Migrating data from document-oriented database to graph-oriented database

Lim Fung Jin, Nurulhuda Firdaus Mohd Azmi

Tunku Abdul Rahman University of Management and Technology, Kuala Lumpur, Malaysia, department of Software Engineering and Technology.

University Teknologi Malaysia, Malaysia, department of Advanced Informatics, Razak Faculty of Technology and Informatics.

Abstract In data migration between different types of NoSQL database, data may not be directly transferred to the targeted database in compare to migration of data between the same types of database. This is due to the heterogeneity of storage paradigm of the NoSQL databases. For example, migrating data from a document-oriented database such as MongoDB, which stores data in Json (Java Object Notation) format to Neo4j, a graph-oriented database stores data in node, the differences among these databases’ storage paradigm requires different representation of data model in the targeted graph-oriented database. This paper proposed a sequential approach to migrate data from MongoDB to Neo4j. The approach migrates MongoDB data to Neo4j and verifies the migrated data using a comparative method. The paper discusses on the migration algorithm and how complex field in MongoDB such as nested document is presented in Neo4j.

Keywords: data migration, NoSQL database, MongoDB, Neo4j, sequential comparison

1. Introduction

Graph-based database is one of the NoSQL type databases that stores data in nodes. Another type of NoSQL, which is the document-based database stores data in Json form. The transfer of data between the two types of databases may be a straight-forward, direct record to record transferring due to the heterogeneities of storage paradigm among the databases. This paper discusses the algorithm on data migration from MongoDB; a document-oriented database, to Neo4j; a graph-oriented database.

2. Document Oriented Databases and Graph Oriented Databases.

Document oriented NoSQL saves data using JSON or BSON. A key value set is used to identify documents. For document-oriented database, searches may be conducted using both the key and the value. The graph database stores data as nodes connected by edges. The edge illustrates the connection between nodes. There is a pointer in the nodes that directs to the subsequent nodes (Swaroop 2016). Table 1 shows some summary on storage architecture for the two types of database (Zafar 2016)

<table>
<thead>
<tr>
<th>Types of NoSQL</th>
<th>Data Model</th>
<th>Strength</th>
<th>Weakness</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document oriented</td>
<td>Groups of relationships between key values</td>
<td>Tolerate on incomplete data.</td>
<td>Do not have standard query syntax.</td>
<td>CouchDB, MongoDB</td>
</tr>
<tr>
<td>Graph oriented</td>
<td>Data Nodes – “Property Graph”</td>
<td>Apply graph algorithm – identify shortest path, connected ness.</td>
<td>Difficult in clustering, traverse whole graph.</td>
<td>InfoGrid, InfiniteGraph</td>
</tr>
</tbody>
</table>

3. Data Migrations

The process of transferring data from one data source to another, such as from one database to another or within the same database, is known as data migration. For NoSQL databases, there are differences (Dharmasiri and Goonetillake 2013) among these databases and the differences are:

- Each type of NoSQL database has its own data model, i.e. data model is different among types of NoSQL databases.
- Implementation is different even for the NoSQL databases which has the same data model.
- Difference query language
- Supporting different features of CAP
• Apply different model of consistency

In the migration of data between NOSQL databases may require the consideration of the heterogeneities of the databases’ data model. Inappropriate design of data model in the target database may not be able to support the data schema from the original data source and will affect the integrity of data. This paper discusses on the migration of data between different types of NoSQL with the objectives:

Obj1: Examine the sequential algorithm that migrates data between NoSQL databases.
Obj2: Examine data schema of the targeted database to support data model of the original data source.

4. Related Works

Data migration between diverse databases was studied and methods were developed. On-demand conversion between the old and new schema fields is possible with the lazy migration strategy. The period of time the application is offline will be reduced as a result of this (Klettke 2016) However, in terms of managing the application's code, this technique will create a load on the developers. Another solution, which uses slow migration but reduces developer load, isolates migration components into a database library, allowing apps to access data normally (Saur et al 2016) With the assistance of a decision tree, a technique called Build Schema Profile (BSP) was proposed for detecting schema differences between database instances. This is a schema-less document-based database approach (Gallinucci et al 2018) for the goal of transferring data between multiple columnar databases, a data migration system was suggested and implemented. Heigra4Cloud transforms data into an intermediate format before converting it to the destination databases (Scavuzzo et al 2014) the migration mechanism was improved by the authors by adding fault tolerance and support for several database types. Data migration issues, such as duplicated data and data loss, are detected by these fault tolerance characteristics (Scavuzzo et al 2016).

5. Experiment Configurations and execution

5.1 Experiment Setup

The experiment is performed on a single machine with databases run as local services. MongoDB 4.2.3 is selected to represent the document-based database and Neo4j 3.5.18 is representing graph-based database. The databases are run on different ports; MongoDB service is run at local port 27017 and Neo4j service is available at port 7474. The dataset from is a curated list in GitHub which contains fourteen collections of Json/Bson files (Ozler 2019) Within the fourteen (14) collections, the Companies collection will be used. The Companies collection consists of 1583 documents where each document consists of thirty-eight fields. The fields within the Companies collection are from simple data type field to complex data type field. Simple data type field contains text data, numbers, data or alphanumeric data. Whereas complex data type field may contain nested documents, nested array, array of document and et cetera. The schema of Companies collection is depicted in Table 2. In Table 2, the mixed data type represents the field contains different types of data in different documents.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id</td>
<td>ObjectId</td>
<td>deadpooload_year</td>
<td>Mixed</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>tag_list</td>
<td>String</td>
</tr>
<tr>
<td>permalink</td>
<td>String</td>
<td>alias_list</td>
<td>Mixed</td>
</tr>
<tr>
<td>crunchbase_url</td>
<td>String</td>
<td>email_adress</td>
<td>Mixed</td>
</tr>
<tr>
<td>homepage_url</td>
<td>String</td>
<td>phone_number</td>
<td>Mixed</td>
</tr>
<tr>
<td>blog_url</td>
<td>String</td>
<td>description</td>
<td>Mixed</td>
</tr>
<tr>
<td>blog_feed_url</td>
<td>String</td>
<td>created_at</td>
<td>Mixed</td>
</tr>
<tr>
<td>twitter_username</td>
<td>Mixed</td>
<td>updated_at</td>
<td>String</td>
</tr>
<tr>
<td>category_code</td>
<td>String</td>
<td>overview</td>
<td>String</td>
</tr>
<tr>
<td>number_of_employees</td>
<td>Mixed</td>
<td>image</td>
<td>Mixed</td>
</tr>
<tr>
<td>founded_year</td>
<td>Mixed</td>
<td>products</td>
<td>Array</td>
</tr>
<tr>
<td>founded_month</td>
<td>Mixed</td>
<td>relationships</td>
<td>Array</td>
</tr>
<tr>
<td>founded_day</td>
<td>Mixed</td>
<td>competitions</td>
<td>Array</td>
</tr>
<tr>
<td>providerships</td>
<td>Array</td>
<td>milestones</td>
<td>Array</td>
</tr>
<tr>
<td>total_money Raised</td>
<td>String</td>
<td>video_embeds</td>
<td>Array</td>
</tr>
<tr>
<td>funding_rounds</td>
<td>Array</td>
<td>screenshots</td>
<td>Array</td>
</tr>
<tr>
<td>investments</td>
<td>Array</td>
<td>external_links</td>
<td>Array</td>
</tr>
<tr>
<td>acquisition</td>
<td>Mixed</td>
<td>partners</td>
<td>Array</td>
</tr>
<tr>
<td>acquisitions</td>
<td>Array</td>
<td>offices</td>
<td>Array</td>
</tr>
</tbody>
</table>
Figure 1 Sample MongoDB document.

Figure 1 depicts the sample of MongoDB document. Figure 1 shows different field types in a document, for example, fields highlighted in A are simple type field which stores only one value in each field, however for highlighted fields in B and C are complex field type, B indicates a nested array and C shows an example of object array. However, each document does not have a standard structure (schema) within the same collection.

5.2 Migration Algorithm

From the structure of data field in MongoDB, due to the non-static structure of MongoDB document, it is necessary to know the structure of all documents and fields. Therefore, the research applies a record-to-record method to migrate data to Neo4j database. Figure 2 depicts the migration algorithm with a flowchart diagram.
The migration process intends to transfer data of a collection in a record-to-record (in this case, document-to-document) basis. The algorithm starts by access the first document. Then, each field within the document will be analyzed for its type. The purpose is to detect the necessity to create additional node for the field. This is due to the different data model supported by Neo4j, for example, Neo4j does not support nested nodes, and therefore for field type such as array of object, a separated node needs to be created to represent the nested field. For simple data field such as test, numeric, alpha-numeric, simple array, the field will be migrated as a property of the target Neo4j node. The nodes are created under Neo4j label *Companies*. However, for complex data type field, extra nodes will be created. For example, the *funding rounds* field in Figure 1; it is an array of object in which each element in the array is a document. Therefore, each element in the array will be migrated as a node in Neo4j. Since the node is created to represent the complex field, a relationship will be created to indicate the connection between nodes.

The creation of node in the algorithm is performed by first generate the Cypher Query Language (CQL) string. This is performed by calling the function *Generate Subquery ()*. The CQL string will be executed to create the intended Neo4j node. After all related-nodes for the complex field are created, the creation of the “mother” node will be performed. “Mother” node means the node that represent a document and consists of those simple field type as property. In this experiment, a “mother” node represents a *Companies* document and the child nodes are the nodes that represent the complex field in MongoDB document. A relationship named “HASSUB” will be created to link all related nodes together. Therefore, the algorithm is creating related-nodes for a single document first, then, create the node for the document itself and link these nodes together. The process will repeat until all documents are migrated.

When all records are migrated, in the match value process, a comparative method is applied to validate and verify the migrated data based on the data origin. This is performed by matching the content value of each node with the value in MongoDB collection. The algorithm of the comparative method is depicted by flowchart in Figure 3.
As depicted by Figure 3, the algorithm starts by retrieving the documents in the MongoDB **Companies** collection. The related nodes will be retrieved from the Neo4j according to the collection name, that is, nodes in the **Companies** label will be retrieved. Each field from the Mongo document and Neo4j node will be compared. If the field is a complex type, then the related nodes will be retrieved and the comparison process will be started again. The comparison of process finished after all records are compared. The comparison result will be written to the **CheckLog.txt** file.

6. Experiment Result and Discussion

The migrated data is viewed by using the Neo4j client browser interface. The labels are retrieved by using the CQL command “**Match (n) return distinct labels(n), count(n)**”. These labels are the results of migrating complex fields that lead to the creation of additional nodes in new label. Figure 4 shows parts of the returned labels and the number of nodes from the command. The highlighted **Companies** label consists of those nodes which contains the simple field data as it is in MongoDB before migration. Other labels are the result of migrating complex field. For example, the **funding_rounds** label, in MongoDB, it is an object array field. When the field is migrated to Neo4j, a node is created for each object element under the label has the same name with the field name.

![Figure 4 Neo4j labels.](image)

In addition, from the highlighted part in Figure 4, it shows that the total nodes in **Companies** label are 1583, which is same as the total document in **Companies** collection.

Next, the research proceeds to read the **CheckLog.txt** file, which consists of the comparison results. Figure 5 and Figure 6 shows parts of comparison results from the file. From the comparison result, it shows that the values between MongoDB documents and the related Neo4j nodes are mapped. To further prove on the algorithms, samples of records form MongoDB and Neo4j of matching result.

![Figure 5 Connected nodes mapping result.](image)
Ji and Azmi (2023)

Figure 6 Sample mapping result.

Figure 5 shows the matching result of the nodes that were created for the Mongo document which id is 52cdef7c4bab8bd675297d8a. The list of numbers represents the connected nodes id in Neo4j. The list of node id is generated by executing the CQL command that retrieve all connected nodes to the node which the oid property is similar to the document id. This oid property (field) of a node represents the Mongo document id in Neo4j which is use for relationship purpose. Figure 6 depicts portion of the matching result of each field.

To further verify the result generated by the matching algorithm, sample records are selected from both MongoDB and Neo4j. Figure 7 and Figure 8 depict the selected fields/properties on the sample document and node respectively. The fields/properties shown in Figure 7 and 8 are the simple fields. The values depicted by Figure 7 and Figure 8 are similar to the value depicted by Figure 6.

![Figure 7 Sample selected Mongo document fields.](https://www.malque.pub/ojs/index.php/msj)

In addition, the verification proceeds to check on the representation of complex field in the document. In this case, the funding_rounds fields are selected. Figure 9 depicts the funding_rounds field form the same document sample. As discussed earlier in the paper, the funding rounds field is an array of object, each object represent a document. Therefore, it is a nested document structure.

Form the sample in Figure 9, the funding_rounds of the document consists of three elements, that is, when migrated into Neo4j, there should be three nodes created under funding_rounds label. For the investments field, nodes are created under investments label in Neo4j. Each element in investments consists of three fields that are company, financial _org and person fields.
Figure 9 funding_rounds field from Mongo document.

Figure 10 and Figure 11 depict the related nodes from Neo4j based on the sample document in Figure 9. From the three figures, it shows that the values of each field in the funding_rounds and investments nodes are matched with the values of document fields in Figure 9. However, Figure 11 does not show the financial_org field due to the field is a document object; therefore, the migrated field becomes a node under the financial_org label. Figure 12 depicts the related financial_org nodes from Neo4j.
Figure 11 Related investments nodes.

```sql
$match (n:investments) where n.id = '52cdef74babbd75297d8a'
```

Figure 12 Related financial_org nodes.

```sql
$match (n:financial_org) where n.id = '52cdef74babbd75297d8a'
```

From the comparisons, it shows that the migration method is able to migrate the data as according to the discussed schema.

7. Conclusions

The record-to-record migration method migrates data from MongoDB to Neo4j in a sequential way. Due to the flexibility of storage paradigm in MongoDB, migrating data requires a predefined structure that are supported by the targeted type of database. For example, in this paper, the Neo4j does not support nested nodes in compare to MongoDB which allowed nested document to be stored as a field. Therefore, the identification of field type is crucial and MongoDB allows greater flexibility in data storing within a document in compare to Neo4j node.

With the record-to-record validator method for the migration data, it needs further verification as manual verification discussed in the paper may not be able to sufficiently verify the effectiveness of the method.

For future works, study will proceed to the migration of more complex field structure and with other NoSQL databases.

Ethical considerations
Not applicable.

Declaration of interest
The authors declare no conflicts of interest.

Funding
This research did not receive any financial support.

References
Dharmasiri HML, Goonetillake MDJS (2013) A federated approach on heterogeneous NoSQL data stores. in 2013 International Conference on Advances in ICT for Emerging Regions (ICTer).


