IJIRMET ISSN (Online): 2456-0448
International Journal Of Innovative Research In Management, Engineering And Technology
Vol. 1, Issue 5, June 2016

POLYGLOT PERSISTENCE FOR CASSANDRA AND NEO4J

[1] Komalpreet Kaur
[1]Thapar University, Patiala, India
[1]2277komal@gmail.com

Abstract: ***The given paper is related to polyglot persistence using Cassandra and Neo4j. The paper firstly discusses the benefits of Cassandra and Neo4j. It also provide us with advantages of Neo4j over Cassandra leading to a need of mapping of data from Cassandra to Neo4j. It then discusses the basic difference between structure of Cassandra and Neo4j and finally, proposes an algorithm for conversion of data from Cassandra to Neo4j..***

Keywords: NoSQL; polyglot persistence; Cassandra; Neo4j ; use-case; migration; column family

I. INTRODUCTION

From past 2-3 decades, technology disruption is happening very rapidly. From pre-stage flat-file system introduced in 70s, to relational and object-relational systems in 90s and then NoSQL databases from 2005 onwards, there have been a lot of changes in how to store huge amount of data. Now-a- days there are a lot many different types of NoSQL databases each having capability of storing different types of data. But for large applications, data is not homogeneous, and in that case its difficult to decide which database to choose. And in such cases, using a single database engine for all of the requirements like data storage or retrieval usually provide non-per formant solutions. Hence, we need a rigid solution that is provided by the concept of polyglot persistence. Talking about its meaning, Poly-glot means knowing or using different languages and persistence means prolonged use of something. Technically, Polyglot Persistence is a fancy term to mean that when storing data, it is best to use multiple data storage technologies, that is chosen on the basis of the way data is being used by individual applications or components of a single application.[2] Different kinds of data are best dealt with different data stores. In short, it means picking the right tool for the right use case.
Polyglot persistence does not only mean using NoSQL databases in combination, but it provides facility of using SQL and NoSQL databases together. Figure 1 shows different type of databases that are best suitable for different applications.
This paper basically talks about the combination of two NoSQL databases namely, Neo4j and Cassandra, where Neo4j is graph database and Cassandra is column database. In Neo4j, data is stored in the form of nodes and relations whereas in Cassandra, data is stored in columns. We are taking the scenario where we have the whole data stored in Cassandra and we need to synchronize and map it to Neo4j. But why we need to do this? As we know that each database has its own advantages and disadvantages, there are situations where Neo4j will be a better choice than Cassandra. Hence, with this

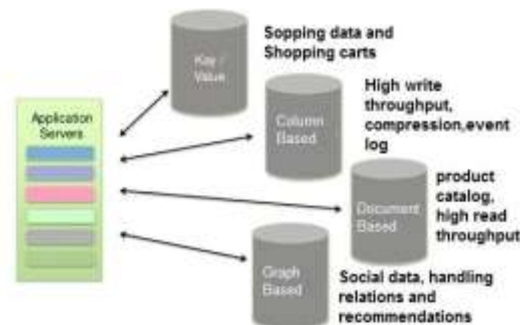

Fig. 1. Different databases suitability with different applications

we can take advantage of the strengths of both databases to enhance functionality of our application.

A. When to choose Neo4j over Cassandra?

Cassandra having master less clustering model, data repli cation ability and reliance on eventual consistency, its biggest strength is the ability to handle a very high read throughput.[3] For this reason, Cassandra is often used to store high volume data such as event logs, which don't require ACID guarantees like what is available with Neo4j.But there are some situations where Neo4j perform better than Cassandra. Such situations are summarized as follows:-

- Finding relationships
  As mentioned above, Cassandra store data in form of columns, which in some ways gives a performance advan- tage but this disconnected construction makes it harder to harness data relationships properly. Although, there is possibility of adding data relationships by embedding aggregate identifying information inside the field of an- other aggregate, but joining aggregates at the application level later becomes just as prohibitively expensive as in a relational database.

On the other hand, graph databases store data relation- ships as relationships. Everything is connected. It has the flexibility to add new nodes and relationships without compromising our existing network or expensively mi- grating our data. With data relationships at their core, graph databases are incredibly efficient when it comes to query speeds, even for deep and complex queries.[1] Hence, graph database like Neo4j are optimized for querying connections between people, things, interests, or really anything that can be connected. With this

TABLE I
TABLE SHOWING BASIC DIFFERENCE BETWEEN STRUCTURE OF CASSANDRA AND NEO4J

Cassandra	Neo4j
Cassandra comprises of table containing columns which is the basic unit of storage.	Graph database comprises of nodes with edges as relationships, each having some properties.
Tables or column families(containing columns that we are likely to query together) are the entities of a key space.	It consists of a label that is used to group nodes into sets. It is just like table name as all nodes labeled with the same label belongs to the same set.
Table further has row key, super column, column name and column value. Hence, it is a list of nested key-value pairs.	Graph contains properties that describe and provide value of a particular node or relationship.
Cassandra schema design focuses primarily on query patterns. Based on the type of data, demoralization and data duplication indirectly show relationships by combining related data.	It has edges that connect the nodes and properties, defining the relationship between them. The value is derived when analyzing the patterns between the nodes and the properties.
The primary key in Cassandra is made up of two parts: the Partition Key and the Clustering Columns. The first maps to the storage engine row key, while the second is used to group columns in a row.	Primary key or node with unique properties is created using unique constraints.
Null fields don't exist in Cassandra unless we add them.	Null value in Neo4j are missing value or undefined value of properties of a node or relationship.

they reveal patterns of similar behavior, influence, and implicit groups.

- Real time recommendations and decision making:- Neo4js native graph storing makes it easier to decipher suggestion data without any intermediate indexing every time. Neo4js native graph processing engine supports high-performance graph queries on large user datasets to enable real-time decision making. With this, it enable users to search for products, services or people based on a host of fine-grained criteria and continually improve recommendations by accommodating new data sources and types. Hence, Neo4j is more suitable than Cassandra for drawing relationships and developing recommenda- tions in applications like retail suggestion engines, fraud detection and social network monitoring.

II. RELATED WORK

The increase in demand for storage of high volume of variety of application specific data has exceeded the capabil- ities of single database. Due to this, the need for polyglot persistence is increasing as organizations prefer to use the best combination database suitable for their application.[2] During initial times many conversions from relational database to graph database, column database took place as most of the organizations have their information, usually stored in relational databases. For this reason, several solutions have been proposed which provide the comparison of schema of different databases as in [8] and mapping and migrating data from relational to Neo4j as in [7,9]. Not only this, the author in

IJIRMET ISSN (Online): 2456-0448
International Journal Of Innovative Research In Management, Engineering And Technology
Vol. 1, Issue 5, June 2016

[10] provides further high query optimization technique based on graph contraction that creates summarized graph in advance and uses it to efficiently query the original dataset. This is a promising technique for retrieving data in semantic aware computing. Similarly conversion of relational to column database is given in [5] and proposal of a systematic approach to database schema design in NoSQL column stores by means of automated schema generation and application-specific schema is given in [6]. This research paper is motivated from the fact that we can devise any type of use case according to

Fig. 2. Cassandra schema for User-Item Keyspace

the requirements of the application and the type of information to be stored. There are many such conversion from one database to another, but to the best of our knowledge, there is no work that tackles specifically the problem of migrating data and queries from a column database (Cassandra) to a graph database (Neo4j) which is the main focus of given research paper. To devise such conversion requires deep understanding of the basic schema/structure and the query language for both Neo4j and Cassandra that is provided by authors in [1] and [3] respectively, which further leads to writing an algorithm for migration of data from Cassandra to Neo4j.

III. PROPOSED ALGORITHM WITH EXAMPLE

Before giving algorithm for mapping and conversion of data from Cassandra to Neo4j firstly, lets talk about the basic difference between the structure of Cassandra and neo4j that is given in Table 1:

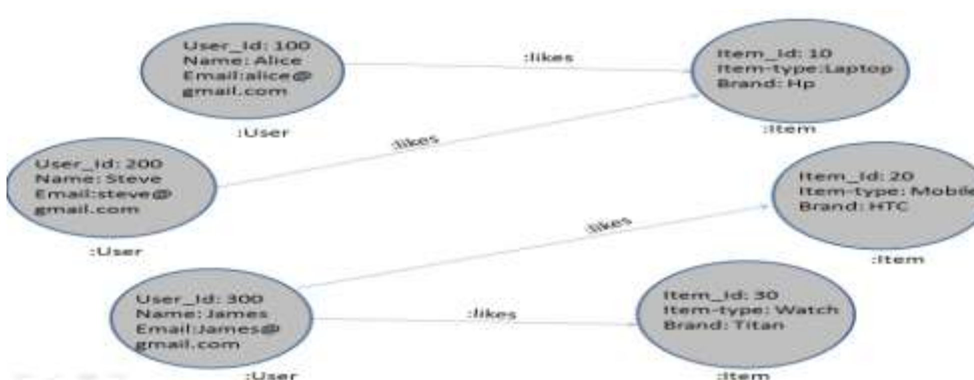

Fig. 3. Resultant Neo4j graph database for User-Item database

Now, lets take an example of an e-commerce system where users can like one or more items. One user can like multiple items and one item can be liked by multiple users, leading to a many-to-many relationship. Initially the data for this application is stored in Cassandra database as shown in figure
2. Its corresponding mapping in Neo4j results in a graph as shown in Figure 3. The migration of data from Column database to graph database is done using algorithm given below, in which Cassandra database is given as input and after processing through all the modules the output generated is the Graph Database G.

Algorithm 1 – Transform Cassandra Database C to Neo4j Graph Database G
Input: Cassandra Database C
Output: Neo4j Database G

```
    Initialisation :
 1: traversed[]<– φ
 2: ColFamily[] <– List of all tables in C
 3: SColFamily[] <– List of tables that gives general infor- mation in C
 4: RColFamily[] <– List of tables that give relational infor-
    mation in C
 5: for each SColFamily ∈ SColFamily[] do
 6:    Call traverseSColFamily();
 7: end for
    traversed[] <– SColFamily
 8: for each RColFamily ∈ RColFamily[] do
 9:    Call traverseRColFamily();
10: end for
    Call createIndexes();
11: return
```

Initially, schema information of column database C will be collected in variables described as follows:-
ColFamily[] - List of all tables or column family of Cassandra. It contains *User, item, User _By_item, item_By _User*.
SColFamily[] - List of tables or column family that gives general information. It contains *User, item* tables.
RColFamily[] - List of tables or column family that give relational information in Cassandra database. It contains *User By _Dept, Dept _By _User* tables.
Traversed[] - Table name is added to this list after that table is traversed. Initially, this list is empty.
traverseSColFamily() - This module is called for each table in *SColFamily[]*- list that has unique row key.
traverseRColFamily()- This module is called for tables that creates relation by using row keys of other tables as column value.
createIndexes() - This module create indexes.
In the end *createIndexes()* module is called to create indexes on all the nodes formed so far. Clearly, the above algorithm makes call to different modules/functions which actually perform the data migration task.
Various modules *(traverseScolumnFamily(), traverseRcolumn- Family(), CreateIndex())* called from Algorithm 1 are briefly explained as follows :-

Algorithm 2 – traverseSColFamily()

```
 1: for each tuple{t} ∈ SColFamily do
 2:    if SColFamily !∈ traversed[] then
 3:       firstNode <– createNode();
 4:       addProperties() <– t.getColumnValues(); 5:          addLabel() <– SColFamily.getName();
 6:    else
 7:       firstNode <– findNode(), where 8:        firstNode.label = table.getName() 9:
          property = t.RowKey
10:    end if
11: end for
```

12: **return**

traverseSColFamily() is called for each table in *SColFam- ily[]* list that has row key. Suppose *traverseSColFamily()* is called for *User* table.
For each *SColFamily* if that table or column family is not tra- versed than its tuples are read. For each tuple t of *SColFamily* a node firstNode is created in graph G, where firstNode has properties same as columns of tuple t. A label is also added to the node as the name of the *SColFamily*.
Such as for tuple (*100,alice)* a node is created with User_id as *100* and name as *alice* and label as *User*. If the *SColFamily* is already traversed than the node firstNode is found from Graph Database G with its property values same as the tuple t's column values. The row key of Cassandra is converted to property of node with unique constraint in Neo4j.

Algorithm 3 – traverseRColFamily()

```
    Initialisation :
 1: importedTables[] <– SColFamily.getImportedKeyInfo()
 2: getCount <– 0
 3: for each tuple {t1} ∈ RColFamily  do
 4:   for each importedTable ∈ importedTables[] & where importedTable.rowKey.value = t1.rowKey.value do
 5:     for all t1.columnKey corresponding to t1.rowKey in RColFamily[] do
 6:       while t1.columnKey != null do
 7:         getCount++
 8:         t1.columnKey++
 9:       end while
10:       while getCount != 0 do
11:         findNode() in G, where
12:         node.property = t1.rowKey.value
13:         node.property = t1.columnKey.value
14:         addEdge();
15:         t1.columnKey++
16:       end while
17:     end for
18:   end for
19: end for
20: return
```

traverseRColFamily():- In our example database *User _By _item, item _By _User* are tables under *RColFamily*. For each table its imported key information is collected. Imported key information contains the table name of those tables whose row keys are being used in column name in the table. In our case, *User* and *item* table are the tables whose row keys are used as column name in tables *User _By _Dept, item _By _User*.

Now for each tuple of *RColFamily (for eg. item _By _User)* its joinable tuples are found in all the imported tables. For table *User_By_Dept*, with row key 300 and column value 20,30 its joinable nodes are *:person(300,james)* to *:item(20,mobile)* and *:item(30,watch)*. The *getCount* variable will give the count of number of instances (column keys) a particular row key is associated. Here the *getCount* is 2. Now, for each joinable tuple find its corresponding node in G, keep on creating edges between the nodes until the *getCount* variable is not 0. Here two edges will be created as given in figure 2 for *:User(200,James)*.

Algorithm 4 – createIndexes()

```
1: for each SColFamily ∈ SColFamily[]  do
2:   Create index such that
3:   label = SColFamily.name
4:   index = SColFamily.RowKey
5: end for
```

In the end, indexes are created for all node labels. The module defined above create indexes on the attributes that were row keys in Cassandra. Hence like this we can migrate the whole data from Cassandra to Neo4j.

IV. CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to automati- cally migrate data from column database Cassandra to graph database Neo4j. Taking into consideration, the shortcomings of Cassandra and its inefficiency in handling large amount of relationship motivated this migration. But at the same time Cassandra's ability of high write throughput can also not be neglected. Applications (eg. fraud detection) that have high write throughput and also need to explore relationships within its data can possibly use such kind of use-case. There are no solutions that really stand out of the lot, there are only solutions adapted to the needs. Hence, this all justifies the need of polyglot persistence for Cassandra and Neo4j. The algorithm presented in the paper has been implemented and tested to prove the feasibility of the approach and efficiency of data mapping and migration. In future works I intend to refine the technique or the algorithm proposed in this paper to obtain a more scalable and compact target database.

REFERENCES

[1] Hongcheng Huang and Ziyu Dong. *Research on architecture and query performance based on distributed graph database neo4j.* In Consumer Electronics, Communications and Networks (CECNet), 2013 3rd Inter- national Conference on, pages 533536, Nov 2013.

[2] R. Sellami, S. Bhiri, and B. Defude. *Supporting multi data stores applica- tions in cloud environments.* IEEE Transactions on Services Computing, 9(1):5971, Jan 2016.

[3] G. Wang and J. Tang. *The nosql principles and basic application of cassandra model.* In Computer Science Service System (CSSS), 2012 International Conference on, pages 13321335, Aug 2012.

[4] S. Prasad. *Application of polyglot persistence to enhance performance of the energy data management systems.* Advances in Electronics, Comput- ers and Communications (ICAECC), 2014 International Conference,10-11 Oct. 2014.

[5] V. Bhagat. *Comparative Study of Row and Column Oriented Database* 2012 Fifth International Conference on Emerging Trends in Engineering and Technology 5-7 Nov. 2012.

[6] D. Bermbach. *Informed Schema Design for Column Store-Based Database Services* IEEE 8th International Conference on Service- Oriented Computing and Applications (SOCA) 19-21 Oct, 2015.

[7] S.Bordoloi and B.Kalita. *Designing Graph Database Models from Exist- ing Relational Databases.* International Journal of Computer Applica- tions (0975 8887), July 2013 .

[8] S.Batra, C.Tyagi.*Comparative Analysis of Relational And Graph Databases* International Journal of Soft Computing and Engineering (IJSCE), May 2012.

[9] R. Chen. *Managing massive graphs in relational DBMS* Big Data, 2013 IEEE International Conference, 6-9 Oct. 2013.

[10] A.Hayakawa. *Efficient Query Processing of Semantic Data Using Graph Contraction on RDBMS* Signal-Image Technology and Internet-Based Systems (SITIS), 2013 International Conference 2-5Dec. 2013.