID, retrieve its name, the drug names that treat or palliate it, the gene names that cause it, and the locations where it occurs—all in a single query.
  2. Identify compounds that can potentially treat a new disease based on regulatory relationships between compounds, genes, and disease locations, excluding compounds already known to treat the disease.
* **Feature Engineering**: Utilizes regular expression generators to clean feature names
* **Batch Upserting**: Uses batching to reduce computation needed.
* **GUI**: Contains a GUI that allows users to run the same CLI commands in a seamless manner.

A screenshot of a computer

AI-generated content may be incorrect.

The system provides a Python command-line interface (CLI) for creating the database and executing queries as well as a Streamlit GUI that allows the same functionality.

**2. Database Design**

**NoSQL Stores Used:**

* **MongoDB (Document Store):**
  + **Purpose:** Store detailed metadata for each node.
  + **Rationale:** MongoDB is usually used for storing JSON-like documents with [flexible schemas](https://www.mongodb.com/resources/basics/unstructured-data/schemaless), which is beneficial for data like our samples that might have variable attributes. JSON is typically the preferred data structure used for NLP LLM metadata storing as well so it makes sense.
* **Neo4j (Graph Database):**
  + **Purpose:** Store nodes and edges to capture the complex relationships among data objects.
  + **Rationale:** Neo4j is optimized for traversing relationships and executing complex queries over heterogeneous networks, making it a natural fit for the HetioNet model.

A diagram of a data flow

AI-generated content may be incorrect.

**3. Overview**

**Code Structure:**

* **main.py:**  
  Provides the CLI for database creation and query execution. It reads TSV files for nodes and edges, then calls functions to insert data into MongoDB and Neo4j.
* **db/document.py:**  
  This ontains the DocumentDB class, which encapsulates my MongoDB functions, including an insert\_node method for upserting node documents.
* **db/graph.py:**  
  Contains the GraphDB class for Neo4j interactions. It provides methods for:
  + Creating nodes (with dynamic labels based on the node's kind).
  + Creating edges between nodes.
  + Executing Query 1 and Query 2 using Cypher queries.

**Query Details:**

* **Query 1:**  
  The query retrieves a disease’s name and aggregates related drug names, gene names, and location names. It uses Cypher's OPTIONAL MATCH to gather data across different relationship types (e.g., TREATS/PALLIATES, CAUSES, OCCURS\_IN/LOCALIZES\_TO).
* **Query 2:**  
  The query implements the rule-based logic: if a compound up-regulates/down-regulates a gene, and the disease’s location down-regulates/up-regulates that gene in the opposite direction, then the compound can potentially treat the disease. The query excludes compounds already linked via a TREATS relationship.

**4. Potential Improvements**

* **Performance Enhancements:**  
  With large datasets (e.g., 23,000 nodes and 1.3 million edges), the current implementation batches the edges before insertion. The reason is because when I ran it for each row I saw an estimated computation time of >2 hours.

A screenshot of a computer

AI-generated content may be incorrect.

However, after implementing batching it still ended up taking a similar amount. This leads me to suspect there is still some bottle-necking (perhaps due to me using Docker for Neo4j/MongoDB.

* This could be improved by:
  + Experiment alternatives to using Cypher’s UNWIND clause.
  + Re-run code on a Linux OS to see if Windows WSL2 was bottlenecking my performance.
  + Using Neo4j’s bulk import tools for one-time database creation.
* **Client Interface Enhancements:**  
  Although there is a basic Streamlit GUI that allows you to select the operations, it can be improved so that users can maybe upload their own files directly. Currently, its only hard coded. The UI could also be improved too.
* **Advanced Error Handling & Logging:**  
  Further improvements could include more robust error handling, logging mechanisms, and performance metrics, as I haven’t implemented all of these safety features in my demo.

**5. Conclusion**

The current implementation meets the project requirements by:

* Modeling HetioNet using two types of NoSQL stores (MongoDB and Neo4j).
* Providing a Python command-line interface for database creation and query execution.
* Implementing two key queries that extract the required biomedical information.
* Presenting a GUI alternative via Streamlit.