Journal of Theoretical and Applied Information Technology
30th June 2021. Vol.99. No 12
© 2021 Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A COMPREHENSIVE APPROACH FOR CONVERTING RELATIONAL TO GRAPH DATABASE USING SPARK

[1]WAEL MOHAMED, [2] MANAL A.ABDEL-FATTAH, [3]SAYED ABDELGABER
[1,2,3]Information Systems dept, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt
E-mail: [1] waelmohamed@fci.helwan.edu.eg, [2] manal_8@hotmail.com , [3] sgaber14@gmail.com

ABSTRACT

Nowadays, data processing requirements is growing exponentially, and relational database is not always the best solution for all situations in big data such as increasing growth of data. Thus, NoSQL databases emerged to overcome the limitations of relational database and work with big data. NoSQL databases have four types of models, namely, key-value model, document database, column database, and graph database. Many approaches have been proposed to convert relational database to NoSQL models. However, most of them map relational database to key-value or column or document. Converting relational to graph database is slightly disregarded by the researchers.

This paper proposes a comprehensive approach, based on Spark framework, for transformation and migration of relational database to graph database without semantic loss. The approach also supports conversions from Sql commands to cypher commands .It is categorized into two parts. The first part is concerned with "transformation and migration using Spark", which encompasses three phases: Meta data analyzer, transformation algorithm, and migration algorithm. The second part focuses on "SQL to cypher", which divides into two phases: SQL parser and Translator. The suggested approach has been applied, results and validation for the proposed approach

Keywords: *Big Data , NOSQL , Graph Database* , Spark , Neo4j

1. INTRODUCTION

with the increasing size of datasets, big data processing has attracted more research attention[1]. Managing big data is a key issue when availability and scalability are required. While traditional database management systems can store large-scale data, they have significant limitations in scalability and availability.

To overcome the limitations of RDB, a new category of database, called NoSQL (Not only Structured Query Language), has been proposed. NoSQL databases are new category of databases that differ from the RDB (Relational database). It supports horizontal scaling, storing complex data, high availability, no fixed schema, fault tolerance, frequent updates to data, and fast development. In addition, it does not depend on Join operations. It is suitable for cloud computing and big data. Therefore, enterprises want to move to NoSQL. There are four categories of NoSQL databases, which are key-value model, document databases, column databases, and graph databases[2]. There is no standard query language for NoSQL databases because each model has different data model and different data access methods. On the other hand, there exist an enormous number of users familiar with SQL. As consequence, migration to NoSQL has a very sloped learning curve. Therefore, the motivation is to use SQL language and big data capabilities of NoSQL stores.

Several approaches have been proposed to convert RDB to target NoSQL database. The approaches create a bridge between SQL and NoSQL that allow users to write SQL on NoSQL database. The main objectives: (i) integrate the world of SQL and big data capabilities of NoSQL (ii) minimize migration cost from RDB to NoSQL. The approaches of mapping and migrating from RDB to NoSQL models are organized into three categories (i) layer that work as middleware between RDB and NoSQL (ii) Storage engine that adjust storage manager in RDB to store relational data in NoSQL database (iii) Migration approaches from relational to graph.

Most of these approaches have common methodology which concluded in these steps: (i)

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

transforms from relational to target NoSQL model (ii) migrate data from relational to NoSQL (iii) provide query mapping from SQL to the language of target NoSQL database. The flaw of first category of approaches, to the best of our knowledge, is that no layer for transformation of RDB to graph database[3]. The existing layer approaches transform only RDB to key-value or column or document models. The weakness of second category is the restriction to specific NoSQL model and did not include graph database. The main flaws of the third category are the experiments didn't conduct in distributed environments [4] and there is no research paper that presents query mapping from SQL to cypher[5].

In this paper, we present an approach, based on spark as a distributed processing engine[6], to transforms and speed up the migration of data from RDB to graph database. The approach also supports translation from SQL to cypher commands. The approach consists of two parts. The first part transforms and migrate data from relational to graph database while the second part translates SQL to cypher commands.

The contributions of this work are listed as follows:

- Proposing an efficient transformation algorithm to transform RDB to graph database.
- Presenting an efficient migration algorithm for migrate data from RDB to graph database using big data processing engine (Apache spark).
- The proposed layer converts RDB to graph database without a semantic loss that is very important for further analysis such as graph mining.
- Proposing query mapping from SQL to cypher query language.
- The proposed layer conducts experiments in distributed environment.

The reminder of this paper is organized as follows. Section 2 provides overview for related work. Section 3 describes the architecture, Transformation algorithm, migration algorithm and query mapping from SQL to cypher language. Experiments and results are outlined in Section 4.The conclusion of this research is offered together with suggestions for future research directions in the final section

2. BACKGROUD AND RELATED WORK

2.1 Definition And Background

This section presents basic definition of relational and graph database. Relational schema is denoted as $R_j = \{A_i, A_i, \ldots, A_M\}$ where R_j is j-th relation , A_i is the set of attributes in the i-th relation and M is number of attributes. In the relational database schema R where R = $\{R_1, R_2, \ldots R_N\}$ where N is number of relations. [7].

Graph database be denoted as G(N,E) such that N = $\{N_1, N_2, \ldots, N_N\}$ is set of nodes and E=$\{e_1, e_2, e_3, \ldots e_n\}$ is the set of edges that connect nodes e = (n_i, n_j) Each node and edge has its own properties[8].

2.2 Related works

The approaches can be placed into three categories: layers approaches, storage engines and migration approaches from relational to graph.

Layers approaches work as middleware between relational model and NoSQL models. Layers allow the transformation, migration and query mapping from RDB to NoSQL model. The motivation of layer approaches is to retain the benefits of SQL in the context of NoSQL. Layers can be classified according to (i) NoSQL models (ii) automatic mapping (iii) support join (iv) SQL support. Layer may support: i) an automatic transformation from relational model to NoSQL model or allow users to customize the transformation process (ii) the translation of SQL join operation to the language of NoSQL model (iii) the translation of all or subset of DML and DDL statements. The functional constrains of NoSQL models make mapping only of subset SQL. Most of the proposed layers support the transformation and migration key-value , column ,or document models such as [9][10][11][12]. The limitation of preceding layer approaches is no layer supports the transformation, migration and query mapping (all DML and DDL) to graph database [3].

Storage engines edit the kernel of RDB management system (RDBMS) to persist relational data in NoSQL model. Three approaches are proposed for storage engines Phoenix [13], CloudyStore[14] and DQE[15]. Phoenix stores only key-value model in MySQL. CloudyStore stores column model in MySQL while DQE stores column model in Derby. The preceding storage engines approaches restrict application to work with particular RDBMS but they provide optimization access and achieve the requirement for managing large-scale data.

Some scholars presented approaches for converting RDB to graph database [8][16-20]. in[16], the authors have proposed an approach to convert RDB to graph database. However, the

© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

approach aggregates the unifiable tuple in the same node that allow node to store large data. The approach violates the semantic of schema that makes the approach is not suitable for further analysis such as graph mining. It is very difficult to determine which type of semantic of this node belongs. If the RDB contains redundant attributes, the data may be losing during migration step. It did not cover the mapping of unary relationship. It uses large volume of database in the experiments but the tests did not conduct in distributed environments.

in[8], an approach for converting from RDB to graph database which includes migration order has been suggested. It determines which table to be migrates as node and which tables that will be migrated as edges. The authors also compare the proposed approach with [16]. The results display reduction of the number of generated nodes. It uses large database in experiments but did not conduct it in distributed environment. The main objective of this approach is to convert relational data to graph database as data preparation for graph analysis. Therefore, the approach did not include query mapping from SQL to any graph query language.

in[17], an improved approach offered for the approach in [16]. The proposed approach called FD2G. FD2G support unary relationship and associative entities. The main disadvantage of FD2G is that it needs to convert the RDB to the third normal form before the migration. FD2G did not use distrusted environment.

in[18], an approach that keeps the semantic from real world in migration of data from relational to graph database has been offered. The approach is the crux for further analysis. It suggests the direction of edge where the starting node is from the many side and the end node is in one side. The starting node contains foreign key as property in node and create edge between two nodes. It stores foreign key in the many so it is considered to be redundant because there is no need to store foreign key as property because the edge already represent the two tuples. The approach conduct experiments on small data. it did not include query mapping from SQL to Cypher.

in[19], transformation rules for mapping one to one , one to many and many to many to graph database have been suggested. The approach did not include automatic transformation and migration and use small data in experiments. It also did not include query mapping from SQL to graph database. The proposed also did not conduct experiments in distributed environment.

in[20] , an approach for migrating data from RDB to neo4j has been offered. It proposes rules for mapping one to one, one to many and many to many relationships. It also explains only the methodology for converting Sql to cypher. It did not conduct experiments in distributed environment and use small database. It did not include recursive relationship in mapping.

The related research works for converting RDB to graph databases have the following shortcomings.

- The lack of research on conducting experiments in distributed environment to reduce time for migration according to systematic literature review[4].
- The absence of research on automatic mapping from SQL to cypher language as stated by this survey[5].

Therefore, an approach that uses Spark as distributed processing engine in converting RDB to graph database is essentially needed.

3 THE PROPOSED APPROACH FOR MIGRATION RDB TO GRAPH DATABASE

The proposed approach uses the benefits of relational in the context of graph database. It consists of two parts (i) Transformation and migration of data using Apache Spark engine and (ii) SQL to cypher. The first part follows the same strategy that transform from relational to NoSQL model then migrating data from RDB to NoSQL model. The first phase in the first part is to perform metadata analysis for RDB. The second phase is the transformation algorithm. The third phase is the migration algorithm. The second part of the proposed approach layer consists of SQL parser and the translator from SQL to cypher equivalent query. The architecture has three types of dictionary. Dictionary1 stores metadata of tables such as table name, primary keys, and foreign keys. Dictionary 2 contains the output from Transformation algorithm that describes how each table is migrated to graph database. Dictionary 3 maintains start node, end node and edge name for all edges that have been emitted from migration phase .SQL to cypher part uses dictionary 1, 2 and 3 in query mapping. Figure1 outlines the architecture of the proposed approach.

3.1 Part I: Transformation and Migration

3.1.1 schema metadata analyzer

In order to get efficient ETL (Extract, transform, load) from the RDB to the graph database each table data and properties should be accessed as minimum as possible. Thus metadata should be inspected before transformation and migration step.

The relationships between tables, recursive relationship, primary key, foreign and other attributes that are not part of primary key or foreign key should be identified. The aforementioned tasks depend on complexity of RDB not data volume of RDB itself. SchemaCrawler[21] is used to achieve those tasks. Therefore, the input to schema metadata analyzer is RDB and the output is relationships between all tables, primary key, foreign key and other attributes of table. The output from this step persisted in dictionary 1 as illustrated in figure. 1.

3.1.2 **transformation algorithm**

This section presents a proposed algorithm to transform from RDB to graph model without semantically lose. The input is the schema metadata analyzer output (dictionary 1 in figure 1) while the output is nodes and edges types. The property graph consists of nodes and edges .The data in the RDB convert as nodes and edges in graph database. The proposed transformation algorithm proposes three types of nodes: (i) first nodes (ii) second nodes (iii) intermediate nodes. Also, it proposes join table edges type which contains join table that represents many to many relationships between two tables. It offers Recursive relationship edges type which contains tables that have recursive relationships.

By using dictionary 1, each table has properties such as primary key attributes, imported foreign keys, exported foreign keys and other fields. For each table in all tables in RDB, we check the following constrains:-

i. If number of imported foreign keys =2 , references two others tables and primary key is not referenced by any others tables is the representation of M: N relationship between entities (join table), table will be converted as join table edges type. So, all tuples of join table will be migrated as edges.
ii. If number of imported foreign keys of table >= 3, table will be converted as intermediate node type.
iii. If number of imported foreign keys =0 that implies the table does not references any other tables, table will be converted to first node type.
iv. If table has imported foreign keys or exported foreign keys, the table will be converted as second node type.
v. If table has recursive relationship, the recursive relationship will be converted to recursive relationship edges type. All tuples in table that have recursive relationship will be migrated as edges between nodes of the same label.

The output from the transformation algorithm is stored in dictionary 2 as in figure1.

If the relationship between two entities is one to many, then the starting node in graph model will be from many side and the end node in one side. Each node type with the list of table's names that belongs to this type is saved in dictionary 2. The pseudocode for transformation is depicted in Algorithm1.

Algorithm 1: transform RDB to graph database
Input: RDB and the output from schema analyzer
Output: list of tables to migrate as first nodes or second nodes or intermediate nodes or recursive edges or join table edges.

```
AllTables [ ] ← list of all tables in database
JoinTablesEdges [ ]← ∅
FirstNodes [ ] ← ∅
SecondNodes [ ] ← ∅
IntermediateNodes [ ] ← ∅
RecursiveEdges [ ] ← ∅
For each table t ∈ allTables {
 If (t.getImportedKeys = = 2 and
t.getExportedKeys = = ∅ ) {
  JoinTablesEdges [ ]← t
 }
 Else if (t.getImportedKeys >=3) {
  IntermediateNodes [ ]← t
 }
 Else if (t.getImportedKeys≠ ∅) {
   FirstNodes [ ]← t
  }
  Else if (t.getImportedKeys≠ ∅) or
t.getExportedKeys≠ ∅)) {
   SecondNodes [ ]← t
 }
 RelatedTables [ ]← t.getRelatedTables
(TableRelationship.parent)
 For each Table rt ∈ RelatedTables{
  If (rt.getName () == t.getName ()) {
   RecursiveEdges [ ]← t
   }}
  End for each
 }
End for each
```

Journal of Theoretical and Applied Information Technology
30th June 2021. Vol.99. No 12
© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

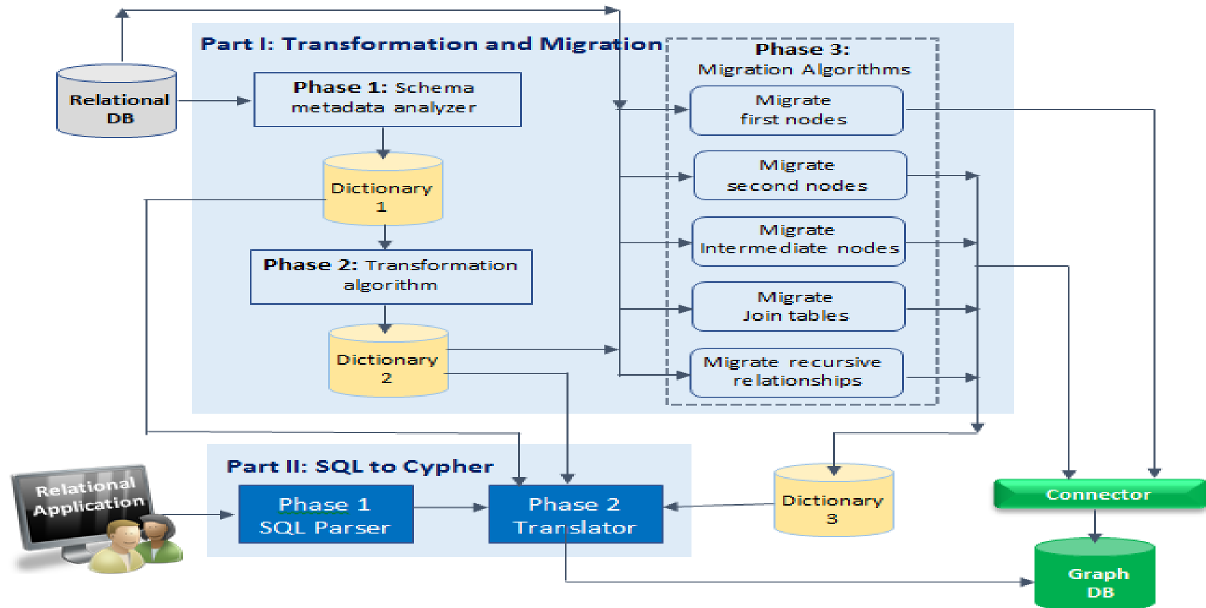


Figure 1.: The proposed approach to migrate RDB to graph database (Synthesis by the authors)

3.1.3 migration algorithm

This section introduces the proposed algorithm to migrate data from RDB to graph database has been proposed. Apache spark is used in the migration step to reduce time for migrating large-scale data because apache spark work in distributed environment. The input to migration algorithm is the output of transformation algorithm (dictionary 2 in figure1). Firstly, spark cluster reads dictionary 2 and creates hash map with five keys as follows: first nodes, second node, intermediate nodes, join tables and recursive. Each key contains the list of all tables belongs to this key. After that spark add this hash map as a broadcast variable to be available to all spark nodes. The main task in migration algorithm is the migration order. The migration order will go through the following steps:

- First nodes type refers to the tables which do not reference any other tables.
- Second nodes type.
- Intermediate nodes type
- JoinTablesEdges list that contains join tables.
- Recursive Edges list that have tables recursive relationships.

Migrating one to many, one to one and many to many relationships are migrated as edges where each edge has start node, end node and edge name. All metadata about edges is persisted in dictionary 3 as shown in figure 1.The pseudocode for the migration steps is shown in Algorithm 2. An example for RDB is illustrated in figure. 2.

Migrating one to many, one to one and many to many relationships are migrated as edges where each edge has start node, end node and edge name. All metadata about edges is persisted in dictionary 3 as shown in figure 1.The pseudocode for the migration steps is shown in Algorithm 2. An example for RDB is illustrated in figure 2.

Algorithm 2: Migrating RDB to graph model
Input: RDB r and JoinTablesEdges [], First nodes [], Second nodes [], Intermediate nodes [] and RecursiveEdges [].
Output: Graph Database g
Migrated ← ∅
For each Table t ∈ First nodes[]
  MigrateFirstNodes (First nodes [],r, g)
End for each
Migrated ← First nodes []
For each Table t ⊂ second nodes
  MigrateSecondNodes (Second nodes [], r, g)
End for each
Migrated←Second nodes []
For each Table t ∈ Intermediate nodes
  MigrateIntermediateNodes (Intermediate nodes [], r, g)
End for each
Migrated←IntermediateNodes []
For each Table t ∈ JoinTablesEdges
  MigrateJoinTablesEdges (JoinTablesEdges [],r ,g),
End for each
For each Table t ∈ Self nodes

© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

MigrateRecursiveRelationship (RecursiveEdges [] , r, g)
End for each

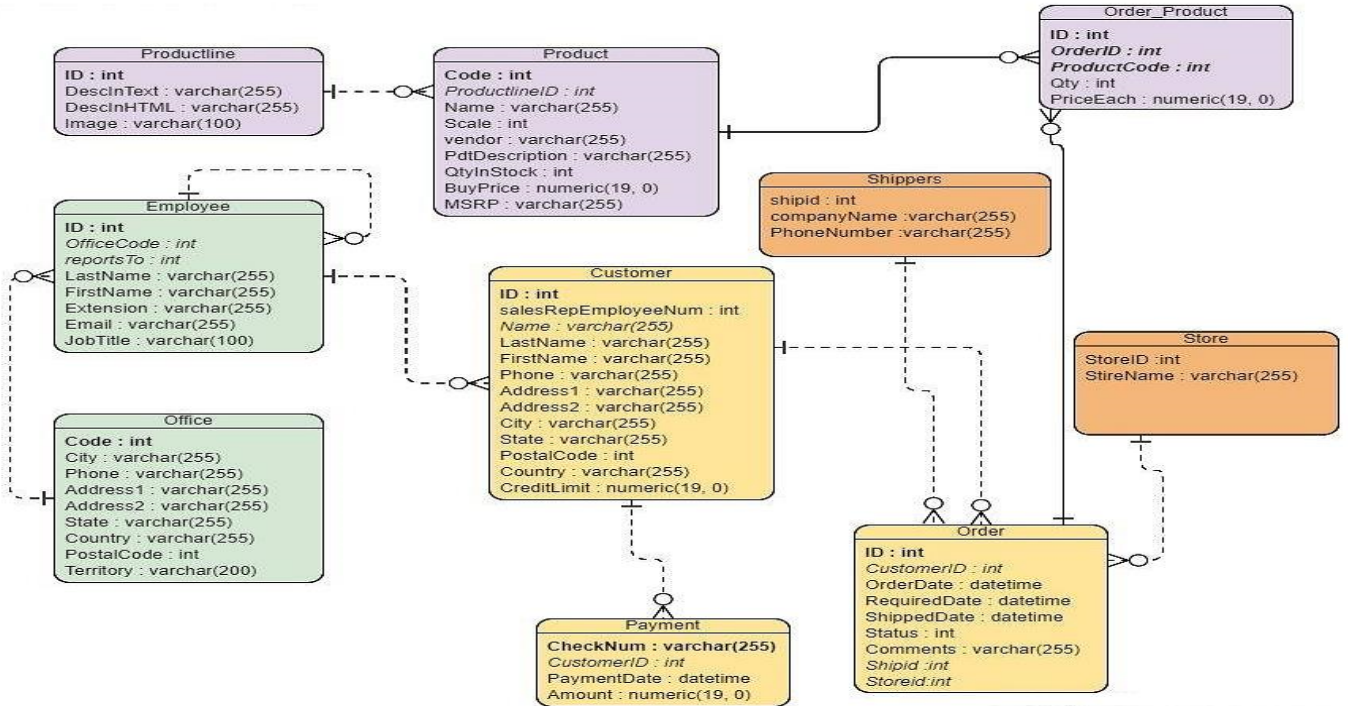


Figure 2: relational database example

3.1.3.1 migrating first nodes

Spark extracts data that belongs to the first nodes type from hash map in the broadcast variable. Therefore, all tables of first nodes type became available to all nodes in spark cluster. Each table in first nodes is migrated by creating nodes where label is table name and each tuple field with value as properties in node. Each node in cluster read first nodes from broadcast variable .Spark job takes table's names and create node for each tuple in each table as depicted in algorithm 2.1. In RDB described in figure 2, tables that will be migrated as first nodes are office, product lines and shipper. The input to this step is first nodes, RDB name and graph database name. The connector receives all tuples in each table of the first node type as spark dataset and create node for each tuple of this dataset. After the end of this step, office and productline tables are stored in migrated list.

Algorithm 2.1: Migrating first nodes to graph database: **MigrateFirstNodes (First nodes [],r, g)**
Input: RDB r, graph database g, First nodes []
Output: tables in first nodes list have been migrated to graph database g

For each Table t ∈ First nodes []
 For each tuple {tuple1} in t
  Node ← createNode ()
  Node.label ← t.getName ()
  Node.properties ←
tuple1.getCoulmnsNamesAndValues ()
 End for each
End for each

3.1.3.2 Migrating second nodes

Spark cluster fetches all tables of second nodes type from the map which exists in the broadcast variable. Second nodes list contains tables that will be migrated as second nodes in this step. The input to this step is second nodes, RDB name and graph database name. The migrated list contains office, shipper and productline, while second nodes list contains Product, Employee, Customer and Payment. Tables in second nodes list are reference tables (Child tables) for tables that already migrated in the migrated list (parent table) will be

© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

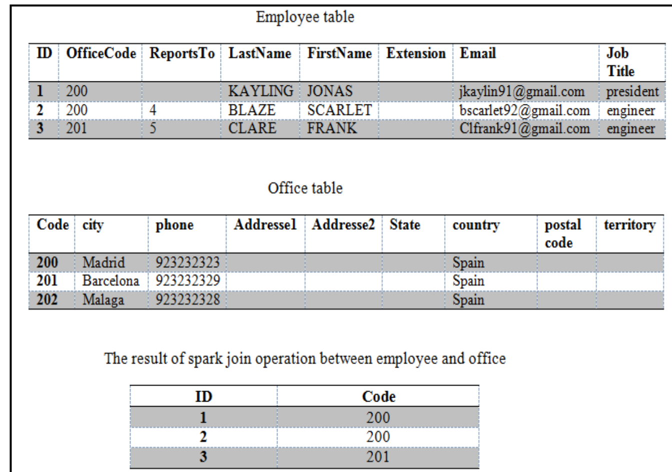
converts first. Migrating second nodes has two parts:
(i) Migrating second nodes that reference first nodes
(ii) Migrating remaining tables in second nodes list

Child tables of the migrated list that contain (office and productline) are Product and Employee tables. The intersection between second nodes list and child tables is Product and Employee. Product table is the child table of productline while Employee is the child table of Office. For each table in the result of the intersection, the following are taken:

i. Each tuple is migrated as node where table name become the node label and properties same as columns of tuple.
ii. Get the parents of table from migrated list and create edges between table and parents.
iii. Add table to migrated list.

The nodes will be created first then the edges. In figure. 3, all tuples of Employee are migrated as nodes where node label is Employee and all fields such as LastName are added as properties to node. For example ProductLine is foreign key for table productline table and officeCode is foreign key for table office. So, edges will be created between product and productline nodes and between Employee and Office nodes. In graph database, two nodes representing the tuples of two tables are connected if and only if the two tuples are joinable .spark Sql join job is used to create edges by joining two tables such as Office and Employee. The result of join process for Office and Employee is the joinable tuples and create edges between Office and Employee nodes where start node from Employee and end node is Office node. For example in figure. 3, an edge is created between employee node with id =1 and office node with code =200 another edge between employee node with id =2 and office node with code =200 and another edge between employee node with id =3 and office node with code =201.

The representation of 1:1 and 1:N and N:1 is very similar in migration to graph database. For example, creating edge between employee and office node is depicted in figure 4. Each tuple in office table is mapped to zero or more joinable tuples in employee table. The direction of the edge is from child to parent such as from Employee to office. Employee and Product are added to migrated list. The start node, edge name and end node is stored in dictionary. Apache spark join job is used to get joinable tuples and create edges that represent two tuples.



Employee table

ID	OfficeCode	ReportsTo	LastName	FirstName	Extension	Email	Job Title
1	200		KAYLING	JONAS		jkaylin91@gmail.com	president
2	200	4	BLAZE	SCARLET		bscarlet92@gmail.com	engineer
3	201	5	CLARE	FRANK		Clfrank91@gmail.com	engineer

Office table

Code	city	phone	Addresse1	Addresse2	State	country	postal code	territory
200	Madrid	923232323				Spain		
201	Barcelona	923232329				Spain		
202	Malaga	923232328				Spain		

The result of spark join operation between employee and office

ID	Code
1	200
2	200
3	201

Figure 3: Creating edges using spark join

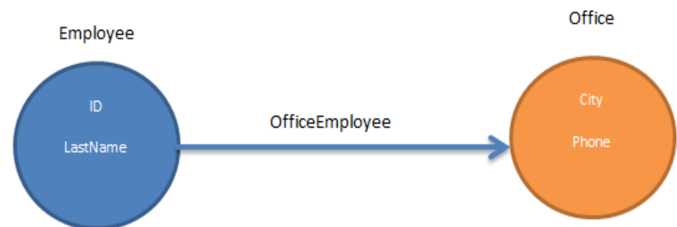


Figure .4: Migrating first part in second nodes type

To migrate the remaining second nodes, we take the difference between the second nodes and migrated list. Migrated list has office, productline, Employee and product while second nodes list has Product, Employee, Customer and Payment. The difference between two lists is Customer and Payment. For each table in the result of the difference, the following are taken:

1. Each tuple is migrated as node where table name become node label and each tuple field with value as properties in the node.
2. Table is added to migrated list.
3. We check if there is child-to-parent relationship between table and its parents in migrated list. if there a relationship , two nodes that represents two tuples of two tables will be connected by edge when the two tuples are joinable.

For example, customer table contains foreign key salesRepEmployeeNumber that referencing Employee table so the start node for the edge that represent joinable tuples is the customer node and end node is employee node.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Payment table also contains foreign key (customer ID) to Customer table. The direction of the edge is also from child to parent. Start node from payment node and end node is customer node. The graph model after migrating Payment and customer is showed in figure. 5. The start node, edge name and end node is stored in dictionary 3.

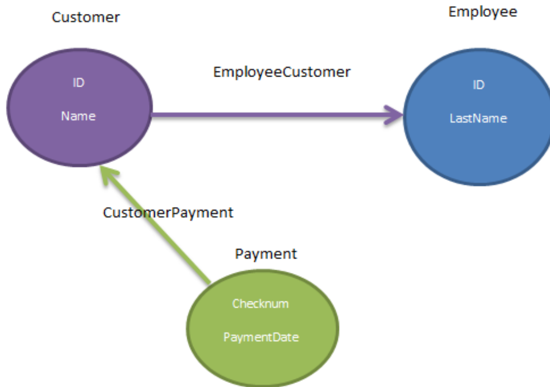


Figure.5: Migration of customer and payment (second part in second nodes type)

Algorithm 2.2: Migrating second nodes to graph database:

```
MigrateSecondNodes (Second nodes [], r, g)
Input: RDB r, graph database g, Second nodes []
Output: tables in second nodes list have been
migrated to graph database g
childNodes 1[]← getChildTables (migrated list [])
secondNodes1 []← childNodes 1 [] ∩ second
nodes []
For each Table t ∈ secondNodes1 []
  For each tuple {tuple1} in t
    Node ←createNode ()
    Node.label ← t.getName ()
    Node.properties ←
tuple1.getCoulmnsNamesAndValues()
  End for each
  Parents []←getParentTables(t)
  For each Table p ∈ Parents []
    If (p.pk == t.fk)
      FirstNode← getNode (p1)
      SecondNode← getNode (t)
      AddEdge (FirstNode, SecondNode)
    End if
  End for each
  Migrated []←t
End for each

secondNodes2 []←second nodes []-migrated list []
For each Table t ∈ secondNodes2 []
  For each tuple {tuple1} in t
    Node ←createNode ()
    Node.label ← t.getName ()
    Node.properties ←
tuple1.getCoulmnsNamesAndValues ()
  End for each
Migrated []←t
End for each
For each Table t1 ∈ secondNodes2 []
  Parents []←getParentTables(t1)
  For each Table p1 ∈ Parents []
    If (p1.pk == t1.fk)
      FirstNode← getNode (p1)
      SecondNode← getNode (t1)
      AddEdge (FirstNode, SecondNode)
    End if
  End for each
End for each
```

3.1.3.3 **migrating intermediate nodes**

Spark cluster reads all tables that will be migrated as Intermediate nodes from hash map. Consequently, all tables in intermediate nodes are available to all nodes of spark cluster. The input to this step is Intermediate nodes, RDB name and graph database name. Intermediate nodes list contain tables that will be migrated as intermediate nodes. For each table in the intermediate nodes list, we do the following:

i. Each tuple is migrated as node where table name become node label and properties of the node are the same as fields of the tuple.
ii. Table is added to migrated list.
iii. We get parents of the table first and for each parent, we create edges between two nodes that represent two joinable tuples from this table and this parent.

The migration of intermediate nodes also conducted using apache spark Sql job. We also use spark Sql join between table and each parent is used to get joinable tuples .In this example, intermediate nodes list contains order table. The parent's tables of order are shipper, Customer, and Store. Creating edges between nodes is achieved using Apache spark SQL as in migrating second nodes. The start node for the edge is from intermediate node (child node) to parent nodes. An edge is created between order and shipper where starting node for edge is from order node to shipper node. Another edge is created between order and Customer where starting node for edge is from order node to Customer node and last edge from order node to Store node. The start node, edge name and end node also stored in dictionary 3 .Figure. 6 shows the migration of order

Journal of Theoretical and Applied Information Technology
30th June 2021. Vol.99. No 12
© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

table as intermediate node. Algorithm 2.3 displays migration steps for intermediate nodes list.

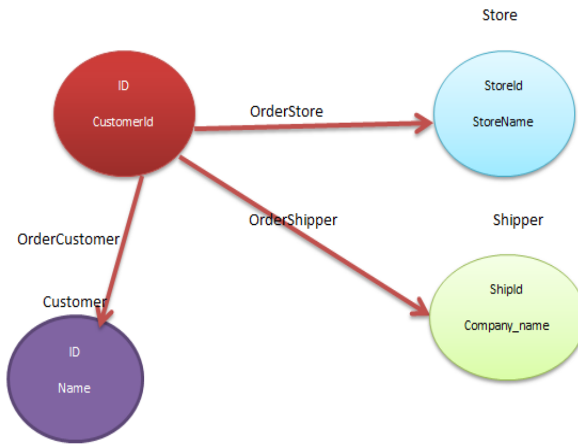


Figure 6. :The migration of order table as intermediate node

Algorithm 2.3: Migrating intermediate nodes to graph database:

Algorithm 2.3: Migrating intermediate nodes to graph database: **MigrateIntermediateNodes (intermediate nodes [], r, g)**

Input: RDB r, graph database g, Intermediate nodes [ ]

Output: Intermediate nodes list [ ] have been migrated to graph database g

```
For each Table t ∈ Intermediate nodes [ ]
 | For each tuple {tuple1} in t
 |  Node ←createNode ()
 |  Node.label ← t.getName ()
 |  Node.properties ←
 tuple1.getCoulmnsNamesAndValues ()
 | End for each
  Migrated [ ]←t
End for each
For each Table t1 ∈ Intermediate Nodes [ ]
 |Parents [ ]←getParentTables(t1)
 | For each Table p2 ∈ Parents [ ]
 |  | If (p2.pk == t1.fk)
 |  |  FirstNode← getNode (p1)
 |  |  SecondNode← getNode (t1)
 |  |  AddEdge (FirstNode, SecondNode)
 |  | End if
 | End for each
End for each
```

3.1.3.4 migrating join tables

In this step, join table list are extracted form hash map in the broadcast variable. No nodes are created in this step because all nodes were created in previous steps. The input to this step is Join table list, RDB name and graph database name. JoinTablesEdges list contain join tables that will be migrated as edges between two nodes. For each table in the Edges list, the following actions are done :

i. Two imported foreign keys are extracted from table where each foreign key references one table in two tables that are connected via join table.
ii. Get two nodes that represent two tables that connected by join table.
iii. Get attributes on the relationship by take difference between all column in table and two imported foreign key columns.
iv. An edge is created between nodes for each tuple in table.

The migration of join tables is conducted using Apache spark job. JoinTablesEdges list contains of Order_Product table. The Order_Product is join table (M: N relation) between order table and product. OrderID is foreign key in Order_Product for order table while ProductCode is foreign key in Order_Product for product table. The attributes qty and priceEach are relation attributes .Each tuple in Order_Product is migrated as edge between order node and product node. The attributes qty and priceEach are migrated as properties on the edge. Migration of Order_Product is depicted in figure 7. All the steps in migrating join tables are depicted in algorithm 2.4. All tasks in this step are conducted using spark Sql join job. The start node, edge name and end node for each join table stored in dictionary 3 in figure. 1.

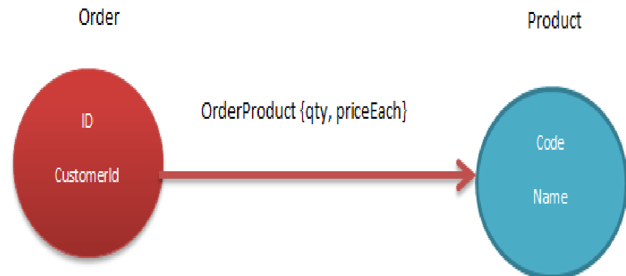


Figure.7: Migration of Order_Product table

Algorithm 2.4: Migrating join tables to graph database: **MigrateJoinTablesEdges** (JoinTablesEdges **[], r, g)**

Input: RDB r, graph database g, Join Tables

Output: join tables in JoinTablesEdges [] have been migrated as edges to graph database g
For each Table t ⊂ JoinTablesEdges []
FKS ←t.getImportedFkslist ()
Fk 1← FKS[0]
Fk 2← FKS[1]
Table1 ←t.getReferencedTable (Fk 1)
Table2 ←t. getReferencedTable (Fk 2)
Node 1←getNode (Table1)
Node 2←getNode (Table2)
AllColmuns [] ← t.getAllCoulmns ()
AttrributesOnRelationship [] ← allColmuns [] -FKS []
For each tuple tuple1 ∈ t
AddEdge (Node 1, Node 2, t.getAttributeOnRelationship ())
End for each
End for each

3.1.3.5 migrating recursive relationship

All tables that have recursive relationships are extracted from hash map. RecursiveEdges list contains tables that have Recursive relationship. The input to this step is RDB name, tables that have Recursive relationships and graph database name. The nodes that represent those tables have been already migrated in previous steps. The objective of this step is to create edges between nodes that represent the tables which have recursive relationship. Apache Spark Sql inner join operation is used to create edges between nodes. RecursiveEdges list in our example contains employee table. Employee table has ID as primary key and reportsTo as foreign key for employee table. The inner join operation is between employee table and itself and the join condition is employee.reportsTo column equal employee. ID column. For each table in RecursiveEdges list, we do the following:

1. Primary key and foreign key of table are extracted and inner join operation between table and itself is created. The output is spark dataset contains foreign key and primary key as columns.
2. An edge is created for each tuple in the result of spark dataset where the name of the edge is the name of foreign key.
3. The starting node of the edge that represents tuple is from employee node which has foreign key as attribute to the employee node which has primary key equals foreign key of starting node.

The migration of recursive relationship of employee table is depicted in figure. 8. The metadata for recursive edges such as edge name, start node and end node is stored on dictionary 3 in figure. 1.

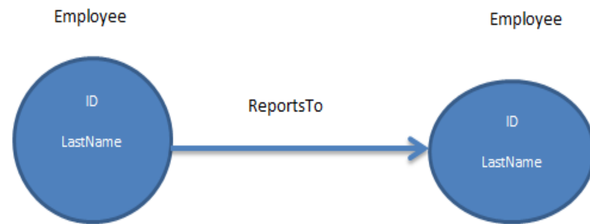


Figure .8: migrating of recursive relationship for employee table.

Algorithm 2.5: Migrating Recursive relationship to graph database: **MigrateRecursiveRelationship (**RecursiveEdges **[], r, g)**
Input: RDB r, graph database g, tables that have recursive relationships []
Output: Edges between table that have recursive relationship have been migrated to graph database g
For each Table t ∈ RecursiveEdges []
For each tuple {tuple1} in t
Pk ←Tuple1.getPk()
Fk ←Tuple1.getFk()
Node1 ←getNode(pk)
Node2 ←getNode(fk)
addEdge **(** Node1,Node 2)
End for each
End for each

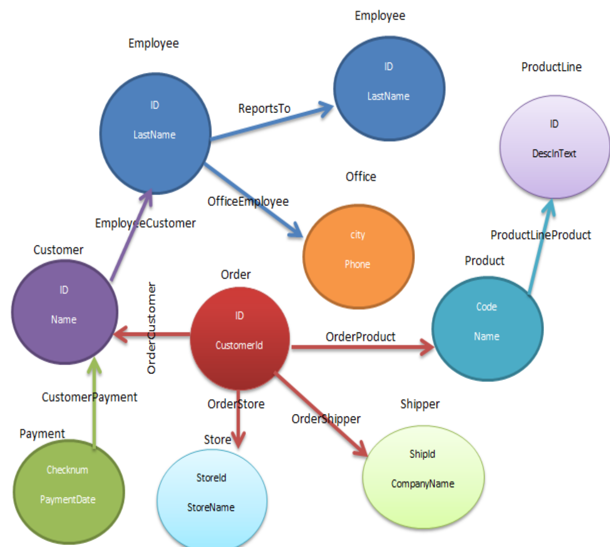


Figure 9: graph model for RDB of figure 2

Figure 9 represents the graph database after transformation and migration of all tables in RDB in the mentioned example in figure 2

3.2 Part II :SQL to Cypher

This part of the approach consists of Sql parser and translator from SQL to cypher language. Jsqlparser [22] used to parse SQL query. It uses the data of dictionary1, dictionary 2 and dictionary 3 .All tables foreign key, primary and remaining attributes key are in dictionary1. All tables that have been migrated as first nodes, second nodes, intermediate nodes, JoinTablesEdges and RecursiveEdges are in dictionary 2. All metadata about edges are in dictionary 3. The SQL to cypher module receives insert, update, or delete or select query and convert it to cypher with the help of three dictionaries. It supports subset of Data modeling language (DML) operation.

3.2.1 Insert

This section explains the translation of SQL INSERT statements to CREATE Nodes in cypher language. Jsqlparser is used to extract table name and columns from INSERT statement. The translation of insert statements is very similar to migration algorithm. There exist two formats of INSERT statements as follows:

i. INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...).
ii. INSERT INTO table_name VALUES (value1, value2, value3 ,...).

In the first format the columns name is written after table name but in the second format the columns are not provided. in this case by using all columns are detected from dictionary 1. Find out which type of node this table belongs to by reading the dictionary 2. The layer automatically generate Cypher query that is equivalent to insert SQL statement.

If the table in the Insert statement belongs to first nodes, the insert statement is mapped to node where node label is the name of table and properties of the node are the same as columns of the insert statement. For example the Sql insert statement " INSERT INTO ProductLine values (1000, ' text data',' html data','image1') "is translated to cypher" CREATE (n: productline {ID:1000, Descintext: text data' ,DescinHTML: 'html data',image:image1}) ".

If table belongs to second nodes, the insert statement is mapped to node and creates edge between this node and the nodes that represent parents of this table. The edge start from the node that represents the table in insert statement , ending at the node that represent parent table , and the edge name is the name of parent table concatenated with table name in the insert statement. For example the insert statement" **insert into product values (1, 1000,'pc',** 2,200,340,'hp','any','b') " is translated to "CREATE(n:product{code:1,productlineid:1000,name:'pc',scale:2,qtyInStock:200,buyPrice:340,vendor:'hp',MSRB:'any',pdtdescription:'b'}) MATCH (a:productline),(b:product) WHERE a.ID = b.productlineid AND b.code = 1 AND b.productlineid = **1000 AND b.name = 'pc' AND b.scale = 2 AND b.qtyInStock = 200 AND b.buyPrice = 340 AND b.vendor = 'hp' AND b.MSRB = 'any' AND b.pdtdescription = 'b' CREATE (a)-[r:productlineproduct]->(b) RETURN type(r)** ".

If table belongs to Intermediate nodes, the insert statement is mapped to node and creates edge between this node and the nodes that represent parents of this table. The edge start from the node that represents the table in insert statement, ending node is the node that represent parent table , and the edge name is the table name in the insert statement concatenated with name of parent table. For example the insert statement " INSERT INTO order VALUES (10,1,'20-10-2018','21-10-2018','22-10-2018',1,'deliverig date',1,1) " is translated to cypher :

" CREATE (n:order{ID:10,customerid:1,OrderDate:'20-10-2018',RequredDate:'21-10-2018',ShippedDate:'22-10-2018',Status:1,comments:'deliverig date',shipid:1,StoreID:1})
MATCH (a:customer),(b:order) WHERE a.ID= b.customerid AND b.ID = 10 AND b.customerid = 1 AND b.OrderDate = '20-10-2018' AND b.RequredDate = '21-10-2018' AND b.ShippedDate = '22-10-2018' AND b.Status = 1 AND b.comments = 'deliverig date' AND b.shipid = 1 AND b.StoreID = 1 CREATE (b)-[r:ordercustomer]->(a) RETURN type(r)
MATCH (a:shipper),(b:order) WHERE a.shipid = b.shipid AND b.ID = 10 AND b.customerid = 1 AND b.OrderDate = '20-10-2018' AND b.RequredDate = '21-10-2018' AND b.ShippedDate = '22-10-2018' AND b.Status = 1 AND b.comments = 'deliverig date' AND b.shipid = 1 AND b.StoreID = 1 CREATE (b)-[r:orderhipper]->(a) RETURN type(r)
MATCH (a:store),(b:order) WHERE a.StoreID = b.StoreID AND b.ID = 10 AND b.customerid = 1 AND b.OrderDate = '20-10-2018' AND b.RequredDate = '21-10-2018' AND b.ShippedDate = '22-10-2018' AND b.Status = 1 AND b.comments

Journal of Theoretical and Applied Information Technology
30th June 2021. Vol.99. No 12
© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

= 'deliverig date' AND b.shipid = 1 AND b.StoreID = 1 CREATE (b)-[r:ordertore]->(a) RETURN type(r)"

If table belongs to JoinTablesEdges, the insert statement is mapped as edge between two nodes that represent two tables for M:N relation. The attributes on the relation are added as properties to the edge. The edge name is the concatenation between two tables that are connected via M: N relation. For example , the following insert statement " INSERT INTO Order_product (orderID,productCode,qty,priceEach) VALUES (2,2,2,267)" is translated to " MATCH (a:order),(b:product) WHERE a.orderid = 2 AND b. productCode = 2 CREATE (a)-[r:orderproduct{qty:2 , priceEach:2}]->(b)"

If table belongs to RecursiveEdges, the insert statement is mapped as edge where edge name is the foreign key that represents recursive relationship. The edge start from node that represents table which has foreign key as attribute and end at the node which has primary key equals foreign key of the starting node.

3.2.2 Update

This section is responsible for translating SQL UPDATE statements to Cypher SET. The format of UPDATE Statement is " UPDATE table_name SET column1 = value1, column2 = value2 ...WHERE condition; ".Table name and where part are extracted by using Jsqlparser. The layer automatically generate Cypher query that is equivalent to update SQL statement .For example the Sql update statement " UPDATE shipper SET companyname = 'Thinklarge', phone ='00220' WHERE shipid =1 " is translated to " MATCH (n:shipper { shipid : 1 }) SET n.companyname = 'Thinklarge', n.phone ='00220' return n ".

3.2.3 Delete

This section describes how to translate SQL DELETE statements to Cypher. Jsqlparser is used to extract table name and where condition. The format of DELETE statement is " DELETE FROM table_name WHERE condition;" .The format of delete in cypher is" MATCH (n { condition }) DETACH DELETE n " where n is the name of node. The layer automatically generate Cypher query that is equivalent to delete SQL statement for example the SQL delete statement " delete from Employee where Email='xavidata@yahoo.com' " is translated to " MATCH (Employee { Email: "xavidata@yahoo.com"}) DETACH DELETE Employee ".

3.2.4 Select

The translator also can translate SQL SELECT statement to Cypher. The Tables names, alias and columns in the query are extracted from SQL query using Jsqlparser. It automatically generates cypher query for the input SQL query by using dictionary1, dictionary 2 , dictionary 3 and Sql parser. The translator translates SQL queries that contains join to cypher.

The translator supports conversion of WHERE and GROUP BY. It also supports translation of LIKE operator to "STARTS WITH", "END WITH" and "CONATINS" in cypher. For example if the input SQL select query is" select * from customer c where LastName like '%w' and ID=1" is translated to " MATCH (p:customer) WHERE p.LastName STARTS WITH 'w' and p.ID=1 RETURN p. LastName ,p.FirstName,p.Name, p.phone , p.ID , p.salesRepEmployee , p. Address1 , p.Address2,p.city , p.city , p.state , p.PostalCode , p.Country ,p.CreditLimit , p.ID".The input Sql statement contains "*" , therefore the translator uses the dictionary 1 to get all columns for query in SQL.

The translator also handles the query contains tables that have been migrated as edges of JoinTablesEdges type (Dictionary 2) in graph database and joining multiple tables .For example if the input SQL select query is" SELECT c.Name , c.ID FROM customer AS c JOIN order AS o ON (c.ID = o.CustomerID) JOIN Order_Product AS od ON (o. ID = od.OrderID) JOIN product AS p ON (od.ProductCode = p.code) WHERE p.name = 'first product'" is translated to " match (c:customer)-[:orderCustomer]-(o:order),(o:order)-[:orderproduct]-(p:product)where p.name = 'Chocolade' return c. Name,c.ID". The input SQL query contains customer, order, Order_Product and product tables. Order_Product table has been migrated as edge. The translator removes Order_Product from table list in query. The translator construct pattern by getting edge name between Customer and order (: ordercustomer) then the edge between order and product (orderproduct) by using metadata of edges in dictionary 3. The join between Customer, order, Order_Product and product tables in SQL is translated as pattern in cypher "(c: customer)-[: ordercustomer]-(o: order),

Journal of Theoretical and Applied Information Technology
30th June 2021. Vol.99. No 12
© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

(o: order)-[: orderproduct]-(p: product)". The translator does not support subquery. The translation steps for select query are illustrated in algorithm 3.

```
Algorithm 3: Translation from SQL to Cypher
Input: SQL query q
Output: Cypher query cq
TabelsInQuery← Ø
Attributes← Ø
TabelsInQuery ←ExtractTables (q)
FilterConditions ←Ø
Patterns ← Ø
joinsConditionList ← Ø
/* relationshipMapping: contains start node, edge
name and end node that clarifies how the
relationship between two tables in relational model
is mapped to graph model (obtained from
dictionary 3)*/
MappingList←relationshipMapping ()
/* getAttributes: extract attributes after select
keyword */
Attributes ←getAttributes (q)
/* getAliasMapping: map that contains alias as key
and table name as value */
AliasMapping ← getAliasMapping ()
/* getJoinsCondition(q): Extract all join conditions
in SQL query*/
joinsConditionList ←getJoinsCondition(q)
If (TabelsInQuery>1)
  {
  For each table t in query
    If (JoinTablesEdges . conatins (t)) then
      Patterns←ConstructPatternEdges
      (joinsConditionList, MappingList,
      JoinTablesEdges , AliasMapping)
      Break;
    Else
      Patterns← ConstructPattern
(joinsConditionList, MappingList, AliasMapping)
      Break;
  End for each
  /* getFilterConditions: extract filters such as AND
, Like from query*/
  FilterConditions ←getFilterConditions (q)
cq ← "match" + patterns + FilterConditions +
"return "+Attributes
End If
Else {
cq ← "match"+(table alias : table name)+
FilterConditions +"return "+Attributes
}
```

SQL query may contains tables that have been migrated as edges in the transformation and migration process. Therefore, the mapping considers it in the translation process from select to cypher. JoinTablesEdges are the relations (join tables) that have been migrated as edges between nodes. If query has any table from edges list, ConstructPatternEdges will be executed as depicted in algorithm 3.2. Otherwise, ConstructPattern will be executed as shown in algorithm 3.1

```
Algorithm 3.1: ConstructPattern
Input: joins Conditions List and table alias map
Output: patterns that represent joins in the SQL
query.
Patterns←Ø
For each condition c ∈ joinsConditionList
  tables[] ←c.split("=")
  /* extractNode : extract table from join condition
*/
  Node1←extractNode (tables [0])
  Node2←extractNode (tables [1])
   /* getRelationship: get relationship (edge) that
join Node1 and Node2 by using the provided
joins Conditions List , dictionary 3 and emits the
relationship as follows (start node) –[edge]-(end
node) */
  Patterns += getRelationship (joinsConditionList,
Node1, Node2)
End for each
```

```
Algorithm 3.2: ConstructPatternEdges
Input: JoinTablesEdges list, joins Conditions List
and table alias map
Output: patterns that represent joins in the SQL
query
Patterns←Ø
/* Edgesjoins contains elements that is part of
relationship that have been migrated as edges
between two nodes*/
Edgesjoins ←Ø
For each condition c ∈ joinsConditionList
  tables[] ←c.split("=")
  /* extractNode: extract table from join condition
*/
  Node1←extractNode (tables [0])
  Node2←extractNode (tables [1])
  If (JoinTablesEdges.conatins (Node1))
      Edgesjoins.Add (Node1: Node2)
  End if
  Else If (JoinTablesEdges.conatins (Node2) )
      Edgesjoins .Add (Node2: Node1)
  End if
```

Else
/* getRelationship: get relationship (edge) that join Node1 and Node2 by using the provided joins Conditions List , dictionary 3 and emits the relationship as follows (start node) –[edge]-(end node) */
Patterns += getRelationship (joinsConditionList, Node1, Node2)
End else
/* getRelationshipEdge: gets the relationship (edge) between two elements in Edgesjoins list that share the same prefix such as { "**Order_Product: order** "," **Order_Product: product**"} by using the provided joins Conditions List, dictionary 3 and emit relationship as follows (start node) – [edge]-(end node) */
Patterns += getRelationshipEdge (joinsConditionList, Edgesjoins)
End for each

4 EXPERIMENTS AND RESUALTS

The proposed approach implemented by java programming language and Apache spark 2.4.1.MySQL has been used as relational data source and Neo4j as graph database. The experiments were conducted on spark standalone cluster with two nodes. Each node has Intel core i7 and 8 GB of RAM. The Neo4j is hosted on a dedicated machine with 8GB of RAM and Intel core i7 (4 cores).

4.1 Database

Imdb is open source database. It is available in a form of plain text files. Only subset of database is used in the experiments. Table 1 describes the characteristics of the databases.

Table 1: database characteristics

Databases	Number of tables	Tuples	Relationships
IMDB[23]	7	5610922	6

4.2 Transformation and migration time

The migration of data from RDB to graph database depend on partitions and batch size. The relational data that will be migrated is saved in spark cluster as dataset. Partitions are the number of Partitions for the spark dataset while batch size defines for each partition the batch size sent to Neo4j. The level of parallelism in spark is dissimilar to the level of parallelism in neo4j. Partitioning in spark refers how the data will be partitioned according to the number of machines in spark cluster.

In neo4j, Partitioning is very dependent on the number of cores in the leader node of neo4j cluster because the writing in neo4j is scaling vertically. The input to spark-neo4j-connector[24] for creating edges is spark dataset, number of partitions, batch size and node operation. The reason for choosing spark-neo4j-connector rather than other import tools in migration is that the connector provides flexibility to shape data and easy to integrate data with other data connected to spark cluster. Also, the connector also supports the integration of neo4j data with spark machine learning. Spark can be used as preprocessing for neo4j import tools such as load csv. Load csv tool can create nodes up to 10 million nodes only.

To achieve high performance in writing data from spark dataset to neo4j database, partitions should be tuned , batch size and memory configuration in neo4j server parameters.it is very recommend to tune partitions to avoid transaction locks when creating nodes or creating edges between nodes. The transaction scope is stretch on a single partition. But in our case, write failures can be occurred in case of writing edges. So the dataset should be repartitioned before writing it in neo4j to avoid write failures. If the dataset is partitioned to only one partition prior sinking them to neo4j, only one core of neo4j server will process the dataset. To get the maximum throughput in the migration process, all available cores in neo4j server should be used .Therefore; the spark dataset should be properly partitioned before writing them in neo4j server. For example, assume the following two tables (Table 2 ,Table 3) order and shipper **.**

All the tuples in the two tables will be migrated to neo4j as nodes without problems in write process. Creating edges between nodes requires only ID of the two nodes. Edges are created by spark inner join job as in Table 4.

The result in table 4 is stored as spark dataset. If the dataset is partitioned to only one partition as in table 4, only one core of neo4j is used in writing edges. In this case, no write problems occur in creating edges because only one partition is writing. If the dataset is partitioned to two partitions as in table 5 , Table 6 and two partition (cores) is writing, the first partition lock the shipper node with shipid=1 and second partition cannot write edges because the shipper node with shipid=1 is

locked by the first partition. The two threads wait each other which may rejects writing with lock error.

Table 2 : order table

ID	customerID	OrderDate	RequiredDate	ShipDate	status	Comments	shipid	Storeid
1	2000	2018-12-08	2018-12-09	2018-12-10	0	Null	100	200
2	2010	2018-12-12	2018-12-13	2018-12-14	1	Null	100	201
3	2110	2018-12-14	2018-12-15	2018-12-16	2	Null	101	202
4	2200	2018-12-13	2018-12-14	2018-12-15	1	Null	100	200
5	2111	2018-12-14	2018-12-15	2018-12-16	0	Null	102	206
6	2123	2018-12-15	2018-12-16	2018-12-17	2	Null	101	203
7	2145	2018-12-16	2018-12-17	2018-12-20	1	Null	100	201
8	2501	2018-12-16	2018-12-17	2018-12-18	0	Null	103	204
9	2608	2018-12-18	2018-12-19	2018-12-20	0	Null	102	200

Table 3 : shipper table

Shipid	CompanyName	phoneNumber
100	Egy service	002019866557
101	Rapid service	002019882234
102	Easy Delivery	002010765448
103	Fast	002011298764

Table 4 : result of spark inner join between order and shipper in one partition.

Shipid	ID
100	1
100	2
100	4
100	7
101	3
101	6
102	5
102	9
103	8

Table 5 : first partition of spark dataset

Shipid	ID
100	7
101	**3**
101	6
102	**5**
102	9
103	8

To avoid writes lock while using all available cores in neo4j server the dataset should be partitioned properly before sinking in neo4j as depicted in figure 10. The spark dataset is repartition and saved in Apache parquet[25] files. Apache parquet is useful to get high performance data pipeline. Each partition contains non overlapping shipid. So the pipeline of data migration is changed as follows :(i)the data is repartitioned using spark and stored in parquet files (ii) read all the partitioned data as dataset and deliver it to spark neo4j connector as depicted in figure. 10.

Table 6 : second partition of spark dataset

Shipid	ID
100	1
100	2
100	4

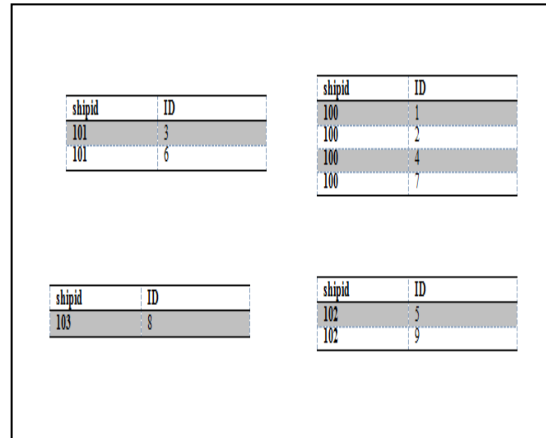


Figure 10. spark dataset after repartitioning

Other important parameters are the neo4j memory configurations heap memory and page cache size. Normalize loading is used in the migration which uses different dataset for each node and edges. Indexes are used for each node to get high performance in migration.

The proposed approach transforms and migrate data from IMDB database and sink data in neo4j database. It creates 1,870,915 nodes and 4,354,581 edges in neo4j database. When the number of partitions is 1 the migration time is 12 minutes. When the number of partitions is 4 the migration time is 8 minutes as depicted in figure. 11. The neo4j database assigned with only 5 GB pf RAM.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

To get high throughput, the best machine is selected and increase size of heap memory and page cache. This allows increasing in batch size in writing to neo4j, consequently, decrease migration time. The partitioning mentioned above is used in mapping of one to many relationships. Creating edges that represent join table for many to many need special optimized partitioning strategy in future works.

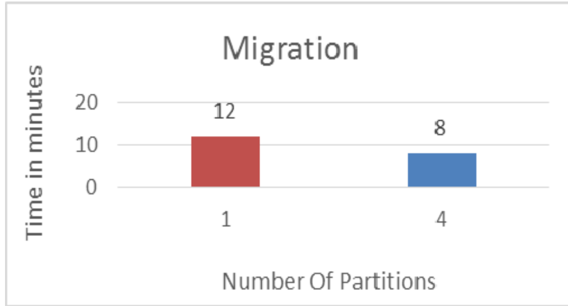


Figure11: Migration time

Table 7 : query translation from SQL to Cypher

statement #	SQL statement	Cypher statement
St1	"select a.first_name ,d.genre from actors a join roles r on r.actor_id=a.id join movies m on r.movie_id=m.id join movies_g enres d on d.movie_id=m.id where a.id=320"	"match (d:movies_genres)-[:moviesmovies_genres]-(m:movies) ,(a:actors)-[:actorsmovies]-(m:movies) where a.id = 320 return a.first_name,d.genre
St 2	"select a.first_name,di.first_name from actors a join roles r on r.actor_id=a.id join movies m on r.movie_id=m.id join movies_directors dvd on m. id=dvd.movie_id join directors di on di.id=dvd.director_id where a.id=410"	match (a:actors)-[:actorsmovies]-(m:movies) ,(m:movies)-[:directorsmovies]-(di:directors) where a.id = 410 return a.first_name,di.first_name"
St 3	"select * from actors "	"MATCH (p:actors) RETURN p.id,p.first_name,p.last_name,p.gender"
St 4	"select * from actors a where a.first_name like '%t'"	" MATCH (p:actors) WHERE p.first_name ENDS WITH 't' RETURN p.id,p.first_name,p.last_name,p.gender"
St 5	" select d.first_name,dg.genre from directors d join directors_genres dg on d.id=dg.director_id where d.id=1000 "	" match (d:directors)-[:directorsdirectors_genres]-(dg:directors_genres) where d.id = 1000 return d.first_name,dg.genre "
St 6	" insert into movies(id,name,`year`) VALUES(412321,'elresala',2008) "	" CREATE (n:movies{id:412321,name:'elresala',`year`:2008}) "
St 7	" insert into movies_genres(movie_id,genre) VALUES(412321,'islamic') "	" CREATE (n:movies_genres{movie_id:412321,genre:'islamic'}) ; MATCH (a:movies),(b:movies_genres) WHERE a.id = b.movie_id AND b.movie_id = 412321 AND b.genre = 'islamic' CREATE (a)-[r:moviesmovies_genres]->(b) RETURN type(r) "
St 8	update movies set year= 2009 where id=412321;	"MATCH (n:movies { id : 412321 }) SET n.year= 2009 return n"
St 9	" delete from movies where mid = 412321"	"MATCH (movies { id : 412321}) DETACH DELETE movies "

4.3 Query Translation and Processing

Queries in Table 7 display the translation form SQL statements to cypher statements using SQL to cypher part. The translator automatically maps insert, update, delete and select statements to cypher language. Select statements are from St 1 to St 5 .Insert statements are St 6 and St 7. Update statement is St 8 and Delete statement is St 9 .The queries were conducted in IMDB database.

Queries in Table 8 are used to ensure that no data loss occurred in the migration process from RDB to graph database. The queries were conducted in IMDB database. The results in table VIII ensure that the transformed data using the proposed layer is complete. The outcome from cypher query is the same as the outcome from SQL query in all statements from St1 to St 5.

As relationships treat as "the first class citizen" in graph databases, the queries that need join between tables in SQL queries such as St 1, St 2 and St 5 take more time than its equivalent in cypher query

as depicted in figure. 12. The relationships in graph database between nodes are actually persisted as edges in physical storage. In contrast, the relationships or joins between tables in RDB are calculated for every query. Therefore, traversing edges in graph database is faster than joining in RDB. The comparison between statements that have join is illustrated in figure 12.

Table 8: queries on relational and graph used in testing phase

Statement #	Relational outcome	Tables involved in the query	Graph outcome
St 1	20	4	20
St 2	1	5	1
St 3	817718	1	817718
St 4	21879	1	21879
St 5	156562	2	156562

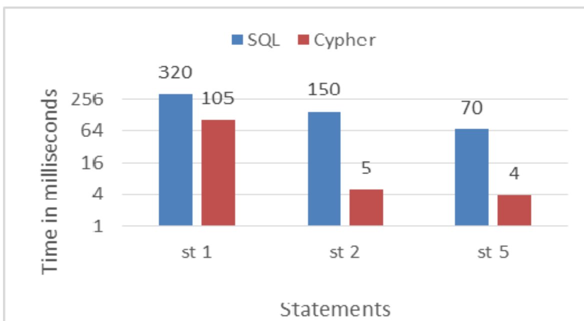


Figure. 12: Time execution difference between SQL and Cypher

4.4 Comparison with similar methods

The main focus of the experiments was to prove the effectiveness of the proposed by comparing it with existing methods. Neo4j ETL[26] is tool to migrate data from RDB to graph database. Neo4j ETL did not provide query translation from SQL to Cypher. It runs on only one machine while our approach run in distributed environment. Therefore, our proposed approach outperforms than neo4j ETL .Table 9 shows the comparison between our proposed approach and Neo4j ETL.

Table 9. :The comparison between proposed approach and Neo4j ETL

property	Proposed approach	Neo4j ETL(online direct import)
Translating SQL to Cypher	Support translation from insert, update ,delete and select statements to cypher	Not support translating from SQL statements to cypher
Elapsed time for migration	8 minutes	12 minutes
Mode	Distributed processing environment	Local machine

5 METHODOLOGY

The following research methodology has been followed:

- Reviewing the previous works related to migrating data from relational to graph database
- Proposing transformation algorithm
- Proposing migration algorithm using distributed processing engine
- Installing spark cluster and neo4j and download database
- Conducting experiments and compare results between the proposed approach and Neo4j ETL

6 CONCLUSION AND FUTURE WORK

NoSQL databases have been proposed to solve big data challenges. NoSQL have four different categories key-value, column, document and graph. Each category has different data model for managing data. NoSQL does not support SQL and there exists a huge base of users familiar with SQL. Therefore, several approaches have been proposed to preserve the benefits of SQL in NoSQL databases. The approaches map the relational data to key-value, column and document database. No approach maps the relational data to graph model. The previous researches migrating data from RDB to graph database have drawbacks such as the experiments did not conduct on distributed environments and query mapping from SQL to graph language. Some approaches also have semantic loss.

This paper presents an approach that transforms and migrates relational data to graph database. It converts RDB to graph database without semantic loss which is very important for further analysis such as graph mining. It retains the benefits of SQL in the context of graph database. The approach also supports the query mapping from SQL to cypher. The architecture of the approach builds on top of Apache spark and conducts the experiments in a distributed environment. It has two parts (i) transformation and migration part and (ii) SQL to cypher part. Transformation and migration consists of three phases schema metadata analyzer, transformation algorithm and migration algorithm. SQL to cypher part have SQL parser and translator .The layer allows enterprises to migrate relational data to graph data without semantically lose or data lose with a lower learning curve. The proposed layer considered being the base for graph mining algorithms.

The experiments ensure that the proposed approach migrates data from relational data to graph without semantic loss or data loss. The experiments also assure that traversing data using graph database is faster than RDB .The research proves the viability of using distributed processing in migration process. For the immediate future, we plan to enhance SQL by adding supports for subquery. We also plan to propose a strategy to repartition data in a join table for many to many relationship.

Applying graph mining algorithms after migrating data from relational to graph database may display connections among data that were formerly not shown. The study ensures the importance of using big data processing engine such as spark in the data migration process.

REFRENCES:

[1] P. Zikopoulos and C. Eaton, Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data. McGraw-Hill Osborne Media, 2011.

[2] P. J. Sadalage, M. Fowler, and U. S. River, NoSQL Distilled : a brief guide to the emerging world of polyglot persistence, 1 st editi. Pearson Education, 2013.

[3] G. A. Schreiner, D. Duarte, and R. dos S. Mello, “When relational-based applications go to NoSQL databases: A survey,” Information (Switzerland), vol. 10, no. 7, pp. 1–22, 2019.

[4] A. T. de Oliveira, A. D. de Souza, E. M. Moreira, and E. Seraphim, “Mapping and Conversion between Relational and Graph Databases Models: A Systematic Literature Review,” in 17th International Conference on Information Technology--New Generations (ITNG 2020), 2020, pp. 539–543.

[5] M. N. Mami, D. Graux, H. Thakkar, S. Scerri, S. Auer, and J. Lehmann, “The query translation landscape: A survey,” arXiv, no. 1. 2019.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark : Cluster Computing with Working Sets,” HotCloud’10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, p. 10, 2010.

[7] W. Hu and Y. Qu, “Discovering simple mappings between relational database schemas and ontologies,” in The Semantic Web, Springer, 2007, pp. 225–238.

[8] O. Orel, S. Zakošek, and M. Baranović, “Property Oriented Relational-To-Graph Database Conversion,” Automatika, vol. 57, no. 3, pp. 836–845, 2017.

[9] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla, and P. Sánchez, “Mortadelo: Automatic generation of NoSQL stores from platform-independent data models,” Future Generation Computer Systems, vol. 105, pp. 455–474, 2020.

[10] E. M. Kuszera, L. M. Peres, and M. D. Del Fabro, “Toward RDB to NoSQL: Transforming data with metamorfose framework,” Proceedings of the ACM Symposium on Applied Computing, vol. Part F1477, pp. 456–463, 2019.

[11] C. Li and J. Gu, “An integration approach of hybrid databases based on SQL in cloud computing environment,” Software: Practice and Experience, vol. 49, no. 3, pp. 401–422, 2019.

[12] S. Ramzan, I. S. Bajwa, B. Ramzan, and W. Anwar, “Intelligent Data Engineering for Migration to NoSQL Based Secure Environments,” IEEE Access, vol. 7, pp. 69042–69057, 2019.

[13] D. E. M. Arnaut, R. Schroeder, and C. S. Hara, “Phoenix: A relational storage component for the cloud,” in 2011 IEEE 4th International Conference on Cloud Computing, 2011, pp. 684–691.

[14] D. Egger, “SQL in the Cloud,” ETH, Swiss Federal Institute of Technology, Department of Computer Science …, 2009.

[15] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira, “An effective scalable SQL engine for NoSQL databases,” in IFIP International

© 2021 Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Conference on Distributed Applications and Interoperable Systems, 2013, pp. 155–168.

[16] R. De Virgilio, A. Maccioni, and R. Torlone, "Converting relational to graph databases," 1st International Workshop on Graph Data Management Experiences and Systems, GRADES 2013 - Co-located with SIGMOD/PODS 2013, vol. 1, no. i, 2013.

[17] Y. A. Megid, N. El-Tazi, and A. Fahmy, "Using functional dependencies in conversion of relational databases to graph databases," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11030 LNCS, no. January, pp. 350–357, 2018.

[18] D. W. Wardani and J. Kiing, "Semantic mapping relational to graph model," in Proceeding - 2014 International Conference on Computer, Control, Informatics and Its Applications: "New Challenges and Opportunities in Big Data", IC3INA 2014, 2014, pp. 160–165.

[19] O. Alotaibi and E. Pardede, "Transformation of schema from relational database (RDB) to NoSQL databases," Data, vol. 4, no. 4, pp. 1–11, 2019.

[20] M. Gupta and R. Rani Aggarwal, "Transforming relational database to graph database using Neo4j," in Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications, Bangalore, India, 2014, pp. 322–331.

[21] "schemacrawler." [Online]. Available: https://www.schemacrawler.com/. [Accessed: 24-Nov-2020].

[22] "JSqlParser." [Online]. Available: https://github.com/JSQLParser/JSqlParser. [Accessed: 08-Dec-2020].

[23] "IMDb. Internet Movie Database." [Online]. Available: www.imdb.com/interfaces. [Accessed: 20-Dec-2020].

[24] "neo4j spark connector." [Online]. Available: https://github.com/neo4j-contrib/neo4j-spark-connector/. [Accessed: 20-Sep-2020].

[25] "Apache Parquet." [Online]. Available: https://parquet.apache.org/. [Accessed: 14-Jan-2021].

[26] "Neo4j ETL." [Online]. Available: https://neo4j.com/developer/neo4j-etl/. [Accessed: 15-Oct-2020].