Neo4j Graph Database Management System

Kushal Gevaria (kgg5247) Akshay Pudage (ap7558)

ABSTRACT

With the age of Big Data, the amount of data that the applications need to deal with has increased significantly over the years. The traditional relational database systems may not be quite efficient when dealing with big data as they perform better only when there is a fixed structure of data. This cannot always be guaranteed and thus a new trend of database systems were developed to handle unstructured data. Such systems are popularly known as NoSQL database systems. Neo4j is one such NoSQL database management system which stores data in the form of graphs. It is easier and efficient to model records and their relationships with graphs. In this paper, we dive deep into the internals of Neo4j to learn how efficiently it uses graph structure to model data.

1. OVERVIEW

Traditional databases like relational database systems are a popular choice for many application domains. They are particularly good for storing structured, form-based data into tables. The query performance is usually good when the data is structured, but it quickly starts to deteriorate if that is not the case. In real world, one cannot always guarantee that the data would be in a structured form with well defined relationships. Relational databases often struggle when new ad-hoc relations pop up while using an application in real world. They are poor at handling relations. Relations in a relational database exist only at the time of modeling the database in the form of table joins. They do not take into account importance or weights of relationships between two or more tables. In addition, these table joins are computationally expensive and this cost multiplies with the size of the data very quickly. It also needs to take into account various null columns when joining tables. In order for the joins to work, foreign key needs to be added to the schema of tables, thus adding more complexity and overhead. For a very simple query that retrieves just a few rows, joining of tables containing large data is usually needed. Reciprocal queries add even more computational cost.

The other alternative to store data is to use NoSQL database systems which uses either key-value, document, or column oriented data stores for storing disconnected values, documents or columns. The problem with such data stores is the lack of relations between data. It is possible to add relationships by adding identifier of a record into other record to connect the two, but this makes the whole process computationally expensive at the application level. These identifiers can be considered as foreign keys. Adding such identifiers complicates the process of adding or deleting of new records. If a record having an identifier in other records is deleted, one must make sure to delete the identifier in all the other records thus adding a considerable amount of overhead. If the record identifier is not deleted, it may lead to dangling references in other records thus affecting the performance and data quality. One other problem in such databases is that representing two-way relations is difficult. To represent such relationships, each record must have a identifier to other record i.e. a backward link. Maintaining such backward links are expensive. While traversing deeper relations, such costs can add up very fast leading to performance issues which can be noticed by the users.

Graph databases are a type of a NoSQL database which stores data in graph semantics using nodes and relationships between the nodes. Each relationship and nodes may have properties assigned to it which are stored in a key-value map. The main advantage of graph database systems is how well the graph databases embrace relationship between nodes. The relationships between different entities in the data are explicitly stored in the graph databases in the form of direct connections between two related nodes. The main reasons why the graph databases are preferred over traditional relational systems and other NoSQL databases is as follows:

- Relations in the traditional relation database systems and other NoSQL databases are implicitly stored in the form of foreign keys spread across multiple tables (in case of RDBMS), and across multiple records (in case of NoSQL databases). In order to access such relationships between nodes in the database, expensive joins are usually needed which usually add costs of sorting and matching records. This cost is computationally expensive and the query performance quickly degrades with increase in data size. In graph databases, relationships among nodes can be quickly retrieved as each relationship is explicitly represented in the graph structure. Any complex query involving multiple relationships usually takes only a constant amount of time, thereby giving a boost to the query performance. This performance is unaffected even if more data and relationships are added to the database over time.

- Relational database systems are not very good with relationships. Relational database systems need highly organized and structured data with distinct relationships identified between the entities while designing of the database system. It often struggles when new relations between data are encountered and the whole

schema design needs to be changed in order to accommodate such ad-hoc relations. On the other hand, graph databases embrace relationships. Each node stores its relationship information in a list. Whenever a new relationship needs to be modeled, one can simply append a new relationship to this list of relationships without having to change the database design in a drastic way.

In Neo4j, graph structures have four distinct entities which are used for storing data namely: nodes, relationships, properties and labels. Nodes are used for storing entities in the data like, for example, in case of movie database, the nodes can be actors and directors. Labels are used for categorizing a group of nodes like, for example, all the actor nodes can have 'Actors' label. Relationships are used for joining related nodes, like for example, an actor node can be connected to a movie node in which the actor acted in. Each node and relationship may have properties assigned to it, which is nothing but a key-value pair for storing information like, for example, actor node can have a property of name for storing actor's name in the node. All these entities together represent the overall graph structure.

In the following sections, we dive deeper into the internals of Neo4j by taking a look at how the data is stored internally, how it is indexed. We also look at how the queries are optimized and processed. We then explore transaction management and security aspects of Neo4j. Finally, we look at an application which uses Neo4j for storing data. This application creates and stores a movie database. It has following labels inside it: actors, directors, movies and genres. This database was chosen as it is easier to understand the working of Neo4j database without complicating the graph structure too much. Using Neo4j for this type of data also enables us to search for degrees of separation between two nodes, like for example, finding a mutual actor say A who has worked with both actors B and C. Here, the degree of separation for actor B and C is one via actor A. Doing the same operations in a relational database would involve self joins of the tables, which can quickly add up computational costs. Thus, finding degrees of separation between entities can be efficiently done in graph databases.

2. DATA STORAGE AND INDEXING

2.1 Indexing

Graph databases consists of several nodes and relationships between them. Nodes and its relationships also have labels and properties assigned to it to distinguish between different nodes and relationships. There are multiple ways of encoding the graph structure and preserving relationship among nodes. If a database system uses index-free adjacency, it is said to have native processing capability.

In index-free adjacency, each node is responsible for storing the actual reference of the adjacent nodes. This reference in a way acts as an index to other nodes. One other way of encoding graph structure is to use global indexes for linking nodes. The database systems that use such indexes are called as non-native graph database systems. Native graph database engines have an advantage over non-native graph database engines. Native graph database engines have lower computation costs and the query time is independent of the total size of the graph and it only amounts to the size of the graph searched. In native graph database systems, relationships can be traversed in a *O(1)* time irrespective of the direction of the relationship, while in non-native graph database systems, for traversing a relationship, the index for a node must be first searched in an index file which requires *O(log n)* computations and if the direction of the relationship is reversed the cost can go as high as *O(m log n)*, where m is the number of nodes that are connected to a node and n is the total number of nodes in the graph. If the number of nodes are high, even $log\ n$ can lead to a very high value. Thus, index-free adjacency is the key for efficient graph traversals.

While index-free adjacency speeds up the query performance while finding the relationships between nodes, it is also important to find the correct nodes in a vast graph efficiently to begin the graph traversal. Traversing an entire graph might not be feasible to find a node, say A, in a graph of about thousands of nodes and then traversing relationships from node A to other nodes. Neo4j uses indexes for quickly searching a node within the graph for quick traversal. It uses hash indexes as well as inverted indexes [9]. Using hash indexes, node look-ups can be made in a *O1* time. Neo4j also has a support indexing relationships in a similar way. Neo4j also has a support to store indexes in a generation-aware B+ tree [7]. Generation is nothing but a checkpoint of changes made to the indexes while the database is in use. Whenever a node in the tree is modified, it is copied from a stable generation to a new unstable generation. Further changes to the node are made directly on to the unstable version. This node also has a pointers to stable and unstable versions of other nodes in the tree. Such generation aware trees allows for reliable recovery of indexes in case of failure. Thus, unlike relational databases where indexes are used for locating records and also for joins, indexes in graph databases are mainly used for locating the correct node or relationship in a vast graph. The rest of the graph can be traversed using the indexes stored within the nodes and relationships themselves.

Indexes in Neo4j can be defined as follows. This creates indexes for two nodes and a relationship between them.

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
Index<Node> movies = index.forNodes( "movies" );
RelationshipIndex roles = index.forRelationships(
    "roles" );
```

Listing 1: Creating Indexes, from Neo4j Documentation[5]

To each index, we can then add any number of key-value pairs as follows.

```
Node node = graphDb.createNode();
node.setProperty( "name", "Leonardo DiCaprio" );
actors.add( node, "name", node.getProperty( "name" ) );
```

Listing 2: Adding key-value pairs to indexes, from Neo4j Documentation[5]

We can then search for the above indexed key-value pair as follows.This will retrieve the single node that matches the search string.

```
IndexHits<Node> hits = actors.get( "name", "Leonardo
    DiCaprio" );
Node node = hits.getSingle();
```

Listing 3: Getting indexed key-value pairs, from Neo4j Documentation[5]

To delete a index on a node, following code can be used.

```
actors.remove( node, "name", "Leonardo DiCaprio" );
```

Listing 4: Deleting Indexes, from Neo4j Documentation[5]

The same process can be used for creating indexes for relationships. Relationships can be indexed based on its start and end nodes in addition to its properties [5].

2.2 Data Storage

In this section, we take a look at how the graph data is stored physically on disk. When it comes to storage, Neo4j stores data in different store files. There are stores for storing nodes, relationships and properties for a graph.

All the nodes that are created in the graph are stored in node store. The physical file that is responsible for storing this node store is `neostore.nodestore.db` [8]. Each node is stored within the store in a record. This record is of fixed size, where each record is of length nine bytes [8]. This design decision of having the records of fixed size leads to faster traversal of records. It basically eliminates the need of deploying a search algorithm for finding nodes stored within an array of records. The idea is similar to an array in a programming language, in which, any element within an array can be accessed with *0(1)* time. Similarly, the address of any node within an array of records can easily be accessed if we know its ID by simply multiplying the id with the size of each record (nine bytes). This operation takes *O(1)* time leading to high performance traversal.

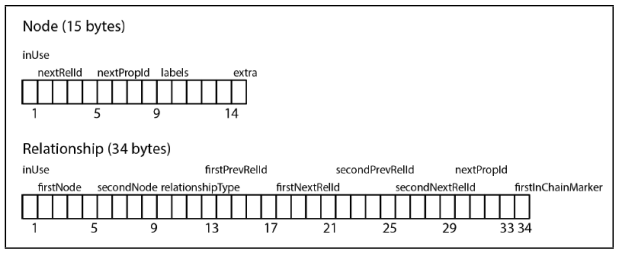


Figure 1: Node and relationship store file record structure, from *Graph Databases* [8].

The node record structure is shown in figure 1. The first byte is reserved for an in-use flag which simply indicates if that particular record is currently holding a valid node of the graph. The next four bytes hold the ID of the first relationship of this node. The next four bytes hold the ID for the first property for this node. The next five bytes are reserved for storing the address the address to point to the label store for this node. The final byte is an extra byte, reserved for flags. One such flag keeps track if the node is densely connected in the graph. Some of the remaining space is reserved for future use [8].

The relationship record structure is shown in figure 1. The relationship records are stored in relationship store and the can be found in `neostore.relationshipstore.db` physical file [8]. The first nine bytes are used for storing IDs of the nodes at the either end of the relationship. There is also a pointer to the relationship type, this information is usually stored in a separate store called as relationship store. Following this are pointers that point to the next and previous relationship records for each of the start and end nodes. The last byte is used for storing a flag which is used for determining if the the current relationship record is the first in relationship chain.

Node and relationship store are mainly responsible for storing the graph structure. There is one other store for storing properties called as properties store. Properties store user data in the form of key-value pairs just like in a map data structure. Both the nodes and relationships can have properties assigned to them. Records in the properties store are physically stored in `neostore.propertystore.db` file [8]. Records in the property store are of fixed size for quick traversals. Each records can hold four properties and it has a pointer to the next property in the property chain. Thus, the property chain is a singly linked list. Neo4j allows to have properties of primitive JVM types [8]. Properties can be stored using a dynamic store record or they can be stored inline. If storing inline, a record can hold a maximum of four properties. Dynamic store can hold more than four properties by creating a separate linked list of its own. Storing properties in such dynamic arrays may involve considerable amount of I/O time as compared to inline properties.

Pointers in records act as singly linked list, in case of node record structure, and as double linked list, in case of relationship record structure. These pointers combined with the design decision of having fixed size records have enabled faster and efficient graph traversals in Neo4j.

Along with good and efficient storage layout, it is also important to manage the flow of data from to and fro from the main memory and secondary memory. Most of the new machines have larger main memories and faster secondary drives, but it can still result in poor I/O times when dealing with large amount of data. To mitigate this, Neo4j uses an LRU-K page cache[8]. It divides the main memory into k pages. When the memory is full, it chooses the page that is least frequently used for replacement, thus ensuring optimal use of resources.

3. QUERY PROCESSING AND OPTIMIZATION

Given below demonstrates the query processing and optimization techniques performed underneath the Neo4j database application.

3.1 Query Processing

Neo4j built their own graph query language Cypher, which was later made open source with the hopes of making it SQL for the graph databases [3]. Cypher was made the most declarative language, meaning all the CRUD operations like create, read, update and delete operations can be done on the database using simple Cypher queries where only what needs to be done should be mentioned in the query and not how. This extensive support is provided by Cypher using ASCII-art representation of nodes and patterns.

Consider a simple example where we want to find all the actors who acted in movies directed by 'Yash Raj'. This can be simply achieved by using the query shown in listing 5.

```
MATCH
(a:Actor)-[:actedIn]->(m:Movie)<-[:directed]-(d:Director)
WHERE d.name = "YashRaj"
RETURN a
```

Listing 5: Cypher query to find all the actors who acted in movies directed by 'Yash Raj'

The best part of Cypher query language is we can represent all the complex relations and structures in a meaningful and concise way [3]. In the above query, the query is processed as follows. First, the nodes with label Actor are selected then all the edges/relationships with label "actedIn" are selected. Furthermore, the other nodes with label Movie are chosen. With this results, we select nodes with incoming edges directed and other node matching the label Director. Additionally, a filter is applied to the name attribute of the director name.

If this query were to perform on a traditional relational database, we would have to join at least 3-4 tables in order to get the desired information. Remember joins are expensive operations. If the queries are finding neighbors of a particular node, then writing such queries in SQL becomes cumbersome, really complex and lengthy. On the other hand, Cypher provides a special operator to find the neighborhood nodes the way we want [3].

3.2 Query Optimization

Regular search over the entire data can be costly in terms of time and resources. Cypher provides an option of indexing. We can create an index on a single attribute or composite indexes are also available. Composite indexes allow index creation on more than one attribute. By default, the unique ID of each node is indexed and any operation that uses the ID is already optimized. Besides, we can also add hints in the query to let the query engine know that we would like to use an index while executing the given query [2]. These hints are optional though.

The Neo4j tries to execute the queries as fast as possible and in order to do that sometimes a query tuning is needed [2]. Now, this can also be done manually given the domain knowledge. It is common that filtering should happen as early as possible in order to work with fewer data. Each cypher query gets optimized and transformed into an execution plan. The execution plan that uses minimal resources is chosen from this set of plans. If the queries are parameterized as oppose to hardcoded literals the query engine also re-uses the same execution plan instead of building all plans and choosing the best. Furthermore, there are several operators that aid in choosing the best query plan.

Even though SQL provides indexing and query optimization using several types of join sometimes writing query becomes really complex and lengthy. A classic example is finding the neighborhood of a certain attribute value. Writing these queries are pain let alone the optimization on top of it. What makes Neo4j convenient as compared to SQL is its dynamic nature and expressiveness to represent information [2].

4. TRANSACTION MANAGEMENT AND SECURITY SUPPORT

Given below demonstrates the transaction management and security support provided by the Neo4j database application.

4.1 Transaction Management

Transactions are important in database management systems to ensure integrity and consistency of the data stored in the databases. Most of the NoSQL databases do not have support for transactions. There is an unvalidated assumption that transactional systems do not scale well [8]. This is partially true as the two-commit phase may cause unavailability at times during simultaneous access [1].

Transactions in Neo4j are similar to traditional database systems. Write locks are acquired on the nodes and relationships that are involved in the transaction to ensure consistency. If a transaction is completed successfully, the changes are flushed to the disk and the write locks are removed. In case of transaction failure, the changes are discarded and the write locks are removed. This is implemented internally with the help of an in-memory object whose state represents writes to database. This object is supported by lock manager, which applies write locks to nodes and relationships involved in the transaction. These locks are acquired whenever the nodes and relationships are accessed by the query either for read, write, update or delete operations. On successful completion of transaction, the write locks are removed, changes are committed to disk and then the transaction object is discarded. On transaction failure, the write locks on each node and relationship involved is first removed and then the transaction object is discarded. For committing changes to disk, Neo4j uses *Write Ahead Log* [8]. On successful transaction, the changes are appended at the end of the commit log. After the changes are logged in the log, the changes are flushed to the disk. This ensures fault tolerance in the database system. On failure of the system, the state of the database system can be recovered with the help of these logs.

4.2 Security Support

Security is of utmost importance in any database application to prevent unauthorized access to underlying sensitive data. In Neo4j, security is controlled by authentication and authorization and is enabled by default. Authorization is managed by role-based access control (RBAC) [6]. Neo4j supports third party authentication providers like LDAP and Kerberos [6]. Every user in the database is required to have a password. Each user account also has a role associated with it, which controls the tasks the user is authorized to perform while using the database system. There are various predefined roles in Neo4j having different authorization permissions. Some of these are reader, editor, publisher, architect and admin [6]. In addition to these predefined roles, there is a option to create custom role with granular permissions. Neo4j also has a sandboxing feature to restrict access to insecure APIs or APIs which are still under development [6]. All such security features ensure that the data in the database system is safe and secured.

5. NOSQL DATABASE APPLICATION

For our Neo4j NoSQL database application, we are uti-

lizing the Neo4j community version 3.2.5 application that operates conveniently across all the platforms. Neo4j provides reliable support in Java with all the built-in application packages that can be used to communicate with the database [8]. We are adopting functionality from the package "org.neo4j.graphdb". To establish the connection with the database, we simply use "GraphDatabaseService" class which is present in the same package mentioned earlier. [1]. Given below is the example of creating a neo4j database. Also, it shows useful packages for creating nodes and relationships between the nodes:

```
// Main.java
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Relationship;
import org.neo4j.graphdb.RelationshipType;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.neo4j.unsafe.batchinsert.BatchInserter;
import org.neo4j.unsafe.batchinsert.BatchInserters;

public class Main {
    private static GraphDatabaseService dbsiIMDB = new
         GraphDatabaseFactory().newEmbeddedDatabase(new
         File("fileDirectory"));
    public GraphDatabaseService getDbsiIMDB() {
      return dbsiIMDB;
   }
    // Additional supporting functionality
}
```

Listing 6: Java Packages for Neo4j

Let us explain the most important functionality we used from the java packages from the Neo4j database in code 6. The "GraphDatabaseService" package provides an instance of Neo4j graphDB to perform CRUD operations on the nodes and relationships in java. These CRUD operation methods in java performs Cypher queries behind the scenes [4]. Cypher queries are the query writing techniques in Neo4j database, similar to the SQL queries in mySQL database. The Node package is used to create nodes in the graph database. Given below is the snippet for creating a node for a director.

```
Node dNode = getDbsiIMDB().createNode();
dNode.setProperty("name", director.getDirector());
```

Listing 7: Create Node

Also, we need to set certain properties for the created node. The properties as compared to the SQL database are nothing but the column attributes for each person. Node package has the method "setProperty()" to assign these properties. In code 7, each director has a property called name, i.e. the name of the person. After creating the node we need to label each node. The "addLabel()" method provides this functionality. Given below is the snippet to add label to the created nodes.

```
Label directorLabel = Label.label("Director"); // label
     of director
dNode.addLabel(directorLabel);
```

Listing 8: Create Label

This will recognize each identity with label "Director" as the director of some set of movies. After creating set of nodes, we need relationships between them. First, we create those relationships. In Neo4j, relationship is the directed edge between two nodes. This can be achieved, by the java package "Relationship" and "RelationshipType". Code 9 demonstrates the creation of relationship between two nodes and provide a type to the same. This will create a relationship between a director node and a movie node with the relationship type as "directed". This will create a directed edge pointing from the director node to the movie node in the Neo4j graph database.

```
RelationshipType directorMovieConnection =
     RelationshipType.withName("directed");
dNode.createRelationshipTo(mNode,
     directorMovieConnection);
```

Listing 9: Create Relationship

This completes a basic creation of the Neo4j nodes and relationships using java. We would also like to perform other operations like searching for certain nodes and relationships as per the requirement and also update of delete the content. To find the nodes and relationships for those nodes, the code 10 demonstrates certain java methods for the same.

```
Node dNode = getDbsiIMDB().findNode(directorLabel,
     "name", directorName);
for(Relationship r : dNode.getRelationships()) {
    // get nodes on other end of this relationship
    Node tempMovie = r.getEndNode();
}
```

Listing 10: Find Nodes and Relationships

This snippet above, will find a director with mentioned director name and all it's existing relationships. We know that directors currently have relationship with different movie nodes and the name of those relationships is "directed". The "getEndNode()" will give nodes on the other side of the relationship, which in this case are movie nodes. After we get nodes and it's relationships from the database, we can perform update or delete operations on the same. To update the information for a given node, code 11 is an example.

```
Node dNode = getDbsiIMDB().findNode(directorLabel,
     "name", oldName);
dNode.setProperty("name", newName);
```

Listing 11: Update Node Information

Here, we are updating the name of the director node by finding it using the old name and then set the property by using the "setProperty" method on the node to set the "name" property of the director node. We can also delete certain node or relationship between two nodes using the method "delete()". Code

```
Node dNode = getDbsiIMDB().findNode(directorLabel,
     "name", name);
for(Relationship r : dNode.getRelationships()) {
```

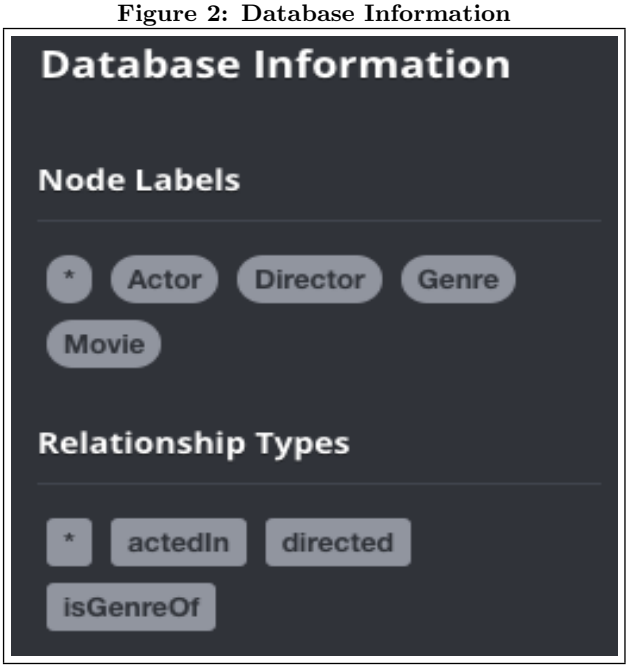
```
    r.delete();
}
dNode.delete();
```

Listing 12: Delete director node and it's relationships

Here, before deleting a node, we need to first delete all it's relationships, otherwise those relationships will just float around with one side of edge pointing to some existing node, while other side pointing to nothing. Hence, we first get all the relationships and delete them first, then we delete the node to perform the clean delete.

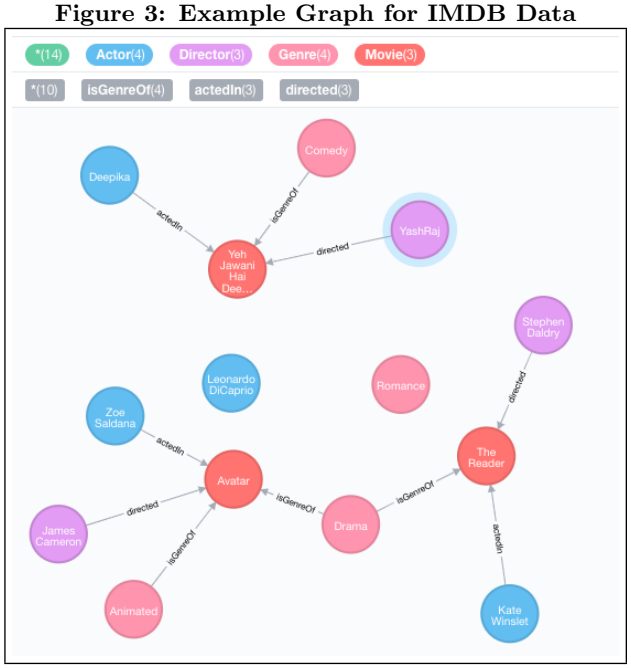
As we perform all the above-mentioned operations on the Neo4j database, we can keep the track of the updates through their interacting graphical user interface. In this GUI you can have a look at all the nodes with labels for the same as well as the relationship type between those nodes.

Figure 2: Database Information



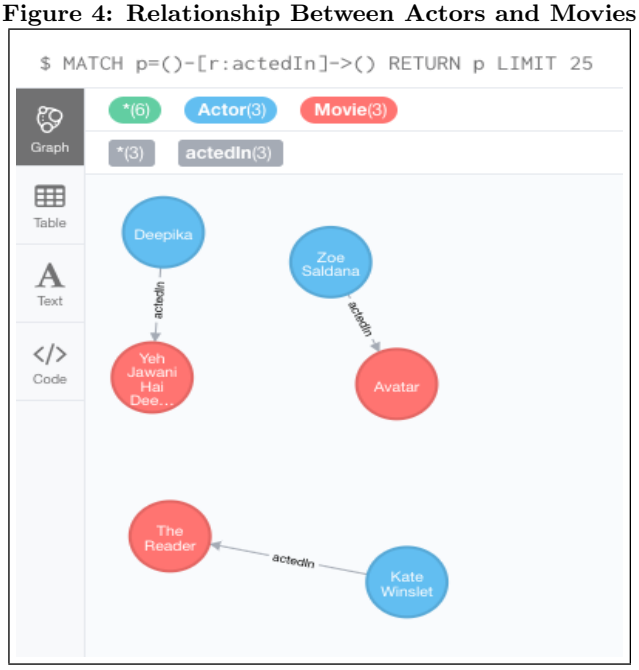
The figure 2 shows the inserted node labels and relationship types in the Neo4j database representing the IMDB information. A basic representation of graph database of the IMDB dataset is show in the figure 3 where we have some relationships between the nodes of actors, movies directors and genres.

Figure 3: Example Graph for IMDB Data



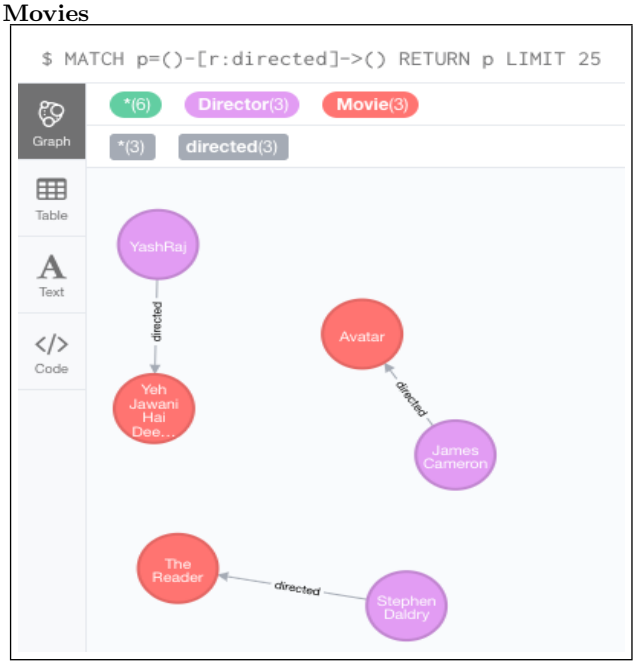
First, we created some actor nodes which are shown in the figure 4 that have relationships with certain movies. For example, actress "Kate Winslet" acted in movie "The Reader".

Figure 4: Relationship Between Actors and Movies



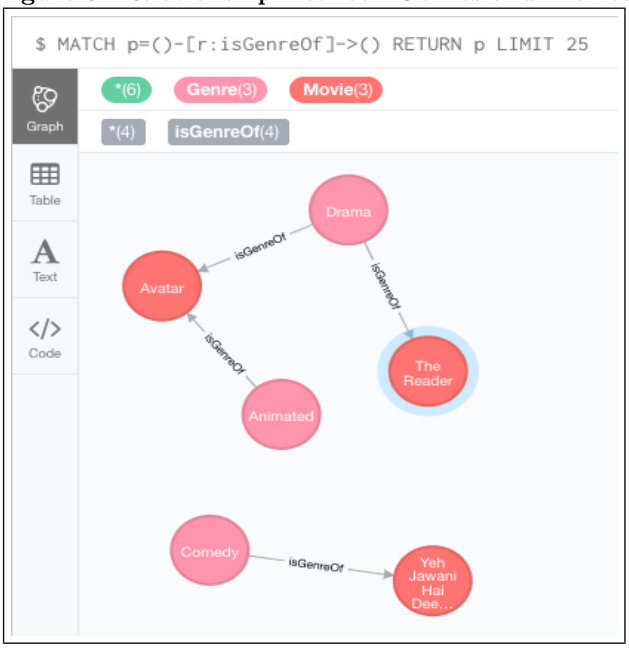
Second, we created some director nodes which are shown in the figure 5 that have relationships with certain movies. For example, director "Stephen Daldry" directed movie "The Reader".

Figure 5: Relationship Between Directors and Movies



Finally, we created some genre nodes which are shown in the figure 5 that have relationships with certain movies. For example, “Drama” is a genre for movies “Avatar” and “The Reader”.

Figure 6: Relationship Between Genres and Movies

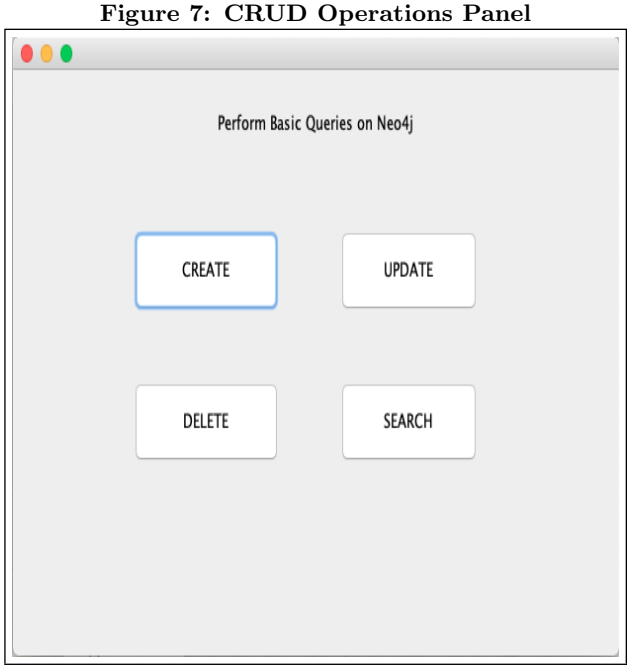


6. NEO4J GUI FOR BASIC CRUD OPERATIONS

The GUI of our application is developed using Java Swing API.The GUI provides user interface for performing basic CRUD operations on the database.

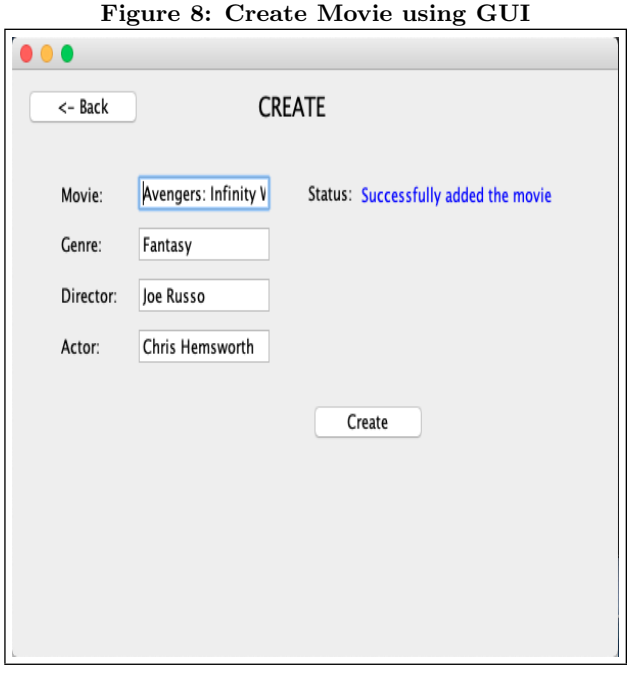
The dashboard of the GUI application shows buttons for create, update, delete and search for records.

Figure 7: CRUD Operations Panel



Clicking ’Create’ button in the dashboard shows the following pane, wherein the user can create new nodes for movie, actor, genre and director by giving string inputs. The underlying code will automatically define relationships between them.

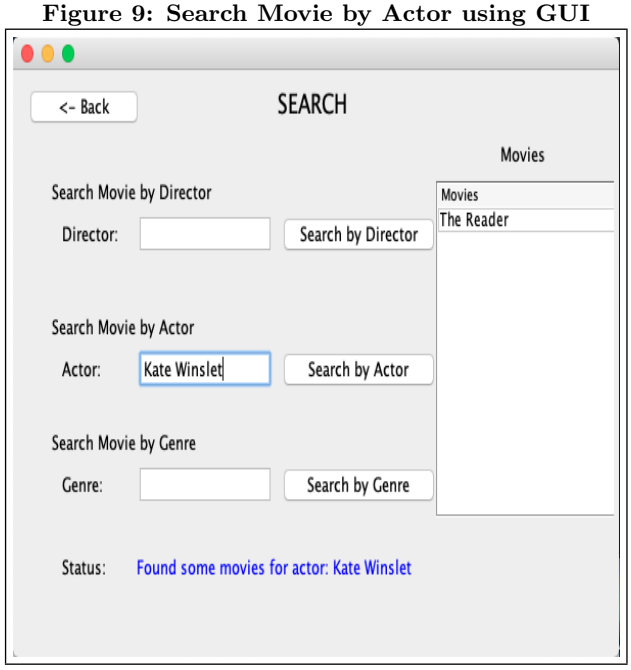
Figure 8: Create Movie using GUI



Clicking ’Search’ button in the dashboard shows the following pane, wherein the user can search for a movie by

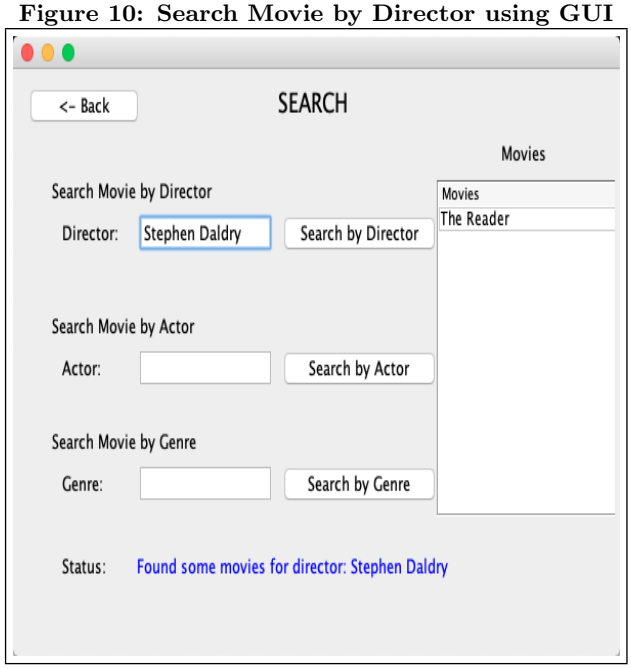
actor, by giving a string input. The results will be displayed in a list format in the same pane.

Figure 9: Search Movie by Actor using GUI



Clicking 'Search' button in the dashboard shows the following pane, wherein the user can search for a movie by director, by giving a string input. The results will be displayed in a list format in the same pane.

Figure 10: Search Movie by Director using GUI

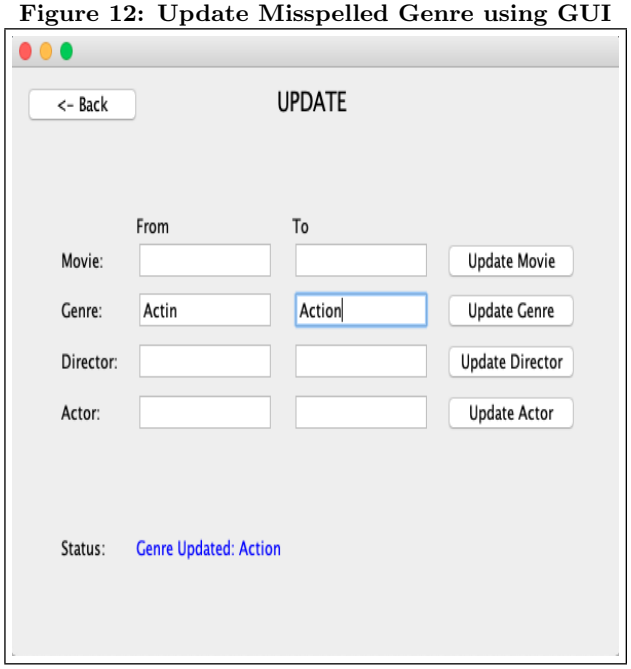


Clicking 'Search' button in the dashboard shows the following pane, wherein the user can search for a movie by genre, by giving a string input. The results will be displayed in a list format in the same pane.

Figure 11: Search Movie by Genre using GUI

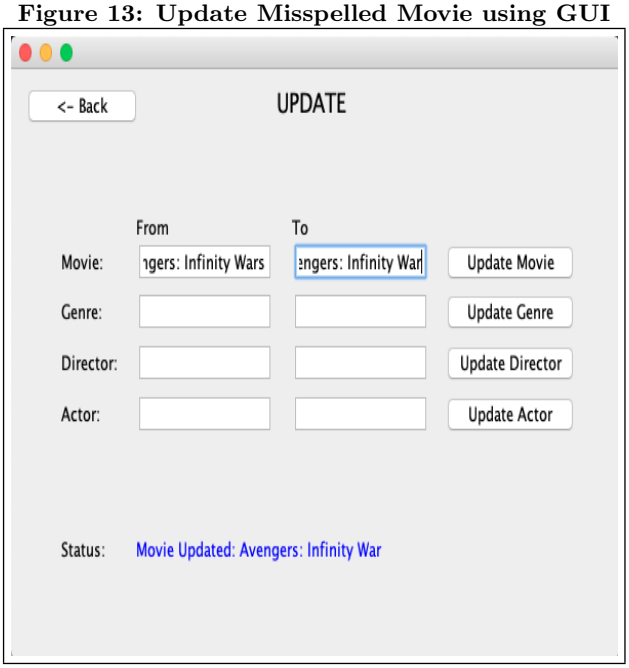
Clicking 'Update' button in the dashboard shows the following pane, wherein user can make changes to the properties of nodes like for example updating misspelled genre name.

Figure 12: Update Misspelled Genre using GUI



Clicking 'Update' button in the dashboard shows the following pane, wherein user can make changes to the properties of nodes like for example updating misspelled movie name.

Figure 13: Update Misspelled Movie using GUI



Clicking 'Delete' button in the dashboard shows the following pane, wherein user can delete tne nodes in the graph database application. It also deletes relationships of the node that is being deleted.

Figure 14: Delete Director using GUI

7. FINAL REMARKS

For the graphical user interface we are planning to create a Java Swing application to perform these four basic CRUD operations. We chose to using swing application since it is very convenient to integrate the swing GUI with the existing Java back-end application which also provides best support of the Neo4j graph database. We completed with supporting the back-end with simple user friendly graphical user interface performing CRUD operation on the NEO4J IMDB database. Further steps in our application process, is to support the back-end with complex queries and also simultaneously provide the feature to perform those queries using by extending our application with additional features.

8. WORKLOAD DISTRIBUTION

8.1 Teammate 1

Teammate 1 was responsible for writing the logic to perform the create and search operations using the java eclipse IDE and also perform the database connectivity. Other responsibilities were to create the graphical user interface for basic CRUD operation and write about the query processing and optimization, transaction management and NoSQL database application creation part of the report. This teammate was successfully able to follow everything mentioned above.

8.2 Teammate 2

Teammate 2 was responsible for writing the logic to perform the update and delete operations using the java eclipse IDE and also look up about the graphical user interface we are going to use in future. Other responsibilities were to write about the abstract, overview, data storage and indexing, transaction security support part of the report. This teammate was successfully able to follow everything mentioned above.

9. REFERENCES

[1] E. Eifrem. Graph database service, 2002.
[2] E. Eifrem. Neo4j cypher query language, 2002.
[3] E. Eifrem. Neo4j cypher query tuning, 2002.
[4] E. Eifrem. Neo4j cypher refcard 3.4, 2002.
[5] Neo4j. Chapter 6. manual indexing. Accessed: 2018-11-18.
[6] Neo4j. Chapter 7: Security. Accessed: 2018-10-26.
[7] Neo4j-GitHub. Gbptree code. Accessed: 2018-11-18.
[8] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. O'Reilly Media, Inc., 2013.
[9] V. Soloshchuk. Neo4j community edition: Worth using? Accessed: 2018-11-18.