EPiC Series in Computing
Volume 89, 2022, Pages 21–30
Proceedings of 35th International Conference on Computer Applications in Industry and Engineering

Mutation Event and Fuzzy Active Rule Processing in a Graph Database System

Ying Jin and Vaidehi Rakesh Shah
Department of Computer Science,
California State University, Sacramento,
Sacramento, CA 95819-6021, USA
jiny@csus.edu

Abstract

A graph database system is a type of NoSQL databases that is based on the graph data model using nodes and arcs. Existing graph databases are passive and can only handle crisp data. Information in the real world can be imprecise and vague rather than crisp. Integrating fuzzy logic into database systems allows users to use uncertain data, which presents the degree to which something is true. Compared with passive databases, active databases support event handling by monitoring and reacting to specific circumstances automatically. This paper describes our approach of incorporating fuzzy concepts and active rules into a graph database system. Our recent publication has described our work of temporal event processing. This paper focuses on mutation events handling and active rule processing in a fuzzy system, covering the language model, the execution model, and architecture design. The language model defines the rule structure and contains the metadata for rule processing. Architecture design identifies the system's architectural components and user interfaces including rule specification interface and query interface. The execution model handles rule processing and execution at run time. A supply chain application is used to demonstrate the examples of active rule specification and execution.

1 Introduction

Traditional databases only handle crisp and precise data. To handle imprecise data, fuzzy database systems extend the capabilities by allowing fuzzy values be to stored and queried. Fuzzy logic computes the degree to which something is true [1]. Compared to that, crisp logic is considered binary, in which a value is either true or false.

Passive database management systems execute database operations only upon commands issued by users. Traditional database systems are passive as they lack the functionality of automatic reaction to events. An active database system provides declarative specification and automatic event handling using active rules [2]. An active rule consists of three parts: an event, a condition, and an action. When an event occurs, the condition is evaluated. If the condition is fulfilled, the action part conducts a sequence of operations on the database. Active rules are also known as event-condition-action or ECA rules. Triggers are simplified active rules in relational database systems.

Graph database systems are one type of NoSQL databases. Graph database systems are based on the graph model, using node and arcs to present data and its relationships. This research incorporates fuzzy logic and active rules into Neo4j [3], a predominant graph database system. Two types of events can trigger rules: mutation events and temporal events. Timer rules are triggered by temporal events that occur at a specific time or time interval. Mutation events are raised by the Create, Update, and Delete operations, which are write operations to the databases. Our previous research described our design and implementation on temporal event handling and timer rule processing in a fuzzy graph database [4]. This paper focuses on processing rules triggered by mutation events. Specifically, we will present our research on the language model, system architecture design, and implementation of rule processing and execution. The language model covers rule definition and metadata for language processing. The condition-action part of a rule is based on a Cypher query [5]. In addition to using crisp values in a rule, users can use fuzzy expressions to specify their business logic. Architecture design identifies the architectural components and user interfaces in the system. The execution model handles rule processing at static time and rule execution at run time. Our system is a general system, which is not restricted to a specific application. To illustrate the rule definition and execution logic, we use a supply chain example with products, distributors, retailers, and customers. Retailers order products from distributors. Customers will go to the retailers to purchase products. Further details can be found in [6].

The rest of the paper is organized as follows. Chapter 2 provides the background of fuzzy logic and discusses related research. Chapter 3 presents the language model with the rule structure. Chapter 4 explains the system architecture and rule processing. Chapter 5 summarizes the research and provides future work.

2 Related Work

Information can be ambiguous, imprecise, and uncertain in the real world. Fuzzy data presents the property that an attribute is not associated with one unique value. For instance, the term "hot" can refer to a numerical range, from 60 to 80 Celsius in a given context. Fuzzy set theory is a generalization of crisp sets that allows the presentation of information in a matter of degrees. Membership functions are used in the fuzzy set theory to compute crisp values. Linguistic variables and their respective linguistic values are defined in a membership function. For instance, in the sentence "The soup is hot”, "soup" is a linguistic variable or fuzzy variable; and "hot" is a linguistic value. With the use of membership functions, linguistic values can be converted to numerical values. Different membership functions can be used by different distributions. This research covers three types of distribution: 1) trapezoidal distribution, 2) possibility interval, and 3) triangular distribution. We described three types of distributions in our previous publication [4] for temporal event handling. We use the same way to handle the distributions in this paper.

Our previous publications have presented our research on fuzzy XML databases [7]. There is limited research on incorporating fuzzy logic or active rules into graph databases. The research in [8] described the approach of "Trigger-By-Example" to implement triggers in graph databases, including both BEFORE INSERT trigger and AFTER INSERT trigger. This approach is a variation of the traditional Query-By-Example approach. Users can specify rules using its graphical interface. The work in [9] introduced a system architecture to handle triggers in the graph databases. The architecture contains a one-time query processor to handle the simple type of events such as insert, update, and delete. It also has a continuous query processor to save the subgraph-action rules to perform specified actions. Both [8] and [9] only allow users to specify the rules containing crisp values. The research in [10] added fuzzy logic to Neo4j, which defined "fuzzy data" as a new data type. It presented its system using a social network application. The work in [11] and [12] also introduced fuzzy logic into graph databases. Compared with the related research, this project incorporates both active rules and fuzzy logic into a graph database management system. In our system, users can specify fuzzy terms in an active rule, thus allowing active behaviours to be specified in a fuzzy manner in a graph database environment.

3 Language

The structure of rules is shown in Figure 1. Rules are specified based on the Cypher Query Language. Cypher is the query language of Neo4j, which allows users to query and manipulate nodes and arcs (relationships) in the graph. The details of Cypher can be found in [5]. A mutation event is raised from operations of creating nodes and/or relationships, update nodes and/or relationships, and delete nodes and/or relationships. BEFORE/AFTER specifies when to execute the condition and action part of the rule. "Before" rule executes the condition and action before executing the Create, Update, Delete operation. "After" rule executes the condition and action after the operation. The condition and action part of the rule is specified using extended Cypher, which extends Cypher by allowing fuzzy logic in a Cypher query.

```
CREATE RULE RuleName ($parameter 1, …$parameter n)
EVENT
            Cypher query such as Create, Update, Delete
BEFORE/AFTER
CONDITION AND ACTION
            Extended Cypher query
```

Figure 1: Rule Structure

A rule example is shown in Figure 2, which describes the business logic related to distributors and retailers. A retailer usually orders a product from its "primary" distributor that is its first preference. When a distributor increases the cost, if its "secondary" distributor offers a better price and the rating is "high", then the preference of "primary" and "secondary" will be switched. Four parameters are defined in the rule. For example, the "distName" parameter is for the distributor. The same rule can be used by multiple distributors with different distributor names without the need to define a rule for each distributor to cover the same business logic. The event part is raised by the operation of cost changes, defined by a Neo4j Cypher query. The condition-action part compares the price and checks the rating. If the conditions of price and rating are true, then action is performed to modify preference. The condition and action are also based on Cypher query. Cypher query is extended to accept fuzzy terms. In this example, the linguistic variable is "rating" and the linguistic value is "high". Threshold can also be specified, such as "dis.dRating = #high WITH THOLD = 0.4" to combine with the fuzzy term to control the range of the corresponding crisp values. More fuzzy terms can be specified in other rules. For example, when a new shipment arrives, a rule can check if profit = [10,20], popularity = ~ 2, and the product is perishable, then a 10 percent discount is applied to the product's original sale price.

```
CREATE RULE RetailerDistributorRelationship ($itemSoldbyDist, $cost, $distName, $retailerName)
EVENT
MATCH n=(d:Distributor)-[de:Delivers]->(r:Retailer)
WHERE d.name = $distName AND
de.productName = $itemSoldbyDist AND
r.name = $retailerName
SET de.cost = $cost
AFTER
CONDITION and ACTION
WITH d, r, de MATCH n1= (dis:Distributor)-[del:Delivers]->(ret:Retailer)
WHERE del.productName = $itemSoldbyDist  AND
ret.name = $retailerName AND
del.type = "Secondary" AND
dis.dRating = #high AND
del.cost < de.cost
SET de.type = "Secondary" , del.type = "Primary"
```

Figure 2: Rule Example

4 System Architecture and Execution Model

4.1 System Architecture and Execution Flow

As shown in Figure 3, the architecture components include rule interface, query interface, rule parser, fuzzy rule repository, crisp rule repository, mutation event handler, and rule engine. The system has two interfaces: Active Rule Specification Interface and Query Interface. The Rule Interface allows users to specify rules. The Query Interface is used to enter a Cypher command that may raise an event to trigger a rule. Users can define a rule through the rule specification interface, according to the rule structure defined in Figure 1. The rule parser parses the rule and stores the rule in the fuzzy rule repository. Next, the fuzzy rule is converted to a crisp rule. Crisp rules are stored in the crisp rule repository.

The other interface, Query Interface, allows users to enter Cypher commands. If the command is a read operation to query the database without data modification, and if this operation only contains crisp values, then it is executed in the Neo4j database directly. On the other hand, if the read operation contains fuzzy terms, it is processed using fuzzy query processing method first, before querying the Neo4j database. If the command is a write operation, e.g. Create, Update, or Delete, then it can be a candidate to trigger a rule. The mutation event handler processes the event and passes it to the rule engine. Rule engine fetches the rules from the Crisp Rule Repository if there are any. If at least one rule is matched, the fetched rules are executed based on the "before/after" definition. That is, the rule is executed either before or after the Cypher command.

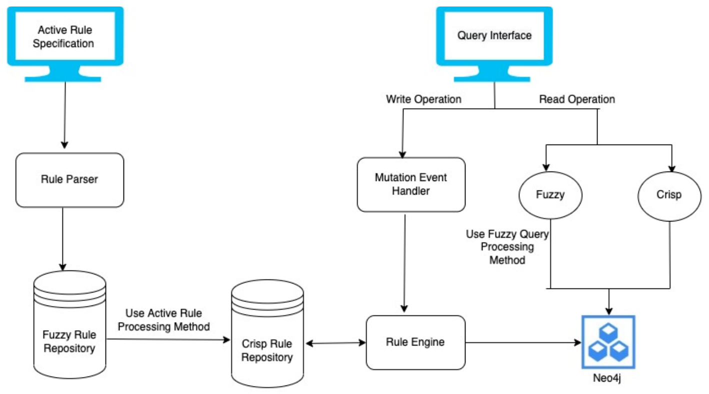


Figure 3: System architecture of mutation event handling

4.2 Fuzzy Rule Repository

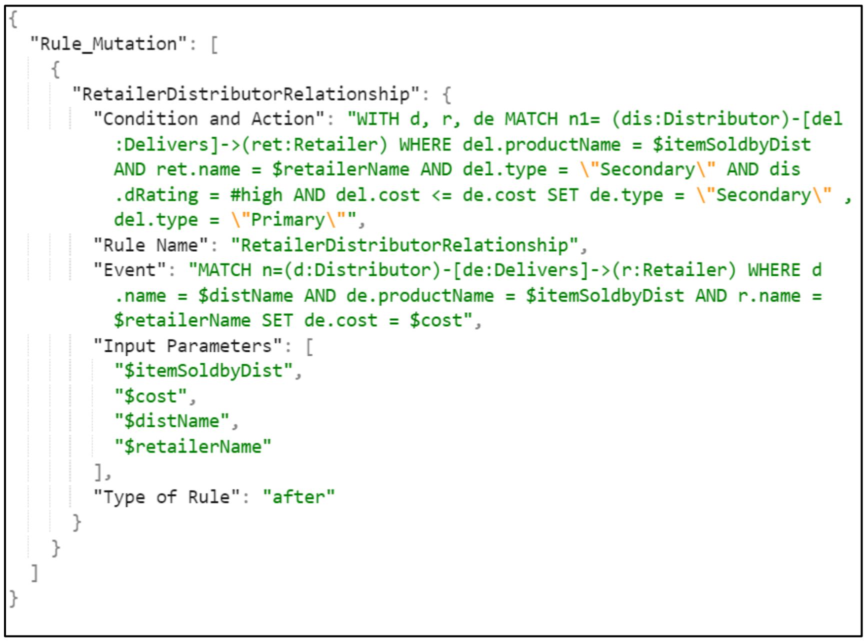
Fuzzy Rule repository stores fuzzy rules. The Fuzzy Rule Repository is in a JSON format and contains two lists based on the rule types. “Rule_Mutation” is the list of fuzzy rules which are triggered by mutation events. “Rule_Timer” is the list of fuzzy timer rules that are triggered by temporal events. The “RetailerDistributorRelationship” rule shown in Figure 2 is an example of rules triggered by mutation events, specifically, the update operation. Figure 4 shows the stored structure of this rule. “Rule Name” is the name of the rule. “Input Parameters” has the list of the input parameters. Input parameters start with a ‘$’ symbol. “Event” is the Cypher command such as create or update. The block of “Condition and Action” contains a fuzzy query. In this case, “dRating” is a linguistic variable. “Type of rule” specifies as “Before” or “After”. “Before” rule is executed before the operation that raises the event; “After” rule is executed after the operation.

4.3 Fuzzy to Crisp Conversion

The “Condition and Action” part of the rule can contain fuzzy terms. We need to convert the fuzzy terms into their crisp presentation, because Neo4j can only accept the crisp values. As described in Section 2, our system covers three types of distributions: possibility interval, trapezoidal distribution, and triangular distribution. These linguistic values are converted to their respective crisp values using the distribution defined in Fuzzy Meta Knowledge Base (FMKB). FMKB contains the values of alpha, beta, gamma, delta, and margin which are used by the three types of distribution to handle the fuzzy information. Figure 5 shows the trapezoidal distribution, in which the fuzzy to crisp conversion can be specified in (1) and (2). THOLD is the threshold. A fuzzy value is converted to a range of crisp values of [min, max].

$$Min = (\beta - \alpha) * THOLD + \alpha \qquad (1)$$

$$Max = [(\delta - \gamma) * (1 - THOLD)] + \gamma \qquad (2)$$



```
{
  "Rule_Mutation": [
    {
      "RetailerDistributorRelationship": {
        "Condition and Action": "WITH d, r, de MATCH n1= (dis:Distributor)-[del
          :Delivers]->(ret:Retailer) WHERE del.productName = $itemSoldbyDist
          AND ret.name = $retailerName AND del.type = \"Secondary\" AND dis
          .dRating = #high AND del.cost <= de.cost SET de.type = \"Secondary\" ,
          del.type = \"Primary\"",
        "Rule Name": "RetailerDistributorRelationship",
        "Event": "MATCH n=(d:Distributor)-[de:Delivers]->(r:Retailer) WHERE d
          .name = $distName AND de.productName = $itemSoldbyDist AND r.name =
          $retailerName SET de.cost = $cost",
        "Input Parameters": [
          "$itemSoldbyDist",
          "$cost",
          "$distName",
          "$retailerName"
        ],
        "Type of Rule": "after"
      }
    }
  ]
}
```

Figure 4: Example of rule storage structure

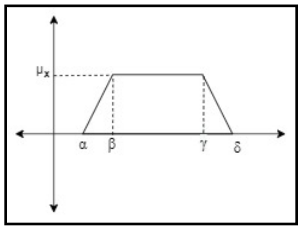


Figure 5: Trapezoidal Distribution

Figure 6 is the possibility interval distribution. The linguistic variable is defined on an interval [m, n]. For example, we can define profit on [20, 30]. Triangle distribution is shown in Figure 7, where the conversion is defined in (3) and (4).

$$Min = d - (margin * (1 - THOLD)) \qquad (3)$$

$$Max = d + (margin * (1 - THOLD)) \qquad (4)$$

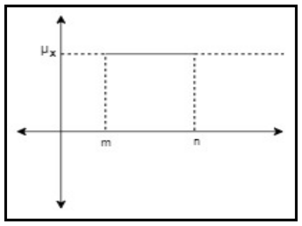


Figure 6: Possibility Interval

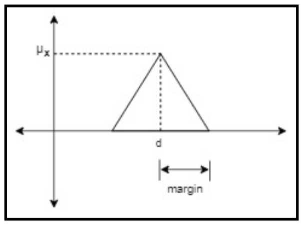


Figure 7: Triangle Distribution

Special characters such as "#" are used before a linguistic variable to distinguish types of distribution. Trapezoidal, possibility interval, and triangle distributions are signified by "#", "[]", and "~", respectively. The parser is based on our previous work in [4]. It is built based on JavaCC to identify the linguistic variables, parse the fuzzy query, and generate the crisp presentation.

4.4 Crisp Rule Repository

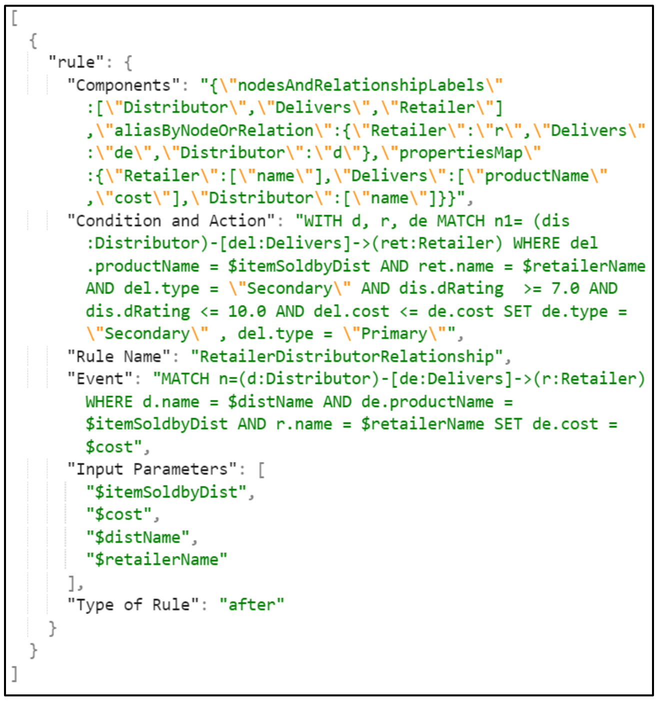
After a rule is converted from fuzzy to crisp, it is stored in the crisp rule repository. For the fuzzy rule example in Figure 2, the crisp rule generated is shown in Figure 8. In the crisp rule repository, a rule has the information of "Rule Name", "Input Parameters", "Event", and "Type of rule". The "Condition and Action" block contains the converted crisp query. The "Components" captures the key information of an event. Our system uses the "component" structure to match the Cypher command entered by a user with the rules stored in the crisp rule repository.

The "component" structure contains three important parts. "Nodes and Relationship labels" is a list of node names and relationship names used in the Cypher command. The "Alias By Node or Relation" is a list of pairs of values containing name of the node/relationship and the corresponding aliases used. "Properties Map" is a list of pairs of values containing the name of the node/relationship and its properties being accessed.

Our system builds the "component" structure for the event part of a rule. Then it is stored in the crisp rule repository. At run time, when a user enters a Cypher command using the query interface, another "component" structure is created for this user command. Only when these two component structures match are the matching rules fetched from the crisp rule repository and executed.

4.5 Rule Execution

When a user enters a Cypher command using the Query Interface, the event handler processes the event. The rule engine checks if this command can raise an event that matches the event part of a defined rule in the crisp rule repository. If yes, the rule can be executed according to the rule type. The execution flow is as follows: a user's Cypher command is sent to the searchRules() method to find the matching rules in the repository. In searchRules() method, for every rule presented in the crisp rule repository, the "component" structure in the rule and the "component" structure of the user's command are compared. We use the matchStructure() method to compare nodes and relationship labels. If the labels match, then the respective properties are checked. If both the labels and the properties match, then the rule is considered to be a matching rule.



```
[
  {
    "rule": {
      "Components": "{\"nodesAndRelationshipLabels\"
        :[\"Distributor\",\"Delivers\",\"Retailer\"]
        ,\"aliasByNodeOrRelation\":{\"Retailer\":\"r\",\"Delivers\"
        :\"de\",\"Distributor\":\"d\"},\"propertiesMap\"
        :{\"Retailer\":[\"name\"],\"Delivers\":[\"productName\"
        ,\"cost\"],\"Distributor\":[\"name\"]}}",
      "Condition and Action": "WITH d, r, de MATCH n1= (dis
        :Distributor)-[del:Delivers]->(ret:Retailer) WHERE del
        .productName = $itemSoldbyDist AND ret.name = $retailerName
        AND del.type = \"Secondary\" AND dis.dRating  >= 7.0 AND
        dis.dRating <= 10.0 AND del.cost <= de.cost SET de.type =
        \"Secondary\" , del.type = \"Primary\"",
      "Rule Name": "RetailerDistributorRelationship",
      "Event": "MATCH n=(d:Distributor)-[de:Delivers]->(r:Retailer)
        WHERE d.name = $distName AND de.productName =
        $itemSoldbyDist AND r.name = $retailerName SET de.cost =
        $cost",
      "Input Parameters": [
        "$itemSoldbyDist",
        "$cost",
        "$distName",
        "$retailerName"
      ],
      "Type of Rule": "after"
    }
  }
]
```

Figure 8: Crisp Rule Example

After the matched rules are fetched from the rule repository, the rules are sent to the replaceAlias() function to replace the aliases used in the user's query. We use the regular expression and pattern matching of Java to handle alias replacement. After the alias are replaced, switched WHERE condition is handled in the system. Next, we use the replaceTokens() function to pass the actual values to the input parameters and use these values in the "Condition and Action" part of the rule. After the replacement logic is executed, a final query is generated by combining the "Event" and "Condition and Action" parts of the rules.

4.6 Alternative Design

The overhead of using an active rule system is that the system spends time to match rules, retrieve rules, and execute rules. We are working on evaluating various structures and storage options to reduce the overhead. Instead of using JSON files as the fuzzy and crisp repository, one option is to use the graph database itself to store the rules. The following attributes are in a node: rule name, event, input parameters, component of node and relationship labels, component of alias, component of property maps, condition and action, and type of rule.

Another option is to use other database systems' power of query processing and optimization. When the size of the rule base is large, the query processing time to match "component" structure is critical for rule retrieval. Using relational database for query processing is a good candidate. When the schema is designed properly, we can avoid substring processing within queries to reduce the time of matching "component" structure. We design relational database tables as follows:

1) Rule (rule_name, event, condition_action, rule_type, component_alias), in which rule name is the primary key;

2) Parameter (rule_name, rule_parameter), in which the combination of rule name and rule parameter is the primary key and rule name is the foreign key referencing to the Rule table;

3) Component_property_map(rule_name, label, property_name), in which the combination of rule name, label, and property_name is the primary key and rule name is the foreign key referencing to the Rule table.

This allows the retrieval of matching rules quickly using the database engine, however, it requires additional database system.

5 Summary

Graph databases are gaining more and more attention because of their unique data model that is suitable for processing highly interconnected data. This research incorporates active rules and fuzzy concepts into Neo4j graph database system. This paper describes our language model and execution environment for fuzzy active rules. Rule definitions are based on Cypher query language. Users can specify fuzzy terms in the "Condition and Action" part of a rule. There are two types of events: temporal event and mutation event. We have represented our work of temporal event processing in [4]. This paper presents the details of handling mutation events and corresponding rule processing. To the best of our knowledge, this is the first project that incorporates fuzzy logic and active rules into a graph database system. Our future work is to refine the system, such as design a more user-friendly interface and extend our approach to other database systems.

References

[1] LA Zadeh. "Fuzzy sets," *Information and control*, vol. 8 (1965), pp. 338–353.

[2] L. Liu, T. Özsu (editors), Encyclopedia of Database Systems, Second Edition. Springer 2018

[3] Neo4j, "Neo4j Documentation," [Online]. Available: https://neo4j.com/docs/.

[4] A. Vadwala, Y. Jin, "Event and Query Processing in a Fuzzy Active Graph Database System," in *Proceedings of 37th International Conference on Computers and Their Applications*, vol 82, March 2022, pp. 82-91.

[5] Neo4j, "The Neo4j Cypher Manual v4.4," [Online]. Available: https://neo4j.com/docs/cypher-manual/current/.

[6] V. R. Shan, "Mutation Event Handling in Neo4j," Master project report, California State University, Sacramento.

[7] Y. Jin, H. J. Mehta, C. Madalli, "An Active Rule-based Fuzzy XML Database System," *Journal of Computational Methods in Science and Engineering* (JCMSE) 12 (2012), IOS Press, The Netherlands, pp. 103-111.

[8] K. Rabuzin and M. Šestak, "Creating Triggers with Trigger-By-Example in Graph Databases," in *Proceedings Of the 8th International Conference on Data Science, Technology and Applications*, 2019, vol. 1, pp. 137–144.

[9] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. "Graphflow: An Active Graph Database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, Association for Computing Machinery, New York, NY, USA , 2017, pp. 1695–1698.

[10] B. P. Costa and L. D. V. Cura, "An Neo4j implementation for designing fuzzy graph databases," in *Proceedings of the 23rd International Database Applications & Engineering Symposium*, Association for Computing Machinery, New York, NY, USA, June 2019, pp. 1–6.

[11] O. Pivert, E. Scholly, G. Smits, and V. Thion, "Fuzzy Quality-Aware queries to graph databases," *Information Sciences*, 2020, 521, pp. 160-173.

[12] A. Castelltort and T. Martin "Handling scalable approximate queries over NoSQL graph databases: Cypherf and the Fuzzy4S framework," *Fuzzy Sets and Systems*, 2018, 348, 21-49.