# Creating Triggers with Trigger-By-Example in Graph Databases

Kornelije Rabuzin[1][a] and Martina Šestak[1][b]

[1]*Faculty of Organization and Informatics, University of Zagreb, Pavlinska 2, 42000 Varaždin, Croatia*
*{krabuzin, msestak}@foi.hr*

Keywords: Trigger-By-Example, Graph Databases, Triggers, Active Databases.

Abstract: In recent years, NoSQL graph databases have received an increased interest in the research community. Various query languages have been developed to enable users to interact with a graph database (e.g. Neo4j), such as Cypher or Gremlin. Although the syntax of graph query languages can be learned, inexperienced users may encounter learning difficulties regardless of their domain knowledge or level of expertise. For this reason, the Query-By-Example approach has been used in relational databases over the years. In this paper, we demonstrate how a variation of this approach, the Trigger-By-Example approach, can be used to define triggers in graph databases, specifically Neo4j, as database mechanisms activated upon a given event. The proposed approach follows the Event-Condition-Action model of active databases, which represents the basis of a trigger. To demonstrate the proposed approach, a special graphical interface has been developed, which enables users to create triggers in a short series of steps. The proposed approach is tested on several sample scenarios.

## 1 INTRODUCTION

The idea of active mechanisms able to react to a specified event implemented in database systems dates from 1975, when it was first implemented in IBM's System R. The idea was quite simple; it was important to implement a mechanism that could be able to react to different types of events that occur primarily within the database or in its surroundings. In database theory, such behaviour was described by the concept of active (Event-Condition-Action, ECA) rules. ECA rules are interpreted in the following way: if some event occurs in the system, and some conditions are fulfilled, then a given action (or set of actions) is automatically executed as a system's reaction to the event. The events defined in ECA rules can vary in their complexity, ranging from simple (e.g. basic data manipulation statements such as INSERT, UPDATE and/or DELETE) to complex events, which can be defined by means of simple events, or events such as a sequence of events, negation, etc. In database systems, ECA rules are most often implemented through the trigger mechanism. A trigger represents a database object written in a given procedural language, which executes automatically when a given event occurs.

To avoid manually writing the trigger function code for inexperienced users, the Trigger-By-Example approach has been introduced to simplify the process of designing database triggers. The approach uses the Query-By-Example (QBE) as a graphical interface for creating triggers (Lee et al., 2000b), and makes the entire trigger design process more user-friendly.

Nowadays, modern database systems need to handle important challenges, such as large data volume, data integrity, scalability, variety of data sources, unstructured data, etc. Hence, in some application domains, traditional relational databases have been "replaced" by their NoSQL counterparts designed to better adapt to these challenges. Graph databases are a category of database solutions within the NoSQL ecosystem designed to efficiently store and manage highly interconnected data (for instance, social network data).

Triggers as database objects have only been recently introduced in a very few Graph Database Management Systems (GDBMSs) (e.g. Neo4j and JanusGraph), and they still require users of different levels of expertise to have a certain level of query language syntax knowledge. This research is motivated by this issue, and the aim of this paper is to make the trigger design process in graph databases easier, faster and more understandable for different users.

The main contribution of this paper is an approach, which describes how to easily design and im-

[a] https://orcid.org/0000-0002-0247-669X
[b] https://orcid.org/0000-0001-7054-4925

plement triggers in graph databases. To implement the proposed Trigger-By-Example approach in graph databases, we implemented a graphical user interface (GUI), which enables users to define triggers stored as ECA rules, i.e., Cypher query language statements, in a graph database (specifically, Neo4j).

The rest of the paper is organized as follows: Section 2 contains an overview of existing research papers related to active databases, Trigger-By-Example approach and developed graph-based rule specification engines. In Section 3, a theoretical background is given to help readers understand the concept of active and graph databases and the TBE approach. In Section 4 the proposed TBE approach in graph databases is presented and demonstrated on simple Neo4j use cases. Finally, we conclude this paper by describing future research directions.

## 2 RELATED WORK

Over the years, there has been a number of research papers, which introduced graph-based rule specification engines and visual interfaces. During the 90s, Dayal, Widom and Ceri published several relevant research papers and books related to active database systems. In these publications, the authors discussed active database systems in general and their possible application domains (Widom and Ceri, 1996), using declarative approach to specify the model for active rules execution (Ceri, 1992), rule execution semantics and implementation (Dayal et al., 1994), etc.

Nowak, Bak and Jedrzejek developed a graph-based prototype implementation of a graphical interface for specifying rules (Nowak et al., 2012). Apart from rule creation, the interface is able to perform rule reasoning in order to obtain results. The rule creation process is carried out by specifying the body (left hand side, LHS) and the head (right hand side, RHS) of the rule in a form of two separate graphs, which are then used to build ”if LHS then RHS” statements. Created rules are able to check if there is a given fact with specific attribute values in the knowledge base, or if there exists a relationship between two existing facts. The first condition type is supported in the current state of our implementation, while the relationship existence check will be part of our future work. The authors used Jess rule engine for rule creation and reasoning, which requires users to specify rule conditions by following Jess language syntax similar to RDF. In the proposed approach, the users can specify rules by simply following ”natural” semantics, i.e., they do not need to be familiar with any kind of syntax.

Next, Grüner, Weber and Epple explored the possibility of using rule-based systems for industrial automation (Grüner et al., 2014). Specifically, the authors used Neo4j GDBMS and Cypher to build a rule-based system, which will demonstrate the benefits of using graph concepts to specify rules. In their approach, a rule consists of a premise (Cypher query) and a conclusion (series of operations executed based on results of the query evaluation), and it is stored in XML format as a statement written in a descriptive syntax similar to RuleML. However, in this approach, the rules are created either manually on demand or by the system, which lowers the need for users' engagement in the rule specification process. On the other hand, our graphical interface for rule specification provides more flexibility for the users, and it further engages the user in the process by making it more simple.

Furthermore, an interesting rule-based engine has been introduced in (Rapsevicius and Juska, 2014). The authors developed an expert system for the Compact Muon Solenoid (CMS) Cathode Strip Chambers detector at the Large Hadron Collider (LHC). One of the system's components is the rule-based complex event processing (CEP) engine, which consists of rules written in SQL syntax forming a decision tree. In the context relevant for this paper, a rule presents a named computational expression that results in a conclusion if expression conditions are met (Rapsevicius and Juska, 2014). The CEP engine ensures that, for each incoming data stream, a relevant rule is fired, which evaluates the conditions, and returns a conclusion based on that evaluation. The conclusion can then be configured to perform a certain action, such as sending notifications, execute commands, etc. The rules definitions are stored within tables in a relational databases, and the authors developed a GUI interface for rules specification. Nevertheless, the interface requires users to specify rules by using specific operators with no explicit syntax guidelines. Compared to our proposed approach, the GUI requires users to still have a level of syntax knowledge, whereas our approach enables users to specify rules in a completely visual manner regardless of their level of knowledge and experience.

The idea of active graph databases is still in its early years of development. The most significant contribution in this field has been made by Kankanamge et al., who developed Graphflow, an active graph database (Kankanamge et al., 2017). The system is built on Neo4j, and uses Cypher++, a declarative Cypher extension, which supports triggers as subgraph-condition-action rules. Cypher++ support the specification of rules, which are triggered on executing MATCH, CREATE, DELETE, UPDATE and SHORTEST PATH queries, and such rules can result in creating new versions of the underlying property graph or writing a subgraph into a local file. The underlying syntax of Cypher++ is equivalent to Cypher query language. Moreover, Cypher++ represents one of several implementations of the openCypher project (Neo4j, 2018), which goal is to continually improve Cypher query language, and make it a standardized graph query language.

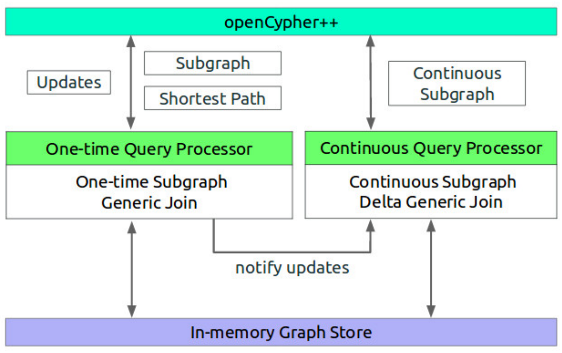


Figure 1: Graphflow system architecture (Kankanamge et al., 2017).

In (DSG, 2017), the authors presented an overview of Graphflow system architecture shown in Figure 1. The system uses two query processors depending on the event type, that triggered the rule. The one-time query processor is responsible for handling Cypher statements (MATCH, UPDATE, INSERT etc.), and perform updates on the underlying graph store, whereas the continuous query processor stores and evaluates subgraph-action rules in order to perform given actions. In our case, all currently supported rules are processed by a single query processor.

## 3 BACKGROUND

### 3.1 Active databases and active rules

Active databases represent database solutions, which rely on active (ECA) rules in order to automatically react to certain events. In the context of active rules, an event represents some change of state that requires an intervention. The simplest types of events are:

- Statements such as INSERT, UPDATE, and/or DELETE,
- Time events (absolute, relative and periodic),
- User-defined events,
- Transaction events (beginning/end of a transaction),
- Method events (in object-oriented databases), etc.

Simple events can be joined (E1 and E2, E4 or E3, etc.) and produce a complex event. Other types of complex events would include:

- negation - an event did not happen within a given time interval *t*,
- repeat - something repeated several times within a given time interval *t*,
- sequence - several events occurred in a predefined sequence, etc.

In order to model an active system, one has to take care of objects, events and transactions, which transform between database states by operating on objects (Kangsabanik et al., 1997). A good Active Database Management System (ADBMS) usually refers to a system, which supports more and different event types.

The ECA rule execution process contains several steps. Once the event has been detected, it triggers one or more rules. The conditions for triggered rules have to be evaluated. We say that the rule is triggered, but it is not a guarantee that it will be executed because the rules condition component needs to be evaluated. Based on the condition evaluation, rules actions are executed (if the condition evaluation was successful). Actions that are being executed could trigger other (new) rules. Additionally, in (Herbst, 1996), the concept of ECAA rules was introduced as an extension to ECA rules. In ECAA rule model, if the condition for an occurred event is successfully evaluated, the first action will be executed; otherwise, an alternative action is executed.

### 3.2 Trigger-By-Example

The Trigger-By-Example (TBE) approach was first introduced by Lee, Mao and Chu in (Lee et al., 2000a). Since its roots can be traced to the Query-By-Example approach/language, the main purpose of TBE is to help user to write trigger rules more easily through a graphical interface. As the authors suggest, an important benefit of TBE is that, in its implementation as a layer between a visual interface and database triggers, the TBE approach is loosely dependent on the underlying database system used. Additionally, database triggers written in SQL language are procedural in nature, but the visual interface enables users to specify rules in a declarative manner.

In (Lee et al., 2005), the same authors discussed TBE properties in more detail. Each component of the ECA rule is represented in a skeleton table differentiated by their prefix (for instance, condition skeleton table is prefixed by C). In its early years, SQL supported only INSERT, UPDATE and DELETE trigger event types. Trigger condition definitions can be categorized as either parameter filter or general constraint type. Parameter filter definition type is represented in the event skeleton table, and uses transition variables (BEFORE or AFTER) to specify the condition. The general constraint type is used to represent general triggers regardless of event types, so they are represented in the condition skeleton table.

TBE can also be used as an integrity constraints enforcement mechanism. The reason for this lies in the fact that the underlying implementation enables rule specification and processing necessary to maintain database integrity. The idea of maintaining graph database integrity by following the TBE approach will be implemented and discussed in the next sections.

### 3.3 Graph databases

Graph databases represent a NoSQL category, in which data is stored as nodes and relationships between nodes. This idea is appropriate for many different scenarios, including social network analysis, fraud detection, IT infrastructure (computer networks), recommendation engines, etc. (https://neo4j.com/use-cases/). Unlike a relational database, where during query execution tables need to be joined to retrieve values from more than one table, graph databases use physical pointers between nodes. This eliminated the need for complex joins, and increases the graph query execution speed, making graph databases much faster, especially when searching whether nodes are connected, or when the shortest path between two nodes needs to be found.

Various graph query languages have been developed for graph databases; the most often used languages are Cypher and Gremlin. In this paper, we focus on Cypher query language, because it is used much more often than Gremlin due to its declarative nature and syntax similar to SQL.

There are many different statements supported in Cypher, but at this point of time there is no native Cypher statement for creating triggers, and the level of support for triggers as specific database objects in modern GDBMSs is rather low. For this reason, we implemented a GUI, which enables users to specify simple ECA(A) rules, and store them in Neo4j database for future usage during query execution.

At the moment, in Neo4j, triggers are supported only as *TransactionEventHandler* objects, which analyze submitted database transactions and perform certain actions (De Marzi, 2015). The official Neo4j documentation supports three hooks, i.e., states, in which trigger evaluation can be performed: *beforeCommit*, *afterCommit* and *afterRollback*. On the other hand, in JanusGraph, triggers can be implemented by using user transaction logs, where triggers are registered for a certain change in data (e.g. a new edge of given type is added), and fire an external event or make additional changes to the graph (JanusGraph, 2017).

## 4 METHODOLOGY

To demonstrate the proposed approach, we built an application prototype by using Java 8 and Neo4j graph database, which communicate via *neo4j-java* driver. The prototype can be used as a graphical interface to create active rules in Neo4j graph database.

There are two main approaches, which can be used when building a system with active capabilities, namely integrated and layered approach. The layered approach includes extending the existing Database Management System (DBMS) with an additional layer, which adds the active capability to the DBMS. This layer is responsible for event detection, rule execution, etc. Conversely, the integrated approach means that the DBMS core needs to be changed in order to be able to detect events, evaluate rules and manage transactions in a more advanced manner.

With the ADBMS implementation approaches in mind, the following methodology has been used when designing and implementing the Trigger-By-Example approach:

- ECAA rules are used to gain better control of the system's behaviour in case when the condition is not evaluated successfully,
- the application is built by following the layered implementation approach, i.e., the application is built as an extension to the Neo4j GDBMS,
- the condition component of active rules is performed immediately,
- the action and alternative action execution component of active rules is performed immediately,
- simple types of events are supported (INSERT, UPDATE and DELETE operations).

To summarize, in the proposed TBE approach, a graph database trigger is an ECAA rule evaluted by the rule processing engine built on top of Neo4j GDBMS to maintain database integrity. The engine does not require any additional query processors, because the ECAA rule is not stored processed directly in the database. Instead, the rule processing is carried out on the application level, where the engine identifies rules needed to be checked for a given user's query.

At the moment, the proposed TBE approach supports the specification of BEFORE INSERT and AFTER INSERT triggers. In both cases, the TBE approach includes two steps:

1. Trigger (rule) specification, which includes defining the components of the ECA(A) rule, and
2. Trigger validation, which includes inserting nodes/relationships to activate the trigger and test its correctness.

## 4.1 Prerequisites

Before demonstrating the proposed TBE approach, we built a Neo4j graph database based on a real dataset available at https://www.kaggle.com/new-york-city/new-york-city-current-job-postings. The dataset contains information about New York City job postings retrieved from the official New York jobs site. In total, there are 3.238 rows in the downloaded CSV file. By analyzing the contents and structure of the dataset, we developed a property graph data model to be implemented in Neo4j, and used for validation purposes. A sample graph database entry represented as nodes and relationships is depicted in the property graph model shown in Figure 2.

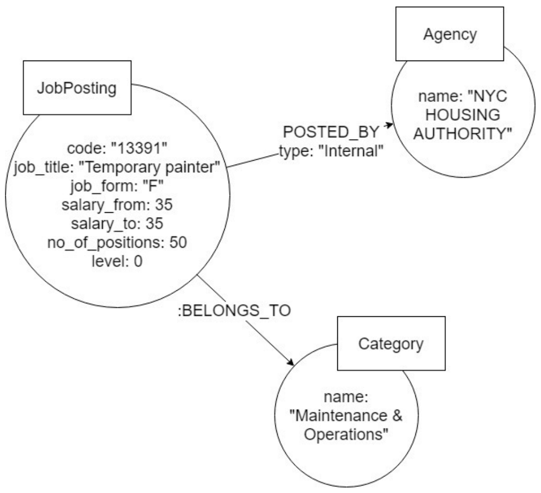


Figure 2: A sample Neo4j database entry represented through graph data model.

## 4.2 BEFORE INSERT trigger

The BEFORE INSERT trigger, which is activated before a new node/relationship is inserted into the database, can prevent users from inserting data inconsistent with business rules (e.g., the user can add a new node only if some other node is already present in the database).

Therefore, in the context of graph databases, to create a new node of a specific label, it is first necessary to check if there is a node of some other label with a given property value in the database.

As shown in Figure 3, the specification step of a BEFORE INSERT trigger includes selecting the label of a node, which will be monitored when inserting new nodes, and entering data about the node, which needs to exist in the database with a given property value in order to insert a node into the database. The ECAA rule specified through a graphical interface (Figure 3) can be visualized as a simple diagram shown in Figure 4.

The rule will ensure that the user cannot add a new job posting if there is no agency named ”NYC HOUSING AUTHORITY”, which was mentioned on a sample database entry shown in Figure 2. Also, note that node labels displayed as dropdown options are retrieved from the database by executing the following Cypher query:

```
MATCH (n) RETURN distinct labels(n)
```

In the underlying implementation, the created rule (trigger) will be saved in a global list of rules, and activated in the rule validation step, when a user tries to create a new node labeled Node1. To check if there is an existing node of a given label with given property value, before committing the database transaction to insert a node labeled Node1, the following program code of the method will be executed:

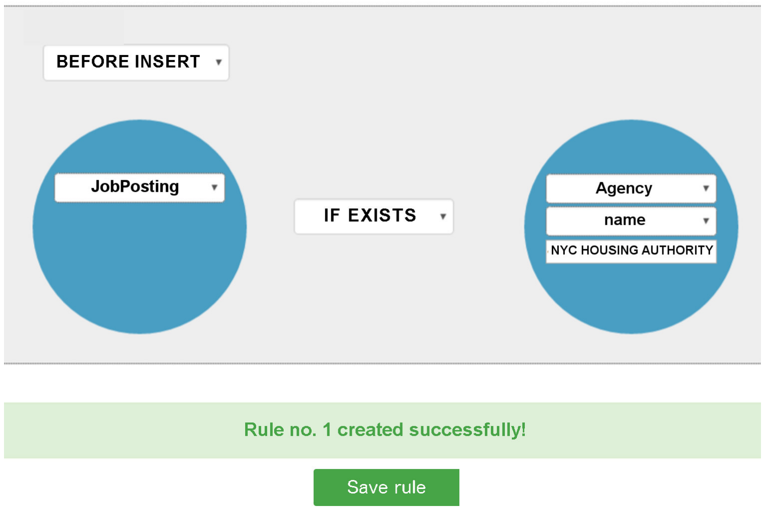
```
String query="MATCH (n: " + label + ")
    WHERE n." + property + "='" + value + "'
    RETURN n";
Result result = db.execute(query);
ResourceIterator<Node> resourceIterator =
    result.columnAs("n");

if(resourceIterator.hasNext())
{
    exists = true;
    result.close();
}
```

In the method, the Cypher query will return the given node if it exists as a Node object in the *ResourceIterator* iterator object. If the iterator has an element, i.e., the iterator is not empty, we can conclude that the conditional node with given property value exists. In this case, the value of variable *exists* indicates whether the condition is fulfilled or not, which affects the action component of the rule.

## 4.3 AFTER INSERT trigger

As its name indicates, AFTER INSERT trigger is activated after the INSERT event is detected and the specified condition fulfilled. In this paper, we specify



Rule no. 1 created successfully!

Figure 3: Creating a BEFORE INSERT trigger through graphical interface.

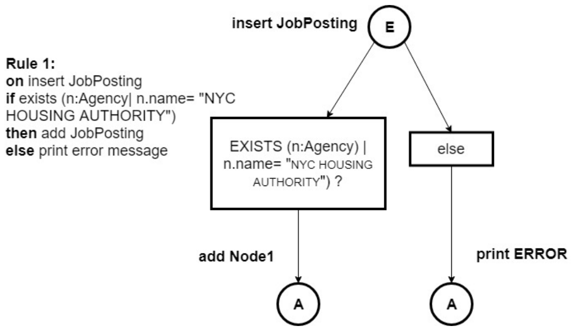


Figure 4: ECAA rule diagram of the BEFORE INSERT trigger.

an AFTER INSERT trigger, which automatically updates a node property value after a node with a given label is inserted into the database.

This kind of update operation is often used for automated value insertions for given properties (in this case, job level). Also, the AFTER INSERT trigger can be used when performing data aggregations in data warehouses, and help automatically maintain the values of aggregated nodes in graph databases after the values are aggregated by applying mathematical functions (e.g. SUM()).

As depicted in Figure 5, the AFTER INSERT trigger rule is specified by first entering the property value required for a given node label, followed by specifying which property value of a given node should be updated and to which value. Note that in this example, there were no alternative actions specified for this trigger, so the underlying rule model is ECA-based.

The specified rule for the AFTER INSERT trigger is also implemented as a method, which executes after committing the transaction, in which a given node is created in the database. Specifically, in this example, the method first retrieves the nodes specified in the rule by executing a Cypher query, which returns all nodes labeled JobPosting.

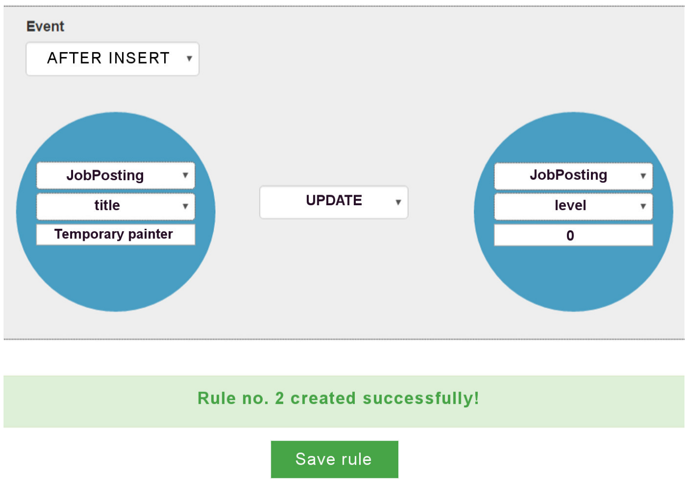
For each node with a given label the value of level property will be set to 0 after a new node labeled JobPosting with *title* property value ”Temporary painter” is inserted into the database by executing the following Cypher query:

```
MATCH (n:JobPosting {job_title: "Temporary
painter"}) SET n.level = 0
RETURN n
```

## 5 RESULTS

The rule validation step of the proposed TBE approach is performed by inserting nodes through the developed graphical interface. Given the database structure developed based on a real dataset and created rules (triggers) depicted in Figures 3 and 5, we entered node data, which could violate those rules (in case of the BEFORE INSERT trigger), or simply activate the rules (in case of the AFTER INSERT trigger).

The BEFORE INSERT trigger presented in Section 4.2 is created to ensure that a new node labeled JobPosting can only be inserted if there exists a node labeled Agency having *name* property value set to ”NYC HOUSING AUTHORITY”. At the initial database state, when a user tries to create such node, the condition of the ECAA rule is not satisfied, because there is no node labeled Agency with the given property value in the database. Hence, the alternative action of the rule is executed, which results in an error message displayed to the user (Figure 7).



Rule no. 2 created successfully!

Figure 5: Creating an AFTER INSERT trigger through graphical interface.

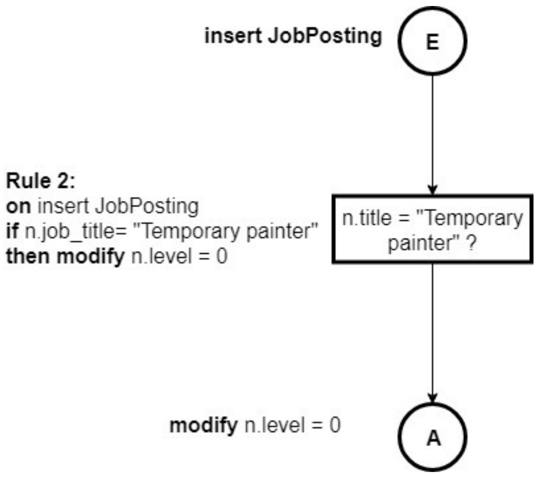
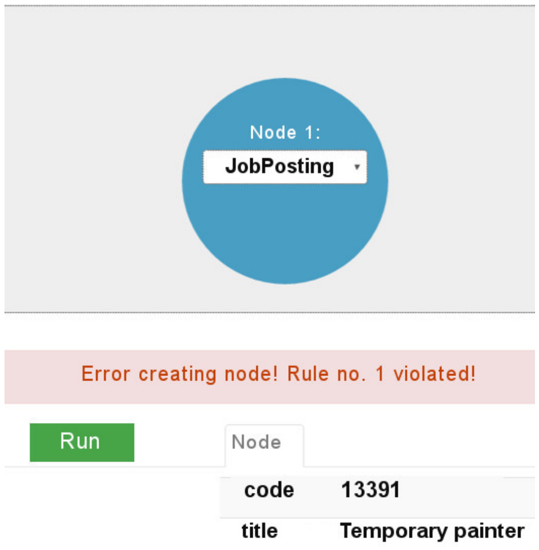


Figure 6: ECA rule diagram of the AFTER INSERT trigger.



Error creating node! Rule no. 1 violated!

| | |
|---|---|
| code | 13391 |
| title | Temporary painter |

Figure 7: Error message displayed when a trigger's rule condition is not fulfilled.

On the other side, the AFTER INSERT rule (trigger) is validated automatically when we insert a node labeled JobPosting with a *title* property value ”Temporary painter”. After the node insertion process is performed through the graphical interface, the action specified by the ECA rule is performed, which updates existing JobPosting node, and sets the node's *level* property value to 0. Hence, the AFTER INSERT trigger will result in updated values of *level* property to 0 for all job postings for ”Temporary painter” jobs.

## 6 DISCUSSION

As shown in the previous sections, the proposed TBE approach in graph databases does not bring any additional complexity to users when creating triggers in graph databases. It has already been mentioned that there is still no specific Cypher query language syntax dedicated to creating triggers. Therefore, our approach, which includes a graphical interface similar to e.g. Microsoft Access presents a simple and understandable way to create ECA(A) rules. Additionally, all queries specified through the interface executed in short time.

In this paper, we have demonstrated how the approach can be used to implement database rules on simple events, such as adding new nodes into the database. Since the underlying implementation includes methods, which check each ECA(A) rule component when the user tries to execute a database operation, our approach can be easily extended to also include complex events, such as SEQUENCE and NEGATION.

The SEQUENCE event operator can be used to check if several events occurred in a predefined sequence. In terms of graph databases, the SEQUENCE operator could be used to ensure that specific action(s) are automatically performed as a reaction to the event when several nodes are created in a specified order. For instance, if a student fails an exam in the first course (E1), second course (E2) and third course (E3), this would mean that he is most likely not going to finish the semester (A). However, such case could be easily modeled as an ECA(A) rule for a SEQUENCE operator and implemented by using our TBE approach.

Furthermore, the NEGATION event operator can also be important in graph databases because it can prevent certain anomalies. For instance, once a user creates two nodes, but fails to create a relationship between these nodes, the NEGATION operator could be used to notify the user that the relationship has not been created between the nodes.

Our future work includes extending the proposed TBE approach to support trigger specification for relationships and complex events, such as SEQUENCE and NEGATION. Also, we will conduct a more detailed query performance test to gain insight into how much the TBE approach affects the query time execution with the increasing complexity of events (rules) implemented.

## 7 CONCLUSIONS

In this paper, we discussed the need for building active database systems able to automatically react to certain events and perform different actions. Triggers as active mechanisms still face a low level of support in graph databases with limited features. Current research approaches to build graph-based rule engines require users with different levels of knowledge and expertise to be familiar with query language syntax, which may require a certain learning period from the users. Hence, in this paper, we propose to use Trigger-By-Example approach/language for rule specification in graph databases. We developed a prototype implementation of the proposed approach as a layer built on Neo4j GDBMS, which enables users to specify triggers as active rules. At the moment, our implementation supports handling simple events, such as INSERT, UPDATE and DELETE, so, as part of our future work, we plan to extend this support on complex events, such as SEQUENCE and other. The proposed TBE approach can be used to perform certain actions in graph databases automatically, such as node data aggregation or integrity enforcement mechanism.

## REFERENCES

Ceri, S. (1992). A declarative approach to active databases. In *[1992] Eighth International Conference on Data Engineering*, pages 452–456. IEEE.

Dayal, U., Hanson, E., and Widom, J. (1994). Active database systems. Technical report, Stanford InfoLab.

De Marzi, M. (2015). Triggers in neo4j. Available at https://maxdemarzi.com/2015/03/25/triggers-in-neo4j/.

DSG, U. (2017). Graphflow. Available at http://graphflow.io/.

Grüner, S., Weber, P., and Epple, U. (2014). Rule-based engineering using declarative graph database queries. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 274–279. IEEE.

Herbst, H. (1996). Business rules in systems analysis: a meta-model and repository system. *Information Systems*, 21(2):147–166.

JanusGraph (2017). Transaction log. Available at https://docs.janusgraph.org/latest/log.html.

Kangsabanik, P., Mall, R., and Majumdar, A. K. (1997). A technique for modeling applications in active object oriented database management systems. *Information Sciences*, 102(1-4):67–103.

Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., and Salihoglu, S. (2017). Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM.

Lee, D., Mao, W., Chiu, H., and Chu, W. W. (2000a). Tbe: A graphical interface for writing trigger rules in active databases. In *Advances in Visual Information Management*, pages 367–386. Springer.

Lee, D., Mao, W., Chiu, H., and Chu, W. W. (2005). Designing triggers with trigger-by-example. *Knowledge and information systems*, 7(1):110–134.

Lee, D., Mao, W., and Chu, W. W. (2000b). Tbe: Trigger-by-example. In *International Conference on Conceptual Modeling*, pages 112–125. Springer.

Neo4j, I. (2018). opencypher. Available on May 15, 2019 at https://www.opencypher.org/about.

Nowak, M., Bak, J., and Jedrzejek, C. (2012). Graph-based rule editor. In *RuleML (2)*.

Rapsevicius, V. and Juska, E. (2014). Expert system for the lhc cms cathode strip chambers (csc) detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 738:126–131.

Widom, J. and Ceri, S. (1996). *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann.