———Big Data Project 1: HetioNet

Team Members:

Asafe Brandao - Neo4j Implementation

Fabiola Li Wu - MongoDB Implementation

Course: Big Data

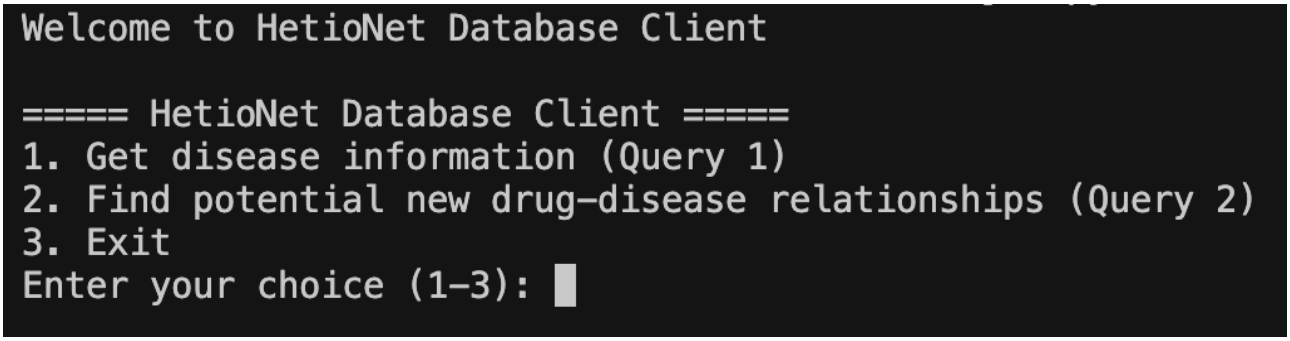
Instructor: Arezoo Bybordi

Due Date: Monday, March 17th, 2025

Project Overview:

The HetioNet is a tool to understand the complex relationships between diseases, genes, compounds, and anatomy. This project focuses on building a database to efficiently handle queries related to disease location, genes, and treatment by utilizing MongoDB and Neo4J NoSQL data stores. The system should be able to answer crucial questions regarding disease information, treatment options, and their biological connections using quick response times.

Client Interface:



This is a simple but functional command-line interface (CLI). The interface loops until the user selects option 3 (Exit).

MongoDB Implementation (Document Database)

Setup on macOS

We followed several resources to install MongoDB on macOS and integrate it with Visual Studio Code for ease of development and database management:

- ➔ **MongoDB Installation on macOS:**

  We followed the instructions from the official MongoDB documentation to install MongoDB on macOS:

  MongoDB Installation for macOS

- ➔ **MongoDB Integration with Visual Studio Code:**

  We also followed the steps outlined in the Visual Studio Code documentation for MongoDB:

  MongoDB in Visual Studio Code

- ➔ **Additional Video Tutorial:**

  We watched YouTube tutorials such as this one for further guidance on installing and connecting MongoDB with Visual Studio Code.

- ➔ **Connecting MongoDB with Visual Studio Code:**

  We followed the MongoDB VSCode extension setup process provided here:

  MongoDB VSCode Extension

MongoDB Class

We created the MongoDB class to store all functions related to MongoDB such as cleanDatabase(), loadNodes(), loadEdges(), diseaseInfo(diseaseID), and missingEdges()

Cleaning the Database:

Before uploading data, we ensured the MongoDB database was clean because we did not want unnecessary or old collections that might interfere with the new data upload process. We used the drop() method to remove it and print something when it was done.

Loading Data into MongoDB:

loadNodes()

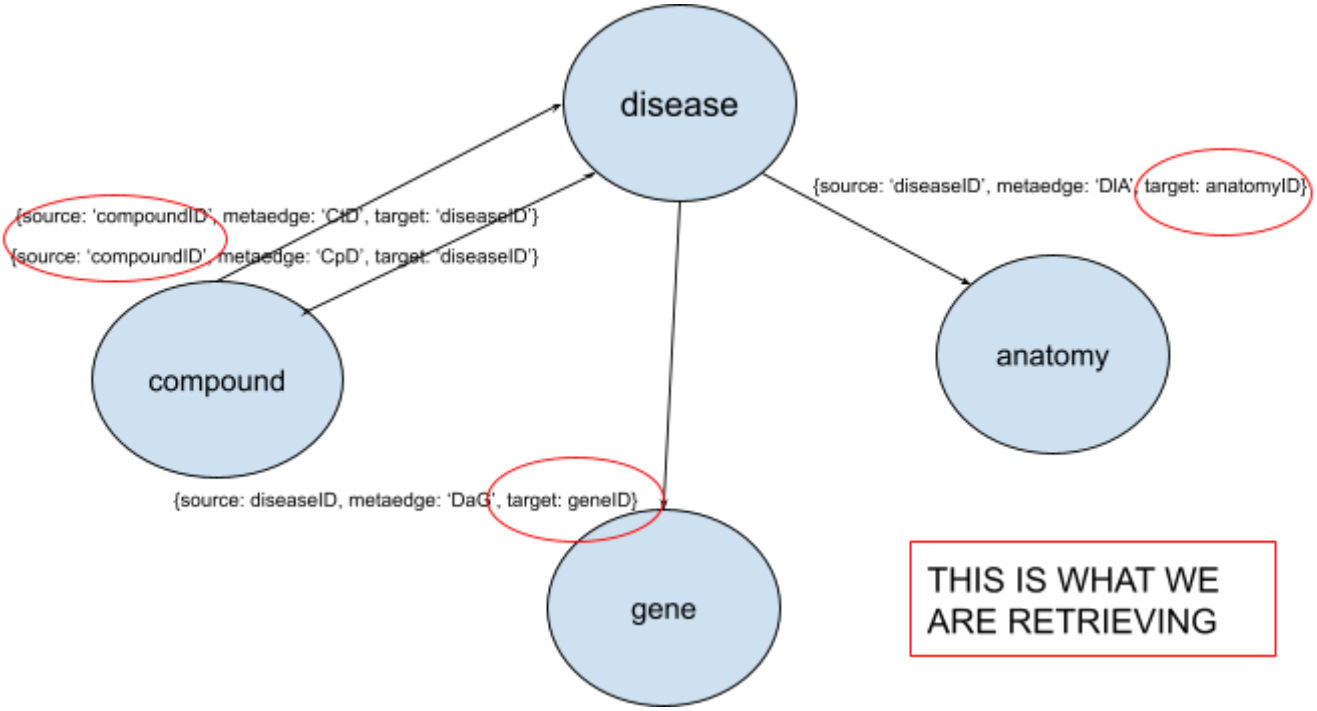
- It reads the data from the file nodes.tsv.
- Instead of storing id in the database, it stores it as _id, otherwise, it would have two IDs created which are _id(object ID) and id(one of the members).
- It stores the data by column and in order [‘_id’, ‘name’, ‘kind’].
- It converts the DataFrame into lists of dictionaries.
- Then, it inserts data into the nodes collection in MongoDB.
- Outputs the total of nodes uploaded to the MongoDB database.

loadEdges()

- It reads the data from the file edges.tsv.
- It stores the data by column and in order [‘source’, ‘metaedge’, ‘target’].
- It converts the DataFrame into lists of dictionaries.
- Then, it inserts data into the edges collection in MongoDB.
- Outputs the total of edges uploaded to the MongoDB database.

Query #1: diseaseInfo(diseaseID)

For this query, we are given a disease ID and we are requested to print the ID, name, drugs that palliates(CpD) and treat the disease(CtD), the genes associated with the disease(DaG), and where the disease can be localized(DlA).



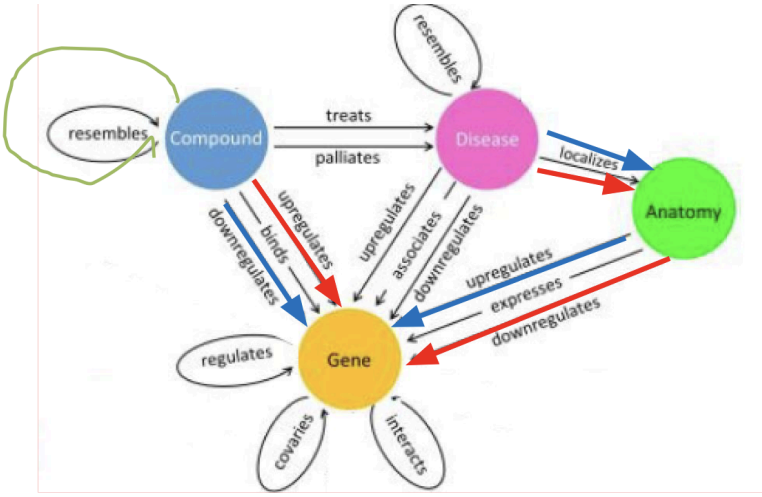
The logic/algorithm behind this is that the disease for CpD and CtD edges is located in the 'target' of the edge and for DaG and DlA, the disease is located in the 'source' of the edge. After looking at all the nodes and placing them into target_node and source_node, we join them with the command $project. Then, we process the data that we have and classify them by the interested 'metaedges'. In the end, we retrieve the data and print them out and it looks like this:

```
----- RESULTS -----
ID: Disease::DOID:263
Disease: kidney cancer
Treating Drugs: Capecitabine, Dactinomycin, Doxorubicin, Vincristine, Everolimus, Sorafenib, Gemcitabine, Temsirolimus, Paclitaxel, Pazopanib,
 Vinblastine, Erlotinib, Sunitinib
Palliating Drugs: Hydrocortisone, Esomeprazole, Omeprazole, Celecoxib, Dexamethasone, Betamethasone
Genes: CALB1, ANGPTL1, DNASE1L3, DNAJC11, ASB9, BPHL, DMRT2, ACACB, CYB5D2, COX5A, ABHD4, AUH, CRYM, CGN, DOLPP1, ANXA9, ABAT, AMFR, CTH, AK3,
 CYP4F3, CGNL1, ACSF2, DNAJC12, DCDC2, ATP1B2, CDKN1C, APOO, COBLL1, ATP6V0D1, ARG2, CENPV, CTSH, AFAP1L2, AGMAT, C9orf24, ADAL, CDO1, CADM1,
ARNT2, C1orf116, AGPAT3, DSP, ADCK1, C3orf52, ACSM3, CFAP221, AFM, ACADSB, CLPTM1, ASAP2, CYP24A1, CYP4A22, ADHFE1, CFAP70, CCBE1, CRABP1, AHC
YL2, ATP6V1B1, AQP3, CNNM2, CMTM4, CDC14B, DNMT3L, CLIC5, DNMBP, CASR, COL4A6, ATP6V0E2, AJAP1, ATP1A1, ACSM2B, CLSTN2, APEH, CCDC64, BCAM, AT
P5G3, CTNNAL1, CHN2, AHCYL1, CORO2B, DPYS, ABCA8, DPEP1, ADK, COL9A2, C6orf203, ACOX2, CLMN, ABCB1, AGXT2, BDH1, DNER, ACOX1, BDKRB2, BLCAP, A
OX1, AMDHD1, ARFGEF1, CACHD1, CRYAA, C3orf18, AASS, ANGPT1, CYP4F12, AREL1, BAP1, ANK3, ACO1, CKB, CPEB4, ABCC5, AZGP1, CNP, CBWD3, ACSS3, ABH
D10, COL4A4, BMP7, AMT, ACAD10, ATPAF1, CAMK2A, CACNA2D2, ACAD8, CDH3, ACOT12, C16orf89, BHMT2, C14orf37, DDC, APOM, C5, AIFM1, ALDOB, APCDD1L
, ANGPTL3, ALDH4A1, ABHD14B, CA8, CAB39L, AQP2, AGXT, CSDC2, DIO1, CEL, ALDH1B1, DNAJC16, AIF1L, CNTN1, CYFIP2, CKMT1A, AQP11, CYP4A11, DCXR,
CALML3, C16orf45, CHL1, ABHD17C, ADTRP, BTD, CYP39A1, DHRS1, DIRAS3, CASP9, ANXA3, ALAS1, ARHGAP24, ARSD, ARSF, ATP6V0A4, BTBD11, AK1, ADH6, D
PP6, ANPEP, B4GALNT3, DHTKD1, C11orf71, CFI, ACPP, C8orf4, CHGB, ALAD, CA4, BAG1, COX7B, CXCL14, ABCD3, ABCC6, CAND2, BAIAP2, COMMD8, CTNS, AC
SL3, CNTN3, C11orf54, CPM, C7, CAT, CYP8B1, CISH, CPNE6, AP1M2, BCL7A, CA10, CDC37L1, C1QTNF7, DNASE1, CXCL12, BAMBI, CNGA1, BDH2, ABLIM1, ATR
NL1, DNAJC6, ACY1, DCN, CPN2, BEND7, AMPH, DHRS11, DACH1, ABHD14A, CYP4V2, CRHBP, ADH1B, CAPS, CHODL, ABCA9, ATP6V1C2, ADORA1, ANKRD46, APOD,
C1orf168, COL4A5, ANO5, CHP1, C1orf115, ATP5S, CKMT2, ALDH5A1, ABCA6, ALB, COBL, DEPTOR, ANK2, BSPRY, DHX30, CLDN10
Locations: renal pelvis, hepatic vein, nephron tubule, ureter, renal system, renal vein, renal artery, gonad, iris, collecting duct of renal t
ubule, nephron, cortex of kidney, cardiac atrium, juxtaglomerular apparatus, posterior vena cava, kidney, urine, renal papilla
```

Query #2: missingEdges()

This query identifies and processes "missing edges" with a biological and chemical network by assuming that a compound can treat a disease if the compound up-regulates a gene, but the location down-regulates the gene and if the compound down-regulates a gene, but the location up-regulates the gene.

This was the initial brainstorm for query 2, but it did not work because it took very long to execute the missing edges.



CtD if{

diseaseLocalizeAnatomy = find all { metaedge: 'DlA'} edges and from that do:

compoundDownregulatesGene = find all { metaedge: 'CdG'}

compoundDownregulatesGene = find all { metaedge: 'CdG'}

diseaseUpregulatesGene = find all edges that have { metaedge: 'AuG'} && source is in diseaseLocalizeAnatomy.target

diseaseDownregulatesGene = find all edges that have { metaedge: 'AdG'} && source is in diseaseLocalizeAnatomy.target

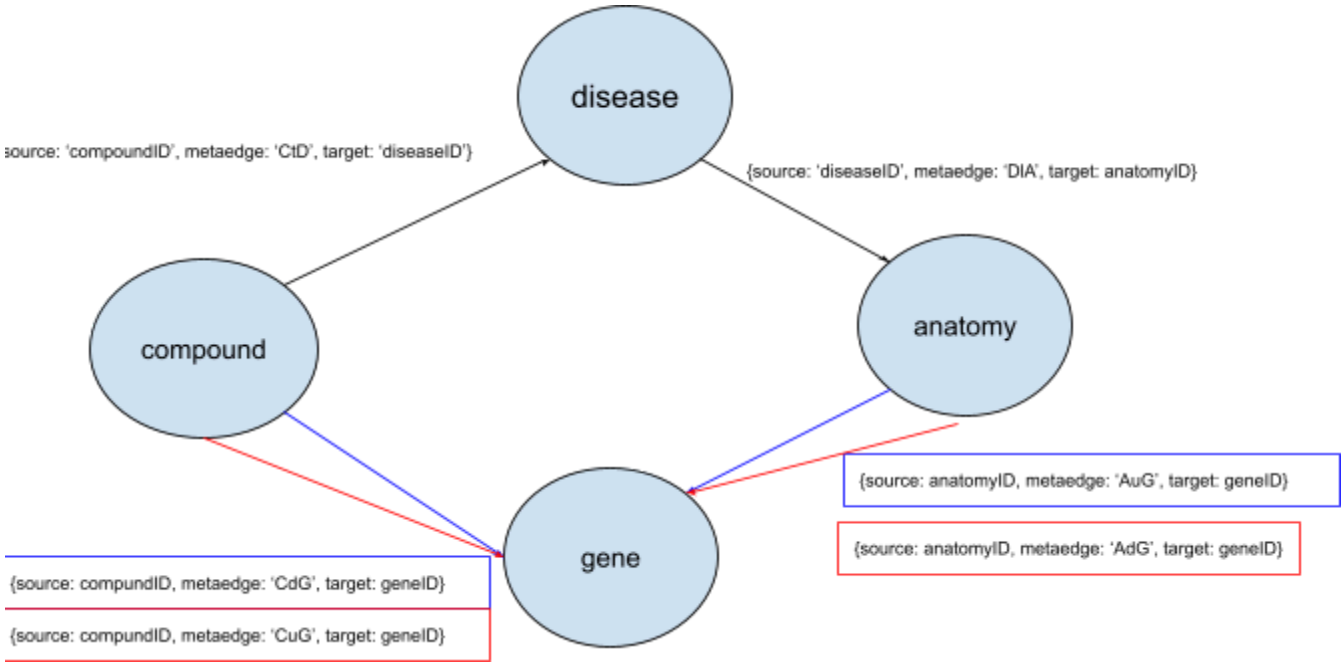
CompoundTreatsDiseaseRoaster = for every element if diseaseUpregulatesGene and compoundDownregulatesGene have the same target gene, then put it in this list as source = diseaseLocalizeAnatomy.source, metaedge = ‘CtD’, target = the gene that these two have in common in their target edge. Do the same thing if diseaseDownregulatesGene and compoundUpregulatesGene have the same target gene.

CompoundTreatsDisease = all the edges with { metaedge: 'CtD'}

Remove the ones from CompoundTreatsDiseaseRoaster that are in CompoundTreatsDisease and return CompoundTreatsDiseaseRoaster

}

Therefore, this was our second try and it worked



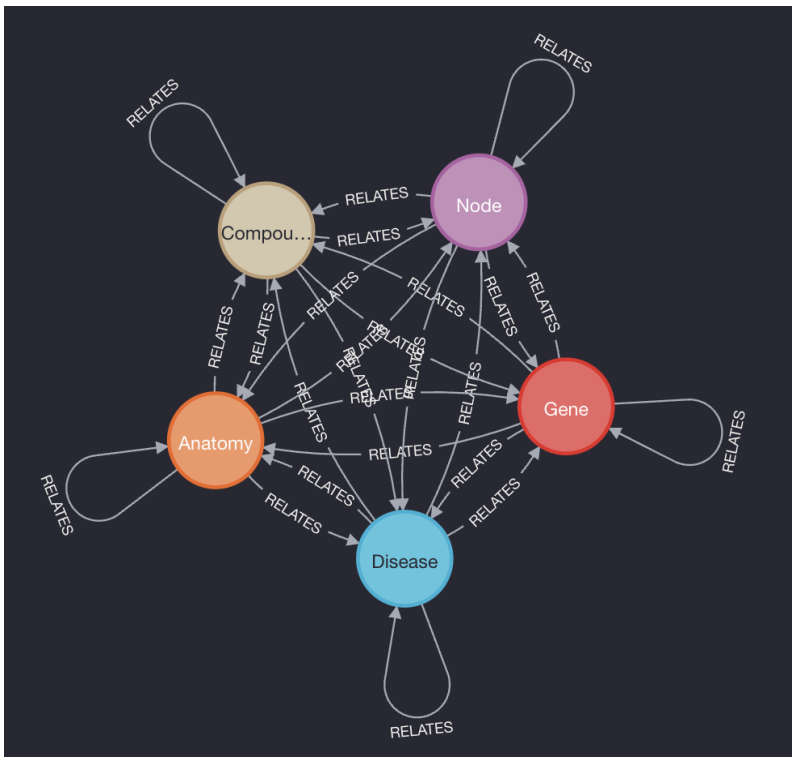
It goes through the edge collection and finds the edges with “CuG" or "CdG", and then, it goes through the edges collection again and matches the target of “CuG” with “AdG” and “CdG” with “AuG”. Then, it goes through a $match command to filter out the AdG and AuG edges that did not match with a CuG/CdG edge. After the filtering, it fetches all the DlA edges and compares the target of each DlA edge with the AuG and AdG edges filtered before, and if it matches, then it means that there is a connection between the source of the DlA edge and the source of the CuG/CdG edge. If there is a connection, then it creates a new edge. It checks in the database if the new edges already exist in the database, and adds it to the CtD_edges list if it does not exist. At the end, it prints out the missing ones.

```
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2994'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:3277'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2394'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:13223'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:11612'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:119'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1725'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1612'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1964'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:4362'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:363'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:635'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:12365'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2531'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:4989'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:0050741'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:784'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:3393'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1459'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:10608'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:5419'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2355'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2043'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:11239'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:363'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1725'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2998'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1612'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:4362'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2394'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:119'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:11612'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:2994'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1459'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:1964'}
{'source': 'Compound::DB00374', 'metaedge': 'CtD', 'target': 'Disease::DOID:13223'}

----- RESULTS -----
Found 3159822 potential new treatment compounds:
Total matching CtD edges: 3228795
Found 755 CtD edges in the database.
There are 3159822 missing edges.
```

Neo4j Implementation (Graph Database):

1. Design Diagram



For Neo4j implementation we started by clearing the database to remove any existing data. Nodes were then added from the TSV file, each assigned a general 'Node' label and a more specific label like 'Disease' or 'Compound'. An index is created on the “id” property for faster searches. Edges were added in batches of 25000 to prevent performance and memory issues. Relationships RELATES, with a “type” to indicate the connection (ex: CtD for Compound-treats-Disease).

2. Queries

Query 1:

```
query = """
    MATCH (d:Disease {id: $disease_id})
    OPTIONAL MATCH (drug:Compound)-[r1:RELATES {type: 'CtD'}]->(d)
    OPTIONAL MATCH (gene:Gene)-[r2:RELATES {type: 'DaG'}]->(d)
    OPTIONAL MATCH (d)-[r3:RELATES {type: 'DlA'}]->(anatomy:Anatomy)
    RETURN d.name AS Disease,
           collect(DISTINCT drug.name) AS Drugs,
           collect(DISTINCT gene.name) AS Genes,
           collect(DISTINCT anatomy.name) AS Locations
"""
```

The query first matches the disease node with the given disease ID. Then, it performs three optional matches to find the nodes for: compounds that treat the disease, genes associated with the disease, and anatomical locations where the disease occurs. We use OPTIONAL MATCH to make sure that even if some of these relationships are missing, the query still returns results without errors. Finally, the query collects the names of the entities while eliminating duplicates by using DISTINCT.

Query 1 Output:

```
Enter disease ID (ex: Disease::DOID:263): Disease::DOID:263

----- RESULTS -----
Disease: kidney cancer
Drugs: Sorafenib, Gemcitabine, Erlotinib, Vincristine, Vinblastine, Dactinomycin, Doxorubicin, Capecitabine, Paclitaxel, Sunitinib, Everolimus, Temsirolimus, Pazopanib
Genes: None
Locations: ureter, gonad, renal system, posterior vena cava, urine, renal vein, hepatic vein, renal artery, renal pelvis, cortex of kidney, renal papilla, nephron tubule, collecting duct of renal tubule, nephron, iris, cardiac atrium, kidney, juxtaglomerular apparatus
```

Query 2:

```
query = """
    MATCH (compound:Compound)-[:RELATES {type: 'CuG'}]->(gene:Gene)
    MATCH (disease:Disease)-[:RELATES {type: 'DlA'}]->(location:Anatomy)-[:RELATES {type: 'AdG'}]->(gene)
    WHERE NOT (compound)-[:RELATES {type: 'CtD'}]->(disease)
    RETURN compound.name AS Compound
    UNION
    MATCH (compound:Compound)-[:RELATES {type: 'CdG'}]->(gene:Gene)
    MATCH (disease:Disease)-[:RELATES {type: 'DlA'}]->(location:Anatomy)-[:RELATES {type: 'AuG'}]->(gene)
    WHERE NOT (compound)-[:RELATES {type: 'CtD'}]->(disease)
    RETURN compound.name AS Compound
    ORDER BY Compound
"""
```

For Query 2, the first MATCH statement finds compounds that up-regulate certain genes meaning they increase the gene's activity. Then, the second MATCH filters diseases whose anatomical locations down-regulate the same genes. This opposition of drug increasing activity while the disease's location decreases it, can suggest that the compound might counteract the disease. The WHERE NOT ensures that these drug-disease relationships don't already exist in the database (where the compound is already known to treat the disease). Next, we reverse the logic, we search for compounds that down-regulate a gene and link them with diseases where the anatomical location up-regulates that gene. The UNION merges results from both scenarios. Finally, we ORDER BY Compound to return the drugs.

Query 2 Output (Top 25 drugs):

```
----- RESULTS -----
Found 784 potential new treatment compounds:
1. Cyclosporine
2. Biotin
3. L-Glutamine
4. Adenosine monophosphate
5. Alpha-Linolenic Acid
6. Calcitriol
7. Riboflavin
8. Calcidiol
9. L-Tryptophan
10. Dihomo-gamma-linolenic acid
11. Icosapent
12. Vitamin A
13. Vitamin E
14. Pyridoxine
15. Menadione
16. Pravastatin
17. Fluvoxamine
18. Valsartan
19. Flunisolide
20. Nicotine
21. Esmolol
22. Bortezomib
23. Carbidopa
24. Phentermine
25. Tramadol
```

Potential Improvements for both MongoDB and Neo4j:

Possible improvements are a GUI which would be more user friendly than our command-line interface and implementing specific relationship types instead of using our generic RELATES relationship with a type property. By creating direct relationship types like TREATS, CAUSES it would eliminate the need for extra type filtering which we believe would improve the performance for queries like Query 2.