Future Generation Computer Systems 115 (2021) 459–474

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Applying *k*-vertex cardinality constraints on a Neo4j graph database

Martina Šestak *, Marjan Heričko, Tatjana Welzer Družovec, Muhamed Turkanović

Faculty of Electrical Engineering and Computer Science, University of Maribor, Koroška cesta 46, 2000 Maribor, Slovenia

A R T I C L E I N F O

Article history:
Received 18 September 2019
Received in revised form 16 July 2020
Accepted 28 September 2020
Available online 2 October 2020

Keywords:
k-vertex cardinality constraints
Graph databases
Property graph data model
Procedures
Business rules
Graph schema

A B S T R A C T

As with any other database solution, graph databases also need to be able to implement business rules related to a given application domain. At the moment, aside from integrity constraints, there is a limited number of mechanisms for business rules implementation in Graph Database Management Systems (GDBMSs). The underlying property graph data model does not include any formal notation on how to represent different constraints. Specifically, this paper discusses the problem of representing cardinality constraints in graph databases. We introduce the novel concept of *k*-vertex cardinality constraints, which enable us to specify the minimum and maximum number of edges between a vertex and a subgraph. We also propose an approach, which includes the representation of cardinality constraints through the property graph data model, and demonstrate its implementation through a series of stored procedures in Neo4j GDBMS. The proposed approach is then evaluated by performing experiments on synthetic and real datasets to test the influence of checking cardinality constraints on query execution times (QETs) when adding new edges. Additionally, a comparison is performed on synthetic datasets with varying outgoing vertex degrees in order to gain an insight into how increasing the vertex degree affects QETs. In general, the results obtained for each test scenario show that the implemented *k*-vertex cardinality constraints model does not significantly affect QETs. Also, the results indicate that the model is dependent on the order of the underlying *k*-vertex cardinality constraints and outgoing vertex degree in the dataset.

1. Introduction

Graph databases have been in use for a few decades now [1–4]. Compared to other database solutions, graph databases store real-world entities and their connections in the form of a graph, i.e. vertices connected by edges. However, the ability to store additional information regarding connections between entities as edge properties makes them a good solution in domains with highly connected data (e.g. social networks).

Even though graph database systems are continuously developed to make them more stable in performance and richer in features, there is still much work to be done before they can meet the maturity level of other data management solutions (e.g. relational databases). In this paper, we focus on graph database mechanisms, which can be used to bridge the gap between business logic and information systems in order to increase the applicability of graph database technology in a real-world business context.

Namely, in each business environment, there exists a precisely specified set of formal steps, which define how a given business process is performed, i.e. in which sequence the process actions are executed, or limit some aspect of business process execution. These steps are generally referred to as business rules [5], and they are by nature atomic, i.e. defined in such a way that they cannot be further decomposed. In [6], Simsion and Witt include enforcement of business rules as one of the measures of quality of data models, and identify business rules as “top-level object classes” generated during the business requirements gathering process, which must be defined before the enterprise starts doing business in order to specify how its information system (IS) behaves in a particular situation. According to various authors [6,7], business rules’ scope can vary from specifying how enterprise data is stored and handled to constraining how a given business process is executed. In graph database context, various business rules might need be implemented, such as vertex/edge property existence, unique and obligatory (not null) property values, etc. Nevertheless, given the fact that edges are seen as “first-class citizens” in graph databases [8], it is important to discuss business rules focusing on how edges are created in order to e.g. prevent/detect fraudulent behaviour in graph-oriented domains, and make them a more reliable solution for varying-size enterprises and domains. For this purpose, traditional notions of integrity constraints developed in other database solutions (mostly relational databases) might be re-used with some adjustments to the

* Corresponding author.
E-mail addresses: martina.sestak@um.si (M. Šestak), marjan.hericko@um.si (M. Heričko), tatjana.welzer@um.si (T.W. Družovec), muhamed.turkanovic@um.si (M. Turkanović).

https://doi.org/10.1016/j.future.2020.09.036
0167-739X/© 2020 Elsevier B.V. All rights reserved.

context of graphs, which are mostly related to how connections are established between entities. Even so, in the case of integrity constraints like UNIQUE or NOT NULL, the syntax of modern graph query languages somewhat resembles the SQL syntax in terms of their specification.

A common type of business rule is one that specifies how many instances (occurrences) of entities are linked to each other (e.g., one employee has only one manager, while one manager can manage more employees). In the context of databases, these kinds of business rules define the so-called degree of an edge (relationship), i.e. the cardinality between linked entities. Therefore, a cardinality constraint determines "the participation of an entity type E in a relationship type R" [9]. Since they affect the database structure, cardinality constraints are considered to be structural constraints [10]. Cardinality constraints can be further categorized into two types [10]:

1. *participation* - an entity type must be involved in a given number of relationship types, and
2. *look-across* - specify the minimum and maximum number of relationship types, in which a given entity type can be involved.

In relational databases, cardinality constraints are often specified during the conceptual database design, and represented as labels within the Entity-Relationship (ER) diagram [9,11].

At the moment, most GDBMSs use several integrity constraints and other mechanisms to enforce graph database integrity. For instance, Neo4j GDBMS supports the definition of constraints on unique vertex property values, vertex and edge property existence, etc., as well as a simple trigger mechanism to enforce database integrity [12]. On the other hand, OrientDB supports the so-called "in" and "out" constraints, which enable specifying which edge type can be created between which two exact vertex types [13] (e.g. Person and Car vertices can be connected only by an OWNS edge type in an exact Person → Car direction). Next, in OrientDB, duplicate edges between two vertices are prevented by creating UNIQUE indexes. The combination of in/out constraints and unique indexes can then be used to implicitly implement cardinality constraints. A JanusGraph (formerly known as TitanDB) allows the specification of edge label multiplicity, i.e. how many edges of a specific label are allowed between two vertices [14]. Most other GDBMSs (AllegroGraph, InfiniteGraph, HyperGraphDB) rely on user-defined mechanisms implemented on the application level to maintain integrity. In general, since graph databases rely on the "flexible schema" principle [15], indexes, integrity constraints and triggers are mechanisms used most often to maintain graph database integrity.

Motivation. Even though integrity constraints represent a reliable mechanism for maintaining database integrity, business rules related to a given domain can often be more complex to be represented and implemented in any type of databases. In this paper, we focus on the rules that specify how many instances of entities can be connected to each other, i.e. cardinality constraint rules. As an example, these types of rules can be used to limit people who can apply for a job posting, or how many tasks an employee can be assigned to in a given period of time. In relational databases, these kind of constraints are often represented in the ER diagram as one-to-one (1:1), one-to-many (1:M) or many-to-many (M:N) relationships [16]. However, there has been little research done in representing and implementing cardinality constraints in graph databases [17–19]. In his recent paper [20], Angles presents a formal definition of the property graph database model, and mentions cardinality constraints as a valuable extension to the graph database schema. So far, most authors have focused their research efforts towards limiting the number of binary edges allowed between two vertices. However, the integrity of a piece of information ensured by cardinality constraint rules in graph-driven domains often encompasses more than two vertices at once, which is a challenging scenario to represent and implement using current approaches. We tackle this challenge by making the *k*-vertex cardinality constraints concept able to represent higher order cardinality constraints between multiple vertices. Our motivation was further supported by the fact that there are a limited number of GDBMSs with the ability to implement cardinality constraints (e.g. JanusGraph [14]).

Contributions. In general terms, the main contributions of this paper are the formal definition of the *k*-vertex cardinality constraints, a novel cardinality constraints model for their representation, and the definition of procedures for the model implementation within a GDBMS. The introduced model enables the specification and the representation of higher order cardinality constraints, exploits the advantages of graph data representation, and enhances the graph database architecture. To achieve this, we present the following specific contributions:

- We introduce the concept of *k*-vertex cardinality constraints along with its formal definition and syntax for specification,
- We present a novel *k*-vertex cardinality constraint model, which enables the representation of higher order cardinality constraints (between several vertices),
- We present an implementation approach for the proposed *k*-vertex cardinality constraint model in graph databases, focusing on the Neo4j GDBMS,
- We evaluate our model implementation on synthetic and real datasets,
- We analyse the query execution times with and without checking the cardinality constraints, and
- Based on the results of the performance analysis, we discuss the positive influence of checking *k*-vertex cardinality constraints on graph query performance.

The rest of the paper is organized as follows: in Section 2, we give an overview of previous research papers related to cardinality constraints in graph databases. Section 3 contains the theoretical background necessary to define *k*-vertex cardinality constraints (introduced in Section 4) and possible implementation approaches in Neo4j GDBMS. In Section 5, we describe the set of operations through which the proposed *k*-vertex cardinality constraint model can be implemented within a GDBMS, whereas Section 6 contains a description of how the evaluation of the model's performance within the Neo4j GDBMS has been carried out. In Section 7, we discuss the results of the performance analysis and mention some limitations and future work. Finally, in Section 8, we conclude this paper by summarizing the most important findings.

2. Related work

In general, the topic of representing and enforcing cardinality constraints in different database solutions has been widely studied over the years. Intuitively, cardinality constraints have mostly been studied within the context of relational databases, and most approaches build on representing cardinality constraints through the ER model [11,21–24]. Camps focused his work on transforming *n*-ary relationships to the relational database schema through functional dependency patterns for different cardinality combinations (e.g. 1:1:1, 1:N:N for ternary edges, etc.) [25]. In object-relational databases, various cardinalities between relations can be implemented by pointers or via a hybrid approach (pointers and foreign keys) [26].

Additionally, several approaches have recently been introduced for modelling NOSQL databases in general [27]. In [28], the authors proposed a common conceptual model for NoSQL solutions, in which the minimum and maximum cardinalities are specified as the multiplicity property of the relationship definitions, and include (cardinality) constraint validation as one of the validation methods for the proposed model.

The topic of integrity constraints in graph databases has been discussed in [17]. The authors explored the possibilities available in Neo4j and other GDBMSs to express integrity constraints, and presented a prototype implementation of new constructs developed to express these constraints. According to their overview, Titan (currently renamed as JanusGraph at the moment) is the only GDBMS that offers the possibility of specifying cardinality constraints on a vertex and edge property, as well as different edge cardinalities (1:1, 1:N, M:N). The authors propose extending Cypher[1] with integrity constraints related to edge cardinality, which could be used to express the limitations on the number of edges a given vertex can be mapped to. However, the authors have still not included this constraint in their prototype implementation. The authors present the role of cardinality constraints as a mechanism for limiting how many edge types a given vertex needs to have, but this topic is not discussed in further detail, nor do they consider the possibility of specifying higher order cardinality constraints between several vertices.

In [30], the authors studied the processing of label-constraint reachability (LCR) queries. According to the authors, the goal of these queries is to check the existence of a directed path consisting of given edge labels between two vertices. For instance, this category of reachability queries can be used in social networks to only find members (vertices) that are relatives, as the label constraint in that case would be "isRelativeOf". In most cases, LCR queries are answered by computing transitive closures, i.e. matrices, in which values indicate whether there is a path between two vertices in a graph. In their approach, the authors combined transitive closures with a real-time graph traversal to optimize the computing of the path-label transitive closures [30]. The topic of LCR queries is somewhat relevant for our research, as the goal of such queries (checking the existence of a path) can be seen as a special type of cardinality constraint, which ensures the participation of two vertices in a given edge type. Therefore, such a scenario is supported by our enhanced model when semantically formulated in a slightly different manner.

There have been a few papers discussing the topic of conceptual modelling in graph databases, in which different approaches have been proposed for implementing business rules and constraints in graph databases. Specifically, in [31], Daniel, Sunyé and Cabot proposed the usage of an intermediate graph metamodel to map OCL[2] conceptual schema to the Gremlin graph query language and Blueprints abstraction layer in some graph databases. The authors proposed a UMLtoGraphDB framework for converting conceptual schemas into their graph representation and a series of Gremlin[3] queries based on business rules and constraints expressed with OCL. In our approach, we avoid using additional standardized languages such as OCL to express constraints. Instead, we introduce a novel concept of k-vertex cardinality constraint, which is based on graph concepts, to specify and represent constraints as vertices in the graph database.

Furthermore, in [33], the authors introduced an approach for modelling graph database (GD) schema based on the Entity Relationship (ER) diagram of the application domain by following two steps: (1) adjusting the ER diagram to be able to create GD schema and (2) mapping the adjusted ER diagram to GD schema. During the first step, the authors use the same 1:1, 1:N and M:N ER notation to represent cardinality constraints between two vertices in GD schema. In the next step, the authors propose representing cardinality constraints in GD schema through edges (relationships). However, compared to our research, the authors do not specify how exactly these constraints should be physically stored, and do not offer a concrete implementation of the proposed approach. The same conclusion can be made by looking at the research done by Ghrab et al. who introduced a generic graph database model called GRAD, which supports more complex graph structures and integrity constraints, including cardinality constraints represented as semantic constraints [34]. The authors use UML notation to indicate cardinalities between classes of vertices. Nevertheless, as the authors note, details regarding physical implementation of the model are still to be resolved in their future work. De Sousa and Cura [35] use the Extended Binary ER model for the conceptual modelling phase, based upon which the logical graph database model is specified. In their approach, cardinality constraints are denoted as *edgeRestricted* properties of vertices, and they propose a set of mapping rules for transforming the conceptual to the logical scheme that also takes cardinality constraints into account. However, the approach focuses on the binary edges between two vertices only.

Recently, Hartig and Hidders presented an interesting approach for defining graph database schema by using the GraphQL Schema Definition Language (SDL) [36]. The authors demonstrate how annotations available in GraphQL SDL can be used to specify 1:1, 1:N, N:1 and M:N cardinality constraints. In their approach, the limit on edge cardinality can be noted by specifying whether a value of property, which represents an association between two vertex types, is atomic (scalar or enumeration type) or a list of atomic values. Compared to this paper, we developed an enhanced k-vertex cardinality constraints model, which enables specifying cardinality constraints inside the database itself without using any external, third-party solutions or APIs. Also, our proposed enhanced model is powerful enough to allow for the specification of cardinality constraints of a higher order, which include three or more vertex types.

Sedlmeier and Gogolla proposed and evaluated a graph-based approach for conceptual data modelling and business requirements analysis called TEGel (Tibet Entity Graph Language) [37]. In their paper, they present a prototype database called Lhasa DB, which supports several concepts defined within TEGeL language, such as classification, inheritance, structured types representation, etc. In addition to rules related to entity structure representation, the proposed graph type also includes a mechanism for specifying rules related to integrity constraints, specifically value and structure constraints. Such rules can be defined through edge types as so-called participation cardinality, which specifies how many times a given entity instance can participate in a given edge [37]. The authors also introduced the cardinality concept to binary edges of different orders, in which first order edges connect two entity types, whereas higher order (*hyper*) edges connect first order edges with 1:1, 1:M and M:N mappings. Our model represents a novel approach, thus further improvement in specifying cardinality constraints of higher order and complexity. Instead of using individual edges of different orders to specify cardinality, we use a subgraph to specify the minimum and maximum number of edges of a given type between a vertex and that subgraph in general. This brings more flexibility to the entire model by allowing us to specify cardinality constraints of a higher order, which conform better to business rules (an example use case is mentioned in 4).

[1] Cypher - Neo4j official graph query language based on pattern matching paradigm [29].

[2] Object Constraint Language (OCL) - declarative language for describing business rules.

[3] Gremlin — traversal-based graph query language included in the TinkerPop3 API, which can be used to query the underlying Blueprints graph data model or Neo4j GDBMS [32].

3. Preliminaries

In order to fully understand the concept and definition of k-vertex cardinality constraints, it is necessary to first define relevant graph concepts, such as the property graph, property graph database schema, directed walk and cardinality constraints in general.

Let us assume that **L** denotes an infinite set of vertex/edge labels, **N** is an infinite set of atomic values, **P** represents an infinite set of vertex/edge property names, **T** is an infinite set of data types, whereas the function $SET^+(X)$ denotes a set of all finite subsets of a given non-empty set X. [20].

Definition 1. A property graph is a tuple $G = (V, E, \rho, \lambda, \sigma)$ [20] where:

1. V is a finite set of vertices;
2. E is a finite set of edges such that E has no elements in common with V;
3. $\rho : E \to (V \times V)$ is a total function that associates any edge in E with a pair of vertices in V;
4. $\lambda : (V \cup E) \to SET^+(L)$ is a partial function that associates a vertex/edge with a set of labels from L;
5. $\sigma : (V \cup E) \times P \to SET^+(N)$ is a partial function that associates vertices/edges with properties, and for each property it assigns a set of values from N.

Definition 2. A property graph schema is a tuple $GS = (T_V, T_E, \beta, \delta)$ [20] where:

1. $T_V \subset L$ is a finite set of labels representing vertex types;
2. $T_E \subset L$ is a finite set of labels representing edge types, where $T_V \cap T_E = \emptyset$;
3. $\beta : (T_V \cup T_E) \times P \to T$ is a partial function that defines the properties of vertex/edge types, as well as the data types of the corresponding values;
4. $\delta : (T_V, T_V) \to SET^+(T_E)$ is a partial function that defines the edge types allowed between a given pair of vertex types.

Furthermore, the notion of k-vertex cardinality constraints is currently dependent on the existence of a directed walk between vertex and edge types included in the *kCard* definition. A directed walk is a graph theory notion important for directed graphs, whose definition we adjusted to our context as described in Definition 3.

Definition 3. In a directed graph, a directed walk W from v_0 to v_n, where n represents the size of T_V, $v_0, \ldots, v_n \in T_V$ and $e_0, \ldots, e_{n-1} \in T_E$, is an alternating sequence of vertex and edge types defined as $W = \langle v_0, e_1, v_1, \ldots, v_{n-1}, e_{n-1}, v_n \rangle$, such that $tail(e_i) = v_{i-1}$ and $end(e_i) = v_i$, for $i = 1, 2, \ldots, n$ [38].

In general, Olivé [39] defined cardinality constraints as the minimum and maximum number of edges allowed between two entity types (Definition 4). In his definition, a given binary edge type R is defined as $R(p_1 : E_1, p_2 : E_2)$, where p_1 and p_2 represent entity (vertex) types E_1 and E_2, respectively, participating in R.

Definition 4. In the property graph G, a cardinality constraint specified for a binary edge type is defined as $Card(p_1; p_2; R) = (min, max)$ [39] where:

1. $p_1, p_2 \in T_V$ are vertex types representing entity types E_1 and E_2, respectively;
2. $R \in T_E$ is a binary edge type specified between two entity types;
3. $min \in \mathbb{N}$ is a positive number representing the minimum number of allowed edges;
4. $max \in \mathbb{N}$ is a positive number representing the maximum number of allowed edges;

4. *k*-vertex cardinality constraint

Due to the high connectedness of data stored in property graphs, a single piece of information capturing a business activity is often spread across more than one edge between two vertices. A basic example to illustrate this situation is capturing students' course enrolment in different semesters. In this case, the complete information would be captured by connecting the *Student*, *Course* and *Semester* vertices within the constructed property graph. Nevertheless, let us assume that there exists an underlying business rule, which prohibits students from enrolling into more than three courses in a given semester. In the context of cardinality constraints, this implies that it is necessary to limit how many edges can be created between a Student and a Course, but with regard to a given Semester. In this case, the Course and the Semester form a subgraph as a semantic unit, and the cardinality constraints need to limit the number of edges between the Student and this subgraph, as they store the information about a given student enrolling into a given course in a given semester. By limiting only the number of edges between the Student and Course vertices would not capture the entire information, but generally limit in how many courses a student can enrol.

To apply the concept of cardinality constraints to n-ary edges between multiple vertices, i.e. a vertex and a subgraph, we extend Definition 4 and propose the concept of k-vertex cardinality constraints, where a formal definition is given in Definition 5. Informally, a k-vertex cardinality constraint specifies how many edges (minimum and maximum) of type R are allowed between a vertex labelled E and a subgraph S, which consists of k vertex and edge types forming a directed walk.

For property graph G, let $\mathbf{S} = (\mathbf{T_{VS}}, \mathbf{T_{ES}}, \mathbf{S_i})$ denote a subgraph consisting of vertex and edge types T_{VS} and T_{ES} ($T_{VS} \in T_V$ and $T_{ES} \in T_E$), which form a directed walk W, where k represents the order of subgraph S (i.e. number of vertex types in subgraph S), $2 \le i \le k$ and $S_i \subset S$. Consecutively, the definition of a binary edge type R is modified into $R(p_1 : E_1, p_2 : S)$, where p_1 and p_2 represent the vertex and subgraph types E_1 and S, respectively, participating in edge type R.

Definition 5. In the property graph G, a ***k*-vertex cardinality constraint** specified for a binary edge type is defined as $\mathbf{kCard(p_1; p_2; R) = (min, max)}$ where:

1. $p_1 \in T_V$ is a vertex type representing the entity type E_1;
2. p_2 is a subgraph S consisting of vertex and edge types;
3. $R \in T_E$ is a binary edge type specified between the vertex type p_1 and the subgraph p_2;
4. $min \in \mathbb{N}$ is a positive number representing the minimum number of allowed edges;
5. $max \in \mathbb{N}$ is a positive number representing the maximum number of allowed edges;

Note that k-vertex cardinality constraints may also be included in the property graph schema definition listed in Definition 2 as an extension of the δ function. In such a case, the δ function would define the minimum and maximum number of given edge types allowed between a given vertex and subgraph type (instead of two vertex types). Formally, the extended δ function could then be defined as $\delta : (T_V, S) \to SET^+(T_E) \times \mathbb{N} \times \mathbb{N}$.

Based on Definition 5, several facts about k-vertex cardinality constraints can be established.

Fact 1. *For a binary edge type R between two vertex types E_1 and E_2 in graph G, T_{ES} and S_i elements of subgraph S do not need to be specified.*

Therefore, a 1-vertex cardinality constraint includes a subgraph S, which consists of only one element, i.e the vertex type T_{VS}.

Fact 2. *Specifying one k-vertex cardinality constraint is semantically not equivalent to specifying $\underline{k}$ 1-vertex cardinality constraints.*

As already shown in Section 2, current approaches and tools mostly focus on the representation and implementation of cardinality constraints on edges between two vertices. The need to include a subgraph into the cardinality constraint definition stems from the high connectedness of information stored in graph databases, as a result of which defining multiple constraint rules between single vertices is not semantically equivalent to defining one higher order constraint rule between a vertex and a subgraph.

Specifically, modern graph-oriented domains must be able to store information including multiple vertices, at once. For instance, a single higher order constraint rule, which limits the number of cash withdrawals a client can make in a given day is not semantically equivalent to two constraint rules, one of which limits the number of withdrawals a client can make, and the other limits the number of withdrawals (in general) within a given day. For instance, a 1-vertex cardinality constraint specified as $kCard(Client; \{Withdrawal\}; MAKES) = (0, 5)$ would denote that a client may make, at most, five withdrawals in total. Conversely, a 2-vertex cardinality constraint specified as $kCard(Client; \{Withdrawal, IN\_DAY, \{Day\}\}; MAKES) = (0, 5)$ would limit the number of withdrawals that a client can make in a given day, which is a more appropriate rule for the domain.

Fact 3. *Within a given kCard constraint rule, a given vertex or edge type can be included more than once as a different definition element.*

In the case of specifying *kCard* constraints on cyclic edges, which connect vertices of the same types, it is possible to repeat the same vertex type in multiple levels of the definition. For instance, to specify that a given employee may have only one supervisor, the *kCard* constraint rule would be specified as $kCard(Employee; \{Employee\}; HAS\_SUPERVISOR) = (0, 1)$, where the *Employee* vertex type is included in both the E_1 and T_{VS} elements.

4.1. Properties of k-vertex cardinality constraints

In this section, we analyse three properties of *k*-vertex cardinality constraints: semantics preservation, idempotency of their verification and semantic independence.

Property 1 (*Semantics Preservation*). *A k-vertex cardinality constraint kCard specified for a given input business rule σ is preserving semantics if their property graph schema representations Mod_σ and Mod_{kCard}, respectively, are isomorphic.*

The semantics preservation indicates that the *k*-vertex cardinality constraints can vividly represent the semantics of the underlying business rules by directly mapping them into the elements of the property graph.

Property 2 (*Idempotency of kCard Verification*). *Let the function τ defined as $\tau : kCard \rightarrow \mathbb{N}$ denote the k-vertex cardinality constraint verification function, which returns 1 or 0 depending on if a given kCard rule is satisfied or not. The function τ is an idempotent operation if $\tau(\tau(kCard)) = 1$.*

The idempotency of the *kCard* rules verification indicates that the verification process has no additional effect on the underlying graph database regardless of the number of times it is executed, i.e. it returns 1 (the rule is satisfied) each time. For instance, this situation occurs when there are no patterns in the database that match any *kCard* constraint pattern.

Property 3 (*Semantic Independency*). *Two given k-vertex cardinality constraints kCard1 and kCard2 are semantically independent if there is no functional dependency between $\tau(kCard1)$ and $\tau(kCard2)$ and vice versa.*

The semantic independency indicates that, for given two *k*-vertex cardinality constraints, the verification must be carried out separately even if they partially include the same vertex and edge types. In such cases, one *kCard* rule cannot be thought of as a generalization of the other *kCard* rule since they carry different meanings. For instance, a rule $kCard(Student; \{Course\}; ENROLLS) = (0, 3)$ ("A student can enrol at most three courses.") is semantically independent and must be verified independently of the rule $kCard(Student; \{Course, HELD\_IN, \{Semester\}\}; ENROLLS) = (0, 3)$ ("A student can enrol at most three courses in a given semester.").

4.2. Model approach

The proposed approach provides a mechanism for transforming and representing input business rules, which limit the cardinality of connections between entities within the property graph data model by following a series of steps and representing these constraints by the property graph data model. In general, the proposed approach consists of two activities:

1. Cardinality constraint specification, whose goal is to build a list of specifications of *k*-vertex cardinality constraints for each input business rule, and
2. Cardinality constraint representation, whose goal is to build a property graph data model, which represents the specified constraints.

During the first activity, a list of *k*-vertex cardinality constraints is built in accordance with the business rules by identifying the attributes of the k-vertex cardinality constraint, i.e. the vertex type (E), subgraph (S), edge type (R), and minimum and maximum number of edges allowed for each given input business rule. At the moment, the subgraph S is identified based on the semantic interpretation of input business rules, i.e. the vertices and edges affected by the cardinality constraint form the subgraph path (vertex of type E excluded). However, various graph theory algorithms could be used to make this step more objective and automatic (e.g. highest density). In a given graph database, any *kCard* rule is stored a vertex labelled *CardinalityConstraint*, whose properties represent the elements of the formal definition (i.e. vertex, edge and subgraph type and the minimum and maximum number of allowed edges). Specifically, a given 2-vertex cardinality constraint rule formally defined as $kCard(p_1; p_2; R) = (min, max)$ would be mapped into a *CardinalityConstraint* vertex with the following syntax (i.e. vertex properties): $\{E_1, R_1, \{E_2, R_2, \{E_3\}\}, min, max\}$.

Due to the possibly high effort and syntax knowledge necessary to specify constraint rules, we plan to investigate the possibility of using TextRank and other machine-learning algorithms (available in Neo4j Graph Data Science library) to simplify and automatize this step in our future work. For now, due to the focus of this research, we have confined ourselves to the manual approach of identifying the business rules and defining the constraint rules.

As an example, given the sample property graph shown in Fig. 1, the following set of business rules have been defined

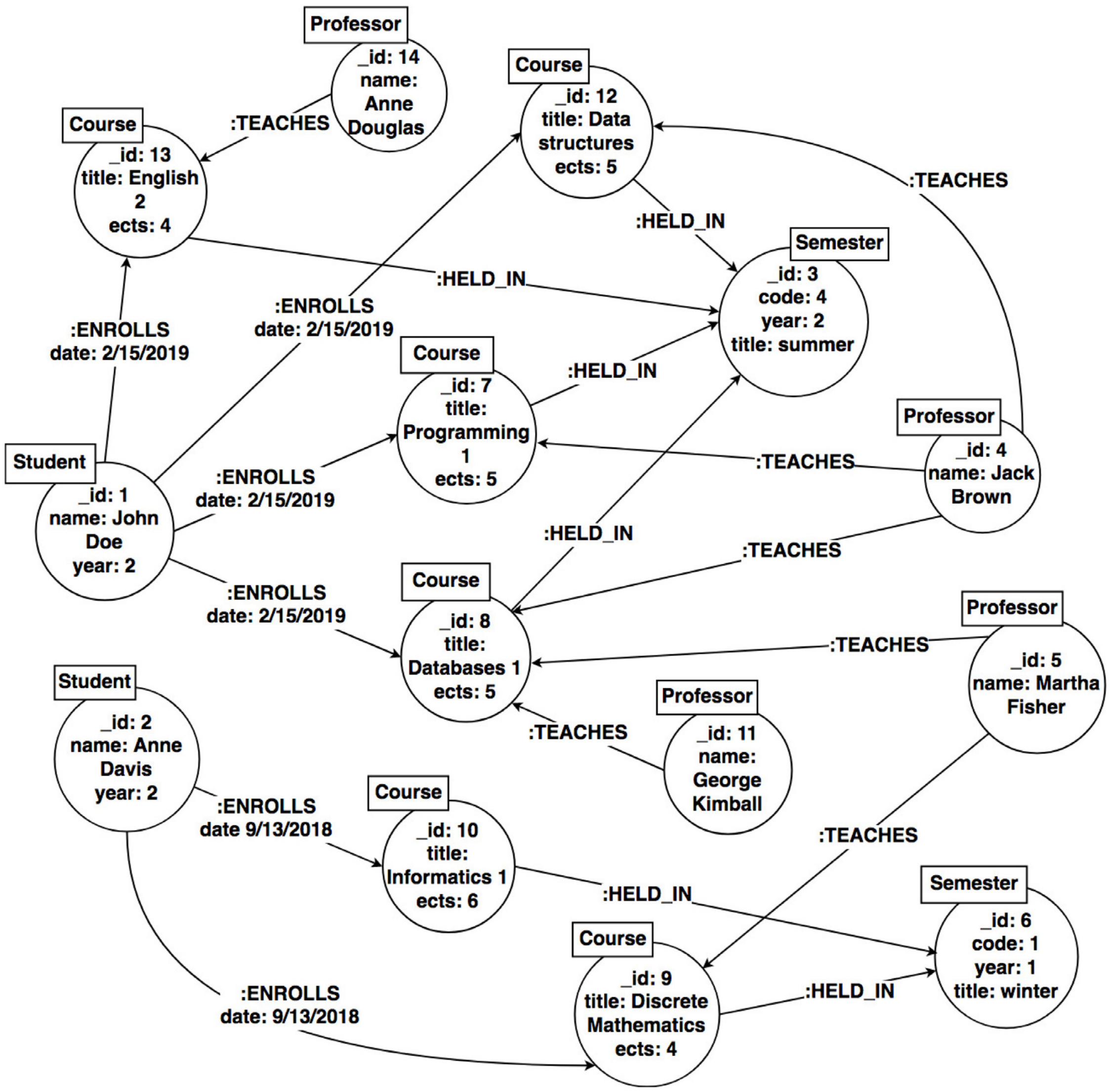


Fig. 1. Sample property graph representing course enrolment process without applying business rules.

to specify some limitations related to course enrolment process represented by the model:

(*BR*1) *A course may be enrolled in by 0 to 150 students at any time.*
(*BR*2) *A course may be taught by 1 to 2 professors at any time.*
(*BR*3) *A professor may teach between 0 and 2 courses in one semester.*
(*BR*4) *A student may enrol in between 0 and 3 courses in one semester.*

In our sample scenario, the result of the first activity is a set of *k*-vertex cardinality constraints for input business rules:

(BR1) $kCard(Course; \{Student\}; ENROLLED\_BY) = (0, 150)$
(BR2) $kCard(Course; \{Professor\}; TAUGHT\_BY) = (1, 2)$
(BR3) $kCard(Professor; \{Course, HELD\_IN, \{Semester\}\}; TEACHES) = (0, 2)$
(BR4) $kCard(Student; \{Course, HELD\_IN, \{Semester\}\}; ENROLLS) = (0, 3)$

In the *kCard* definitions for rules BR3 and BR4, the subgraph *S* consists of a directed edge labelled *HELD_IN* that connects vertices of type *Course* and *Semester*, where *Semester* is the only element of the subgraph S_2. Formally, *S* corresponds to a subgraph structure in the form $\{Course, HELD\_IN, \{Semester\}\}$, and it is included here in the *kCard* specification for the purpose of their property graph schema representation. For instance, the syntax for specifying the *kCard* rule for BR3 has the following form: "*E: Professor, R: TEACHES, S: {E: Course, R: HELD_IN, S: {E: Semester}}, min:0, max:2*". This structure is passed as an argument to the *create_kCard()* procedure responsible for creating a vertex representing the *kCard* rule in the graph database, which will be described in later sections.

Once extracted from the domain's business rules, *k*-vertex cardinality constraints can be included in the property graph schema represented by the property graph data model during the second activity, which can then be implemented in a given graph database. An example of such a schema is shown in Fig. 2. The schema presents specifications of four *k*-vertex cardinality constraints for the above given scenario. For instance, a 1-vertex cardinality constraint for *BR1* is formally defined as *kCard(Course;{Student};ENROLLED_BY) = (0,150)*, which means that a given course may be enrolled in by 0 to 150 students. Similarly, a cardinality constraint of a higher order can be specified. For instance, a 2-vertex cardinality constraint for *BR4* is defined

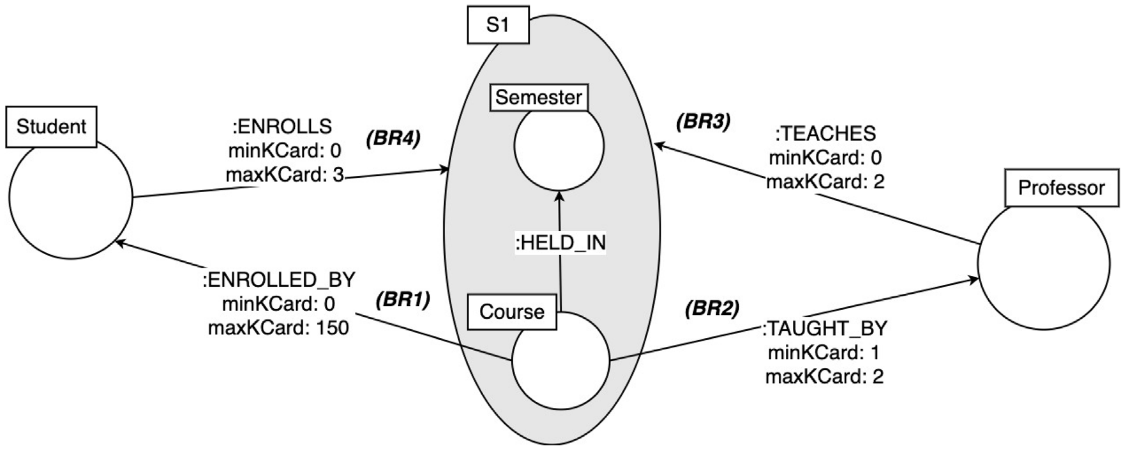


Fig. 2. Property graph schema representation of *k*-vertex cardinality constraints.

as *kCard(Student;{Course, HELD_IN, {Semester}};ENROLLS) = (0,3)*, which means that a given student can enrol between 0 and 3 courses in a given semester.

It is important to note that in our proposed approach 1-vertex cardinality constraints are semantically equivalent to cardinality constraints specified between two vertices since subgraph *S* in its definition contains a single vertex. However, by introducing the subgraph structure in the *k*-vertex cardinality constraints definition, we introduce the possibility of specifying both simple (1-vertex) and higher order cardinality constraints (multiple vertices), since subgraph *S* can have as many nested subgraphs as necessary.

After building the *k*-vertex cardinality constraint model as a property graph schema representation, we focus on its implementation in a Neo4j GDBMS.

5. The proposed implementation model

This section deals with the implementation details of the proposed *k*-vertex cardinality constraint model. Due to the high-level definition of the concept, our proposed model can be implemented in any GDBMS through the usage of its extension mechanisms, which allow users to implement additional business logic into the system. For instance, in the Neo4j GDBMS, the model can be implemented by using the user-defined procedure mechanism.

Once implemented, the following three operations are available to create and evaluate *k*-vertex cardinality constraints:

1. *Create_kCard*({constraint}) - used for creating vertices representing the *kCard* rules within the graph database,
2. *Create_relationship*(query_pattern, constraint_mode) - used to insert a new edge within a given query pattern with/without checking if the existing *kCard* rules are satisfied, and
3. *Check_constraint*(output_file_path) - used to evaluate the existing data within a graph database against the *kCard* rules and identify edges violating the constraints.

At the moment, the *kCard* constraint rules are specified in a given graph database instance manually by invoking the *Create_kCard({constraint})* operation, where the *constraint* argument must be specified by following a custom syntax that relies on their definition (already described in Section 4.2). However, in future extensions, this human error-prone approach might be supplemented by either developing an intuitive user interface, which will translate user inputs into the underlying required syntax, or by developing a "standardized" statement in a given query language (e.g. Cypher) with a clear syntax to follow. For the purpose on our research objective, we focused on the model definition and its evaluation and left the aforementioned automatizing options for possible future research.

Prior to invoking the *create_relationship* or *check_constraint* operations, an initial set of defined *k*-vertex cardinality constraints must be stored as vertices in the graph database. Please note that the model is flexible enough to allow users to add new constraint vertices to the graph later if necessary.

A detailed description of procedures implementing the *create_relationship* or *check_constraint* operations is given in the following subsections. Since we used the Neo4j GDBMS to demonstrate this approach, the procedures can be used by invoking the CALL Cypher statement on a Neo4j graph database instance.

5.1. Create_relationship procedure

Create_relationship is called when a user wants to add a new edge between two vertices in a graph database. The first argument of the procedure is the query pattern, which forms the path between two given vertices in the graph. For instance, to create an edge of type ENROLLED between a student 'John Doe' and course 'Web programming', the input query pattern would be:

```
(n1:Student {name: 'John Doe'})-[r1:ENROLLED]->(n2:Course {title:
'Web programming'})
```

The second argument, *constraint_mode*, indicates whether a cardinality check should take place when adding new edge to the database or not. If the value of this argument is set to "NO_CARDINALITY", the edge corresponding to the input query pattern will be added to the database without checking for a cardinality constraint violation. Otherwise, the procedure will first ensure that inserting the new edge does not violate relevant constraints, defined and stored in a separate graph database (collection).

The input pattern (property values excluded) is matched against patterns of constraints retrieved from the database in order to identify if there is a cardinality constraint specified for this pattern. In terms of *k*-vertex cardinality constraints, a 1-vertex cardinality constraint would be specified in the database with the following definition: *kCard(Student,Course,1,ENROLLED)*. In the graph model representation, this constraint is implemented as a vertex labelled *CardinalityConstraint* with properties E (value "Student"), R (value "ENROLLED"), S (map with property E with value "Course") and the minimum and maximum properties indicating cardinality limitations for a given edge.

If the patterns are equal, then a query is executed to retrieve the current number of edges of a given type for a given start vertex. If this number is still less than the maximum cardinality specified by the constraint, i.e. the new edge would not yet violate the constraint, the edge is added to the database. Otherwise, no action is performed, i.e. a given edge is not inserted. The complete pseudocode of this procedure is presented in Procedure 1 in Appendix A.

5.2. Check_constraint procedure

This procedure is called on a property graph, which may contain edges violating cardinality constraint rules. Hence, the only argument for this procedure is the path of the output file, in which the results of the procedure are written.

Algorithm-wise, the pseudocode of this procedure is slightly different and more complex compared to the *create_relationship* procedure (Procedure 2 in Appendix A).

For each cardinality constraint retrieved from the database, the algorithm executes an additional query to retrieve all vertices and edges, which match the pattern defined in the constraint. Each retrieved start vertex is then put as a key into a map, whereas the map value represents the number (counter) of edges found for which the given vertex is the start vertex (map key). If that number is smaller than the maximum number of edges allowed by the constraint, the value is increased by 1, and the edge is added into a list of edges, which do not violate the constraint rules. Otherwise, the edge is declared to be the one violating the cardinality constraint and added into a specific list of violating edges. Finally, the list of violating edges is iterated, and each member edge can be addressed as deemed necessary.

The strategy on how to address the violating edges can be easily modified depending on the scenario (e.g. removed from the collection, labelled as violating but kept in the collection or moved to another location/collection). Currently, we have focused our efforts on demonstrating how our concept could be used to detect and "purge" data (i.e. edges) in violation of the cardinality constraint rules. One of our future research directions will be on modifying this procedure to include various strategies for handling violating edges once they are all detected in the entire database (e.g. store them in log files, move them to another location, and so on), as this can be a useful feature in various use cases and domains, especially anomaly and fraud detection. Nevertheless, the primary goal of cardinality constraints should be on preventing such data from being stored in the database at all (i.e. the *create_ relationship* procedure with *kCard* constraint check enabled), so this is the main aim of the proposed *kCard* concept.

6. Evaluation

The evaluation of the proposed *k*-vertex constraint model has been carried out in a test environment with both synthetic and real datasets. The experiments described in this section aim at analysing the effectiveness of the proposed *k*-vertex cardinality constraint model and its implementation through the aforementioned procedures.

Throughout the experiment, the query execution times were analysed with and without checking edge cardinalities by using the *k*-vertex cardinality constraints. A comparison was made to evaluate the effect of the implemented solution on the overall effectiveness.

6.1. Prerequisites

To evaluate the proposed model, two types of datasets have been used:

1. Synthetic datasets consisting of 100, 1,000 and 5,000 edges, and
2. Real dataset representing the New York City expense budget.[4]

Synthetic datasets consist of a varying number of vertices and edges manually generated by using Cypher and functions available in the Neo4j APOC (Awesome Procedures on Cypher) library.

In addition to synthetic datasets with varying number of edges (100, 1000 and 5000), the experiments were conducted on synthetic datasets with varying outgoing vertex degrees. Note that the maximum number of edges was rounded to 5000 in order to generally achieve a scale corresponding to the real dataset (from 100 to almost 10,000 relationships combined). In graph theory, the outgoing vertex degree denoted as $d(v)$ represents the number of outgoing edges from a given vertex (node) v [40]. A graph, in which all vertices are adjacent to each other has the (outgoing) vertex degree of *n-1*. The purpose of including this graph property in experiments was to explore how increasing the outgoing vertex degree influences query performance, especially the complexity of finding edges that violate cardinality constraints. Note that, for this research, an outgoing vertex degree of 1 was excluded from experiments in order to give more attention to more complex cases.

Hence, both the graph size, i.e. number of edges in the graph database, and outgoing degree of vertices were used for a performance analysis on a synthetic dataset. A summary table containing database statistics (number of vertices/edges, database size, etc.) is presented in Table 1. The table includes summary information for databases after each test dataset (both synthetic and real) have been imported.

For each of these datasets, a property graph schema was built with an appropriate number of vertex and edge types. The sample property graph schemas used to perform the experiments on a dataset with 100 edges and outgoing vertex degree 3 (*Syn100d3*) and the real dataset (*Real*) are depicted in 3(a) and 3b, respectively.

Please note that synthetic datasets have been generated in random order, i.e. with no specific preferences. Also, in this context, *n* represents the number of vertices in the dataset, so that the *n-1* vertex degree indicates a complete graph scenario, where each vertex is connected to every other vertex in the graph. During the dataset creation, the main objective was to ensure that the outgoing degree for each vertex in the dataset was 2, 3 or n-1, depending on the test scenario. Therefore, it is possible that some vertices have varying ingoing and outgoing degrees, because our focus only went to the outgoing vertex degree.

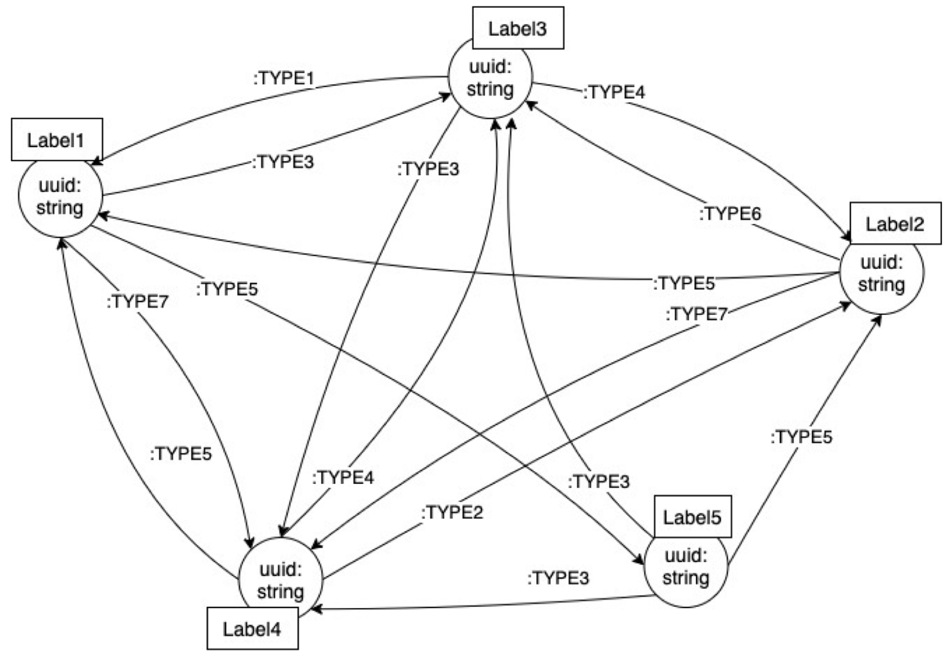
On the other hand, a real dataset downloaded from Kaggle in the CSV format was first analysed and transformed in order to build a property graph data model. The dataset contains data about New York City agencies' expenses. A subset of columns were selected from that CSV in order to formulate meaningful graph vertices and edges. The built graph data model contains vertices labelled Department, Fiscal Year, Account, Source_Fund and Program (3077 vertices in total). There are five types of directed edges in this model (9297 edges in total):

- IS_SUBDEPARTMENT_OF (Department -> Department)
- HAS_PROGRAM (Department -> Program)

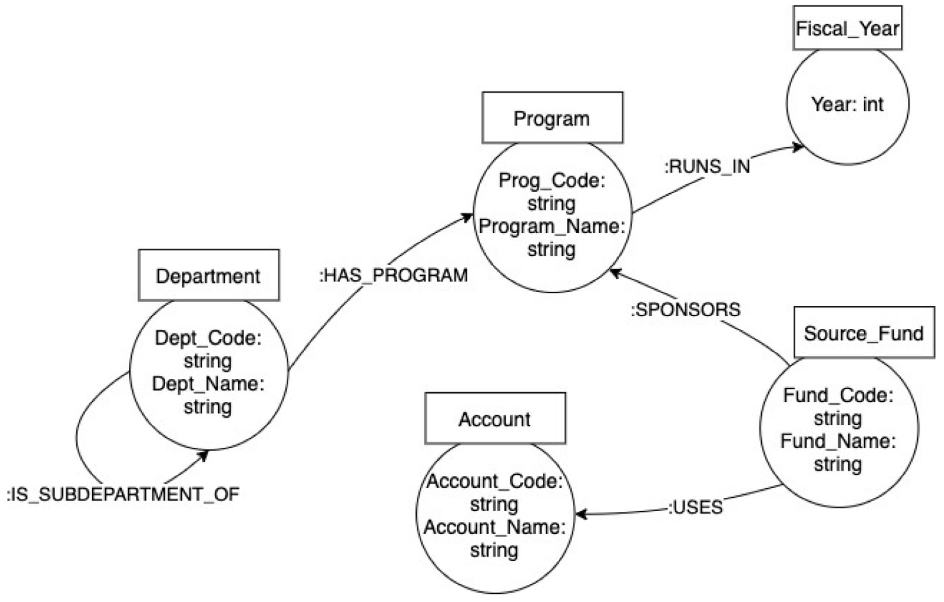
[4] https://www.kaggle.com/new-york-city/new-york-city-expense-budget.

Table 1
List of acronyms and graph database properties for synthetic and real test datasets.

Dataset acronym	Database storage size	Outgoing vertex degree	Number of vertices	Number of vertex labels	Number of edges	Number of edge types
Syn100d2	129.63 kB	2	50	3	100	6
Syn100d3	128.01 kB	3	34	5	100	7
Syn100dn	126.25 kB	n-1 (complete graph)	25	3	100	7
Syn1000d2	265.84 kB	2	500	3	1000	6
Syn1000d3	242.58 kB	3	340	5	1000	7
Syn1000dn	228.64 kB	n-1 (complete graph)	250	3	1000	7
Syn5000d2	871.31 kB	2	2500	3	5000	6
Syn5000d3	751.80 kB	3	1700	5	5000	7
Syn5000dn	683.72 kB	n-1 (complete graph)	1250	3	5000	7
Real	1.45 MB	–	3077	5	9297	5



(a) *Syn100d3* synthetic test dataset.



(b) *Real* test dataset.

Fig. 3. Property graph schemas for test datasets.

- RUNS_IN (Program -> Fiscal_Year)
- SPONSORS (Source_Fund -> Program), and
- USES (Source_Fund -> Account).

Before analysing the query performance of the implemented procedures, as mentioned in Section 5, it was necessary to specify *k*-vertex cardinality constraints.

To test the proposed model, we decided to define cardinality constraints of a different order for each test scenario and dataset. We chose to observe the model performance in scenarios when there are both single cardinality constraints of a specific order (1-vertex or 2-vertex), and when there is a combination of constraints of different orders (1-vertex combined with 2-vertex constraint) to see how large the difference in query execution times is between these scenarios.

Therefore, the query performance was tested on three scenarios, which differ in the number and order of cardinality constraints stored in the database:

1. A single "simple" cardinality constraint specified for an edge between two vertices (henceforth denoted as `SingleSimpleCard`),
2. A single "complex" cardinality constraint specified for an edge between a vertex and a subgraph (henceforth denoted as `SingleComplexCard`) and
3. A combination of the two previous constraints (one "simple" and one "complex" constraint) (henceforth denoted as `CombCard`).

For synthetic datasets, arbitrary 1- and 2-vertex cardinality constraints have been specified for edges between vertices with an existing path between them. On the other hand, in a real dataset, we specified a meaningful 1- and 2-vertex cardinality constraint appropriate for the selected dataset domain (i.e. New York city budget). The following *k*-vertex cardinality constraints have been specified on test datasets:

- Synthetic datasets:
  - 1-vertex constraint:
    kCard(Label1; {Label2}; Type6) = (0,1)
  - 2-vertex constraint:
    kCard(Label2; {Label1, Type3, {Label3}}; Type5) = (0,1)
- Real dataset:
  - 1-vertex constraint ensuring that a department may be a subdepartment of at most one other department:
    kCard(Department; {Department}; IS_SUBDEPARTMENT_OF) = (0,1)
  - 2-vertex constraint ensuring that a source fund may sponsor at most three programs in a given fiscal year:
    kCard(Source_Fund; {Program, RUNS_IN, {Fiscal_Year}}; SPONSORS) = (0,3)

Furthermore, the experiments were conducted in a local test environment comprised of a Neo4j Community Edition GDBMS (version 3.3.5) running on a 1,6 GHz Intel Core i5 machine with 8 GB RAM running macOS High Sierra and Java 8. The user-defined procedures used to implement the *k*-vertex cardinality constraints model were written in the Java programming language by following guidelines listed in the official Neo4j documentation available at [41]. The JAR file, which contains compiled procedure code, was put in the */plugins* directory of a given Neo4j database instance.

6.2. Performance analysis

The implemented procedures (creating edges with and without a cardinality check and checking existing edges against cardinality constraints) have been called 12 times on each dataset, as described in Section 6.1. The database server was restarted and the database itself was recreated after each iteration, to ensure that the database cache is clear and not affected by any residual data. Furthermore, while executing each iteration, all services and jobs, not related to the experiment and running on the machine, were forced to stop. The final results of the performance test contain query execution times in milliseconds for each test scenario (e.g. create_relationship with cardinality check procedure on (`Syn1000d3`) with a `SingleSimpleCard` cardinality constraint in the database). To eliminate any outliers that might affect the correctness and reliability of the performance analysis, we excluded the minimum and maximum values of query execution times for each test scenario, which in the end resulted in 10 values available for further processing and in-depth analysis.

7. Results and discussion

7.1. *k*-vertex cardinality constraints model

When applied to the sample property graph discussed in Section 4.2 during the so-called data instance checking process, edges violating the predefined *k*-vertex cardinality constraints will be identified (in Fig. 4, such edges are marked with dotted lines). Note that, in this scenario, a course titled Informatics 1 (id 10) is also marked separately as it violates the cardinality constraint, which requires that each course held in a given semester to be taught by at least one professor.

At the moment, there is no general strategy specifying which edge is to be marked as violating during the validity check. In our current implementation, this depends on the order in which the edges are retrieved and processed on the application level.

7.2. *k*-vertex cardinality constraints performance analysis

This section contains an overview and analysis of selected summary results obtained through experiments performed on the datasets described in Section 6.1.

7.2.1. Query execution times for creating edges with and without checking cardinality constraints

Overall, the results of measuring query execution times for creating new edges with and without checking cardinality constraints on synthetic datasets, are listed in Table B.4 in Appendix B.

Next, as shown in Fig. 5, the results indicate that there is a slight increase in query execution times (QETs) for the *create_relationship* procedure in general as the number of edges in the synthetic datasets grows. Such results were to be expected because the underlying graph traversal algorithm needs to visit more vertices and edges in order to perform necessary queries during procedure execution. More interestingly, it can be observed that the average difference in QET decreases with the growing number of edges in the dataset. Also, the average QET is the highest on datasets with an outgoing vertex degree of 2, and it is inverse to the outgoing vertex degree, i.e. it decreases as the outgoing vertex degree increases. By taking the observed correlations into account, it can be assumed that the QET of *create_relationship* procedure will decrease with the increasing number of edges and increasing outgoing vertex degree. The reason for this lies in the underlying query execution plan

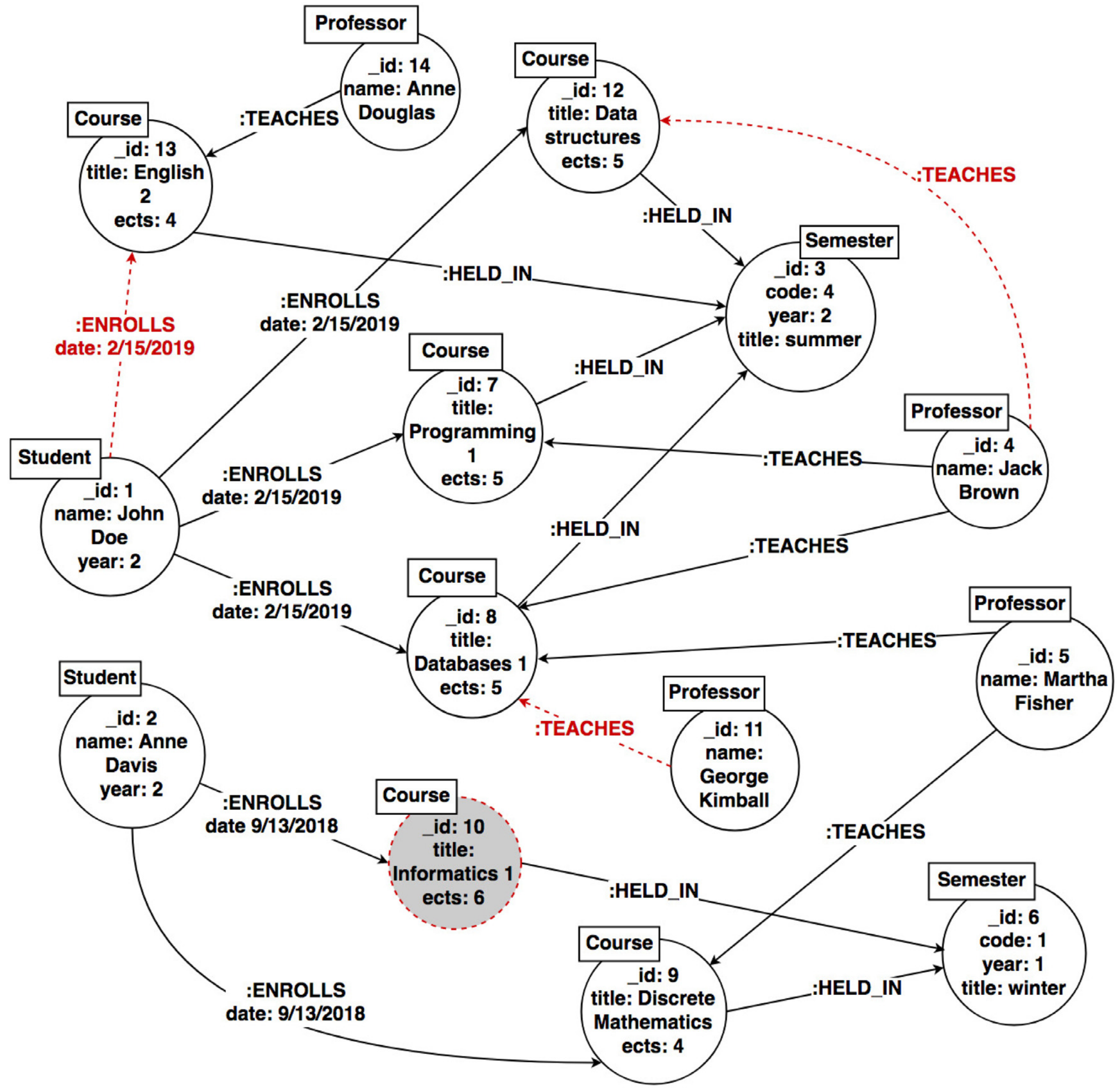


Fig. 4. Sample property graph representing course enrolment process after applying business rules.

constructed by Neo4j GDBMS, which is affected by the graph traversal strategy and its cache memory management.

A somewhat similar behaviour can be observed in QETs of the procedure in case of varying order of specified cardinality constraints; the procedure with the combined cardinality constraint (simple + complex) scenario takes less time to execute than in the single complex constraint scenario. Moreover, in some cases (e.g. 100 edges and d(v) = 3) it executes even faster than when only a single 1-vertex cardinality constraint is present in the database. Another interesting fact is that the evaluation of a single cardinality constraint of a higher order results in highest QETs compared to other two cases with varying constraint order. This behaviour is most likely caused by the underlying recursion, which requires more steps to build the constraint pattern. The same behaviour can be observed in experiments, which include testing the *create_relationship* constraint against a real dataset (Table 2). The usage of recursive methods in the underlying implementation also influences the overall time complexity of the *create_relationship* procedure with additional costs caused by their logarithmic running time. Hence, the overall time complexity of the entire procedure can be estimated to $O(n^2 + n \log n)$.

Table 2
Query execution times of *create_relationship* procedure on real dataset with and without checking cardinality constraints (in milliseconds).

Constraint order	Cardinality check	Query execution time (ms)	QET change (%)
SingleSimpleCard	No Yes	492.5 496.5	0.8
SingleComplexCard	No Yes	509.1 667	23.67
CombCard	No Yes	500.7 521	3.89

Overall, the results obtained by these experiments show that the QET overhead caused by the implemented *k*-vertex cardinality constraint model can be measured within approximately 10–170 ms, which is acceptable for the current research scope, since the objective of this paper was to analyse the influence

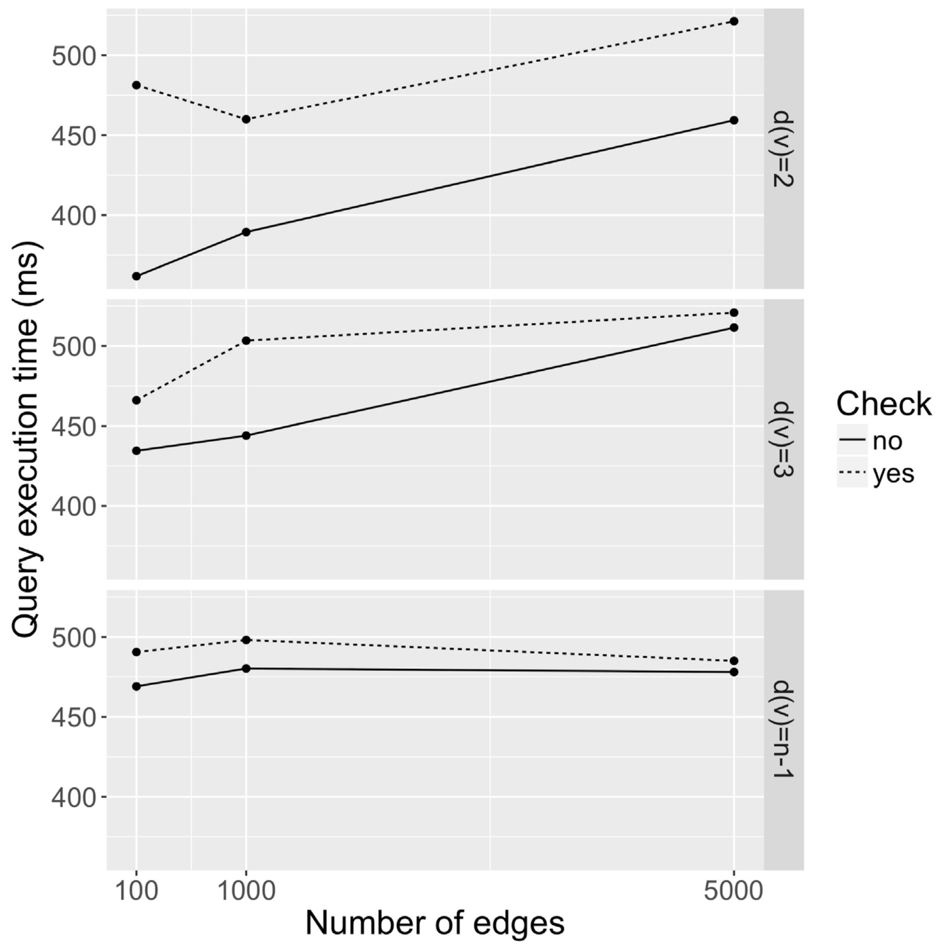


Fig. 5. Query execution times of the *create_relationship* procedure on synthetic datasets with and without checking cardinality constraints (in milliseconds) for the *CombCard* scenario.

of our novel *k*-vertex cardinality constraints model on query execution performance in Neo4j GDBMS.

7.2.2. Query execution times for identifying and eliminating violating edges

In our case, any edge, which exceeds the limit of maximum number of edges of a given type, allowed between a vertex and a subgraph, is considered to be violating cardinality constraint rule(s). However, currently the order, in which all edges are processed, corresponds to the order, in which they are retrieved from the database, which is to be further improved in future work (e.g. to include edge priority, and similar).

As expected, the results listed in Table 3 show that the QET of the procedure is highest in cases of combined cardinality constraints. Also, it is possible to notice that the queries take around 50 ms longer to execute with an increasing number of edges in the dataset. However, we can observe that in some cases, the QETs of the procedure on a dataset with 1000 and 5000 edges are almost the same (or even lower on 5000 edges). Once again, this can be attributed to the underlying graph traversal algorithm. However, in general, we can conclude that the QET of the procedure increases along with the increasing number of edges, outgoing vertex degree and cardinality constraint order in the dataset. The experiment results prove that the time complexity of the *check_constraint* procedure is exponential, i.e. $O(n^2)$.

7.2.3. Scalability analysis

To further analyse the performance of the proposed concept on larger datasets and more complex scenarios, additional synthetic datasets with a larger number of edges have been created (10,000, 100,000 and 500,000 edges). Additional experiments were performed to test the model scalability. In order to gain insights into the model performance in the most challenging environment, the experiments included only the most complex test scenarios, i.e. complete graphs with all vertices inter-connected (i.e. d(v) = n-1) and a simple and complex cardinality constraint (*CombCard* scenario) included in the dataset.

The experiment results presented in Fig. 6 include QETs of both *create_relationship* and *check_constraint* procedures on synthetic datasets with 100, 1000, 5000, 10,000, 100,000 and 500,000 edges.

The results indicate an exponential influence of increasing the number of edges on the *check_constraint* procedure, whereas the QET of the *create_relationship* procedure does not reveal a particularly significant increase. In the case of the *check_constraint* procedure, little difference can be observed between QETs when the number of edges is smaller than 10,000. On the largest dataset, with 500,000 edges, this procedure takes more than 1.5 s to execute. The higher increase in QET for this procedure is largely affected by the chosen strategy for handling edges violating cardinality constraint rules. Specifically, the strategy includes deleting

Table 3
Query execution times of *check_constraint* procedure on synthetic datasets (in milliseconds).

No of relationships	Constraint order	d(v) = 2	d(v) = 3	QET change (%)	d(v) = n−1	QET change (%)
100	SingleSimpleCard	514.4	523.5	1.74	562.2	8.50
	SingleComplexCard	604.7	641.7	5.76	649.7	6.92
	CombCard	747.4	636.4	−17,44	709.5	−5.34
1000	SingleSimpleCard	567.5	541.2	−4.86	584.7	2.94
	SingleComplexCard	626.2	614.9	−1.84	623	−0.51
	CombCard	705.9	720.4	2.01	751.1	6.01
5000	SingleSimpleCard	555.6	590.7	5.94	570.1	2.54
	SingleComplexCard	626.2	675.8	7.34	658.4	4.89
	CombCard	758.4	784	3.26	665.2	−14.01

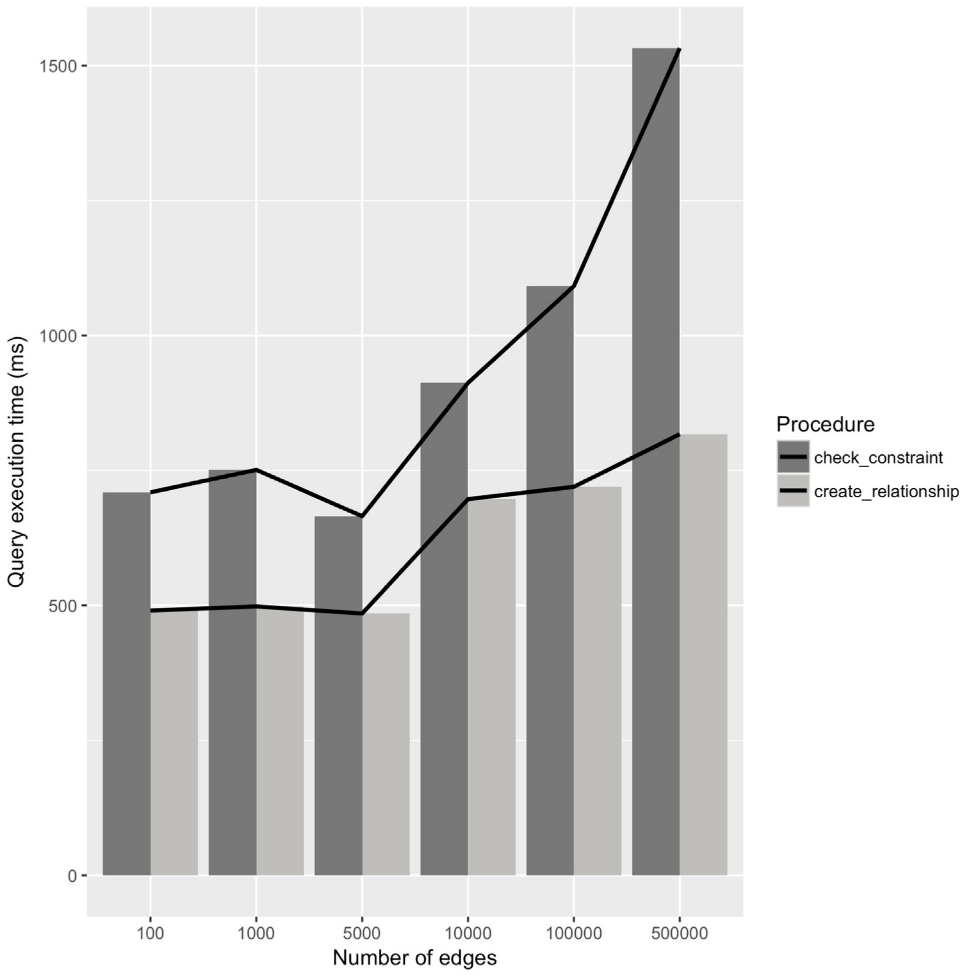


Fig. 6. Query execution times of procedures on an increasing number of edges in datasets.

violating edges from the database, which brings higher QET overhead with the increasing number of edges in the database. On the other side, the *create_relationship* procedure shows the most significant increase in QET after the number of edges to be traversed is over 10,000, which is most likely affected by the underlying page cache size.

8. Conclusion and future work

In this paper, we addressed the topic and challenges related to representing and implementing cardinality constraints in graph databases. The overview of related work and the analysis of current GDBMSs' features revealed that this topic has been poorly explored so far. Therefore, we focused on exploring the possibility of specifying higher order cardinality constraints on edges, which could include specifying cardinality constraints between a vertex and a subgraph.

The main original contribution presented in this paper is a novel concept and a definition of *k*-vertex cardinality constraints, which enables the specification of a minimum and maximum number of edges of type *R* between a vertex labelled *E* and subgraph *S* consisting of *k* vertices. We also demonstrated an approach to represent these constraints by using the property graph data model. Next, we demonstrated the model implementation by using stored procedures in the Neo4j GDBMS, which enable the creation of edges with and without checking cardinality constraints, as well as the identification of edges, which violate those constraints. The source code of the model implementation procedures is available on Github (https://github.com/MartinaSestak/Neo4jCardinalityConstraints).

We conducted a set of experiments on synthetic and real datasets to test the query performance of implemented procedures. Specifically, we evaluated how the query execution time is affected by the number of edges in the graph database, outgoing vertex degree (2, 3 or *n-1*) and the order of *k*-vertex cardinality constraints (1-, 2-vertex cardinality constraints and their combination).

In general, the model implementation does not significantly affect query performance, while ensuring that the database remains consistent and conformed with business rules, i.e. *k*-vertex cardinality constraints. The experiments on the *create_relationship* procedure showed that its QET decreases with the increasing number of edges and outgoing vertex degree. An interesting observation based on experiment results is that the *create_relationship* procedure with checking cardinality constraints on synthetic datasets executes faster in cases with combined 1- and 2-vertex cardinality constraints. As part of our future work, we will investigate the causes of this behaviour in more detail in order to prove that the underlying procedure implementation and graph traversal algorithm employed are the most probable causes of this behaviour. On the other hand, the QETs of the *check_constraint* procedure, which identifies edges violating the cardinality constraints, are as expected. The procedure takes longer to execute as the number of edges and outgoing vertex and *k-vertex* cardinality constraint orders increase (with some exceptions).

As part of our future work, we plan to further improve the *k*-vertex cardinality constraint model. At the moment, the *k*-vertex cardinality constraints definition focuses on limiting the number of edges with vertex and edge types in mind. Nevertheless, a valuable extension of the concept would be to include vertex/edge properties as well. The extended concept will be able to capture the business rules more vividly (e.g. that a credit card may be issued only to the owner of a given bank account). In our future research, we will evaluate the influence of adding vertex/edge properties to the proposed model on QETs of our procedures. We will investigate a more automatic approach to specifying *k*-vertex cardinality constraints from input business rules and explore the possibility of using text mining techniques for this purpose. Currently, the model implementation offers only the possibility of checking the upper limit (maximum) on the number of allowed edges between a vertex and a subgraph. Thus, we plan to implement the minimum aspect as well. Also, as already mentioned, different mechanisms for processing and detecting violating edges will be proposed in future work. Furthermore, the model will be extended to enable different actions to be performed on edges violating *k*-vertex cardinality constraints besides deleting them (e.g. moving them to another destination). Regarding the performance of the model implementation, we will also investigate if there are other solutions and possibilities to minimize the influence of physical-level aspects (e.g. using an in-memory graph database instead of Neo4j). With all this in mind, we can conclude that there is still a lot of space for future contributions to this topic, which will certainly increase the maturity of graph database technology. Once all these improvements are made to the *k*-vertex cardinality constraints model, the model will represent a flexible and easy-to-implement mechanism for maintaining graph database integrity. The underlying *k*-vertex cardinality constraints bring a richer semantics to cardinality constraints already supported in relational databases. For instance, by using these constraints, we can limit to how many calls for proposals published by a given European fund in a given program and topic that a specific type of user can apply for simultaneously. Therefore, it will enable us to use the full power of graph databases in representing relationships.

CRediT authorship contribution statement

Martina Šestak: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Marjan Heričko:** Conceptualization, Methodology, Software, Validation, Writing - review & editing, Supervision. **Tatjana Welzer Družovec:** Resources, Writing – review & editing. **Muhamed Turkanović:** Conceptualization, Methodology, Validation, Formal analysis, Writing - original draft, Writing – review & editing, Visualization, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was funded by the Slovenian Research Agency (research core funding No. P2- 0057), and also in part by EC H2020 Project CONCORDIA G.A. No. 830927.

Appendix A. Pseudocodes of the *k*-vertex cardinality constraints implementation algorithms

Procedure 1: create_relationship(queryPattern, constraintMode)

```
db ← Neo4j database connection object;
if constraintMode = "no_cardinality" then
    db.createRelationship(queryPattern)
else
    inputPattern ← property values omitted from queryPattern ;
    constraints ← db.retrieveConstraints() ;
    foreach Constraint c : constraints do
        constraintPattern ← buildConstraintPattern(c) ;
        if inputPattern = constraintPattern then
            firstNodeLabel ← label of the first vertex in inputPattern ;
            noRelationships ← db.getNumberOfRels(firstNodeLabel) ;
            if noRelationships < c.maxValue then
                db.createRelationship(queryPattern)
            end
        end
    end
end
```

Appendix B. Query execution times of the *create_relationship* procedure on synthetic datasets

See Table B.4.

Table B.4
Query execution times of *create_relationship* procedure on synthetic datasets with and without checking cardinality constraints (in milliseconds).

No of edges	Constraint order	Cardinality check	d(v) = 2	QET change (%)	d(v) = 3	QET change (%)	d(v) = n−1	QET change (%)
100	SingleSimpleCard	No Yes	380.4 465.6	18.29	385.8 485.9	20.6	422.5 498.1	15.18
	SingleComplexCard	No Yes	424.6 550.1	22.81	475.2 606.7	21.67	499.2 603.5	17.28
	CombCard	No Yes	361.8 481.3	24.82	434.5 466.1	6.77	469 490.5	4.38
1000	SingleSimpleCard	No Yes	393.5 496.3	79.28	402.5 481.1	16.34	501.5 527.3	4.89
	SingleComplexCard	No Yes	444.9 541.3	17.81	464.8 619.6	24.98	582.5 650.6	10.46
	CombCard	No Yes	389.4 460	15.35	444 503.4	11.8	480.2 498.1	3.59
5000	SingleSimpleCard	No Yes	430.9 474.3	9.15	488.5 509.4	4.10	502.1 514.1	2.33
	SingleComplexCard	No Yes	501.7 539.2	6.95	568.1 639.9	11.22	553.8 642.5	13.80
	CombCard	No Yes	459.4 521.3	11.87	511.6 520.9	1.78	478 485	1.44

Procedure 2: check_constraint(outputFilePath)

```
db ← Neo4j database connection object ;
constraints ← db.retrieveConstraints() ;
initialize dataMap < Relationship, Integer > ;
initialize listRegularRels ;
initialize listViolatingtRels ;
foreach Constraint c : constraints do
    clear listRegularRels ;
    clear listViolatingtRels ;
    constraintPattern ← buildConstraintPattern(c) ;
    vertexRelationshipResult ← db.retrieveResults
    (constraintPattern) ;
    foreach Result r : vertexRelationshipResult do
        if dataMap.contains(r.getStartVertex() then
            currentNoRels ← r.getStartVertex().value ;
            if currentNoRels < c.maxValue then
                increase r.getStartVertex().value by 1 ;
                add r to listRegularRels ;
            else
                add r to listViolatingtRels
            end
        else
            put (r, 1) to dataMap ;
        end
    end
end
print check result to file(outputFilePath) ;
foreach Relationship r : listViolatingtRels do
    db.deleteRelationship(r) ;
end
```

References

[1] R. Angles, C. Gutierrez, Querying rdf data from a graph database perspective, in: European Semantic Web Conference, Springer, 2005, pp. 346–360.
[2] R. Angles, C. Gutierrez, Survey of graph database models, ACM Comput. Surv. 40 (1) (2008).
[3] R.H. Güting, Graphdb: Modeling and querying graphs in databases, in: VLDB, Vol. 94, 1994, pp. 12–15.
[4] S. Flesca, S. Greco, Querying graph databases, in: International Conference on Extending Database Technology, Springer, 2000, pp. 510–524.
[5] D. Hay, K.A. Healy, J. Hall, Final Paper: DeFining Business Rules-What are They Really, Tech. Rep. 34, Business Rule Group, 2000.
[6] G. Simsion, G. Witt, Data Modeling Essentials, Elsevier, 2004.
[7] H. Herbst, G. Knolmayer, T. Myrach, M. Schlesinger, The specification of business rules: A comparison of selected methodologies, in: Methods and Associated Tools for the Information Systems Life Cycle, Citeseer, 1994, pp. 29–46.
[8] I. Robinson, J. Webber, E. Eifrem, Graph Databases, " O'Reilly Media, Inc.", 2013.
[9] F. Boufares, H. Bennaceur, Consistency problems in er-schemas for database systems, Inform. Sci. 163 (4) (2004) 263–274.
[10] B. Thalheim, Entity-Relationship Modeling: Foundations of Database Technology, Springer Science & Business Media, 2013.
[11] M. Balaban, P. Shoval, Enforcing cardinality constraints in the er model with integrity methods, in: Advanced Topics in Database Research, Volume 1, IGI Global, 2002, pp. 1–16.
[12] Neo4j Inc., Chapter 5. schema, 2019, accessed April 22, 2019 at https://neo4j.com/docs/cypher-manual/current/schema/.
[13] OrientDB, Using schema with graphs, 2019, accessed June 4, 2019 at http://orientdb.com/docs/last/Tutorial-Using-schema-with-graphs.html.
[14] JanusGraph Authors, Chapter 5. schema and data modeling, 2017, accessed April 22, 2019 at https://docs.janusgraph.org/latest/schema.html.
[15] Expero, Schema support in three property graph databases, 2018, accessed April 22, 2019 at https://medium.com/@experoinc/schema-support-in-three-property-graph-databases-1beff569855e.
[16] C. Allen, C. Creary, S. Chatwin, Introduction to Relational Databases, McGraw-Hill, New York, NY, 2003.
[17] J. Pokornỳ, M. Valenta, J. Kovačič, Integrity constraints in graph databases, Procedia Comput. Sci. 109 (2017) 975–981.
[18] G. Fletcher, J. Hidders, J.L. Larriba-Pey, Graph Data Management, Springer, 2018.
[19] R. Angles, A comparison of current graph database models, in: 2012 IEEE 28th International Conference on Data Engineering Workshops, IEEE, 2012, pp. 171–177.
[20] R. Angles, The property graph database model., in: Alberto Mendelzon International Workshop on Foundations of Data Management, 2018.
[21] D.C. Fernandez, P.M. Fernandez, E.C. Galan, Dealing with relationship cardinality constraints in relational database design, in: Effective Databases for Text & Document Management, IGI Global, 2003, pp. 288–317.
[22] A. McAllister, Complete rules for n-ary relationship cardinality constraints, Data Knowl. Eng. 27 (3) (1998) 255–288.
[23] J. Galindo, A. Urrutia, R.A. Carrasco, M. Piattini, Relaxing constraints in enhanced entity-relationship models using fuzzy quantifiers, IEEE Trans. Fuzzy Syst. 12 (6) (2004) 780–796.
[24] F. Currim, N. Neidig, A. Kampoowale, G. Mhatre, The card system, in: International Conference on Conceptual Modeling, Springer, 2010, pp. 433–437.
[25] R. Camps, Transforming N-ary Relationships to Database Schemas: An Old and Forgotten Problem, Research Repot LSI-5-02R of Universitat Politècnica de Catalunya (Spain), 2002.
[26] C. Soutou, Modeling relationships in object-relational databases, Data Knowl. Eng. 36 (1) (2001) 79–107.
[27] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, Nose: Schema design for nosql applications, IEEE Trans. Knowl. Data Eng. 29 (10) (2017) 2275–2289.
[28] S. Banerjee, A. Sarkar, Modeling nosql databases: from conceptual to logical level design, in: 3rd International Conference Applications and Innovations in Mobile Computing (AIMoC 2016), Kolkata, India, February, 2016, pp. 10–12.

[29] Neo4j Inc., Cypher query language, 2020, accessed March 26, 2020 at https://neo4j.com/developer/cypher-query-language/.

[30] L. Zou, K. Xu, J.X. Yu, L. Chen, Y. Xiao, D. Zhao, Efficient processing of label-constraint reachability queries in large graphs, Inf. Syst. 40 (2014) 47–66.

[31] G. Daniel, G. Sunyé, J. Cabot, Umltographdb: mapping conceptual schemas to graph databases, in: International Conference on Conceptual Modeling, Springer, 2016, pp. 430–444.

[32] A. TinkerPop, The gremlin graph traversal machine and language, 2019, accessed March 26, 2020 at https://tinkerpop.apache.org/gremlin.html.

[33] N. Roy-Hubara, L. Rokach, B. Shapira, P. Shoval, Modeling graph database schema, IT Prof. 19 (6) (2017) 34–43.

[34] A. Ghrab, O. Romero, S. Skhiri, A. Vaisman, E. Zimányi, Grad: On graph database modeling, 2016, arXiv:arXiv:1602.00503.

[35] V.M. de Sousa, L.M.d.V. Cura, Logical design of graph databases from an entity-relationship conceptual model, in: Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services, 2018, pp. 183–189.

[36] O. Hartig, J. Hidders, Defining schemas for property graphs by using the graphql schema definition language, in: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), ACM, 2019, p. 6.

[37] M. Sedlmeier, M. Gogolla, Design and prototypical implementation of an integrated graph-based conceptual data model, in: EJC, 2014, pp. 376–395.

[38] J.L. Gross, J. Yellen, Graph theory and its applications, CRC press, 2005.

[39] A. Olivé, Cardinality constraints, in: Conceptual Modeling of Information Systems, Springer Berlin Heidelberg, 2007, pp. 83—102.

[40] J. Mattingley, Graph theory (basics), 2014, accessed April 17, 2019 at https://www.cs.colorado.edu/~srirams/courses/csci2824-spr14/graphs-29.html.

[41] Neo4j Inc., 1.2.4. user-defined procedures, 2019, accessed April 17, 2019 at https://neo4j.com/docs/java-reference/current/extending-neo4j/procedures-and-functions/procedures/.

MARTINA ŠESTAK received her Master's degree in Information and Software Engineering from the Faculty of Organization and Informatics, University of Zagreb in 2016. She is currently a Ph.D. student in Computer Science at Faculty of Electrical Engineering and Computer Science in Maribor. She is currently a Teaching Assistant and a member of Laboratory for Information Systems at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Her main research interests include graph databases, data analytics and knowledge graphs.

MARJAN HERIČKO received the Ph.D. degree in computer science and informatics from the University of Maribor in 1998. He is currently a Full Professor with the Faculty of Electrical Engineering and Computer Science, University of Maribor, where he is also the Head of the Institute of Informatics. His main research interests include all aspects of IS development with emphasis on software and service engineering, software process improvement, data, and process modelling.

TATJANA WELZER is a Full Professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. She is the Head of the Database Technologies Laboratory of the Institute of Informatics. She received her Ph.D. in Computer Science in 1995. Her research work covers areas of database technologies, cross-cultural communication and problems in media communications

MUHAMED TURKANOVIĆ received the B. Sc. degree and the Ph. D. degree in Computer Science and Informatics from the University of Maribor in 2011 and 2016, respectively. His Ph.D. thesis was on authentication protocols for the Internet of Things. He has authored several highly cited scientific articles, published in renowned journals with an impact factor in the field of computer science. He was a Managing Director and a CTO of an IT company from 2013 to 2016. In 2017, he joined the Faculty of Electrical Engineering and Computer Science, University of Maribor, as an Assistant Professor of information technology. He is the head of a multidisciplinary group of researchers called Blockchain Lab:UM. His current research interests include Advanced Database Technologies, Cryptography and Blockchain.