APPLICATION OF ACTIVE RULES ON GRAPH DATABASE

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

MASTER OF SCIENCE

in

Computer Science

by

Vadlamannati Lakshmi Venkata Sai Raja Bharath

FALL
2018
APPLICATION OF ACTIVE RULES ON GRAPH DATABASE

A Project

by

Vadlamannati Lakshmi Venkata Sai Raja Bharath

Approved by:

__________________________________, Committee Chair
Dr. Ying Jin

__________________________________, Second Reader
Dr. Haiquan Chen

__________________________________
Date
Student: Vadlamannati Lakshmi Venkata Sai Raja Bharath

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

__________________________, Graduate Coordinator
Dr. Jinsung Ouyang

Date

Department of Computer Science
Abstract

of

APPLICATION OF ACTIVE RULES ON GRAPH DATABASE

by

Vadlamannati Lakshmi Venkata Sai Raja Bharath

A graph database presents data and the relationship among data based on the graph model. Graph Databases uses graph structures for semantic queries with nodes, edges, and properties. A graph database has two defining elements: 1) Node, which represents an entity, and 2) Arc or a Relationship, which is the connection between two nodes. There are many benefits of using a graph database, such as performance and flexibility. The most widely used graph database is Neo4j, which is used by many organizations and companies around the world, such as Wal-Mart and Lufthansa.

Although there are many advantages in graph database systems, it can be improvised with features that have been implemented in relational database systems. One of such improvisations is active rules. Relational database systems use active rules for constraint management, especially at complicated application level. This project incorporates active rules in a graph database, focusing on using active rules to specify business logic.
Once a rule is defined, the database will react to the predefined event and execute the business logic as necessary. This project focused on the language model of the rule system. It also implemented a prototype for rule execution system.

_______________________, Committee Chair
Dr. Ying Jin

_______________________
Date
ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Dr. Ying Jin Ph.D. for guiding me throughout this project. I can say that this project would not appear in its present form without his kind assistance and support. I owe a deep debt of gratitude to my advisor for her belief in my abilities, her enthusiasm for this project, and her continuous support during the entire project. She encouraged me with sound advice and lots of good ideas.

Also, I would like to thank my second reader Dr. Haiquan (Victor) Chen Ph.D. for his continuous encouragement and support during the project.

Finally, I would also like to thank the Department of Computer Science at California State University, Sacramento for giving me an opportunity to pursue my Master's in Computer Science.
DEDICATION

To my loving parents (Kanaka Durga and Durga Prasad), brother (Sarath), guardians (Sravya and Ravi), cousins (Srikanth, Sravanthi, Raghu, Abhinay), all my friends and my beautiful niece (Aira).
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td></td>
<td>Dedication</td>
<td>viii</td>
</tr>
<tr>
<td></td>
<td>List of figures</td>
<td>xi</td>
</tr>
<tr>
<td></td>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Neo4j</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Major Components of Neo4j</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Applications of Neo4j</td>
<td>4</td>
</tr>
<tr>
<td>2.4</td>
<td>Language</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>CQL Structure</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Active Rules</td>
<td>8</td>
</tr>
</tbody>
</table>
3. LANGUAGE .............................................................................................................. 10

3.1 Example ............................................................................................................... 10

3.2 Example Database ............................................................................................... 14

3.3 Rule Syntax .......................................................................................................... 15

3.4 Rule Examples ...................................................................................................... 16

4. PROTOTYPE FOR RULE EXECUTION .................................................................. 25

5. TESTING .................................................................................................................. 28

6. SUMMARY .............................................................................................................. 35

6.1 Challenges .......................................................................................................... 35

6.2 Conclusion and Future Work .............................................................................. 36

References ............................................................................................................... 37
LIST OF FIGURES

Figures Page
1. Neo4j Graph Model .................................................................................................................. 4
2. Nodes and Relationships based on the example data in Neo4j DB ......................... 14
3. Prototype Design.................................................................................................................... 25
4. The database before execution of Rule 2 ........................................................................... 28
5. The database before execution of Rule 2 ........................................................................... 29
6. The database after execution of Rule 2 .............................................................................. 30
7. The database before execution of Rule 4 ........................................................................... 31
8. The database before execution of Rule 4 ........................................................................... 31
9. The database after execution of Rule 4 .............................................................................. 32
10. The database after execution of Rule 4 ............................................................................ 33
11. The database before execution of Rule 5 ......................................................................... 34
12. The database after execution of Rule 5 ............................................................................ 34
1. INTRODUCTION

In today’s world where everything depends on data, NoSQL databases (no-SQL or Not Only SQL) are most needed. Graph Databases can manage and query highly connected data. “Graph Databases” are used to model in a broad range of business domains like Supply Chain markets and human interconnections, and can be applied in numerous fields like science, government, business, etc. This makes companies that develop Graph databases like Neo at the top of the market.

The goal of this project is to understand and apply the concept of "Triggers" or "Active Rules" on Graph Databases. After going through some theoretical background information, including the illustration of the technical side of Graph Databases and the concept of Active Rules, Neo4j Database is considered as it is the most popular graph database. A Simple Supply Chain Model with a list of Suppliers, Producers, Factories, Retailers, and Customers is used as the dataset as an example to illustrate the features our system.

As a prototype for the proof of the concept of this project, a Rule system is implemented, and the system consists of two parts. They are:

1. Language model – This part focuses majorly on the rule language and the rule syntax
2. Execution model – This part focuses on the prototype which is used to execute the rules
With this rule system, rules can be added to a repository via the language model and can be tested with the ones in the repository via the execution model. The implementation is written in Java by using various drivers like Bolt from Neo4j.

The rest of this report is organized as follows: Chapter 2 provides the background of Neo4j. Chapter 3 briefs about the rule language model, the structure of active rules, and example rules. Chapter 4 provides details of the prototype application. Chapter 5 explains the evaluation of the rules on the prototype in the project. Chapter 6 summarizes the project and provides future suggestions.
2. BACKGROUND

2.1 Neo4j

With the latest trend of social network and the value of understanding relationships between people, ideas, places, etc., gives out a highly specialized subset of data. Handling and management of this data has received a lot of attention and graph database model is the perfect solution for handling this data.

Neo4j, Inc. introduced Neo4j in the year 2007. It is a graph-based NoSQL database that stores data in the form of nodes and edges. This database provides simple, exceptionally improved interfaces to look at the connectivity of any part in the graph. Its engineers portray Neo4j as an ACID-compliant transactional database with native graph data storage and data processing. [1]

2.2 Major Components of Neo4j

There are four main components of Neo4j [2]. They are node, relationship, label, and property.
As shown in Figure 1, nodes are the primary data elements connected to other nodes. Relationships are directional connections between two nodes. Properties are named values for any component in the graph. They can be indexed and constrained. We can also create Composite indexes from multiple properties. Labels are used to group nodes into sets.

2.3 Applications of Neo4j

We all know that everything we have around us is connected and can be easily represented in the form of a graph. This gives us so many use cases where Neo4j can come into the picture. Before choosing supply chain management as a choice for this project, there were few other use cases considered to illustrate the use of Neo4j, such as:

- **Real-Time Recommendations:** Recommendation algorithms find relationships between people, products, and other services related to the purpose based on user's previous behaviors. Neo4j stores interconnected data about customers and
products, and since Neo4j doesn't need indexing at every suggestion, it provides a fast and effective algorithm to deal with real-time data. Walmart uses Neo4j for this purpose.

- **Master Data Management:** In large organizations, different systems store information about customers, employees, titles, and supply chain. With the graph model, it is easy to bring data from disparate systems together to create visual representation of customers or keep track of all the information about the organizational system itself. Cisco uses Neo4j for this purpose, and the company also uses Neo4j for their help desk solution.

- **Fraud Detection:** Fraud detection is crucial in the finance industry. One-way people avoid being detected by bank fraud algorithms is by opening several bank accounts with valid information and doing routine transactions without being an outlier. People open false bank accounts with the same identity token and withdraw all the money in all bank accounts. While it is hard to detect this behavior manually, it is straightforward to see it with a graph because people opening several bank accounts using the same identity token can quickly be recognized as a pattern in a graph.

- **Graph-Based Search:** Metadata is available for things like products, articles, etc. Being able to model metadata as a graph allows the system to enhance its search function, meaning users can find more relevant stuff. For example, in LinkedIn,
when users perform a search, they are provided results based on relevancy, not random or alphabetically sorted results. Lufthansa uses Neo4j for this matter.

- **Network & IT Operations**: If the data center is modeled as a graph, then dependency analysis can easily be applied on network systems. For example, if one virtual machine goes down, information is provided regarding how many applications will be affected. HP uses Neo4j to model their network for some large telecommunication providers.

- **Identity & Access Management**: Within large organizations, there are hundreds of users and controlling who has access to which information is crucial for security reasons. Creating groups and roles for each user comes in handy in this situation. This kind of data is vibrant and connected and can be easily handled by Neo4j.

2.4 **Language**

The language used in Neo4j is Cypher Query Language. Cypher is an open-source, declarative query language and based on Java. It is used to create the graph ecosystem and follows ASCII-art style syntax. With Cypher Query Language, user can follow a simple format to map patterns of nodes and their relationships into graph datasets. Cypher is similar to SQL through which the user will be able to perform actions such as match, insert, update or delete upon their graph data with simple programming logic.
The following are some of the features of CQL:

- **Cypher is a declarative language:** This means that we can specify the data input. We do not define how to get that data from the database.

- **Cypher is human readable language, and it is accessible not just for developers but also layman to quickly learn and use it.**

- **Cypher has expressions similar to SQL like WHERE, ORDER BY and simple conditional statements like <, =, and >.** The difference between Cypher and SQL is that Cypher is designed to represent graph data patterns. For example, it has the MATCH property which finds and specifies patterns in the data.

### 2.5 CQL Structure

Nodes represent data entities, and they can have labels, and each node represents different single data entities. It is equivalent to records in a relational database. Nodes can also have properties which are the attributes. Nodes are shown with parentheses like (p: Product).

In Cypher, between the nodes, we have lines which represent the relationship between each node. Relationships can also have properties just like nodes which is something that is much different than SQL. Also, relationships have directions. A relationship is shown as "–>" between two nodes.
2.6 **Active Rules**

Conventional database systems are passive: they just execute queries or transactions explicitly submitted by a user or a program. During execution of queries, it is very important to monitor these queries as events and to trigger a timely reaction when some conditions occur. For example, in an inventory system, the inventory there should be a proper monitoring on the stock and when the stock of an item falls below a threshold amount, an activity for reordering should be initiated.

But, in an active database system, the database system itself monitors the events, and when they occur, triggers an appropriate reaction promptly. This behavior is declared in production rules (also called event-condition-action rules) as per [4], which are defined and stored in the database.

An active database is a database that has an event-driven architecture (like E-C-A) which reacts based on the transactions in the database. As per [4], Event-condition-action rules for active database systems are of the form:

\[
\text{on event} \\
\text{if condition} \\
\text{then action} \ [4]
\]

An event is a database action that triggers a rule, A Condition is a check done to see if the active rule should be executed and an Action is also a database action which has to be taken
when an event occurs. Together, these Event-Condition-Action Rules are called as “Active rules.”

Events are classified into temporal and mutation events.

- **Temporal Events** - These events are time–based. So, the rule will be triggered based on the time

- **Mutation Events** - These events are query based or non-time based. So, the rule will be triggered based on a database transaction like CREATE, UPDATE, DELETE
3. LANGUAGE

3.1 Example

To demonstrate our approach, Supply Chain Management System is selected as the use case. A simple supply chain management system using Neo4j is implemented with essential components, such as Distributor, Retailer, Customer and Product.

The following CQL commands are executed to enter the data and manipulate the scenarios to test the rules.

1) The following CQL script can be used to create Customers:

```
CREATE (c:Customer { name: "Bharath" });
```

The customer node is created with a “name” property.

2) The following CQL script can be used to create Distributors:

```
CREATE (n:Distributor { name: "Ronzoni", Rating: 3.5 });
```

The distributor node is created with a “name” property, a “Rating” property.

3) The following CQL script can be used to create Retailers:

```
CREATE (r:Retailer { name: "Walmart", openTime : "9:00 AM", isStoreOpenforOrdering : "false", Rating : 4.0 });
```
The Retailer node is created which has a “name” property, an “openTime” property to store the open time of the retailer, a property “isStoreOpenforOrdering” to show whether the store is open, a “Rating” property for the retailer.

4) The following CQL script can be used to create Products:

```
CREATE (p:Product{ name: "rice" });
```

The product node is created with a “name” property.

5) The following CQL scripts can be used to create Relationships between all the created nodes:

**Delivers:**

This relationship maps the distributors, retailers with a relationship to show which products are getting delivered by a distributor to a retailer.

```
MATCH (d:Distributor{name: "Kirkland"})
MATCH (r:Retailer{name:"Walmart"})
MATCH (p:Product{name: "rice" })
CREATE UNIQUE (d)-[:Delivers{Product:p.name}]->(r);
```
Orders:

This relationship maps the retailers and distributors with a relationship “Delivers” to show the scenario where the products are ordered by a retailer to a distributor. The Orders relationship indicates the normal ordering procedure by the retailer to a distributor.

MATCH (d:Distributor{name: "Kirkland"})-[de:Delivers]-
>(r:Retailer{name:"Walmart"})-[s:Sells]->(p:Product{name: "rice" })
CREATE UNIQUE (r)-[o:Orders{Product:p.name}]->(d)

RestockOrders:

This relationship maps the retailers and distributors with a based on a relationship “RestockOrders” to show the scenario where the products are reordered by a retailer to a distributor to restock the items. The RestockOrders relationship indicates the orders made by the retailer to restock an item which will be out of stock in the future.

MATCH (p:Product {name: "rice"})<-[s:Sells]-(r:Retailer{name: "Walmart"})-[o:Orders]->(d:Distributor{name: "Kirkland"})
CREATE UNIQUE (r)-[rs:RestockOrders{
Product: "rice",totalQuantity:50, nextOrderAfterDays:5 }]->(d);
Sells:

This relationship maps the retailers and products to show which products are sold by a retailer with various properties on the relationship to maintain the counts of the products in the respective retailers

MATCH (r:Retailer{name: "Walmart"})
WITH r
MATCH (p:Product)
WHERE p.name = "rice"
CREATE UNIQUE (r)-[s:Sells {ProductPresentQuantity: 10,
ProductReqMinQuantity: 8, ProductOneDayQuantity: 20,
ProductTwoDayQuantity: 30, ProductFiveDayQuantity: 50, Price: 10.5}]->(p);

In this relationship, ProdReqMinQuantity is the minimum number required by the retailer to sell an item. ProductOneDayQuantity is the quantity required by the retailer for a day. ProductTwoDayQuantity is the quantity required by the retailer for 2 days. ProductFiveDayQuantity is the quantity required by the retailer for a week.
CustomerGoesToRetailer:

This relationship maps the retailers and customers to show the favorite retailer of a customer.

MATCH (c:Customer{name:"Bharath"})
WITH c
MATCH (r:Retailer{name:"Walmart"})
CREATE UNIQUE (c)-[ :CustomerGoesToRetailer {Product: "rice"} ]->(r);

3.2 Example Database

![Diagram of nodes and relationships](image)

Figure 2 Nodes and Relationships based on the example data in Neo4j DB
Figure 2 shows sample data of the database. There are Distributor Nodes, namely, “Ronzoni” and “Kirkland”. There is a “Walmart” node which is a retailer. “Kirkland” and “Ronzoni” deliver to “Walmart” via the “Delivers” relationship. The products “yoghurt” and “rice” are sold by the retailer through the “Sells” relationship. Customer “Bharath” goes to the retailer “Walmart” via the relationship “CustomerGoesToRetailer”. The “Orders” relationship indicates the normal ordering procedure by the retailer to a distributor and the “RestockOrders” relationship indicates the orders made by the retailer to restock an item which will be out of stock in the future.

3.3 Rule Syntax

The followings show the rule syntax for non-time-based rule and time-based rule respectively. All rule parameters start with “$”.

For a non-time-based rule:

```
CREATE RULE nontimebasedrulename($parameter1, $parameter 2 …..)
```

```
EVENT
Any CQL Query like CREATE, UPDATE or DELETE
```

```
CONDITION
A CQL Query
```
ACTION

A CQL Query

For a time-based rule:

CREATE RULE timebasedrulename($parameter1, $parameter 2 .....)

EVENT

Time or a Time interval

CONDITION

A CQL Query

ACTION

A CQL Query

3.4 Rule Examples

To show the ways to define rules based on the syntax, these rules were used in the project which came from the scenarios that can occur in a Supply Chain Management system:

1. At the open time of a store, check to see if the store has enough item quantities to processing customer orders. If that is true, then the store will be open to accept orders. This is an example for a time-based rule. Each store should have an attribute called “openTime”

CREATE RULE enableSystem($storeName, $storeOpenTime)
EVENT

SET r.openTime = $storeOpenTime FROM r: Retailer

WHERE r.Name=$storeName

CONDITION

MATCH (r:Retailer)-[sb:Sells]->(p:Product)

WHERE toInteger(sb.ProductPresentQuantity) >

toInteger(sb.ProductReqMinQuantity) AND

r.isStoreOpenforOrdering = "false"

ACTION

MATCH r

SET r.isStoreOpenforOrdering = “true”

WHERE r.Name = $storeName

2. If a new shipment has arrived to be delivered to a retailer, then there should be a check to see if the quantities last for a required period of time and based on the preset quantities ordering to restock the inventory will be placed from the retailer to the distributor. This is an example for a non time-based rule. Due to a limitation while parsing the property “s.ProductPresentQuantity” from the event from Java application to Neo4j we are giving the total sum(existing quantity of the product + new quantity) as $newItemCount. Solving this issue will be a
future direction. In Neo4j, the case statement used in this rule has the same logic as “if...else if”.

CREATE rule checkInventory($storeName, $item, $newItemCount)
EVENT
MATCH (r:Retailer)-[s:Sells]->(p:Product)
WHERE r.name = $storeName and p.name = $item
SET s.ProductPresentQuantity = $newItemCount

CONDITION
WITH r,p
MATCH (r:Retailer)-[sb:Sells]->(p:Product)
WHERE r.name = $storeName AND p.name = $item AND
sb.ProductPresentQuantity < sb.ProductReqMinQuantity

ACTION
MATCH (d:Distributor)-[de:Delivers]->(r:Retailer)-
[sb:Sells]->(p:Product)
WHERE r.name = $storeName AND de.Product = $item
WITH r,d,sb
MATCH (r)-[rs:RestockOrders]->(d)
SET rs.nextOrderAfterDays =
CASE WHEN sb.ProductOneDayQuantity > sb.ProductPresentQuantity THEN 1
WHEN sb.ProductTwoDayQuantity > sb.ProductPresentQuantity THEN 2
WHEN sb.ProductFiveDayQuantity > sb.ProductPresentQuantity THEN 5
ELSE 7 END
RETURN r;

3. At a system specific time, like the close time, there should be a check to see if the quantities last for a required period of time. Based on the preset quantities, order to restock the inventory will be placed from the retailer to the distributor. This is an example for a time-based rule. Each store should have an attribute called “closeTime”.

CREATE rule checkInventoryBasedonTime($storeName, $storeCloseTime)
EVENT
SET r.closeTime=$storeCloseTime FROM r: Retailer
WHERE r.Name=$storeName
CONDITION

WITH r,p
MATCH (r)-[sb:Sells]->(p)
WHERE sb.ProductPresentQuantity < sb.
ProductReqMinQuantity

ACTION

MATCH (d:Distributor)-[de:Delivers]->(r:Retailer)-
[sb:Sells]->(p:Product)
WHERE r.name = $storeName AND p.name = $item
WITH r,d,sb
MATCH (r)-[rs:RestockOrders]->(d)
SET rs.nextOrderAfterDays =
CASE WHEN sb.ProductOneDayQuantity >
sb.ProductPresentQuantity THEN 1
WHEN sb.ProductTwoDayQuantity >
sb.ProductPresentQuantity THEN 2
WHEN sb.ProductFiveDayQuantity >
sb.ProductPresentQuantity THEN 5
ELSE 7 END
RETURN r;
4. If an item is distributed to a retailer and the rating of the distributor is decreased, then there should be a check to see if there exists another distributor that supplies the same item, then the retailer orders the item from the distributor with better rating. This is an example for a non time-based rule.

CREATE rule
orderFromNewDistributorforRetailerBasedOnRating($distName,
$itemSoldbyDist, $newRatingDist)

EVENT

MATCH n=(d:Distributor)-[de:Delivers]->(r:Retailer)-
[s:Sells]->(p:Product) WHERE d.name = $distName AND
de.Product = $itemSoldbyDist
SET d.Rating = $newRatingDist
WITH p

CONDITION

MATCH (d:Distributor)-[de:Delivers]->(r:Retailer)-[s:
Sells]->(p:Product)
WHERE de.Product = $itemSoldbyDist
AND d.Rating > $newRatingDist
WITH collect(d) as goodDistributors, collect(r) as retailers
ACTION

MATCH n=(r:Retailer)-[o:Orders]->(d:Distributor)
WHERE $itemSoldbyDist in o.Product AND d.name = $distName
DELETE o
WITH goodDistributors
FOREACH (d in goodDistributors) CREATE UNIQUE (r)-
[o:Orders {Product: $itemSoldbyDist}]->(d)
RETURN r;

5. If a customer usually goes to retailer for a specific item (which is the favorite store for this item), but that retailer started selling the item for more price. There should be a check if there is another retailer who is selling the same item for less price and has a better rating than the previous retailer, then the customer will be going to this new retailer for that item. This is an example for a non time-based rule.

CREATE rule
makeCustomerGoToNewRetailerBasedonPrice($itemSoldbyRetailer,
$storeName, $price)
EVENT
MATCH (c:Customer)-[gt:CustomerGoesToRetailer]->(r: Retailer)-[s:Sells]->(p:Product)
WHERE s.Product = $itemSoldbyRetailer AND r.name = $storeName
SET s.ProductPrice = $price

CONDITION
WITH r.rating as n_rating, p
MATCH (c:Customer)-[gt: CustomerGoesToRetailer]->(r: Retailer)-[s: Sells]->(p:Product)
WHERE s.Product = $itemSoldbyRetailer
AND toFloat(s.ProductPrice) < toFloat($newPrice)
AND r.rating > n_rating
WITH collect (r) as goodRetailers

ACTION
MATCH n= (c:Customer)-[gt: CustomerGoesToRetailer]->(r: Retailer)-[s: Sells]->(p:Product)
WHERE s.Product = $itemSoldbyRetailer AND r.name = $storeName
DELETE gt;
WITH goodRetailers
FOREACH (r in goodRetailers | CREATE UNIQUE (c)-
[gt: CustomerGoesToRetailer

{Product:$itemSoldbyRetailer}]->(r))

RETURN r;
4. PROTOTYPE FOR RULE EXECUTION

There are three software systems which were used in this project. They are:

1. For the Graph database, Neo4j Desktop 1.1.10 for OSX has been used.

2. For building the Proof of Concept application, Java SDK 1.8 and Neo4j 3.3.4 for Java has been used.

3. As an environment development tool for this project, NetBeans IDE 8.2 for OSX has been used.

![Figure 3 Prototype Design]

As shown in Figure 3, there are seven major components of the prototype. They are:

1. Rule Specification Interface – In this component, users can define new rules through the interface
2. Event and Rule Parser – In this component, the input rule from the Rule Specification Interface is parsed and the result is saved to the Active Rule Repository.

3. Active Rule Repository – This component stores the rules.

4. End user Query Interface – Through this interface, users can write a CQL query.

5. Event Handler – This component reads the CQL query from the user as an event and identifies the actual values from the user’s CQL query.

6. Rule Engine – This component matches the event from the user with events present in the rule repository and fetches the matched rules, then the values of the parameters are mapped and bonded in the rules. Then the rules will be executed over the Neo4j Database.

7. Neo4j Database – Neo4j Database has the required data related to the use case.

The rules specified in Section 3.4 are entered and stored in the rule repository. Firstly, the rule type i.e. time-based or non-time based is selected. Then the required components for the rule, namely, the input parameters, an event, a condition and an action are entered. Finally, the rule is saved to the Active Rule repository. For a time-based rule, the system time itself will be hardcoded as an event and rest will be the same.

Through the End User Query Interface, a CQL event is entered into the console. The application will check if there exists any rule with the same event. The application works in such a way that even if the user enters actual values instead of parameters in the event, the application will parse the event and map the values with the parameters. All of this is
done in the Event Handler component. Then in the Rule Engine, the rule(s) which match the event entered by the user will be fetched from the Active Rule Repository. If there is no Rule with the event, then the application will run the CQL query. If there are rules retrieved that are triggered by the event, the rules will also be executed.
5. TESTING

This section describes the test cases for testing sample rules that are listed in Chapter 3.

**For Rule 2:**

\[
\text{MATCH (r:Retailer)-[s:Sells]->(p:Product)}
\]

WHERE r.name = “Walmart” AND p.name = “yoghurt”

SET s.ProductPresentQuantity = 50

**Before triggering the rule:**

In Fig 4, the value of the property “ProductPresentQuantity” for the retailer node “Walmart” is 10

![Image of database before execution of Rule 2](image.png)

**Figure 4 The database before execution of Rule 2**
In Fig 5, the value of the property “totalQuantity” on the relationship “RestockOrders” for the retailer node “Walmart” is 10 and the value of the property “nextOrderAfterDays” is 5.

![Diagram](image)

**Figure 5 The database before execution of Rule 2**

**After executing the trigger:**

In Fig 5, on the relationship “RestockOrders” from the retailer node “Walmart”, the value of the property “totalQuantity” is 50 and the value of the property “nextOrderAfterDays” is 7
For Rule 4:

MATCH n=(d:Distributor)-[de:Delivers]->(r:Retailer)-[s:Sells]->(p:Product)

WHERE d.name = "Kirkland" AND de.Product = "rice"

SET d.Rating = 3.0

WITH p

Before executing the trigger:

In Fig 7, the rating is 4.5 which will be changed to 3.0
Figure 7 The database before execution of Rule 4

Figure 8 The database before execution of Rule 4
After executing the trigger:

In Fig 9, the rating of the distributor node “Kirkland” is changed to 3.0, which causes the relationship “Orders” to be removed from the retailer nodes “Costco” and “Safeway”. In Fig 10, “Orders” link is added from the retailers “Costco” and “Safeway” to distributor “Ronzini”.

Figure 9 The database after execution of Rule 4
For Rule 5:

```
MATCH (c:Customer)-[gt:CustomerGoesToRetailer]->(r:Retailer)-
    [s:Sells]->(p:Product)
WHERE s.Product = "rice" AND r.name = "Walmart"
SET s.ProductPrice = 14
```

**Before executing the trigger:**

In Fig 10, the event is a product price change and before this event the customer nodes “Bharath” and “Sharath” have a relationship “CustomerGoestoRetailer” to retailer node “Walmart”
Figure 11 The database before execution of Rule 5

After executing the trigger:

In Fig 11, the customer nodes have no relationship to retailer “Walmart”

Figure 12 The database after execution of Rule 5
6. SUMMARY

6.1 Challenges

Neo4j is a rapidly developing framework. There were multiple challenges during the development of this project. For example, frequent releases of Neo4j versions made the code to break frequently. The issues were solved after extensive research and intense code refactoring. Proactive steps were taken to keep the software, Neo4j version, and other framework/plugins to be up to date in the project.

Firstly, implementing Neo4j with Java was straightforward, but parsing Neo4j queries from Java console was very difficult. Even a small mistake like a syntax error in the query from the console will not be executed properly on Neo4j. So, during the design of the prototype, proper care was taken in such a way that the application executes the rules on Neo4j in an efficient way and any error during the execution of the transaction was rolled back. In addition, the code was written in such a way that a Neo4j session was always closed after a transaction, to avoid memory leaks due to the presence of garbage data caused by an unclosed session.

Secondly, validating Queries from a user and to map the values from the user with the parameters in the rules from the repository was challenging. To execute the queries on Neo4j database from the prototype, the Neo4j Desktop Application should also run in parallel because the prototype gets connected to Neo4j database via the Neo4j Desktop
Application. After execution of the queries, the results should be displayed from the Neo4j database to the Java console. This process was a challenge because the results have to be translated from an advanced diagrammatical representation of the database in Neo4j Desktop Application to a simple non-diagrammatical representation in Java console. A future direction is a better user interface design for combining two consoles.

Lastly, the application was kept up to date with all the Neo4j versions. The code was manually changed as per the updates and then the code was built to match the updates. But this issue was permanently solved in the Neo4j version 3.3.4.

6.2 Conclusion and Future Work

Throughout this project, efforts were made to learn various uses and applications of Neo4j. Multiple applications was surveyed and Supply Chain Management was selected as the use case for this project. Rules were defined from various scenarios from Supply Chain Management system. A prototype was designed where these rules were defined, stored in a rule repository, and are validated on a graph database. Future direction is to refine the language model and build an execution model based on this prototype.
References


