EPiC Series in Computing

Volume 69, 2020, Pages 134–140

Proceedings of 35th International Conference on Computers and Their Applications

Active Rules in a Graph Database Environment

Ying Jin, Vadlamannati Lakshmi Venkata Sai Raja Bharath, and Jinaliben Shah

Department of Computer Science, California State University, Sacramento
Sacramento, CA 95819-6021, USA
jiny@csus.edu

Abstract

With the rapid growth of data nowadays, new types of database systems are emerging in order to handle big data, known as NoSQL databases. One type of NoSQL databases is graph database, which uses the graph model to present data and the relationships among data. Existing graph database systems are passive compared to traditional relational database systems that allow automatic event handling through active rules. This paper describes our approach of incorporating active rules into graph databases, allowing users to specify business logic in a declarative manner. The active system has been built on top of a passive graph database to react to events automatically. Our focus is to specify business rules declaratively rather than enforce integrity constraint using rules. Our system consists of a language framework and an execution model. Language specification will further be illustrated by on a motivating example that shows the use of rules in an application context. The paper also describes the design and implementation of the execution model in detail.

Keywords: Active Rule, Graph Database System, Execution Environment

1 Introduction

The amount of data in the world has been increasing dramatically in recent years. NoSQL systems are emerging to handle big data management. NoSQL stands for Not Only SQL. One type of NoSQL system is graph database. Graph database systems are based on the graph model, in which the fundamental components are nodes and arcs. Nodes represent entities while arcs represent relationships between entities. Compared to relational database systems that commonly use JOIN operations, graph databases use graph traversal to handle similar issues. The demand for graph databases is increasing, because it is beneficial to use graph databases to handle relationships.

Active database systems support reactive behavior by building an event-driven architecture on top of traditional passive database systems. Active rules allow users to specify the action to handle different

types of events. Once a rule is specified and stored, the active database system will automatically execute it when the event occurs. . A database user does not need to constantly monitor business operations in order to react to an event. Instead, a rule specified by the user is monitored and executed by the active rule system. The concept of active rules has been incorporated by commercial relational database systems in a simplified format of triggers.

Many applications, such as fraud detection and supply chain management, have adopted graph databases for implementation. There are various business rules within those applications. If these applications were implemented in a relational database system, the relational database would be able to handle it automatically using triggers. However, using a graph database would require the business rules to be handled and enforced manually. This motivates the research of incorporating active rules into graph databases in order to transform them from passive to active systems. This paper describes our approach of incorporating active rules into a graph database. We will use the widely used graph database Neo4j [1] as the implementation system. Our rule system consists of two parts: a language model and an execution environment. The language model is for rule definition. The execution model handles the run time of the rule executions.

The rest of the paper is organized as follows. Section 2 is the brief overview of graph databases, active rules, and related work. Section 3 describes the language model. Section 4 presents our system architecture, as well as the design and implementation of the execution model. Section 5 concludes the paper with a summary of our work.

2 Background and Related Work

Neo4j is an open source Graph Database. The main blocks are Nodes, Relationships, Properties and Labels. Nodes represent the entities and Edges represent the relationships. Edges and Nodes can contain properties. Cypher is the query language for Neo4j.

An active rule is composed of three parts: Event, Condition and Action. It is often known as ECA Rule. The event part describes the happening of an activity. The condition part examines the conditions that need to be satisfied when the event takes place. The Action is a set of operations or tasks which will be executed only if the condition is satisfied. Relational database systems have adapted active rules in the simplified form of triggers. Triggers handle mutation events such as insert, delete, and update operations. Active rules have been applied in relational database systems and object-oriented database systems, detailed in [2]. Our past research investigated the incorporation of active rules in a component integration environment [3] and fuzzy XML databases [4].

The research in [5] is about constraint checking on a graph database, which provides an application program to check an input field such that the input data is within the domain to enforce the domain constraint. The research in [6] handles the reaction of emergence or deletion of subgraphs. Specifically, it is a subgraph-condition-action trigger for a graph database system. Our research described in this paper focuses on handling business logics using Event-Condition-Action rules. Our system can react to both mutation events (insert, delete, and update) and as well as temporal events.

3 Language Model

This section describes the language model of our system. A supply chain application with a list of retailers, distributors, products, and customers are used to illustrate the example of rules.

An active rule in our system consists of an event, a condition, and an action. There are two types of events supported by our system: temporal events and mutation events. Temporal events are based on time or time intervals. Temporal events trigger timer rules. Mutation events are raised by create

(insert), delete, and update operations. This paper focuses on described rules that are triggered by mutation events. More rules can be found in [7, 8].

The basic structure of the rules are:

CREATE Rule RuleName ($parameter 1, …, $parameter n)
Event
    Cypher query such as CREATE, UPDATE & DELETE
Condition & Action
    Cypher Query

Rules are defined using the Cypher Query Language [1]. Condition and Action are specified together in the Condition & Action component to describe the reaction to the event. The following is an example of the rule that describes the business logic of placing an order. Retailers sell products. The database keeps track of the present quantity of a product that a retailer sells, as well as the threshold quantity to restock an item in one day, two days, and five days. For example, if the quantity is less than the threshold of one day, the retailer should restock the next order after one day. The rule shows that when a shipment is accepted by the retailer, the present quantity of the product will be changed. Depending on the amount of inventory at that time, the retailer can adjust when to order more items from the distributor.

```
CREATE rule checkInventory ($storeName, $item, $newItemCount: number)
EVENT
    MATCH n=(r:Retailer)-[s:Sells]->(p:Product)
    WHERE r.name = $storeName AND p.name = $item
    SET s.spPresentQuantity = $newItemCount  + s.spPresentQuantity
AFTER
CONDITION AND ACTION
    WITH r,p,s
    WHERE s.spPresentQuantity < s.spReqMinQuantity
    MATCH n1=(d1:Distributor)-[de1:Delivers]->
    (r1:Retailer)-[s1:Sells]->(p1:Product)
    WHERE r1.name = $storeName AND de1.Product = $item
    WITH r1,d1,s1
    MATCH (r1)-[rs:RestockOrders]->(d1)
    SET rs.restockNextOrderAfterDays = CASE
    WHEN  s1.spPresentQuantity < s1.spOneDayQuantity THEN 1
    WHEN s1.spPresentQuantity < s1.spTwoDayQuantity THEN 2
    WHEN s1.spPresentQuantity < s1.spFiveDayQuantity THEN 5
    ELSE 7 END
    RETURN r1,d1;
```

Figure 1: Rule Example

The rule has three parameters: the store, the item, and the new shipment quantity of the item. Numeric type that involves arithmetic operations within the rule should be restricted as “number” in the parameter. The event is a Cypher update statement, which changes the present quantity of the item sold by the retailer because of the arrival of new shipment. The condition and action part of the rule firstly checks if the present quantity value is less than the minimal quantity requirement of the item. Although the new shipping causes the increase of inventory, more items than expected could have already been sold before the shipment arrives. If condition evaluation results in inadequate inventory, there is a need to place an order. As explained in the previous paragraphs, depending on the result of comparing with

the thresholds, a restock order can be placed within one, two, five, or seven days. By default, each rule is an "after" rule. We will describe "before" and "after" in the next section.

4 Execution Environment

This section describes the system architecture and the rule execution environment. As shown in Figure 1, there are six main components in the system: 1) Active Rule Specification Interface, 2) Query Interface, 3) Rule Parser, 4) Rule Engine, 5) Event Handler, and 6) Active Rule Repository.

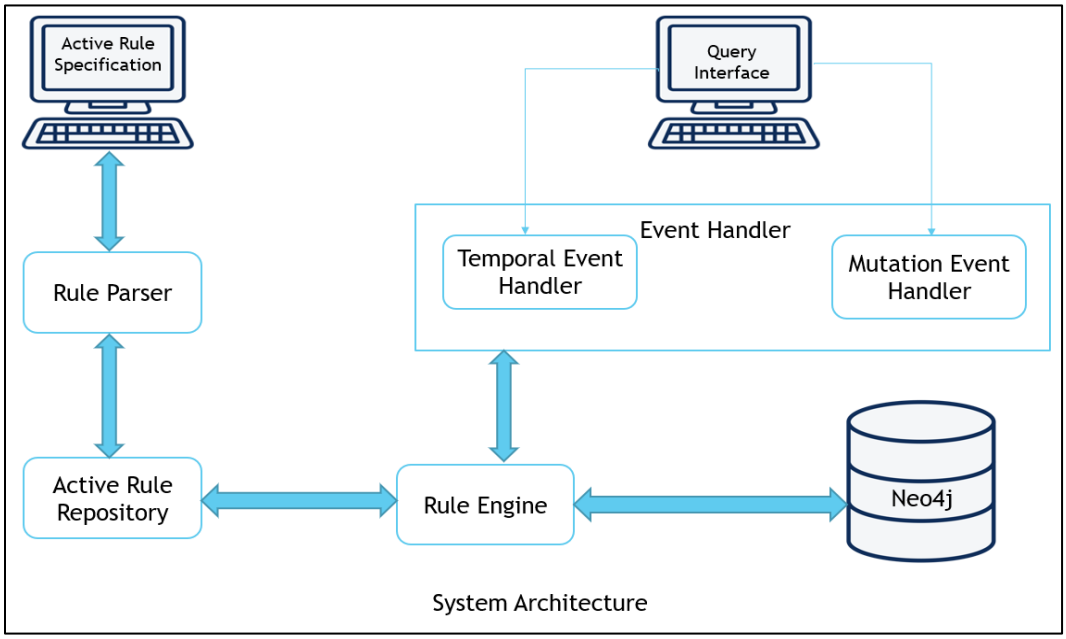

Figure 1: XACML Policy Module

Two interfaces provided by the system are Active Rule Specification Interface and Query Interface. The Active Rule Specification Interface allows users to specify rules such as the rule in Figure 1. The Query Interface allows a user to input Cypher commands such as create, update and delete. The flow of the execution model is as follows: a user specifies an active rule through the Active Rule Specification Interface. The Rule Parser then parses the user-defined rule. Next, the parsed rule is stored in the Active Rule Repository. When a user inputs a Cypher command from the Query Interface, Event Handler communicates with the Rule Engine upon event occurrence. The Rule Engine attempts to retrieve rules that are triggered by the event from the Active Rule Repository. If the rules match according to the event, the Rule Engine retrieves the rules and executes the rules.

In our current implementation, after rules are specified, the rules are stored as files. The rule parser parses a rule and then stores the rule with the following information: rule name, parameters, "before" or "after", event, and condition & action.

There are two types of events: mutation events and temporal events. The Temporal Event Handler handles Timer rules. Timer rules are executed based on a scheduled time or time interval. Java inbuilt

class TimerTask and Timer are used in the current implementation. Scheduler accepts the 24-hour format time regarding rule execution time. When the system time matches with the predefined time in the rule, an event is raised. The Temporal Event Handler notifies the Rule Engine for further rule retrieval and execution.

A mutation event can occur when a user inputs a Cypher command to create, update, or delete data. Users input Cypher commands through the Query Interface. Once a Cypher query is entered, the Mutation Event Handler passes the command to the rule engine. The Rule Engine searches the existing rules to see if there is any event in a rule that matches the cypher command. This was implemented using pattern matching and regular expression using JAVA. A regular expression is a special sequence of characters that matches the set of strings or patterns from the string.

Pattern matching has two main classes: 1) Pattern class and 2) Matcher class. Pattern class is a compiled representation of the regular expression. To create a pattern, we must first invoke its compile() method. After that, a pattern object is returned, and the method accepts the regular expression as the first argument. Matcher class interprets the pattern and performs a match operation against the input string by the user. A matcher () method is used to perform this logic. The java.util.regex package is used for matching the patterns with the regular expression.

The format of the regular expression used to match the input string is shown in Figure 3. For each different logic, new regex is used. Using this regex, the Cypher command is matched with an event in the rule file.

```
Regular Expression

"<([{\\^-=$!|]})?*+.>"
```

Figure 3: Format of Regular Expression

After using the pattern (regular expression), the event will be modified as per the regular expression syntax. Figure 4 illustrates that all the special characters (i.e. “=”, “.”, “-”, “(”, “)”, “$”, “+”) are written by placing a “\” symbol in front of the characters, such that the structure is accepted by the regex. Based on this concept, this project matched the user’s cypher command and event in the rule files.

```
MATCH n\=\(r:Retailer\)\-\[s:Sells\]\-\>\(p:Product\)

WHERE r\.name \= \$storeName1 AND p\.name \= \$item1

SET s\.spPresentQuantity \= \$newItemCount1 \+ s\.spPresentQuantity
```

Figure 4: Format of Event Using Regex

Using regex, a map is created. If the values from the parsed user command and events from files match, then the matchInput() function will extract the values of parameters, or in other words, extract the variable values from the user event. If there is a match, then the condition and action part are retrieved. Since conditions and actions are defined using Cypher Query Language, they can be executed over the Neo4j database. If more than one rule is retrieved, then the system will execute multiple triggered rules according to priorities. The current system uses the rule creation time as a rule priority. The lowest priority is given to the newly created rules. The details can be found in [8].

Rules triggered by mutation events can be categorized according to the time of execution as a “before” rule or an “after” rule. This is similar to the “before” trigger and “after” trigger in relational databases. All timer rules are “after” rules. When a rule is defined as a “before” rule, the rule will be

executed before the Cypher command. If a rule is defined as an "after" rule, the rule is executed after the Cypher command. The processing logic can be summarized as follows:

A Cypher command C raises an event E.

1) Retrieve all rules that are triggered by E, save them to set R,
2) For rules within R that are defined as "before", order "before" rules according to their priorities, and then produce an ordered List as BeforeRule_List
3) For rules within R that are defined as "after", order "after" rules according to their priorities, and then produce an ordered List as AfterRule_List
4) For each rule in the BeforeRule_List, execute the rule one by one
5) Execute the cypher command C
6) For each rule in the AfterRule_List, execute the rule one by one

An example of a Cypher command that raises an event to trigger the "checkInventory" rule (defined in Figure 1) is shown in Figure 5. The challenging part is that we do not match the command with the event literally since a command can be defined differently. Instead, we match them by patterns as described above.

```
Match n= (rr:Retailer) – [se:Sells] -> (p: Product)
Where rr.name= "Costco" and p.name = "Pasta"
SET se.spPresentQuantity = 200 + se.spPresentQuantity
```

Figure 5: Cypher command example

The Rule Engine retrieves the "checkInventory" rule from the rule repository by matching the event part of the rule to the Cypher command. The values, such as "Costco", are passed to the rule parameters. These parameters values are passed as a binding structure from the rule event part to the condition and action part of the rule. If there is no "before" rule triggered by this event, then the Cyhper command (shown in Figure 5) executes first. Since "checkInventory" rule is defined as an "after" rule, this rule is in the AfterRule_List. According to the priorities in the AfterRule_list, the condition and action part of the "checkInventory" rule will be executed when it is its turn.

5 Summary

This paper described our approach of incorporating active rules into a graph database, focusing on business rules. The current version of our system was implemented using Neo4j. Our system consists of a language framework and an execution environment. Active rules are defined based on Cypher Query Language, reflecting the business logic that a rule definer wants to express. Once a rule is defined, it will be parsed and stored in the Rule Repository. The execution environment automatically react to event occurrence by executing the condition and action of each triggered rule. Two types of events are handled by the system: temporal events and mutation events. As we know, building an active rule system is very challenging. Our current system is a proof of concept implementation of an active rule system on top of a passive graph database. Future work includes refining the rule execution environment, building user friendly interfaces, and performance evaluation.

References

[1] Neo4j Graph Platform. https://neo4j.com/
[2] L. Liu, T. Özsu (editors), *Encyclopedia of Database Systems*, Second Edition. Springer 2018
[3] Y. Jin, S. D. Urban, S. W. Dietrich, "A Concurrent Rule Scheduling Algorithm for Integration Rules," *Data and Knowledge Engineering*, Elsevier Science, Vol. 61/1, March 2007. pp. 530 – 546.
[4] Y. Jin, H. J. Mehta, "Composite Event Processing in an Active Rule-Based Fuzzy XML Database System", in *Proceedings of the 12th IEEE International Conference on Information Reuse and Integration*, August 3-5, 2011, Las Vegas, NV, USA, pp. 7-10
[5] K. Rabuzin, M. onecki, M. Sestak, "Implementing Check Integrity Constraint in Graph Databases", In *Proceedings of 82nd IIER International Conference,* October 3-4, 2016*,* Berlin, Germany, pp. 19-22
[6] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, S. Salihoglu, "Graphflow: An Active Graph Database", in *Proceedings of SIGMOD 2017,* May 14-19, 2017, Chicago, IL, USA, pp. 1695-1698
[7] Vadlamannati Lakshmi Venkata Sai Raja Bharath, *Application of Active Rules on Graph Database*, Master Project Report, California State University, Sacramento, 2018
[8] Jinaliben Shah, *Active Rule Execution in Graph databases*, Master project report, California State University, Sacramento, 2019