Shadman Rakib & Khandakar Wahiduzaaman
Big Data Technologies Project 1

Design Diagram

The design involves integrating Neo4j and MongoDB to manage and query complex large data sets.

Here is a breakdown of the components:

- We used MongoDB to store and query disease data. A specific disease document is fetched from MongoDB, and relations are determined by performing lookup operations on edges between nodes.
- We used TSV files to populate MongoDB and Neo4j. Nodes and edges from the CSVs are batch-loaded into Neo4j.

The pipeline consists of:

1. Loading data into panda dataframes
2. Load data into MongoDB and Neo4j by batching using insert_many in MongoDB and unwind in Neo4j.
3. Running queries across Neo4j and MongoDB for disease information.

MongoDB Queries

The query essentially finds the disease node, finds all edges that connect to it, finds all other nodes that use those edges, and then it groups them together using the relation type.

The following is the query in python with comments to help break down what the query does.

```
pipeline = [

    # get disease with matching id

    {"$match": {"id": disease_id, "kind": "Disease"}},

    # find relevant edges
```

```
{"$lookup": {
   "from": "edges",
   "let": {"node_id": "$id"},
   "pipeline": [
      {"$match": {
         "$expr": {
            "$or": [
               {"$eq": ["$source", "$$node_id"]},
               {"$eq": ["$target", "$$node_id"]}
            ]
         }
      }},
      # find nodes from relevant edges above
      {"$lookup": {
         "from": "nodes",
         "let": {"related_id": {"$cond": [{"eq": ["$source", "$$node_id"]}, "$target",
"$source"]}},
         "pipeline": [
            {"$match": {
               "$expr": {"$eq": ["$id", "$$related_id"]}
            }}
         ],
         "as": "related_node"
      }},
```

 # turn array of results into documents so we can project again

{"$unwind": "$related_node"},

 # add necessary info about the node and the relation

{"$project": {

"relation_type": "$metaedge",

"related_node": {

"id": "$related_node.id",

"kind": "$related_node.kind",

"name": "$related_node.name"

}

}}

],

"as": "relations"

}},

 # turn array of result into documents so that we can group

{"$unwind": "$relations"},

 # group the relations based on relation type

{"$group": {

"_id": {

"id": "$id",

"kind": "$kind",

"name": "$name"

},

"compounds": {

```
    "$addToSet": {
      "$cond": [
        {"$in": ["$relations.relation_type", ["CtD", "CpD"]]},
        "$relations.related_node.name",
        "$$REMOVE"
      ]
    }
  },
  "genes": {
    "$addToSet": {
      "$cond": [
        {"$eq": ["$relations.relation_type", "DaG"]},
        "$relations.related_node.name",
        "$$REMOVE"
      ]
    }
  },
  "anatomy": {
    "$addToSet": {
      "$cond": [
        {"$eq": ["$relations.relation_type", "DlA"]},
        "$relations.related_node.name",
        "$$REMOVE"
```

```
            ]
          }
        }
    }},
    {"$project": {
        "_id": 0,
        "disease_name": "$_id.name",
        "compound_names": "$compounds",
        "gene_names": "$genes",
        "anatomy_locations": "$anatomy"
    }}
]


result = list(db.nodes.aggregate(pipeline))
```

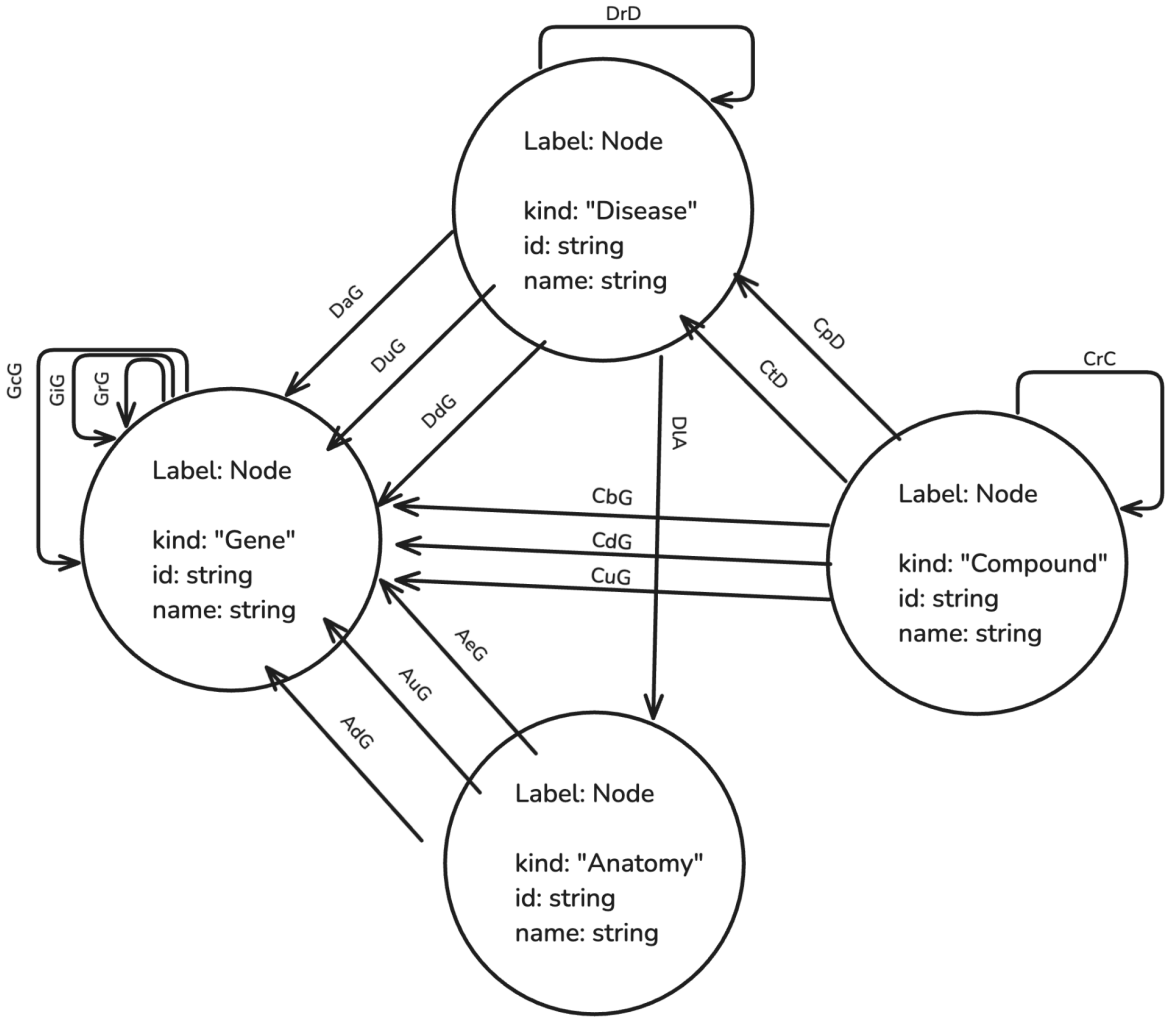
MongoDB Potential Improvements

We can improve performance by narrowing down the edges early on so that we process less data early on in the pipeline. We can index frequently queried fields in MongoDB and Neo4j to optimize search times. We can implement batching and parallelism when performing data loading or querying to boost the process for larger datasets.

Neo4J Design Diagram

Initially, the different nodes had different labels based on the kind. This made batching using UNWIND complicated to make performant, and the creation of the database edges took a very long time. To remedy this, we used a single node label called Node and distinguished

between the types using the kind field and an index on that kind field. There was also a uniqueness constraint on id to improve performance.



Neo4J Queries

There are comments to explain the queries. They use the param $diseaseId.

Query 1 - find drugs, genes, and anatomies associated with a disease id:

```
MATCH (d:Node {kind: "Disease", id: $diseaseId})
// Match compounds that treat or palliate the disease
OPTIONAL MATCH (d)<-[r:CtD|CpD]-(c:Node {kind: "Compound"})
// Match genes associated with the disease
OPTIONAL MATCH (d)-[:DaG]->(g:Node {kind: "Gene"})
// Match anatomical locations related to the disease
OPTIONAL MATCH (d)-[:DlA]->(a:Node {kind: "Anatomy"})
RETURN d.name AS disease_name,
  collect(distinct c.name) AS compound_names,
```

```
  collect(distinct g.name) AS gene_names,
  collect(distinct a.name) AS anatomy_locations
```

Query 2 - find new potential drugs:

```
// get all compounds that can have potential to do opposite of some anatomy on a gene
MATCH (c:Node {kind: "Compound"})-[:CuG|CdG]->(g:Node {kind:
"Gene"})<-[:AdG|AuG]-(a:Node {kind: "Anatomy"})
// narrow to down to just the opposite
WHERE (c)-[:CuG]->(g)<-[:AdG]-(a)
  OR (c)-[:CdG]->(g)<-[:AuG]-(a)
MATCH (d {kind: "Disease", id: $diseaseId})-[:DlA]->(a)
// make sure the compound is a new one
WHERE NOT EXISTS ((c)-[:CtD]->(d))
RETURN DISTINCT c.name as drug_name, c.id as drug_id
```

Neo4J Potential Improvements

To make the graph more descriptive, we can consider bringing back kind labels instead of a generic Node label. However, to do this, we would have to do more research on how to more efficiently batch creation of nodes and edges.

The queries can also be improved by experimenting with what we match first compared to later. Since these different attributes have different quantities, ordering of the matching can have a great impact.

We can potentially speed up queries further by looking into more indexing techniques to help improve the performance. We can also use the EXPLAIN cypher feature to debug and understand the execution of the query to identify bottlenecks.