Cypher by Example: A Visual Query Language for Graph Databases

Kornelije Rabuzin[1][a], Maja Cerjan[1][b] and Martina Šestak[2][c]
[1]*Department of Theoretical and Applied Foundations of Information Sciences, University of Zagreb, Faculty of Organization and Informatics, Pavlinska 2, 42000 Varaždin, Croatia*
[2]*Information Systems Laboratory, University of Maribor Faculty of Electrical Engineeing and Computer Science, Koroška cesta 46, 2000 Maribor, Slovenia*

Keywords: The Graph Databases, NoSQL, Query by Example, Cypher, Query Languages.

Abstract: Considering the increasing connectivity in data exchange between applications and devices, modern IT systems need a storage solution capable of handling connections and patterns between entities. Graph databases emerged as a potential solution in the past decade. Since graph database query language standardization is ongoing, users interact with graph databases using query languages like Cypher or Gremlin, supported by modern Graph Database Management Systems (GDBMSs). Despite well-documented syntax, users with little knowledge of graph databases face a steep learning curve before writing queries on their own data. This limits interest in implementing graph databases due to the lack of a visual tool for maintenance. To address this, the paper introduces *Cypher by Example*, a visual graph query language with an interface and query patterns for interacting with the database. It presents the basic elements of this query language and demonstrates its usefulness in two use cases.

1 INTRODUCTION

The rising connection of data across modern IT systems has made graph databases a popular solution since they allow first-class relationship entities for effective pattern-based analysis such as fraud detection[1]. The main obstacle to increased use remains the absence of a standard query language similar to SQL. OpenCypher and GSQL and PGQL and G-CORE make attempts to create a Graph Query Language (GQL) standard[1] but the standard remains in development.

Until a standard Graph Query Language (GQL) emerges users of graph databases depend on languages like Cypher and Gremlin which major Graph Database Management Systems (GDBMSs) support. Despite its SQL-like declarative syntax Cypher experiences widespread adoption yet developers need to overcome a steep learning curve. Different researchers have developed alternative query methods which reduce the learning burden of syntax through their implementation.

The visual querying system known as Query by Example (QBE) was originally designed for relational database use. The developers of SQL built the language to be user-friendly yet the system became too complicated for average users. Microsoft Access and other database systems employed QBE as a query building tool to create visual interfaces for database interactions.

Based on the QBE visual querying method we suggest Cypher by Example (CBE) as a visual language for graph databases. The system transforms user input queries into Cypher syntax while enabling users to build graph queries through its graphical interface.

The main contributions of this paper are:

- Cypher by Example (CBE) presents a visual query language for graph database operations;

[a] https://orcid.org/0000-0002-0247-669X
[b] https://orcid.org/0009-0003-8825-6050
[c] https://orcid.org/0000-0001-7054-4925

[1] More information available at https://www.gqlstandards.org

Rabuzin, K., Cerjan, M., Šestak and M.
Cypher by Example: A Visual Query Language for Graph Databases.
DOI: 10.5220/0013565400003967
In *Proceedings of the 14th International Conference on Data Science, Technology and Applications (DATA 2025)*, pages 518-525
ISBN: 978-989-758-758-0; ISSN: 2184-285X

- The system architecture of CBE includes its design as well as graph query pattern implementation;
- A user-friendly graphical user interface (GUI) enables users to formulate visual graph queries;
- CBE usability is tested through data insertion and retrieval case studies.

The paper follows this organization: Section 2 provides an overview of visual query systems that are relevant to the topic. The paper provides background information about relational and graph database querying in Section 3. Section 4 describes the CBE language. Section 5 showcases CBE implementation through practical applications. The paper ends with future work and discussion in Section 6.

2 RELATED WORK

Researchers have proposed various database query systems together with Visual Query Systems (VQSs). The paper investigates query formulation since this continues to be the primary difficulty for unskilled graph database users.

Three research directions for visual query formulation were identified by Bhowmick Choi and Li (2017) as the edge-at-a-time method of adding single edges at a time and the pattern-at-a-time method of dragging subgraphs and the Query by Example (QBE) method of using example data for query formulation. The Query by Example (QBE) approach functions as the fundamental basis for the Cypher by Example (CBE) language we propose.

Pabon et al. (2019) presented GraphTQL as a VQS system that enables filtering operations and schema transformation. GraphTQL enhances query formulation but it offers fewer operators than CBE does because it provides extensive query functionality.

AutoG represents a visual query auto-completion framework that Yi along with Choi and Bhowmick and Xu (2016) created. The system includes three main components which are a user interface for input and server-side data indexing and query suggestion functionality. AutoG makes usability better but the system fails to demonstrate the process of converting visual queries into executable Cypher queries.

DAVINCI represents a visual interface which Zhang and his team (2015) developed for constructing subgraph queries. The system extracts patterns from existing graph data before offering the extracted subgraph patterns for user selection. VISAGE represents a system that Pienta et al. (2016, 2017) developed for pattern-based querying along with wildcards and logical operators and GUI-based query construction through a complete client-server system.

Bhowmick, Choi, and Zhou (2013) developed the VOGUE framework by integrating query formulation directly with processing through action-aware indexing which reduces system delays.

Jin et al. (2011) developed GBLENDER as a system which refreshes query results in real-time while users construct queries. GBLENDER creates index structures for subgraph elements to execute query matches through interactive GUI-based operations.

Sharma (2020) presents a hybrid method by showing how GQL syntax enables SQL query translation into graph queries. The early-stage research demonstrates the requirement for an SQL-like graph database query language standard.

The majority of systems focus on visual query formulation yet they restrict their approach to property graph-specific QBE-like methods. Our proposed CBE language targets this specific gap which remains unaddressed.

3 RELATION AND GRAPH DATABASES: SQL, QBE AND CYPHER

Relational databases dominate the market as the leading data management technology since their inception. The Query by Example (QBE) language joins SQL as one of the main methods for database querying operations. Zloof (1977) defines QBE as "a high-level database management language which offers a uniform and simple method to query, modify, create and manage relational databases." The system focuses on making query construction accessible to users who do not understand database principles or syntax.

Through visual tools QBE creates interfaces that follow the user's cognitive patterns during the query process. The system divides database components into two groups: constant elements derived from the database structure and remain unalterable (such as table names and column names) and example elements that users modify to specify their search criteria (such as particular column values). The interface structure enables users to illustrate their desired outcomes while the system creates the corresponding query syntax.

Relational databases can be queried by using the Query by Example (QBE) language which functions as a SQL alternative to simplify database interactions. Users can create queries in QBE by adding example values to construct their queries instead of writing SQL syntax.

Microsoft Access demonstrates this approach. Users can obtain course information with their prerequisites by creating multiple copies of the "courses" table during visual query construction to specify required attributes. This approach eliminates the requirement for intricate SQL syntax involving multiple joins because Figure 1 demonstrates this process.

The process of building a QBE query for courses and their prerequisites reaches its second level (Figure 1.)

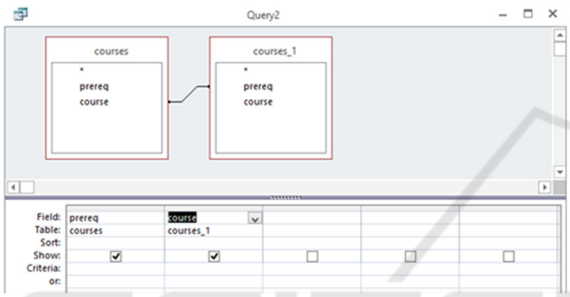

Figure 1: Query formulation for courses and their prerequisites in QBE – the second hierarchy level.

SQL syntax becomes excessively difficult for untrained users to manage as query complexity reaches deeper course hierarchy levels. Recursive queries or nested joins are often required. QBE retains its easy-to-use nature because users can increase the number of visible elements without needing to modify SQL syntax manually. Unstructured and semi-structured data sources including application logs and IoT outputs have increased in popularity thus making fixed-schema databases impractical during recent years. The market transition toward NoSQL technologies occurred because of graph databases which provide flexible schemas through alternative query languages.

The property graph data model serves as the core data model in graph databases which provide both structure and adaptability in data representation. Angles (2018) defines a property graph formally through the tuple PG=(N,E,η,λ,ν), where:

- *$N \subseteq O$ represents a finite set of objects called nodes;*
- *$E \subseteq O$ represents a finite set of objects called edges;*
- *$\eta : E \rightarrow N \times N$ is a total function that assigns an ordered pair of nodes to each edge;*
- *$\lambda : N \cup E \rightarrow P(L)$ is a partial function that assigns a finite set of labels to each node or edge object;*
- *$\nu : (N \cup E) \times K \rightarrow V$ is a partial function that assigns a value to a property of a node or edge object*

A property graph contains nodes and edges that both receive labels and optional key-value properties which describe their features.

The property graph schema, as described by Angles (2018), is a formal construct used to define valid structures within a property graph. It is represented as a tuple $PG_S = (L_N, L_E, \beta, \delta)$, where:

- *$L_N \in L$ is a finite set of labels representing node types;*
- *$LE \in L$ is a finite set of labels representing edge types, where $LN \cap LE = \emptyset$;*
- *$\beta : (LN \cup LE) \times P \rightarrow T$ is a partial function that defines the properties for node and edge types, and the data types of the corresponding values;*
- *$\delta : (LN, LN) \rightarrow SET+(TE)$ is a partial function that defines the edge types allowed between a given pair of node types.*

This model serves as the basis for graph modelling yet maintains flexibility through the absence of rigid structural constraints that relational schemas enforce. Šestak et al. (2021) introduce advanced features including integrity constraints and validation rules that this schema enables through extensions. Most graph databases enable users to interact with property graphs through query languages that include Cypher and Gremlin. The declarative syntax of Cypher enables users who understand SQL to easily use this language for graph queries. GQL is expected to become the future standardized Graph Query Language which will be based on Cypher. The procedural nature of Gremlin makes it suitable for detailed graph traversal operations but it demands advanced knowledge of the domain. Rabuzin, Maleković and Šestak (2016) have introduced a QBE-like method for using Gremlin. The research demonstrates our visual query interface through Cypher. Our Cypher by Example (CBE) system enables users to create graph queries through visual methods instead of manual Cypher query writing and automatically converts these queries into Cypher syntax. This method enables easier learning while maintaining the complete power of graph querying capabilities.

4 CYPHER BY EXAMPLE

In this section, we propose and describe the Cypher by Example visual query language for graph databases. We describe the system architecture of CBE followed by a visual syntax of the CBE language and supported operators, as well as how those operators map to the underlying Cypher queries to be executed within a graph database.

4.1 System Architecture of the CBE Language

The Cypher by Example (CBE) system implements a client-server system which includes graphical user interface components on the client side and server components that interact with the Neo4j graph database. The client contains a data-driven Graphical User Interface (GUI) which enables users to both create and display queries as shown in Figure 2.

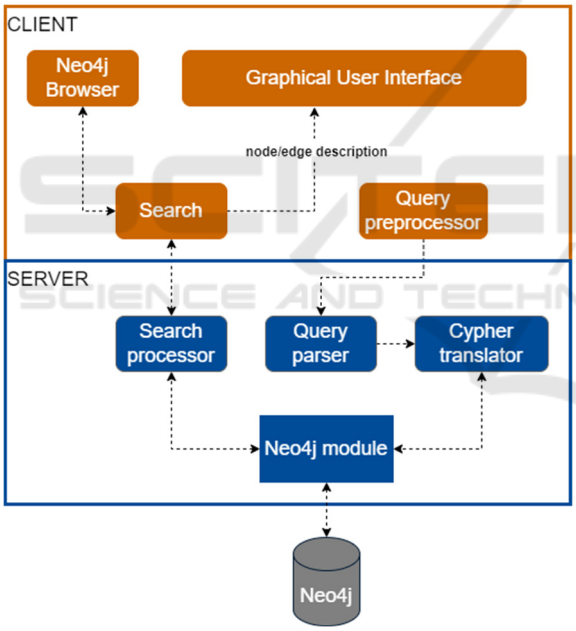

Figure 2: System architecture of the CBE language.

The Search module retrieves graph database metadata (node and edge labels and properties) which allows the GUI to update its interface dynamically. The Query Preprocessor transforms the user's visual query into a structured JSON object before sending it to the server for parsing.

The Search Processor component on the server retrieves metadata by running basic Cypher MATCH queries against the Neo4j database. The Query Parser takes user-generated JSON queries before sending them to the Cypher Translator for conversion into executable Cypher syntax. The Neo4j Module executes the final query through the Neo4jClient library for.NET. The GUI displays results after the architecture receives them from the database execution.

The system used ASP.NET for web application development with Neo4j v3.5 as the database platform and the following sections will provide additional implementation details.

4.2 Supported Graph Query Patterns in the CBE Language

The CBE language enables users to execute basic visual query patterns which directly translate to Cypher queries. Users can use these patterns to perform data insertion and updates and data retrieval operations on graph databases through the interface without needing to write code. The query process begins with retrieving node labels and edge types and properties which then fills the visual interface for additional query development.

Retrieve Node/Edge Metadata

This pattern retrieves the graph structure which includes node and edge labels together with their properties before enabling any additional patterns. Users choose their preferred label through a dropdown selection which reveals properties for modification or filtering in a table.

```
MATCH p=(n)-[e]-()
RETURN DISTINCT LABELS(n) AS node_labels,
PROPERTIES(n) AS node_properties,
TYPE(e) AS edge_types, PROPERTIES(e) AS edge_properties
```

Insert/Update Nodes and Edges

The system enables users to perform both node and edge insertions as well as updates for single nodes or entire paths that include edges. The system enables users to enter properties in any order because it does not enforce a predefined schema structure. The Cypher MERGE command serves both node and edge insertions to maintain data consistency and prevent duplicate entries.

Example for a single node:

```
MERGE (n:NodeLabel {property_name: "value"}) RETURN n
```

Query Nodes and Edges

Users can use these patterns to retrieve graph elements through adaptable retrieval criteria. Users

can perform the following operations in both scenarios:

- Set filter *conditions (e.g., =, <, >),*
- *Include/exclude properties from the result,*
- *Sort results, and*
- *Limit the number of r*eturned records.

Example for querying a node:

```
MATCH (n:NodeLabel)
WHERE n.property_name = "value"
RETURN n.property_name
ORDER BY n.property_name ASC
LIMIT 10
```

For path queries between nodes:

```
MATCH (n1:NodeLabel)-[e1:EdgeType]-
>(n2:NodeLabel)
WHERE n1.property_name = "value"
RETURN n1, e1, n2
ORDER BY n1.property_name
LIMIT 10
```

The fundamental graph query patterns described here serve as the foundation for complex graph operations which receive detailed explanation through practical examples in the following section.

5 CASE STUDY

In this section, we demonstrate the usefulness and the applicability of the proposed CBE language on two use cases, which both include the application of graph query patterns explained in Section 4.2. For the case study, we prepared a sample graph database containing data about users, books and their authors, as well as users' book borrowings. Throughout the use cases, we showcase the usage of the CBE graphical interface when adding data to and retrieving data from the Neo4j graph database.

5.1 Overview of the CBE Prototype Interface

Before showcasing the implementation of selected graph query patterns in practical use cases, we provide a brief overview of the CBE GUI elements, which the user uses to visually formulate graph database queries. The architecture of the CBE language presented in Section 4.1 was implemented as a web application consisting of client- and server-side modules. A simplified overview of the main elements of the CBE GUI is depicted in Figure 3.

The interface consists of three major segments:

1. Graph query patterns list - a menu with listed supported graph query patterns for query formulation;
2. Query formulation editor - a surface, which can be used to add new graph elements (nodes or edges) to the query path. Once added to the editor, the user selects the node label or edge type from a dropdown list available for each element;
3. Query parametrization table - a table, that the user fills to parametrize the query by using different options to adjust specific node or edge property values (Show, Sort, Criteria or Limit).

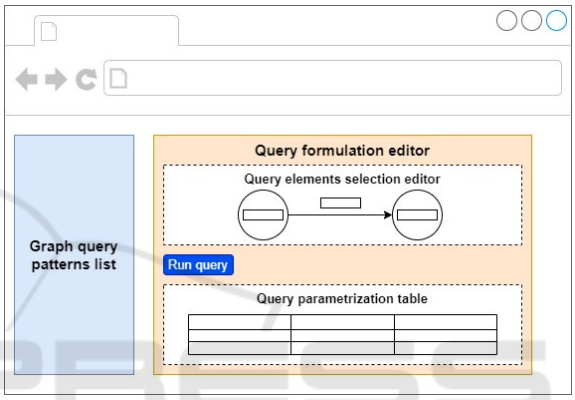

Figure 3: A mockup overview of the CBE GUI elements.

5.2 Use Case 1: Creating Nodes and Edges

The first use case demonstrates how to use the CBE language to insert a node and an edge into the graph database.

To create a new **User** node, the user selects the desired node label from a dropdown list (Figure 4).

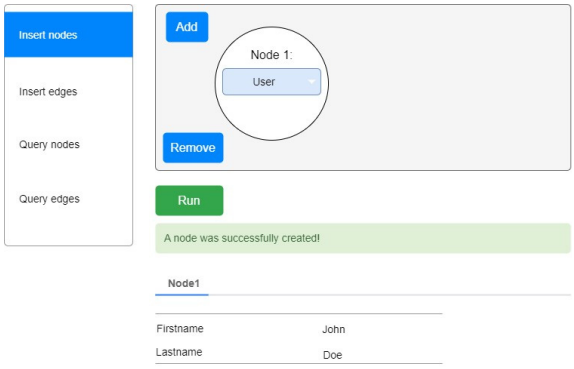

Figure 4: An example of inserting a User node into the database via the CBE interface.

The *Retrieve node metadata* pattern then populates the table with available properties. After entering values for fields like *Firstname* and *Lastname*, the user executes the query.

On the backend, the Neo4j module—built using the Neo4jClient library for .NET—translates the user's input into a Cypher MERGE query and executes it against the database.

Next, to create a WROTE relationship between two nodes (e.g., *Author* and *Book*), the user adds both node types and the edge type to the query editor (Figure 5).

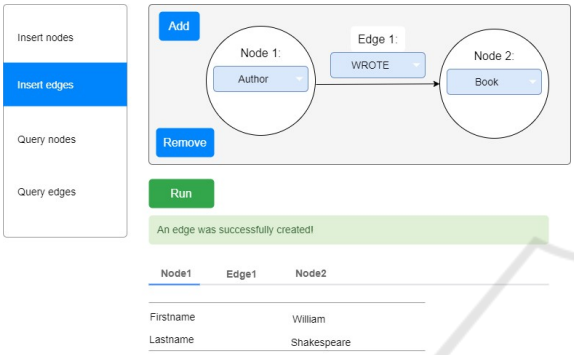

Figure 5: An example of adding a WROTE edge between two nodes.

After setting relevant properties (e.g., author name, book title), the query is executed. If either node doesn't exist, it is created alongside the connecting edge (Figure 6).

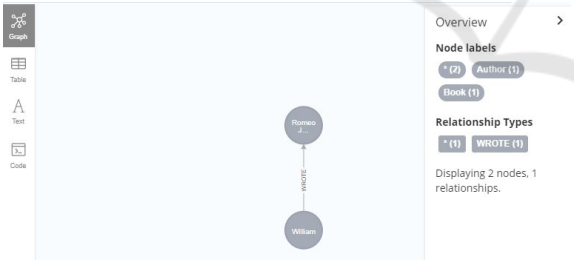

Figure 6: A new WROTE edge is added into the database.

The interface also allows adding more than one edge in a single interaction, enabling the creation of full **graph paths** between multiple nodes (Figure 7).

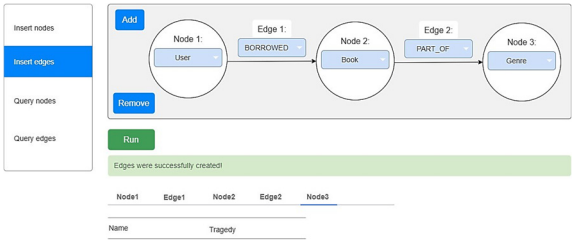

Figure 7: An example of the two edges into the database.

These examples confirm that the CBE interface enables intuitive insertion of both nodes and relationships in the graph database through a visual workflow.

5.3 Use Case 2: Querying Nodes and Edges

After inserting nodes and edges into the database the CBE interface can be used to query the data visually following the QBE philosophy. Users will specify example values of the elements they wish to retrieve and will also set display parameters, sorting, filtering and result limits. In order to demonstrate the "Query a single node" pattern, assume that the user wants to retrieve User nodes. First, the user chooses the User label in the query editor. Then, the Retrieve node/edge metadata pattern populates the table with the properties that are available. The user chooses to display the Firstname and Lastname properties, sorts the Firstname property in ascending order, sets a result limit of 4, and then runs the query (Figure 8).

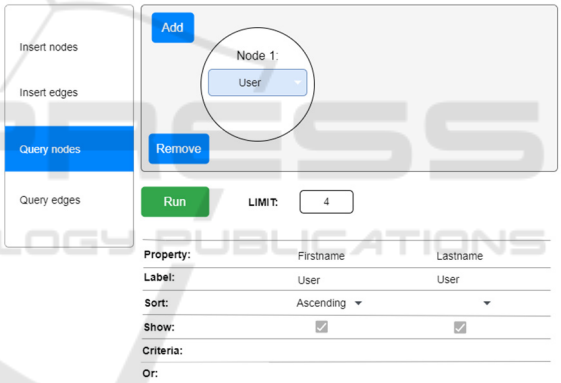

Figure 8: An example formulation of a query on a single node.

The interface generates the following Cypher query:

```
IEnumerable<User> result =
Neo4jDb.Instance.Client.Cypher
  .Match("(u:User)")
  .Return(u => u.As<User>())
  .OrderBy("u.Firstname")
  .Limit(4)
  .Results.ToList<User>();
```

The result is a table of four users, sorted and filtered according to the parameters (Figure 9).

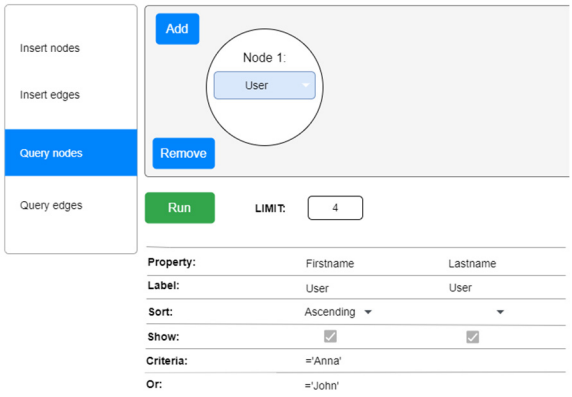

Figure 9: The results of the query for retrieving for users from the database.

Further, the user can specify filter criteria. For example, to return users with the first name "Anna" or "John", values are entered in the *Criteria* and *Or* fields of the *Firstname* row (Figure 10).

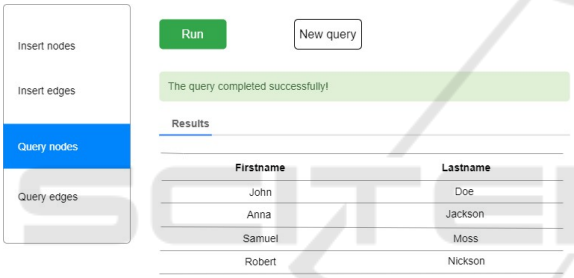

Figure 10: An example formulation of a query on a single node with filtering criteria.

This adjusts the query as follows:

```
IEnumerable<User> result = Neo4jDb.Instance.Client.Cypher
  .Match("(u:User)")
  .Where((User u) => u.Firstname == "Anna")
  .OrWhere((User u) => u.Firstname == "John")
  .Return(u => u.As<User>())
  .OrderBy("u.Firstname")
  .Limit(4)
  .Results.ToList<User>();
```

The edge querying pattern works similarly. For instance, to get users who borrowed books, the user will choose the User, Book, and BORROWED labels in the query editor. The metadata query will fill in the properties for each element. The user will apply filters to Firstname, choose display options for both nodes and the edge (e.g., Date borrowed), and also choose to limit results to 5. The corresponding query is:

```
IEnumerable<Object> result = Neo4jDb.Instance.Client.Cypher
  .Match("(user:User)-[r:BORROWED]->(book:Book)")
  .Where((User user) => user.Firstname == "Anna")
  .OrWhere((User user) => user.Firstname == "John")
  .Return((user, book, r) => new {
    User = user.As<User>(),
    Book = book.As<Book>(),
    R = r.As<BORROWED>()
  })
  .OrderBy("user.Firstname")
  .Limit(5)
  .Results.ToList<Object>();
```

The result displays users and their corresponding borrowings (Figure 12), and the full formulation process is illustrated in Figure 11.

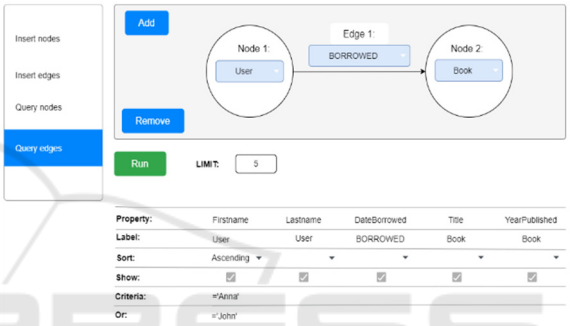

Figure 11: An example formulation of a query on an edge between two nodes.

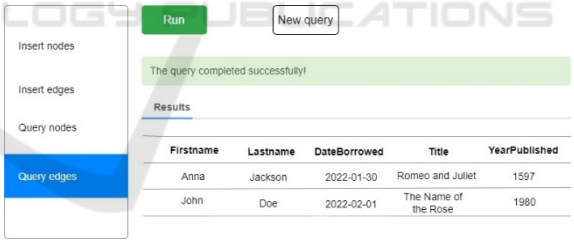

Figure 12: The results of the query for retrieving book borrowings from users named "John" or "Anna".

These examples show that the CBE interface allows users to visually formulate both node and edge queries in the same way as QBE, with real-time Cypher translation and execution against the graph database.

6 CONCLUSION

In this paper, a novel Cypher by Example (CBE) visual graph querying language was proposed. The proposed language follows the design principles of the Query by Example (QBE) language approach,

which has already been used for years as an alternative graphical query language for relational databases. The strong point of the QBE approach is its ability to present inexperienced users with a clear and simple graphical interface for query formulation, thus eliminating the need for a deep understanding of the underlying query language syntax. Our proposed CBE language follows the same thought, and provides a visual graph querying interface for graph database practitioners with little or no knowledge of the Cypher query language syntax or the graph database technology in general. However, unlike other visual graph querying approaches introduced so far, the CBE language follows the QBE design principles strictly, and allows users to adjust graph query parameters in more detail. We described the system architecture of the proposed CBE language, and discussed the currently supported graph query patterns, which can be used to formulate queries via the CBE language interface. Furthermore, we performed a case study on two use cases to demonstrate the usability of the proposed CBE language for inserting and querying nodes and edges in graph databases. As part of our future work, we plan to extend the list of supported graph query patterns to more complex graph structures such as subgraphs as well as other operators for managing different graph database structures (e.g. adding new node labels and edge types, deleting nodes/edges, creating indexes, triggers, stored procedures, etc.). We will also continue our work on improving the visual interface by integrating an appropriate auto-complete framework to additionally simplify the query formulation process for the user.

ACKNOWLEDGMENTS

This work was funded by the Slovenian Research Agency (Research Core Funding No. P2-0057).

REFERENCES

Angles, R. (2018). The Property Graph Database Model. In *Proceedings of AMW 2018*.

Bhowmick, S. S., & Choi, B. (2022). Data-driven visual query interfaces for graphs: Past, present, and (near) future. In *Proceedings of the 2022 International Conference on Management of Data* (pp. 2441–2447). ACM.

Bhowmick, S. S., Choi, B., & Li, C. (2017). Graph querying meets HCI: State of the art and future directions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1731–1736). ACM.

Bhowmick, S. S., Choi, B., & Zhou, S. (2013). Vogue: Towards a visual interaction-aware graph query processing framework. In *Proceedings of CIDR 2013*.

Jin, C., Bhowmick, S. S., Xiao, X., Choi, B., & Zhou, S. (2011). Gblender: Visual subgraph query formulation meets query processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (pp. 1327–1330). ACM.

Pabon, M. C., Millan, M., Roncancio, C., & Collazos, C. A. (2019). Graphtql: A visual query system for graph databases. *Journal of Computer Languages, 51*, 97–111.

Pienta, R., Hohman, F., Tamersoy, A., Endert, A., Navathe, S., Tong, H., & Chau, D. H. (2017). Visual graph query construction and refinement. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1587–1590). ACM.

Pienta, R., Tamersoy, A., Endert, A., Navathe, S., Tong, H., & Chau, D. H. (2016). Visage: Interactive visual graph querying. In *Proceedings of the International Working Conference on Advanced Visual Interfaces* (pp. 272–279). ACM.

Rabuzin, K., Maleković, M., & Sestak, M. (2016). Gremlin by example. In *Proceedings of the International Conference on Advances in Big Data Analytics* (pp. 144–149).

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data*. O'Reilly Media, Inc.

Sestak, M., Heričko, M., Družovec, T. W., & Turkanović, M. (2021). Applying k-vertex cardinality constraints on a neo4j graph database. *Future Generation Computer Systems, 115*, 459–474.

Sharma, C. (2020). Flux: From SQL to GQL query translation tool. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1379–1381). IEEE.

Yi, P., Choi, B., Bhowmick, S. S., & Xu, J. (2016). Autog: A visual query autocompletion framework for graph databases. *Proceedings of the VLDB Endowment, 9*(13), 1505–1508.

Zhang, J., Bhowmick, S. S., Nguyen, H. H., Choi, B., & Zhu, F. (2015). Davinci: Data-driven visual interface construction for subgraph search in graph databases. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering* (pp. 1500–1503). IEEE.

Zloof, M. M. (1977). Query-by-example: A data base language. *IBM Systems Journal, 16*(4), 324–343.