Towards Inheritance in Graph Databases

Kornelije Rabuzin, Martina Šestak
Faculty of Organization and Informatics
University of Zagreb
Varazdin, Croatia
e-mail: {kornelije.rabuzin, msestak2}@foi.hr

***Abstract*—Object-oriented database management systems represent a technology in which the concepts of the object-oriented programming paradigm are implemented in databases. The impedance mismatch problem between different technologies (e.g., relational and object-oriented) has represented a challenge for developers and researchers for many years now. With the growing popularity of graph databases, it is important to resolve differences between the concepts of graph theory (used in graph databases) and the object-oriented paradigm. In this paper, we propose an approach for representing graph nodes as objects, and introduce the concept of node inheritance in graph databases, which we demonstrate on a simple showcase.**

Keywords-object-oriented paradigm; instantiation; inheritance; graph databases; Neo4j; nodes; gremlin by example

I. INTRODUCTION

The object-oriented (OO) paradigm represents an approach which focuses on modular component design and software reuse [1]. Classes and objects represent basic concepts behind this paradigm. Classes can be defined as "extensible templates for creating objects, providing initial values for instance variables and the bodies for methods" [2]. On the other hand, an object represents "a concrete entity based on a class, and is sometimes referred to as an instance of a class" [3]. Thus, a class "is like a blueprint. It defines the data and behavior of a type" [3].

When the concepts of the object-oriented paradigm are implemented in a database management system (DBMS), such a system is then called an object-oriented database management system (OODBMS). However, storing and retrieving data from databases by using OO concepts is not always trivial, since data representations can vary between the two technologies (this problem is called impedance mismatch). In the case of relational databases, this issue was solved by introducing object-relational mappers (ORMs) and object-relational database management systems (e.g., PostgreSQL), which can only be seen as a transitional solution for the problem. Moreover, this problem occurs with other database technologies as well, one of which are graph databases.

In general, OO systems support several reusability mechanisms, which enable developers to use the existing components when developing new components [4]. For the purpose of this paper, we will only mention the instantiation and inheritance mechanisms. The instantiation mechanism enables developers to use the same object definition to generate new objects with the same structure and behavior [5]. Inheritance represents one of the pillars of the OO paradigm (along with polymorphism and encapsulation), and refers to using and extending a definition of an already defined class [6], i.e., deriving data and behavior from the base class [3]. To manage inheritance hierarchies and represent the object database model in OODBMSs, several approaches have been introduced over the years, some of which proposed a graph-based model for the purpose, and will be discussed later in the paper.

On the other hand, the growing amount of highly connected and complex data has led to the development of graph databases as a separate category of NoSQL databases. In graph databases, data is stored and represented as a graph, so different graph-related algorithms can be applied when querying graph databases.

In this paper, we are going to explore basic characteristics of both standard OODBMSs and graph databases. This research is implemented for the purpose of incident command system (ICS). In incident command system different types of resources are available, and some resources are similar. To represent different types of resources and their connections, graph databases could be used. To enable easier management of resources in graph databases, we extend the research and we propose and integrate OO inheritance mechanisms with graph databases. The main contribution of this paper is to provide a theoretical basis for introducing the instantiation and inheritance mechanisms to graph databases.

The rest of the paper is organized as follows: in Section 2, basic concepts of the OO paradigm will be discussed in the context of OODBMSs in general and in the context of relevant features available in PostgreSQL. In Section 3, graph databases and the underlying graph theory will be explained. Section 4 contains the description of several existing approaches regarding representing both object-oriented data model as a graph, and vice versa. In Section 5, the proposed approach for implementing basic OO concepts (class, object, instantiation, inheritance) in graph databases is discussed. Finally, the possibilities for extending this paper and future research will be discussed.

II. OBJECT-ORIENTED CONCEPTS IN DATABASE MANAGEMENT SYSTEMS

A standard OODBMS should provide support for creating classes, objects as class instances, for creating inheritance hierarchies, and to call methods for accessing

978-1-5386-3656-5/18/$31.00 ©2018 IEEE

objects [7], where each object is identified through a unique ID called OID. According to [8], an OODBMS supports the concepts of subtyping and inheritance.

The instantiation mechanism is implemented by creating new objects as base class instances (objects then have the same properties and methods as the base class). Conversely, the inheritance mechanism is usually implemented as structural subtyping [9], whereas the behavioral subtyping is a more complex process, which requires a number of preconditions to be met.

Over the years, some OO concepts have been implemented in relational database management systems (RDBMSs), thus creating object-relational database management systems (e.g., PostgreSQL).

Specifically, table inheritance is supported by several DBMS systems. For instance, IBM's Informix supports table inheritance for tables that are defined on named row types [10]. They also list the benefits of table inheritance [10]:

- It encourages modular implementation of your data model.
- It ensures consistent reuse of schema components.
- It allows you to construct queries whose scope can be some or all of the tables in the table hierarchy.

One example is given in Fig. 1.

```
CREATE TABLE person OF TYPE person_t
(PRIMARY KEY (name))
FRAGMENT BY EXPRESSION
name < 'n' IN dbspace1,
name >= 'n' IN dbspace2;

CREATE TABLE employee OF TYPE
employee_t
(CHECK(salary > 34000))
UNDER person;
```

Figure 1. An example of table inheritance [9]

```
CREATE TABLE cities (
    name          text,
    population    float,
    altitude      int      -- in feet
);

CREATE TABLE capitals (
    state         char(2)
) INHERITS (cities);
```

Figure 2. An example of table inheritance in PostgreSQL [10]

Different objects can be inherited, including constraints, triggers, indexes, etc. In PostgreSQL, inheritance is supported in two forms [11].

The first form is table inheritance (shown in Fig. 2); table "capitals" inherits table "cities", but it is also possible that one table inherits several other tables. When writing queries, one can get the rows from the table, as well as the rows from the table extended by the rows from all tables that inherit the table (such behavior is the default behavior). The word ONLY used in the clause FROM will return only the rows from the table, and exclude the rows from its descendants.

Another option is known as CREATE TABLE … LIKE … After LIKE one has to specify the table which columns, data types and NOT NULL constraints should be copied [11].

III. Graph Databases Characteristics

Recently, it has become clear that social networks, traffic infrastructure, network infrastructure, etc. represent good candidates to be modeled as graphs.

Graph contains nodes, i.e., vertices (points A, B, C, D) and relationships (lines) between the vertices, i.e., edges (Fig. 3). More formally, a simple graph G has a set of vertices V(G) as well as the set of edges E(G). Each edge that belongs to E(G) is a pair of elements that belong to V(G); in simple graphs pair of elements are distinct as loops are not part of simple graphs, as well as multiple edges. Also, in graph theory there is a question whether two graphs are the same. This is known as isomorphism.

The importance of directed graphs is crucial for graph databases. "A directed graph, or digraph, D consists of a non-empty finite set V(D) of elements called vertices, and a finite family A(D) of ordered pairs of elements of V(D) called arcs" [12]. It is important to have in mind that "the ordering of the vertices in an arc is being indicated by an arrow" [12].

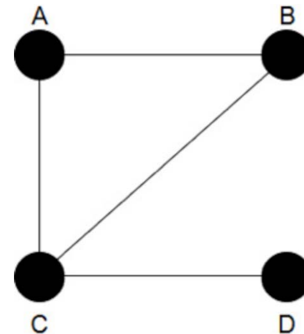

Figure 3. Example of a graph

Nowadays, the aforementioned properties of graphs have been implemented in graph databases. Graph databases are a category of NoSQL databases with a growing popularity among users over the last few years. This is due to the increasing amount of highly connected data, where both the information about real-world entities and the information about the nature of their relationship need to be stored for further processing.

The underlying data model of graph databases is a graph. There are several graph data models, which can be used to store data in graph databases. In this paper, we will use the labelled property graph data model due to its simplicity.

In property graph data model, real-world entities are stored as nodes (vertices) of the graph, while their connections are stored as relationships (edges) between those nodes. Both nodes and edges can have properties in the form of key-value pairs, and labels to distinguish different node and edge types.

Although we plan to represent ICS resources in graph databases, for the purpose of this paper, we have developed a simple graph model of books shown in Fig. 4. The model consists of three nodes labelled "Book", "Author" and "Genre", and two edges of type "WROTE" and "PART_OF". The interpretation of the sample graph would be: William Shakespeare (author) wrote Romeo and Juliet (book published in 1597), which is part of the tragedy genre.

This model was implemented in the Neo4j graph database by using Java Spark framework and different Java Gremlin and Neo4j dependencies discussed in [13].

IV. Related Work

From the appearance of object-oriented databases, several authors explored the possibilities of representing the object database model. In this section, we are going to mention research proposals, which included graph representation of the object database model, and vice versa.

In [14], authors introduced the graph-oriented object database (GOOD) model, in which the object database instance is represented as a graph, and different graph transformation operations are used to manipulate data in the object database. In GOOD, database objects are represented as nodes of the graph, while the object relationships and properties are represented as edges. The authors also developed a graphical transformation language to perform basic graph transformation operations, such as nodes and edges addition/deletion, and abstraction. The language also supports the method mechanism. The authors implemented a database interface for interacting with the GOOD model as well, which enables users to specify graph patterns and GOOD programs, and to visually explore the graph representation of the object base schema.

The GOOD model was later further discussed and extended in several research papers ([15], [16]).

When it comes to representing graphs as objects, at the time of writing this paper, it can be concluded that there has not been much work done regarding this research topic, especially in the context of graph databases. Neo4j Inc. developed an object-graph-mapping (OGM) library called Neo4j OGM, which role is to map graph nodes and relationships to objects in the domain model of an application written in Java programming language [6]. The introduced OGM eliminates the need for using different libraries for storing the domain model in the form of a graph by abstracting the entire database. Like its equivalent Java ORM library Hibernate, Neo4j OGM maps annotated Plain Old Java Objects (POJOs) (e.g., *NodeEntity, Relationship, RelationshipEntity*) to graph nodes and relationships [6].

Dietze et al. developed an open-source OGM framework for Neo4j and Scala language called Renesca [17], which consists of a graph database driver for executing queries and an ER-modeling domain specific language (DSL).

V. Introducing Object-Oriented Concepts to Graph Databases

In this section, we are going to discuss how some OO concepts could be implemented in a graph database, i.e., Neo4j. To interact with Neo4j, queries will be executed through the Gremlin By Example visual query language, which was developed and discussed in previous work [18]. To showcase the proposed approach, nodes represented on the model in Fig. 4 will be implemented by using OO concepts. Also, a simple web interface has been developed by using Java Spark.

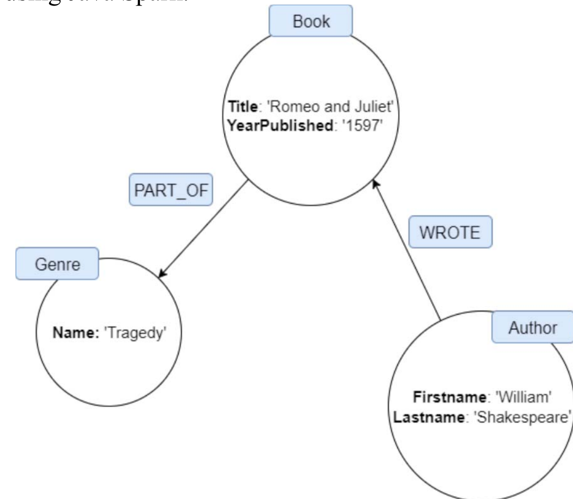

Figure 4. Sample property graph data model

As discussed in Section 3, a node has one or multiple labels (e.g., Author, Book, Person, etc.) and properties stored as key-value pairs (e.g., Firstname, Lastname, Title, etc.). Each node stored in a graph database can be located through its unique identifier. Therefore, the properties of a node resemble the properties of an object, which suggests that nodes can be represented as objects.

Specifically, in our approach, node labels represent the names of classes to be instantiated. Each node label represents a group of nodes with the same role or meaning, i.e., properties, whereas a class represents a type of all objects with the same properties. Therefore, to create new nodes in the database we must instantiate objects of appropriate classes. For instance, to create a new Book node, it is necessary to create a new object of class Book, which has the definition shown in Fig. 5.

```
public class Book
{
    public string label;
    public string title;
    public string yearPublished;

    public Book (string label,
string title, string yearPublished){
        this.label = label;
        this.title = title;
        this.yearPublished =
yearPublished;
    }
}
```

Figure 5. A Java definition of the Book class (node)

As shown in Fig. 6, when the user selects the wanted node label from the dropdown menu (in this case, Book), a method will be called to retrieve properties of the Book class defined earlier. The user then fills in the required values for retrieved properties (book title and year it was published). The entered data is then forwarded to the Book class constructor as parameters, and a new object (Book class instance) is created.

Figure 6. Creating Book node instance

After the necessary object is created, this object is forwarded to the method, which creates nodes in the databases by executing Gremlin queries. Finally, the Gremlin query implemented as shown in Fig. 7 will be executed to create a new Book node.

```
try (Transaction tx = db.tx()) {
  vertex = db.addVertex(book.label);

  vertex.property("Label",
book.label);
  vertex.property("Title",
book.title);
vertex.property("YearPublished",
book.yearPublished);
  tx.commit();
}
```

Figure 7. A Java implementation of creating Book node with Gremlin

This approach enables us to create new nodes by extending existing nodes. In this case, a new class can be created by copying an existing class definition along with its attributes and methods. For instance, the Book class can be used to define the Journal class, i.e., the Journal class can consist of properties copied from the Book class, along with some additional properties specific for the Journal class. The Journal class definition would then be as shown in Fig. 8.

In this example, the Journal class contains all properties of the Book class, but it also contains the additional volume property indicating the volume number of the journal. Thus, when creating a new Journal node, i.e., object, the Journal class constructor calls the Book class instructor as well.

The explained inheritance example can be seen as an equivalent of the SQL LIKE operator in object-relational databases, i.e., once the child node is created, any additional change on the parent node (e.g., added or changed property) will not reflect on the child node. The child node has the same properties of the parent node, i.e., the same structure as the parent node, but it can also be extended with additional properties.

```
public class Journal extends Book
{
    public string volume;

    public Journal (string label,
string title,
       string yearPublished, string
volume){
         this.super(label, title,
yearPublished);
         this.volume = volume;
    }
}
```

Figure 8. A Java definition of the Journal class (node)

VI. Conclusion

In this paper, we have discussed the basic concepts of the object-oriented paradigm, and how these concepts were implemented in relational databases. The main purpose of this paper was to discuss how class instantiation and inheritance could be implemented in graph databases. We have developed a simple showcase to explain how nodes could be stored as objects with the possibility of creating new nodes based on existing nodes by copying existing nodes' definition.

Currently, nodes can inherit both properties and methods of the parent node. In future work, we plan to introduce the possibility to keep a connection between the parent and child node by using some active mechanisms, which will monitor and propagate any changes on the parent node on to the child node. The ideas presented in this paper will be adjusted for next generation incident command system. Furthermore, we plan to support inheritance in its full capacity which means that changes to parent nodes will be automatically propagated to child nodes.

Acknowledgment

This research paper was supported by grant from the North Atlantic Treaty Organization (NATO) Science for Peace & Security Programme project called "Advanced Regional Civil Emergency Coordination Pilot (ARCECP)" (grant No. G498) in cooperation with NATO - Science for Peace and Security (NATO SPS), US Department of Homeland Security Science & Technology Doctorate, MIT Lincoln Laboratory (MIT LL) and Croatian National Protection and Rescue Directorate (NPRD).

REFERENCES

[1] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *ACM Sigplan Notices*, 1986, vol. 21, no. 11, pp. 38–45.

[2] K. B. Bruce, *Foundations of object-oriented languages: types and semantics*. MIT press, 2002.

[3] B. Wagner, L. Luke, W. Maira, and Petrusha Ron, "Classes (C# Programming Guide)." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes.

[4] R. Prieto-Diaz, "Status report: Software reusability," *IEEE Softw.*, vol. 10, no. 3, pp. 61–66, 1993.

[5] E. Bertino and L. Martino, "Object-oriented database management systems: concepts and issues," *Computer (Long. Beach. Calif).*, vol. 24, no. 4, pp. 33–47, 1991.

[6] Neo4j, "Neo4j OGM - An Object Graph Mapping Library for Neo4j v3.0." [Online]. Available: https://neo4j.com/docs/ogm-manual/current/.

[7] W. Kim, "Object-Oriented Database Systems: Promises, Reality, and Future.," in *VLDB*, 1993, vol. 19, pp. 676–692.

[8] S. O. Ogunlere and S. A. Idowu, "Comparison Analysis of Object-Based Databases, Object-Oriented Databases, and Object Relational Databases," *Asian J. Comput. Inf. Syst. (ISSN 2321--5658)*, vol. 3, no. 2, 2015.

[9] D. Maier, J. Stein, A. Otis, and A. Purdy, *Development of an object-oriented DBMS*, vol. 21, no. 11. ACM, 1986.

[10] IBM, "Table Inheritance," 2011. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSGU8G_11.70.0/com.ibm.ddi.doc/ids_ddi_124.htm.

[11] T. P. G. D. Group, "Inheritance," 2017. [Online]. Available: https://www.postgresql.org/docs/10/static/ddl-inherit.html.

[12] R. J. Wilson, *An introduction to graph theory*. Pearson Education India, 1970.

[13] M. Šestak, K. Rabuzin, and M. Novak, "Integrity constraints in graph databases – implementation challenges," in *Proceedings of Central European Conference on Information and Intelligent Systems*, 2016, pp. 23–30.

[14] M. Gyssens, J. Paredaens, J. den Bussche, and D. Van Gucht, "A graph-oriented object database model," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 4, pp. 572–586, 1994.

[15] M. Gyssens, J. Paredaens, and D. Van Gucht, "A graph-oriented object model for database end-user interfaces," *ACM SIGMOD Rec.*, vol. 19, no. 2, pp. 24–33, 1990.

[16] C. Tuijn and M. Gyssens, "CGOOD, a categorical graph-oriented object data model," *Theor. Comput. Sci.*, vol. 160, no. 1–2, pp. 217–239, 1996.

[17] F. Dietze, J. Karoff, A. C. Valdez, M. Ziefle, C. Greven, and U. Schroeder, "An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca," in *International Conference on Availability, Reliability, and Security*, 2016, pp. 204–218.

[18] K. Rabuzin, M. Maleković, and M. Šestak, "Gremlin By Example," in *International Conference on Advances in Big Data Analytics*, 2016.