Project Report on

Sql2Neo: Interconversion of SQL, NoSQL and Neo4j formats

Submitted by

Aayush Jain (16IT101)

Aditi Rao (16IT103)

Under the Guidance of

Shruti J R

Department of Information Technology, NITK Surathkal

Date of Submission: 11 June 2020

in partial fulfillment for the award of the degree

of

Bachelor of Technology

In

Information Technology

At

Department of Information Technology

National Insitute of Technology Karnataka, Surathkal

June 2020

Abstract

In order to leverage data relationships and gain insights into the “big picture” presented by the data, organizations need a database technology that stores relationship information as a first-class entity. This can be done through the use of Graph Databases such as Neo4j. However, many legacy databases are currently stored in SQL or (more recently) non-graphical NoSQL-based systems, thus removing the possibility of enhanced insights brought about through the use of a graph. Thus, this work details the methodology and implementation of Sql2Neo, a command-line tool to convert data between SQL/NoSQL systems to Neo4j. The resulting database effectively stores data relationships but is also flexible when expanding a data model or conforming to changing business needs. It is uniquely suited to visualization for higher-level decision making and inference mining. Sql2Neo also supports query translation to Neo4js Cypher Query Language (CQL) from standard formats such as SQL, increasing the ease of mobility from SQL/NoSQL systems to Neo4j without the rewriting of existing queries.

Contents

1 Introduction

A graph database is an online database management system with Create, Read, Update, and Delete (CRUD) operations working on a graph data model. A graph data model is defined by a collection of nodes representing entities and relationships that connect the nodes. Graph databases are preferred when we want to focus exclusively on understanding the relationship between entities in our database and want to be able to infer data connections without using things like foreign keys or out-of-band processing, such as MapReduce.

In order to leverage data relationships and gain insights into the “big picture” presented by the data, organizations need a database technology that stores relationship information as a first-class entity. The rigid schemas of legacy relational database management systems (RDBMS) make it difficult to add different connections or adapt to new business requirements. Not only do graph databases effectively store data relationships; they are also flexible when expanding a data model or conforming to changing business needs, and are uniquely suited to visualization for higher-level decision making and inference mining.

Relational Databases or SQL databases are a type of DBMS that defines database relationships in the form of tables, also known as relations. They are the most popular DBMS type in the market. Unlike network DBMS, RDBMS does not support many to many relationships. Relational DBMS usually have pre-defined data types that they can support. SQL stands for Structured Query Language and is the standard language for dealing with Relational Databases. It can be used to insert, search, update, and delete database records and many other operations, including optimizing and maintaining databases. Databases like MySQL Database, Oracle, MS SQL Server, Sybase, etc. use SQL.

NoSQL is an upcoming category of Database Management Systems. It is main characteristic is it is non-adherence to Relational Database Concepts. The popularity of NoSQL databases grew with internet giants such as Google, Facebook, Amazon, etc. who use them to deal with gigantic volumes of data. NoSQL systems are preferred in these cases due to their ability to manage large datasets without performance trade-offs. The alternative to upgrading hardware to fix the problem would be to distribute our database load on multiple hosts as the load increases. This is known as “scaling out”. NoSQL databases are non-relational databases that scale-out better than relational databases and are designed with web applications in mind. They do not use SQL to query the data and do not follow strict schemas like relational models. Some examples of NoSQL databases include MongoDB, CouchDB, and HBase. Neo4j is a graph database management system described as an ACID-compliant transactional database with native graph storage and processing. Neo4j is the most popular graph database used today. Neo4j is implemented in Java and accessible from software written in other languages using CQL through a transactional HTTP endpoint, or through the binary “bolt” protocol.

2 Literature Survey

2.1 Related Work

Singh et al. [1] propose a methodology to convert a relational to a graph database by exploiting the schema and the constraints of the source, applied in this case to healthcare data. Here, MySQL was used as the source database with the intended target being Neo4j. They reported the notable efficiency of the system designed for querying. They also note that visualization has also significantly improved with the usage of graph databases.

De Virgilio et al. [2] also propose a methodology to convert a relational database to a graph database by exploiting the schema and the constraints of the source. The approach also supports the translation of conjunctive SQL queries over the source into graph traversal operations over the target. The methodology describes an automatic conversion system from SQL databases to a graph database which allows the created database to perform better than RDB in certain instances of highly complicated schemas.

Maity et al. [3] view the standard problem of converting the RDBMS to NoSQL in the opposite approach and conceptualize a problem where NoSQL is converted back to an RDBMS based system. The approach is illustrated using a case study on MongoDB and Neo4j. They note significant successes in conversion using their algorithm and note the potential for its use in not only NoSQL systems to RDBMS but also for interconversion between two NoSQL systems.

Vicknair et al. [4] report on a comparison of Neo4j with MySQL, for use as the underlying technology in the development of a software system to record and query data provenance information. They noted an almost tenfold increase in speed while using Neo4j for traversal based queries. They conclude that, in general, the graph database did better at the structural type queries and full-text character searches than the relational database. With more development of Neo4j to include integer-based indexing measures as well, it could outperform RDBMS systems in a query on numeric data.

Batra and Tyagi [5] also report on the comparative performance of Neo4j and MySQL, noting that the retrieval times of graph databases is less than relational databases as graph databases look only at records that are directly connected to other records. They conclude noting the flexibility of graph databases when new relationships are added as schema restructuring is not required.

2.2 Outcome of Literature Survey

From the survey, it is clear that the use of graph databases for applications involving a significant focus on entity relationships or visualization would be highly beneficial. It is also evident that it may be useful to transform data from traditional RDBMS to NoSQL systems, such as graph databases solely for the flexibility it offers in terms of the addition of relationships/query speeds for traversal queries. Our proposed methodology aims to allow for quick and easy transformation of existing databases to graph format to take advantage of these benefits.

2.3 Problem Statement

Graph database systems such as Neo4j offer various advantages over traditional RDBMS in terms of traversal queries, visualization, and flexibility in defining new relations without schema restructuring. However, a vast majority of applications could benefit from this store data in traditional SQL setups (while some do in newer NoSQL systems). Thus, there is a need to develop a tool to allow for efficient transformation between these systems to graph databases to enhance decision making and insights into the data.

2.4 Objectives

In order to meet the demands of this problem, our proposed system must meet the following objectives. The system architecture must be designed to allow for *extensibility* in terms of the addition of new SQL/NoSQL systems to convert from. To allow for this, it is necessary to develop a *standard intermediate format* from which conversion to Neo4j can take place. The system must be able to accurately and appropriately *represent relationships* expressed by the use of concepts such as foreign keys in RDBMS, as well as the many to many relationships described in NoSQL systems. The transformation must be done quickly, *without loss of information*, and into a form suitable for direct visualization. There must also be an efficient method for *query translation from SQL/NoSQL based query languages to CQL*, as well as the creation of required indices on identified fields to allow for faster query processing and result retrieval.

3 Requirement Analysis

Sql2Neo is intended to be a command-line tool that enables interconversion between SQL, NoSQL, and Neo4j formats.

3.1 Functional Requirements

F1. SQL databases must be fully converted to a Neo4j database. This includes indexes, constraints, records, and relationships.

F2. NoSQL databases must be fully converted to a Neo4j database. This includes constraints (inferred) and records.

F3. SQL queries must be translated to a Neo4j query to provide interoperability between databases.

3.2 Non-Functional Requirements

NF1. Convert NoSQL database to intermediate SQL format to provide compatibility.

NF2. Manage data loading from databases in order to prevent memory overuse and overflow errors.

NF3. Provide Sql2Neo as an isolated tool (with packaged dependencies) to improve usability.

Most software and hardware requirements intersect with the implementation tools, as discussed in section 6.

4 System Design/Architecture

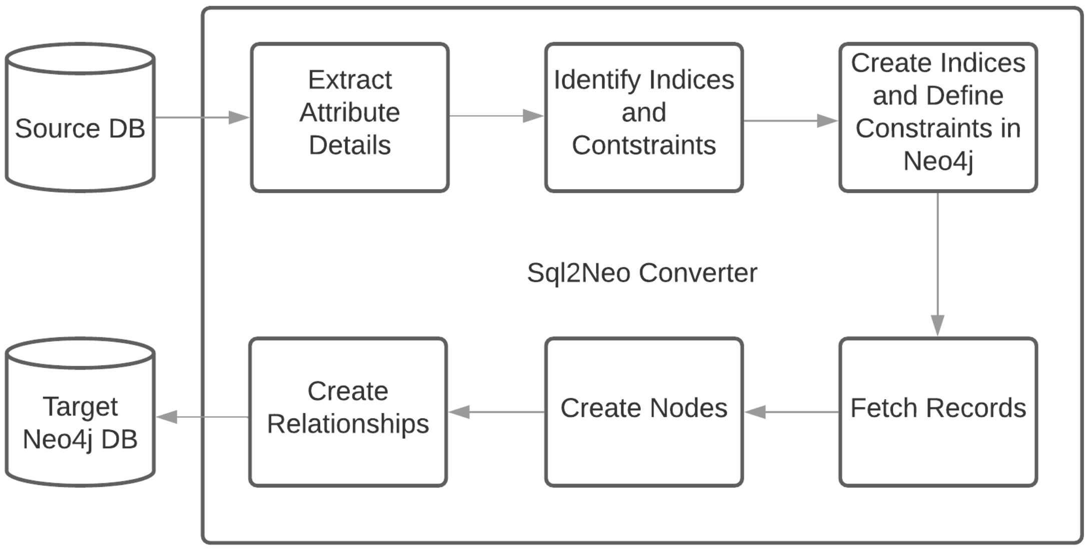

Source DB
Extract Attribute Details
Identify Indices and Contstraints
Create Indices and Define Constraints in Neo4j
Sql2Neo Converter
Target Neo4j DB
Create Relationships
Create Nodes
Fetch Records

Figure 4.1: SQL2Neo Data Converter

Figure 4.1 presents an overview of the proposed system architecture. The aim is to take our source database and start by extracting attribute details. This includes information on attribute name, data type (in case conversion is required as Neo4j does not support some data types), uniqueness, a primary/foreign key, or an index that needs to be built on it. Through this, we can further identify indices and constraints in the data, and subsequently, define them as required. At the end of this step, all data description information has been translated to a format which can be further used while translating records into Neo4j.

Then we begin the population of the Neo4j database. We fetch the records from the source database and, using the data descriptions we have extracted in the previous steps, store it in an intermediate, universal format. This is the last step before node and relationship creation in Neo4j begins.

As Neo4j creates relationships between existing nodes, the nodes are defined first from the identified entities in the data description. Then, relationships or links between the nodes are created and committed to the database.

The system architecture has been designed to encourage extensibility to include compatibility with multiple SQL and NoSQL implementations. All the primary steps before node creation begin to deal with bringing the data to an established universal format, after which conversion to Neo4j entities and relationships is a single pipeline process for all. Thus, it requires that only translation steps to this intermediate format must be determined to include another implementation in the converter.

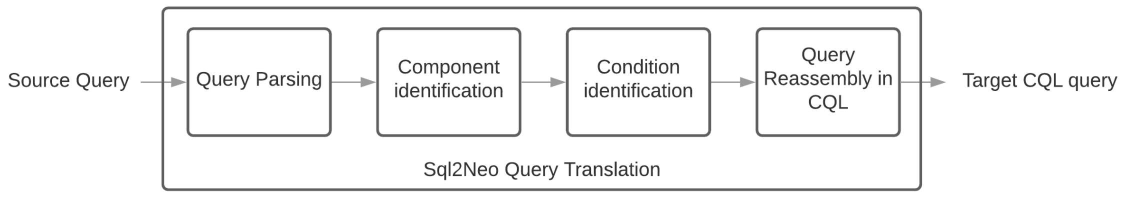


Figure 4.2: SQL2Neo Query Translator

In addition, we also undertake query translation from source (SQL/NoSQL) to CQL or Cypher Query Language. The steps begin with parsing the query to isolate terms of interest (figure 4.2). This is followed by identifying components (such as entities and attributes) from the parsed data. We then identify any conditional clauses to be placed (such as found in an SQL "WHERE" query phrase) and represent them appropriately. Finally, the query is reassembled from its components and placed into a template structure for an equivalent CQL query.

5 Methodology

5.1 Converting SQL-databases to Neo4j format

The first step to convert an SQL database is to extract attribute details of each table. This data is available in the `information_schema` of the database. The data for each attribute answers the following questions:

- should this attribute be *indexed*?
- does this attribute have a *uniqueness constraint*?
- is this attribute a *foreign key reference* to another table's attribute?

Algorithm 1: Extract attribute details of SQL database

```
Input: R, a relational database
Output: AS, an attribute set containing details of each attribute
AS ← $\phi$;                                          /* empty map */
foreach table in information_schema(R) do
    AS[table] ← $\phi$;                               /* empty map */
    foreach attr in table do
        AS[table][attr] ← {index, unique, fk};
    end
end
return AS;
```

The index and uniqueness constraint data enable the Neo4j set up to be as closely modeled to the relational one. Furthermore, indexing on attributes is maintained across systems, and rigid constraints are also satisfied while inserting in the Neo4j database.

Since Neo4j stores data as JSON documents, it does not define a rigid and formal schema. Owing to this property, indices, and constraints can be created before the data is actually inserted. In Neo4j, index creation is not idempotent, meaning that creating the index twice results in an error. Additionally, constraints implicitly create an index on the specified attribute (much like relational databases). Thus constraints are applied only on those attributes that are not indexed.

Algorithm 2: Create indices and constraints in Neo4j

```
Input: AS, an attribute set containing details of each attribute
Result: G, a graph database with applied indices and constraints
foreach table in AS do
    foreach attr in table do
        if attr must be indexed then
            CreateIndex(G, table, attr);
        else if attr has constraint but not indexed then
            /* if attr is indexed, then it meets uniqueness constraint
            */
            CreateUniquenessConstraint(G, table, attr);
    end
end
```

In terms of CQL, `CreateIndex(G, table, attr)` is equivalent to

```
CREATE INDEX index_name ON:table(attr);
```

`CreateUniquenessConstraint(G, table, attr)` is equivalent to

```
CREATE CONSTRAINT constraint_name ON (t:table) ASSERT t.attr IS
    ↪ UNIQUE;
```

Conversion of a table's records to Neo4j nodes is a fairly straightforward task. A naive approach is followed where each table's records are converted to a node. This implies that individual tables that behave purely as relationships are also converted to nodes instead of being retained into Neo4j.

`CreateNewNode(G, table, record)` is equivalent to

```
CREATE (t:table $record);
```

provided that `record` is a map of key-value pairs (record assumed as a parameter).

Finally, relationship conversion is performed. This step takes the foreign key relations from each table and maps them to a relation in Neo4j. This also means that the semantics of the relation is lost since the edge (in Neo4j) does not provide actual data. Instead, it must be inferred from the database.

`CreateRelationship(G, table, attr, fk_table, fk_attr)` is equivalent to

Algorithm 3: Populate the graph database with records

```
Input: R, a relational database
Result: G, a populated graph database
foreach table in R do
    foreach record in table do
        CreateNewNode(G, table, record);
    end
end
```

Algorithm 4: Create relationships between nodes in the graph data

```
Input: AS, an attribute set containing details of each attribute
Result: G, a graph database with relationships
foreach table in AS do
    foreach attr in table do
        if attr is foreign key then
            fk_table, fk_attr ← GetFKReference(attr);
            CreateRelationship(G, table, attr, fk_table, fk_attr);
        end
    end
end
```

```
MATCH (a:table), (b:fk_table) WHERE a.attr = b.fk_attr CREATE (a)
    ↪ -[r:relationship_name]-> (b);
```

5.2 Converting NoSQL-databases to Neo4j format

The primary difference between a NoSQL and relational database is that NoSQL databases do not define a formal schema. If data from such a database were to be migrated, it would have lacked constraints and indices. Thus, an additional step is performed as compared to the procedure for RDBMS migration.

First, the NoSQL database is converted to an intermediate SQL-like format. This intermediate format improves maintainability and allows for seamless management of attribute characteristics over a traditional method.

Algorithm 5: Convert a NoSQL database to a relational format

```
Input: N, a NoSQL database
Output: AS, a relational interpretation of N's attributes
AS ← $\phi$;                                                /* empty map */
foreach collection, C, in N do
    AS[C] ← $\phi$;                                         /* empty map */
    foreach field in C do
        if field has unique values then
            AS[C][field] ← {unique←true};
        end
        if field not present in each record then
            make field nullable
        end
    end
    if unique field exists then
        AS[C][field] ← {index←true}
    else if C has unique id then
        /* uniqueness is absent                                  */
        use auto-generated id as index;
    else
        insert unique id and use as index;
end
return AS;
```

A NoSQL can define records in the same collection with a field missing. To convert to a relational model, all such fields are assumed to be nullable, and the missing records are filled in with `NULL` values. Furthermore, the first unique field is taken as the primary key. Many NoSQL databases auto-generate a unique object identifier, which can prove to be useful if a unique field does not exist. Once `AS` is derived from the database, the migration can proceed as per the algorithms defined for SQL databases.

5.3 Translating SQL queries to Cypher queries

SQL and Cypher have quite a few common keywords that aid in simpler translation between queries. At its core, query translation requires:

1. identification of queried tables
2. identification of projection attributes
3. identification of selection conditions
4. extracting other keywords that can be translated

As mentioned above, few of the keywords shared between SQL and Cypher are `WHERE`, `LIMIT`, `ORDER BY`, `AS`, and `DISTINCT`. In algorithm 6, tokens are considered to be of two types, namely, *keywords* and *identifiers*. Identifiers follow a keyword normally, and are treated as a list of identifiers. This allows for multiple identifiers to be picked up for translation at once.

```
SELECT p.name FROM person p;
SELECT name FROM person;
```

The above queries are equivalent, however, Cypher requires node labels to be explicitly provided. So, `MapToVariableName` maps a given list to appropriate variable names (using 'n' as default variable).

```
MATCH (p:person) RETURN p.name;
MATCH (n:person) RETURN n.name;
```

A similar approach is applied to mapping variable names to the `ORDER BY` attributes.

The algorithm can be extended to include other keywords that exist within CQL, and also to convert other keywords (like JOIN) into Cypher equivalents. The given approach is provided only on `SELECT` queries since commonly used ones may benefit from such a translation.

Algorithm 6: Translate SQL query to CQL

```
Input: Q, an SQL query
Output: C, converted Cypher query
tokens ← Tokenize(Q);        /* split SQL query into individual tokens
 (keywords and identifiers) */
selection ← NULL;
limit ← NULL;
order ← NULL;
for token in tokens do
    if token is SELECT then
        attributes ← TokenAfter(token);
        attributes ← MapToVariableName(attributes);    /* if table name is
         not provided (table.attr), then attributes must be mapped to
         default */
    if token is FROM then
        tables ← TokenAfter(token);
        tables ← MapToVariableName(tables);       /* if variable name not
         given to table, provide defaults */
    else if token is WHERE then
        selection ← TokenAfter(token);
    else if token is LIMIT then
        limit ← TokenAfter(token);
    else if token is ORDER BY then
        order ← TokenAfter(token);
        order ← MapToVariableName(order);        /* if table name is not
         provided (table.attr), then attributes must be mapped to
         default */
end
C ← BuildCypherQuery(tables, attributes, selection, order, limit);
return C;
```

6 Implementation

The proposed approach was implemented on a system running Ubuntu 20.04 LTS. The following additional software is used:

- Python 3.8.2
  - mysql-connector-python 8.0.20
  - py2neo 4.3.0
  - pymongo 3.10.1
  - python-dotenv 0.13.0
  - pandas 1.0.4
  - sqlparse 0.3.1
- MySQL v8.0.20 for testing SQL databases
- MongoDB v3.6.8 for testing NoSQL databases
- Neo4j 4.0.5

The user decides which database he/she wants to migrate. Details of each database (host, username, password, database name) must be present in a `.env` file. Using the configuration provided here, a connection is established with the source and the target database (Neo4j). As per algorithm 1, an attribute set, AS, is maintained that holds data in the following format.

```
‘trained’: {
    ‘certificationdate’: {
        ‘position’: 3,
        ‘index’: False,
        ‘unique’: False,
        ‘foreign_key’: None
    },
    ‘certificationexpires’: {
        ‘position’: 4,
        ‘index’: False,
        ‘unique’: False,
        ‘foreign_key’: None
    },
    ‘physician’: {
        ‘position’: 1,
        ‘index’: False,
        ‘unique’: False,
        ‘foreign_key’: ‘physician.employeeid’
    },
    ‘treatment’: {
        ‘position’: 2,
        ‘index’: False,
        ‘unique’: False,
        ‘foreign_key’: ‘procedure_record.code’
    }
},...
```

From the above data, it is easy to create indices and constraints on any given attribute.

Test migration for a MySQL database is carried out on a sample hospital database containing 45 records across 10 tables. This dataset has 13 different foreign key relationships.

Similarly, for NoSQL, the data is derived in the same format as for MySQL. However, there is an additional cost associated with determining the attribute statistics, i.e., the collected data has to be fully loaded into memory to check for uniqueness. In order to handle loading large datasets in memory, *pandas DataFrames* is used. They can efficiently load large datasets and provide useful methods such as `is_unique` to help determine attribute statistics.

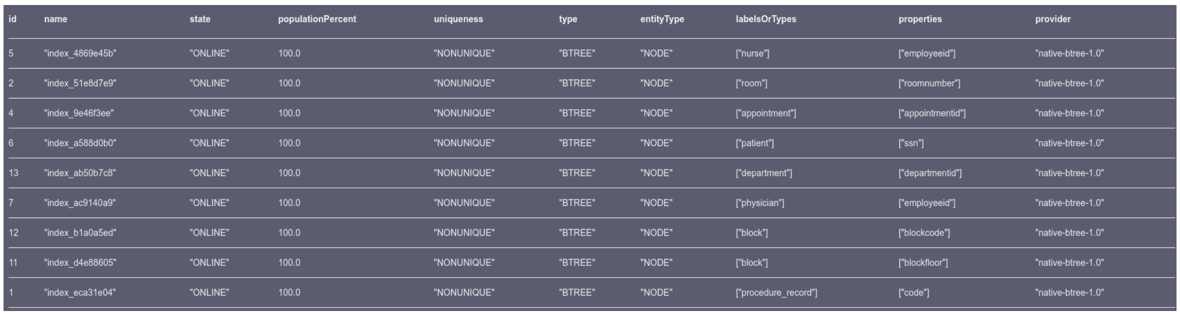
Testing for MongoDB is done on two databases. The first is a database containing US zip code information with 29353 records. Each record has the same attributes and behaves similar to a relational database. Loading this zip code information into a DataFrame takes negligible time, with the main bottleneck being the creation of 29353 nodes. For the second case, a sample database is created with a badly-defined schema. Multiple attributes are missing in the records of the collection. All these values are created and marked as `None` (`NULL` in Python).

As mentioned in section 5.3, the input queries are first tokenized and then converted to CQL equivalents. Tokenization is performed by the *sqlparse* library. The program allows users to run the queries on the migrated database, or perform a *dry run*, i.e., the user may validate the translated queries. As per the algorithm, the following keywords/features exist:

- Allows for projection attribute aliasing, i.e., p.name AS 'Patient', is retained
- Table renaming, i.e., SELECT p.name, nurse.name FROM patient p, nurse n, retains table names defined in SQL
- '*' projection is retained and replaced with 'RETURN n'
- Projection attributes are fully translated
- Order by, and limit keywords are retained

7 Results and Analysis

We perform the conversion on a test MySQL database “hospital” as described in the implementation section. The following images display the results upon conversion to a Neo4j graph database.



id	name	state	populationPercent	uniqueness	type	entityType	labelsOrTypes	properties	provider
5	"index_4869e45b"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["nurse"]	["employeeid"]	"native-btree-1.0"
2	"index_51e8d7e9"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["room"]	["roomnumber"]	"native-btree-1.0"
4	"index_9e46f3ee"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["appointment"]	["appointmentid"]	"native-btree-1.0"
6	"index_a588d0b0"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["patient"]	["ssn"]	"native-btree-1.0"
13	"index_ab50b7c8"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["department"]	["departmentid"]	"native-btree-1.0"
7	"index_ac9140a9"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["physician"]	["employeeid"]	"native-btree-1.0"
12	"index_b1a0a5ed"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["block"]	["blockcode"]	"native-btree-1.0"
11	"index_d4e88605"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["block"]	["blockfloor"]	"native-btree-1.0"
1	"index_eca31e04"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["procedure_record"]	["code"]	"native-btree-1.0"

Figure 7.1: Indexes created on test database “hospital”

In figure 7.1, we can see the indexes created on the test database. It is essential to note the successful translation of uniqueness constraints and attributes to build B-Tree indexes on the fields as required. Index names are auto-generated.

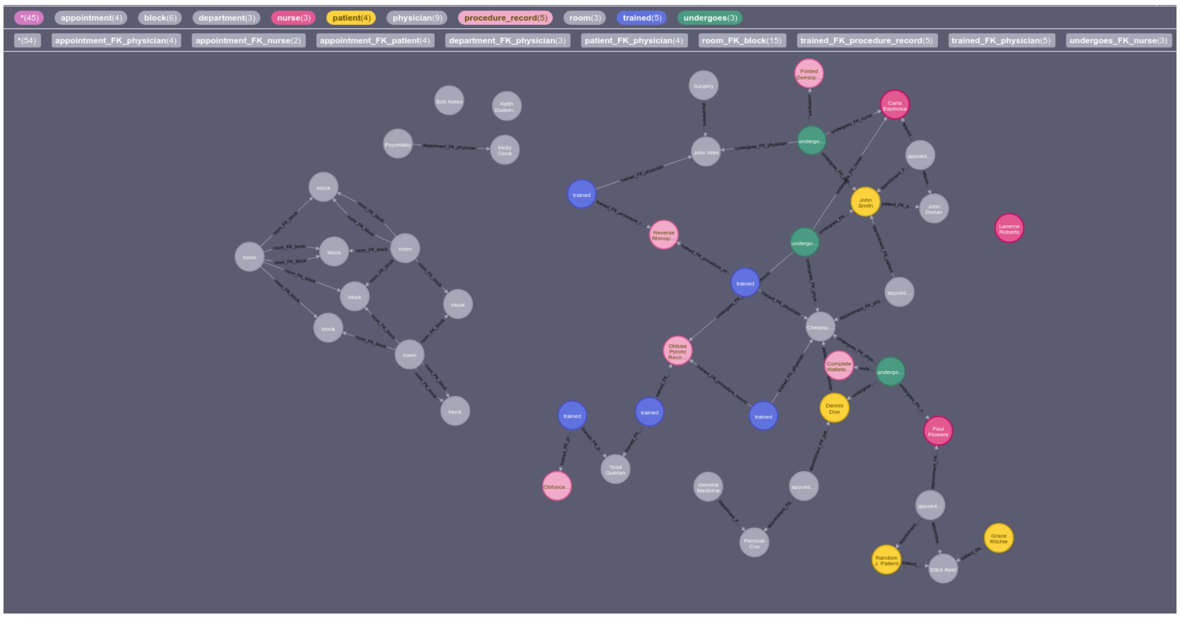


Figure 7.2: Nodes and relationships created on test database “hospital”

In figure 7.2, we show the successful creation of a graph database with all tables and relations being translated to Neo4j nodes and relations. The entity types are represented by different colored nodes (legend can be found in the top bar of the image), and the labeling of the nodes corresponds to the value in the attribute “name”. Relationships are represented by a single line joining two nodes with the relationship name written on it.

Hovering over an entity will display the additional attributes (apart from “name”) of the entity. Similarly, we can see additional attributes of relations by hovering over them.

For a closer look at the structure of the nodes, we look to figure 7.3.

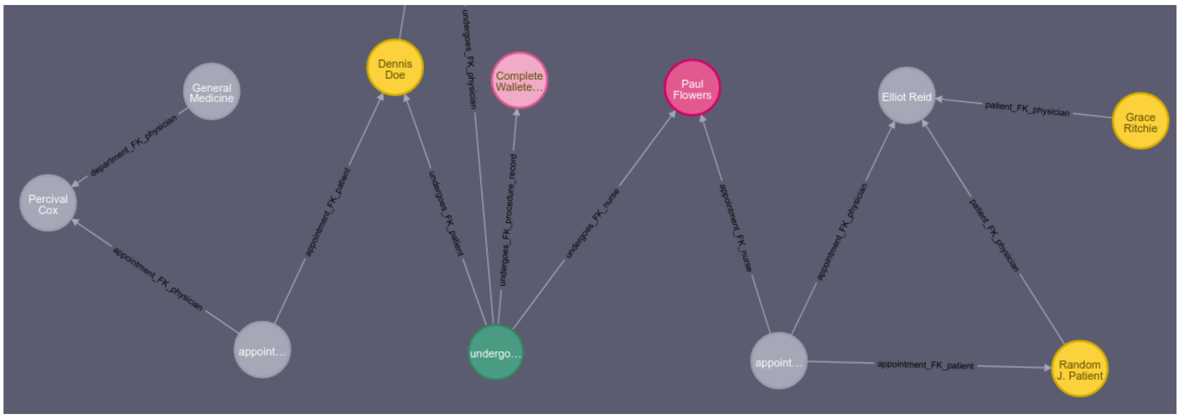


Figure 7.3: View of nodes and relationships created on test database “hospital”

Here we get a better view of the structure of nodes and relationships. As mentioned earlier, nodes are identified by their “name” attribute value, and colors represent the entity types.

Relationships between have been named according to the convention

```
<from_relation>_FK_<to_relation>
```

to describe the foreign key linking the two tables in the MySQL schema form. “Relationship Nodes” have been naively created to represent concepts involving multiple foreign keys such as the “appointment” nodes, created to link physician, patient and nurse nodes. Ideally, these relationships should be modeled as links between the nodes, rather than nodes themselves. However, due to challenges faced in the automatic identification of entities vs. relationships, we currently model them, as shown above.

The output of MongoDB is not shown here since it just creates nodes and indexes. Since no relationship extraction is performed, it is a subset of the output of MySQL migration.

8 Conclusion and Future Works

In conclusion, it is found that it is feasible to undertake such transformation tasks between SQL/NoSQL databases and that it is also quite beneficial when considering improved data visualization and presentation. The implementation of Sql2Neo discussed is additionally easily extensible to include various other source databases as a result of its system architecture and the use of an intermediate SQL-like data storage strategy.

Future works include, as mentioned, the extension of this project to bring more source databases within the transformation fold and improve the quality of relationships derived from the data through more inference driven approaches (rather than the naive approach undertaken here). Furthermore, the implementation of a query translator from SQL to CQL will bridge the gap between the two technologies. It will also allow for a more analytical approach to the SQL database and a graphical approach to the Neo4j database. Currently, all indices are single attribute indices and are created only on the primary key (or first unique value for NoSQL). An improvement would be to support composite attribute index conversion. These indices are present in SQL databases and can provide performance improvements in Neo4j as well.

References

[1] M. Singh and K. Kaur, “Sql2neo: Moving health-care data from relational to graph databases,” in *2015 IEEE International Advance Computing Conference (IACC)*, pp. 721–725, 07 2015.

[2] R. De Virgilio, A. Maccioni, and R. Torlone, “Converting relational to graph databases,” in *First International Workshop on Graph Data Management Experiences and Systems*, GRADES 13, (New York, NY, USA), Association for Computing Machinery, 2013.

[3] B. Maity, A. Acharya, T. Goto, and S. Sen, “A framework to convert nosql to relational model,” in *Proceedings of the 6th ACM/ACIS International Conference on Applied Computing and Information Technology*, ACIT 2018, (New York, NY, USA), p. 16, Association for Computing Machinery, 2018.

[4] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, “A comparison of a graph database and a relational database: A data provenance perspective,” in *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE 10, (New York, NY, USA), Association for Computing Machinery, 2010.

[5] S. Batra and C. Tyagi, “Comparative analysis of relational and graph databases,” in *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, 05 2012.