Modeling and Enforcing Integrity Constraints on Graph Databases [⋆]

Fábio Reina[0000−0002−7193−3639], Alexis Huf[0000−0003−2723−0546], Daniel Presser[0000−0002−2134−20485], and Frank Siqueira[0000−0002−8275−5751]

Graduate Program in Computer Science, Department of Informatics and Statistics
Federal University of Santa Catarina – Florianópolis, Brazil

Abstract. The enormous volume and high variety of information that is constantly produced by computing systems requires storage technologies able to provide high processing velocity and data quality. The suitability for modeling complex data and for delivering performance are characteristics that are making graph databases become very popular. However, existing limitations still prevent database management systems that adopt the graph model to fully ensure data consistency, given that the means for ensuring data consistency are usually nonexistent or at most very simple. This work intends to overcome this limitation by extending the support for defining and enforcing integrity constraints on graph databases, in order to prevent the graph to reach an inconsistent state and compromise the correctness of applications. The proposed integrity constraints are implemented on OrientDB. Experimental results show that the prototype implementation can improve the performance in comparison to verification of constraints on a client application.

Keywords: Graph databases · data consistency · integrity constraints · data integrity · OrientDB.

1 Introduction

A well-known fact regarding the present state of information technology is that the amount of data produced and stored by computer systems is in constant growth. This phenomenon, known as Big Data, is nowadays the subject of a substantial amount of research work. Not only data is produced in larger *volumes*, but it is generated by multiples sources in different formats (*variety*), and must be stored and processed quickly (*velocity*) without compromising its *validity*.

Along with the volume of data, the need for better performance and for more efficient management became very relevant issues [13]. However, the traditional databases (DBs) are not always able to handle all the requirements of such large volumes [17]. As a result, the category of Database Management Systems (DBMSs) known as Not only SQL (NoSQL) has emerged aiming to fulfill these requirements. The main characteristic of NoSQL DBMSs is that data consistency is relaxed with the aim of improving performance. It allows data to be inconsistent for some time, which may be prohibitive for some applications. Among

⋆ Financed in part by CAPES ("Coordenação de Aperfeiçoamento de Pessoal de Nível Superior") - Brasil - Finance Code 001 and by FAPESC ("Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina").

the existing categories of NoSQL systems, graph DBMSs have been increasing in popularity, and are the subject of this work.

In general, when graph DBMSs offer support for specifying and enforcing Integrity Constraints (ICs), the supported ICs are usually very limited. The most common constraints allow database administrators to enforce attributes to have unique values, limit minimum and maximum values, and define attribute types. Due to the lack of support for more complex ICs, data validation often becomes the responsibility of the application that uses the graph DB, resulting in an increase of development effort.

The work presented in this paper aims to tackle the lack of support for complex ICs in graph DBMSs. Thus, it is possible to create rules using two or more elements (attribute, node or edge) at the same time. Therefore, we propose a specification syntax for ICs as well as a mechanism for their enforcement during operations that modify the graph. To achieve this goal we define six new ICs and implement them on the OrientDB, allowing the definition of: (1) conditions on node attributes, (2) required edges, (3) type of in/out nodes of an edge, (4) edge cardinality, (5) bidirectionality of edges, and (6) conditions on attributes of nodes linked by an edge. We evaluate the impact of these constraints over the performance of OrientDB. With these extensions, we intend to transfer the responsibility for validating data constraints from the client application to the DBMS, enforcing data integrity with less effort when developing applications.

The remainder of this paper is organized as follows. Section 2 presents the most relevant concepts used in this work. Section 3 explains our proposal, specifying in more detail the ICs that are supported by our extended version of OrientDB. Then, Section 4 describes the evaluation study performed over our implementation. Next, Section 5 identifies some similar proposals described in the literature and compares them to the solution proposed in this paper. Finally, Section 6 presents the conclusions reached with the development of this work and singles out some open issues that require further research.

2 Fundamental Concepts

A Graph DB is essentially a DB that uses the explicit structure of a graph to store, query and manipulate data. Vertices, also called nodes, represent database records, while edges represent relationships between data [3]. In general, every edge has a label that identifies the relationship it represents. Vertices and edges may also have properties. Therefore, edges are as important as vertices, due to the potentially relevant information they carry. Due to its composition of vertices, edges and properties, this structure is said to be a property graph [19]. This work considers property graphs as defined in Definition 1. This definition was adapted from [2] to remove multiple labels and multiple attribute values, aiming to better align with current graph DBMSs implementations, including Neo4J, OrientDB, InfinityGraph, Trinity, Titan and ArangoDB.

Definition 1. *A property graph is a tuple $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$, such that:*

1. *N is a set of nodes;*

2. E is a set of edges;
3. $\rho : E \rightarrow (N \times N)$ is a function that associates an edge in E with a pair (o, t) of origin and target nodes in N;
4. $\lambda_N : N \rightarrow L_N$ is a function that associates a node with a node label;
5. $\lambda_E : E \rightarrow L_E$ is a function that associates a edge with a edge label;
6. $\sigma : (N \cup E) \times P \rightarrow K$ is a function that given a node or edge together with a property P, associates the pair to a value from K.

The representation of highly connected data in a relational model is possible, however it results in several many-to-many relationships. A query under this conditions may require a big number of joins, which can degrade the performance [18]. In contrast, the graph structure allows a better management of this type of data, resulting in faster query processing [5]. The most recurring example of application that benefit from using graph DBMSs are social networks. Nevertheless, graph DBs are very useful in financial systems, for fraud detection and transaction monitoring [6,18]; on document analysis to analyze speech data and identify stakeholders' intention [15]; on the retail sector, helping with decision making and product recommendation [1]; among several other applications.

Differently from relational DBs, which have a well-defined schema, graph DBs do not have a rigid structure. It means that while in a relational DB an insert operation can only set properties defined in the DB schema, in a graph DB it can add new properties that were not defined on the schema. This characteristic results in faster execution of operations. However, it becomes harder to impose ICs, given that there is no schema to follow. As a result, graph DBs in general either do not provide tools to ensure consistency, or only support very basic ICs.

Graph DBs are categorized under a larger category of DBMSs, named NoSQL DBs. Unlike the acronym suggests, this category does not preclude the use of SQL, but indicates the use of alternatives to the relational model, which backs SQL. They also follow Basically Available, Soft state, and Eventual consistency (BASE) properties, which imply that the DB has to be available most of the time for read and write operations. It also indicates that data may be inconsistent during some time, but will become accurate in a future moment [16].

In DBs with BASE properties, consistency refers to transactions performing reads on up-to-date and committed data. Correctness of data is not a concern of consistency, but rather a concern of data integrity. Thus, data integrity can be defined as the maintenance and assurance of the data correctness during all its life cycle [14]. It is a major concern of many systems, especially those that mange real world data, and can be achieved by the use of a set of rules that specify all the allowed update over the data. In this scenario comes the concept of ICs, which can be described as a set of general rules that define a consistent state of the DB, as well as allowed modifications [3]. These rules must be applied on every inserted data to avoid inconsistency. If the integrity cannot be secured by the DBMS, its assurance must be implemented at the application level.

In relational DBMSs, the constraints are linked to one of the following categories: entity integrity, referential integrity, and domain integrity [9]. The use of ICs ensures that the information stored on the DB conforms to these rules and, as a result, enforces integrity. However, most of the graph DBMSs available on the market nowadays do not support ICs, or only allow elementary rules to be defined, that are unable to provide the level of integrity required by most of the applications that store and manipulate graph data.

3 Extending Support for Integrity Constraints in Graphs

This work proposes an extension of a graph DBMS to allow the specification of complex ICs, which are useful for applications that need to store data in the form of a graph database with a high level of integrity. The graph DBMS chosen to receive this extension was OrientDB[1]. One of the features that motivated its choice is the fact that OrientDB already provides support for the definition of simple constraints. Besides that, it is ranked among the most popular graph DBMSs, according to DBEngines[2].

In OrientDB, every class, also known as node type, is associated to a set of metadata. The proposed extension aims to store the constraint definitions into these metadata. Each IC is defined by the OrientDB client application using our extension of the OrientDB Data Definition Language (DDL). After an IC is stored on the metadata, any modification to the set of instances of that class will be validated against the IC before the corresponding transaction commits.

All ICs already supported by OrientDB are restricted to comparing values between the data being inserted or updated against threshold values defined in OrientDB metadata for a given node type. One approach to implement more complex ICs not limited to node attributes is to dynamically compute node attributes and compile definitions of such complex ICs into simpler ICs that are limited to checking node attribute values. However, this strategy incurs a large overhead in the form of such dynamic attributes and the potentially large amount of metadata required to implement the ICs. To avoid such drawbacks, the extension adds a new first-class component, the constraint manager, to the internal architecture of OrientDB. This component provides its own constraint validation mechanisms that are used by the new IC types, without relying on the constraints already supported by OrientDB. The use of a dedicated constraint manager allows for ICs that also involve edges, in addition to nodes and properties. Its existence also eases the introduction of new constraint types by providing a single interception point for their validation. The extension proposed in this work introduces support for six new IC types: node condition, required edge, in/out, edge cardinality, bidirectional edge and edge condition.

The constraint manager is initialized together with the OrientDB server and uses an SB-Tree [8] to store and manage *Constraint* objects. When the user defines an IC using the `CREATE CONSTRAINT` command, the parser extracts the constraint type, its target and constraint-specific arguments. The result of the parsing is fed into a constraint factory and the resulting Constraint object is

[1] https://orientdb.com/
[2] https://db-engines.com/en/ranking/graph+dbms

serialized into the SB-Tree. In addition to adding the constraint manager, the OrientDB DDL grammar was extended to support the new constraint types.

Once the constraint is declared and stored on the SB-Tree, every write operation on the graph DB triggers the validation of the relevant *Constraint* objects. After locating all relevant *Constraint* objects, the constraint manager collects all relevant data for the write operation and invokes the validation procedure of each *Constraint* object. If any *Constraint* validation fails, an exception is thrown, leading to the abortion of the whole transaction.

The syntax of each new IC supported is presented in the remainder of this section, together with a formal definition. All definitions use the property graph defined by [2] and presented in Definition 1.

Some constraints allow for relational operators, which are denoted by $\alpha \in \mathcal{O}$, where $\mathcal{O} = \{<, \leq, =, \neq, \geq, >\}$. Node and edge types are denoted using the notation $\mathcal{T}_\beta$, where $\beta \in L$ is a label that identifies the type unambiguously. Formally, types are defined by the sets of their instances: $\mathcal{T}_\beta = \{x \mid \lambda_N(x) = \beta \wedge \lambda_E(x) = \beta\}$. Given that, labels for node and edges are disjoint, $\beta \in L_N \iff \mathcal{T}_\beta \subseteq N$ and $\beta \in L_E \iff \mathcal{T}_\beta \subseteq E$.

Node Condition Constraint. The goal of this constraint is to compare properties of a node and to validate values assigned to them according to a previously defined condition. Therefore, this IC is applied to the node class. It is formally specified by Definition 2 and the general syntax is shown in Figure 1.

Definition 2. *Given a property graph $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$, a Node Condition constraint is a tuple Cond $= (\mathcal{T}_o, p_1 \in P, \alpha_1 \in \mathcal{O}, k_1 \in K, p_2 \in P, \alpha_2 \in \mathcal{O}, k_2 \in K, p_3 \in P, \alpha_3 \in \mathcal{O}, k_3 \in K)$, such that:*

$$\forall o \in \mathcal{T}_o \ . \ \begin{cases} \sigma(o, p_2)\ \alpha_2\ k_2, & \text{if } k_1 \in \sigma(o, p_1) \\ \sigma(o, p_3)\ \alpha_3\ k_3, & \text{otherwise} \end{cases}$$

```
<CREATE> <CONSTRAINT> name <ON> class <(> attribute <)> <CONDITIONAL>
<(>  <IF> property (>|<|>=|<=|=|!=) expression
     <THEN> property (>|<|>=|<=|=|!=) expression
     [ <ELSE> property (>|<|>=|<=|=|!=) expression ]  <)>
```

Fig. 1: General syntax of the Node Condition constraint.

Required Edge Constraint. This IC defines that a node class has a mandatory outgoing edge of a given type, that will point to an instance of a target node class. Therefore, if there is a node whose class appears as origin class in the constraint, then there must be at least one edge leaving this node and arriving at a node with the given target class. The constraint is associated with the origin node class metadata. This IC is formally described by Definition 3 and the general syntax is shown in Figure 2.

Definition 3. *Given a property graph $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$, a Required Edge constraint is a tuple Req $= (\mathcal{T}_o, \mathcal{T}_e, \mathcal{T}_t)$ such that: $\forall o \in \mathcal{T}_o \ : \ (\exists e, t \ : \ e \in \mathcal{T}_e \wedge t \in \mathcal{T}_t \wedge \rho(e) = (o, t))$.*

```
<CREATE> <CONSTRAINT> name <ON> origin_class
    <REQUIRED_EDGE> [edge_type] <TO> target_class
```

Fig. 2: General syntax of the required edge constraint.

In/out Constraint. This constraint type restricts the classes of nodes that are connected by an edge class. At the same time, this constraint is also useful to enforce the direction of the represented relationship. For example, a person authors a document, and not the other way around. Unlike the previous constraints, this one is associated to the edge class metadata instead of the node class. This IC can be formally specified as shown by Definition 4 and the general syntax is shown in Figure 3.

Definition 4. *Given a property graph* $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$*, an In/Out constraint is a tuple* $IO = (\mathcal{T}_o, \mathcal{T}_e, \mathcal{T}_t)$ *such that:* $\forall e \in \mathcal{T}_e \ : \ o \in \mathcal{T}_o \ \wedge \ t \in \mathcal{T}_t$*, where* $\rho(e) = (o, t)$*.*

```
<CREATE> <CONSTRAINT> name <ON> edge_type <IN_OUT_EDGE> (
   <FROM> origin_class <TO> target_class | <FROM> origin_class
   | <TO> target_class )
```

Fig. 3: General syntax of the in/out constraint.

Cardinality Constraint. The goal of this IC is to restrict the number of edges of a given class that connect an origin node to target nodes. The cardinality specification consists of two integer numbers, with N serving as a placeholder for “unspecified”, i.e., any number is allowed. Unlike in/out constraints, cardinality constraints are associated with the origin node class and therefore do not allow an unspecified origin node class. Another important difference between both constraints is the form of validation. Since OrientDB associates node identification numbers with classes, in/out validation can be implemented performing no reads on the DB. Figure 4 shows the general syntax of this constraint type. The constraint is always associated with the origin node class and to the edge class. After specification of the cardinality, one may optionally specify the destination node class. This constraint is formally specified by Definition 5.

Definition 5. *Given a property graph* $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$*, a cardinality constraint is a tuple* $Card = (\mathcal{T}_o, \mathcal{T}_e, \mathcal{T}_t, \alpha \in \mathcal{O}, k_o \in K \ \wedge \ k_t \in K)$ *such that:* $\forall o \in \mathcal{T}_o \ : \ ||\{(e, t) \mid (e, t) \in (\mathcal{T}_e \times \mathcal{T}_t) \wedge \rho(e) = (o, t)\}|| \ \alpha \ k_o \wedge \ ||\{(e, o) \mid (e, o) \in (\mathcal{T}_e \times \mathcal{T}_o) \wedge \rho(e) = (t, o)\}|| \ \alpha \ k_t$*.*

```
<CREATE> <CONSTRAINT> name <ON> origin_class <CARDINALITY>
  [edge_type] <(> <INT | N> <..> <INT | N> <)> [<TO> target_class ]
```

Fig. 4: General syntax of the cardinality constraint.

Bidirectional Edge Constraint. The purpose of this IC is to ensure bidirectionality of the direction of a relationship. That is, given two nodes, if there is an edge from A to B, there must be another edge of the same type linking B to A. The constraint is associated with the edge metadata and its validation verifies if the specified nodes are linked by an incoming and an outgoing edge of the required type. This constraint is formally described by Definition 6 and its general syntax is shown in Figure 5.

Definition 6. *Given a property graph $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$, a Bidirectional Edge constraint is a tuple $BiDir = (\mathcal{T}_o, \mathcal{T}_e, \mathcal{T}_t)$ such that the following two conditions hold:*

$$\begin{array}{lll} \forall\ e \in \mathcal{T}_e : & \exists\ \bar{e} : & \rho(e) = (o,t) \implies \rho(\bar{e}) = (t,o)\ \wedge\ t \in \mathcal{T}_t\ \wedge\ o \in \mathcal{T}_o \\ \forall\ \bar{e} \in \mathcal{T}_e : & \exists\ e : & \rho(\bar{e}) = (t,o) \implies \rho(e) = (o,t)\ \wedge\ o \in \mathcal{T}_o\ \wedge\ t \in \mathcal{T}_t \end{array}$$

```
<CREATE> <CONSTRAINT> name <ON> edge_type <BIDIRECTIONAL_EDGE>
    <BETWEEN> origin_class <AND> target_class
```

Fig. 5: General syntax of the bidirectional edge constraint.

Edge Condition Constraint. The goal of this IC is to enforce a condition over attibutes in both ends of an edge. In other words, it compares property values of the two nodes connected by one edge. It is associated with the edge metadata and also enforces the direction of the relationship. It can be formally defined as shown in Definition 7 and its syntax is presented in Figure 6.

Definition 7. *Given a property graph $G = (N, E, \rho, \lambda_N, \lambda_E, \sigma)$, an edge condition constraint is a tuple $ECond = (\mathcal{T}_e, p_o \in P, p_t \in P, \alpha \in O)$, such that: $\forall e \in \mathcal{T}_e\ :\ \rho e = (o,t) \implies \sigma(o, p_o)\ \alpha\ \sigma(t, p_t)$*

```
<CREATE> <CONSTRAINT> name <ON> edge_type <EDGE_CONDITION>
    origin_class<.>property (>|<|>=|<=|=|!=) target_class<.>property
```

Fig. 6: General syntax of the edge condition constraint.

Since all constraints are built on top of the formalization in Definition 1, which is aligned with several other graph DBMSs, the constraints are applicable to those DBMSs as well. In general, applying the method to other DBMS involves modifying the host DBMS for three tasks: parsing the constraints, storing them and validating them. Parsing involves extending the DDL or creating one with the IC syntax presented in this work, so the DB recognizes the `CREATE CONSTRAINT` commands. The presented implementation of the constraint manager uses an SB-Tree to store and manage constraint objects because this is a general use index algorithm already provided by OrientDB. In other DBMSs, this structure could be replaced by similar indexing algorithms, such as B-Trees. In addition, a trigger to the validation routines must be implemented after procedures that recognize the `create`, `delete` and `update` commands of the DBMS.

4 Evaluation

The support for ICs proposed in this paper was implemented on OrientDB 3.1.0. To evaluate the impact of the modifications, the most relevant and quantifiable factor is the execution time of operations in the DBMS. Therefore, this evaluation focuses on the impact of IC validation on query execution time.

The strategy adopted assumes that ICs are originated from business modeling and not only from implementation aspects. As a result, when a developer selects a weakly consistent DBMS or one without support for enforcing ICs, constraint validation must be enforced by logic introduced in the application code. That way, the experiments are built to allow the comparison of three variants of the OrientDB. The first one is the *Original* OrientDB server, without any modification. The *Modified* variant consists in a version of OrientDB modified to incorporate the constraint manager and to accept the definition of the six new ICs described in the previous section. Finally, the *Application* variant corresponds to performing IC validation within the client application and sending the corresponding data manipulation commands to the original OrientDB server.

The source code for reproducing the experiments is available in a public repository[3]. Countermeasures were adopted to avoid spurious interference on the results provoked by the Java Virtual Machine (JVM). A new pair of JVMs is created for every measurement: one JVM executes the OrientDB server while the other executes the test client code (`constraints-tests-client`), which sends the test transaction to the server and measures its execution time. Both JVMs are created on the same host – an Intel i7-4510U dual-core at 2.0 GHz, with 16 GB of RAM, running Ubuntu 18.04.3 LTS 64bit (kernel 5.0). A single measurement consists of two phases, each executed on a new pair of client and server JVMs. In the first phase, the client sets up the DB within OrientDB using administration commands. This setup includes basic schema information, including ICs, and population of the DB where applicable. On the second phase, that is executed on a new pair of client/server JVMs, 100 analogous transactions are executed 3 times. The number of transactions aims to avoid unreliable measurement of fast operations and the 3 executions aim to avoid interference from non-deterministic background tasks, such as the GC (Garbage Collector), disk caching and JIT (Just In Time) compilation. Between each of these 3 executions, disk caches are flushed (using the `sync()` system call) and the GC is requested to run. Only the third execution had its time recorded and was considered for analysis.

The first scenario evaluates the *Node Condition* constraint. In this experiment, each transactions tries to modify the values of properties of one instance of class *Person*, violating the constraint. The constraint imposed in this first experiment is shown in Figure 7.

The next three scenarios evaluate the *Required Edge*, *In/out* and *Cardinality* constraints. Figure 8(a) presents the constraints created in these three scenarios. The DB was populated with 100 replicas of the structure shown in Figure 8(b); then, 100 transactions are executed sequentially, resulting in the structure shown

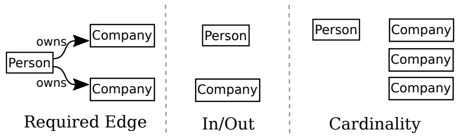
[3] https://bitbucket.org/fmreina/orient-driver/src/master/

```
CREATE CONSTRAINT cond ON Person (attr1) CONDITIONAL
(IF attr2 < 3 THEN attr1 < 2 ELSE attr1 > 4);
```

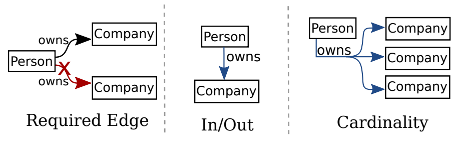
Fig. 7: Node Condition constraint created in the first experiment.

```
CREATE CONSTRAINT req ON Person
REQUIRED_EDGE owns TO Company;

CREATE CONSTRAINT inout ON owns
IN_OUT_EDGE FROM Person TO Company;

CREATE CONSTRAINT card ON Person
CARDINALITY owns N .. 3 TO Company;
```

(a) Constraints created during experiments



(b) DB population template



(c) Transaction templates

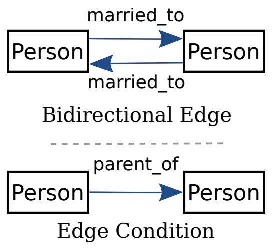
Fig. 8: Constraints (a), models for DB population (b) and transactions (c).

in Figure 8(c). The *Required Edge* IC is created to enforce the existence of at least one edge of type *owns* between nodes *Person* and *Company*. In the *In/Out* case, the IC is created on the edge class *owns* to specify that edges of this type are only valid if they have a node *Person* as source and a node *Company* as target. Finally, the *Cardinality* IC is created on the node class *Person* to limit the number of nodes of type *Company* the same person can own.

The transaction that tests the *Required Edge* scenario, due to the nature of this IC, performs an edge removal rather than an insertion. In the *Original* and *Modified* variants, a single command is sent to OrientDB per transaction. In the *Application* variant, this is not possible. Therefore, the application performs a *SELECT* operation for each node to validate the existing edge cardinality, and then sends a command that creates the edges between the nodes. The same approach is adopted for the *Application* variant in all scenarios, as it is necessary to validate the IC before actually performing the intended operation.

The last experiments evaluate the other two ICs proposed in this paper: bidirectional edge and edge condition. Figure 9(a) illustrates the transactions for each scenario where new edges are created meeting the constraint requirements. As with the first batch, this operations can be executed with a single command in the variants *Modified* and *Original*. However, in the variant *Application*, it is necessary to perform more operations to query the involved nodes and run the validation routine before the command to create the edges is sent to the DB.

The results obtained with the three variants (*i.e., Original, Modified* and *Application*) are illustrated in Figure 10. In this figure, narrow boxes with white background represent the central quartiles, totaling 50% of measurements and are divided into the median. For the limits, from which outliers are found as points, we used $min(Maximum, Median \pm 1.5IQR)$, where IQR is the *Inter-Quartile Range*, to the height of the boxes. In addition to the classic elements of a box diagram, the average, represented by triangles, and the 95% confidence



(a)

```
CREATE CONSTRAINT bidirec ON married_to
BIDIRECTIONAL_EDGE BETWEEN Person AND Person;

CREATE CONSTRAINT econd ON parent_of
EDGE_CONDITION (Person.age > Person.age);
```

(b)

Fig. 9: Models for DB population and transactions (a), and ICs created (b).

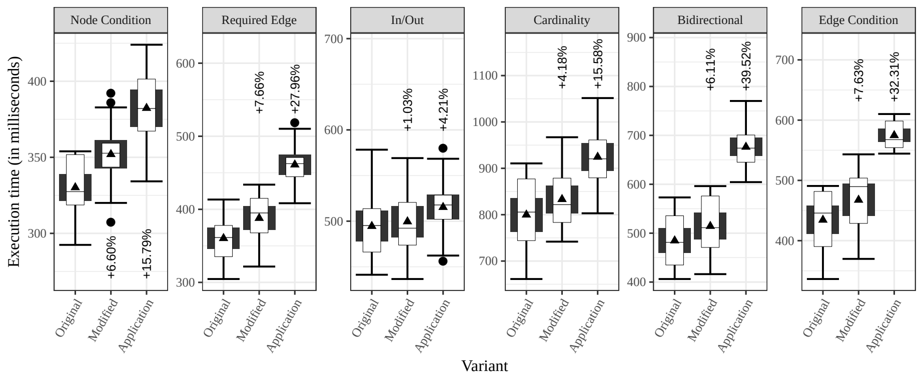


Fig. 10: Impact of constraint checking on the average time required to execute a batch of 100 write transactions.

interval (CI), represented by wide filled rectangles, have been added to the illustration. The labels on the bars show the increase of the runtime as a proportion of the variation *Original*.

The first box diagram in Figure 10 presents the results for the *Node Condition* constraint. There is a performance cost for using the *Modified* OrientBD to impose the constraint, but this cost is significantly lower than doing the required validation at application level.

The second, third and fourth plots show the runtime of the *Required Edge*, *In/out* and *Cardinality* scenarios using the three variants mentioned before. In all scenarios it is also observed that IC validation using the modified OrientDB is, on average, more efficient than validation at the application. Another important conclusion is that there is a large intersection between the performance observed with the *Original* OrientDB and the time with the *Modified* variant. This wide intersection prevents the differences in performance from being considered statistically significant.

The last two plots in Figure 10 present the execution time of transactions in scenarios with the *Bidirectional Edge* and *Edge Condition* constraints. In these two scenarios, the same behavior of the previous ICs is also observed. The validation in the modified version of OrientDB is equally more efficient than the *Application* variant, since the CIs do not intersect. Comparing the other two variants, *Original* and *Modified*, the latter presents a higher positive skew, however the intersection between them is large as well.

Although the *Application* variant has, in all cases, shown worse results than the *Modified* OrientDB, there are scenarios where this difference stands out. In the *Required Edge*, *Cardinality*, *Bidirectional Edge* and *Edge Condition* scenarios, additional database access is required for the application to retrieve the information necessary to perform data validation. New tests with significantly larger graphs and more complex constraints are planned as future work.

The experiments assume that the client application is capable of ensuring adequate concurrency control in order to preserve IC consistency, which can be difficult to achieve in practical scenarios. Two instances of the application can concurrently validate an IC and then concurrently perform operations that together violate the IC. In such situations, where there are applications modifying the graph concurrently, distributed concurrency control techniques should be applied. As a result, there will be greater complexity of implementation, causing additional impacts on applications beyond those shown in Figure 10.

5 Related Work

Some of the characteristics of graph DBs, such as being schema-less and following the BASE properties, make them more flexible and allow them to provide better performance. On the other hand, the same aspects are responsible for the lack of consistency that may be fundamental for some categories of application. However, it is important to remember that a strong schema definition may deteriorate the performance of the DBMS. Thus, the challenge is to find a solution for the lack of consistency without impairing flexibility and performance. Most of the proposals found in the literature that address ICs are developed for relational DBs. Among the few that discuss the graph model, many of them only compare the available implementations, while others suggest supporting new constraints.

Pokorný [10] presents a general overview of graph DBs, covering storage, query, scalability, transaction processing, categories of graph DBs and their limitations. Among the limitations, Pokorný [10] lists features that are not entirely supported by current graph DBMS, such as data partitioning capacity, support for declarative queries, vector operations, and model restrictions that could make possible the definition of data schema. Within the topic of model constraints, ICs play a central role but are not well supported by graph DBMSs.

Barik et al. [4] employ graph DBs to analyze possible attack paths of networked applications. Most of the discussion in this paper centers on the analysis of vulnerabilities and attacks employing graphs. In their analysis, the preconditions necessary to perform an attack form a dependency graph. Therefore, an attack is seen as a progression that satisfies the dependencies. The authors argue that the use of ICs assists the process of generating an attack graph. Thus, they propose an extension of Neo4j[4] in order to create a constraint layer that allows ICs of unique values, primary and foreign keys, value range, in/out (known as edge model) and edge cardinality.

In a comparison between relational and graph databases, Pokorný [11] lists characteristics of relational DBMSs that are not currently supported by graph DBMSs. One of such features is the explicit schema definition, including the specification of ICs. This absence makes verifying accuracy of graph DBs more difficult. Pokorný [11] argues that graph DBs are based on a logical model that has three components: i) a set of types and data structures; ii) a set of inference operators; iii) a set of ICs. Actual graph DBMSs typically lack at least one of these three components, with ICs usually being the missing component. Later, in [12], the authors suggest the definition of ICs in the conceptual or the DB level. For this, they considered property data types, property value ranges, class disjunction (i.e., a node cannot belong to two classes simultaneously), mandatory edges and unique values for a property composition. Support for these ICs is added to Neo4j through the extension of its Cypher language.

Roy-Hubara et al. [13] discuss data modeling and a schema definition. The authors present an approach based on the entity-relationship (ER) model of the application domain and create a mapping from the ER model for a graph DB, along with using a DDL. They argue that the approach could be applied to any graph DB, but do not show nor refer to any implementation.

Lastly, Angles [2] attempts to find common theoretical grounds for the plethora of graph models implemented by several graph DBMSs. The author adopts property graphs as the starting point and provides a logical formalization, including the notion of a schema. This basic notion of schema is extended with ICs and the syntax and semantics of a unified query language are described. All ICs are defined and discussed from a theoretical standpoint, without discussing their support in existing graph DBMSs.

Table 1 shows which ICs are natively supported by Neo4j and which are added to it by the extensions proposed in [4] and [12]. In the case of OrientDB, the table only shows those natively supported by OrientDB and those added by this work, since it has not received such extensions before. In the table, empty circles represent ICs that are natively supported by the DBMS, while filled circles denote the new ICs proposed by the work referenced on the table header. Those that are natively supported but are also repeated with filled circles were re-implemented or improved by the work referenced in the table header.

Some of the ICs that appear in Table 1 require further discussion. In an relational DB, foreign keys are used to represent relationships between tuples from distinct tables. In a graph DB, this can be considered an anti-pattern, since instead of using a foreign key property, one should use edges to represent relationships between nodes, which correspond to tuples in the relational model. Using foreign keys with properties to maintain relationships will yield more complex queries and lead to inefficient query processing, since graph DBMSs are not designed to handle such kind of property joins. The primary key constraint, men-

[4] https://neo4j.com/

Table 1: Comparison between native ICs and proposed extensions.

Constraint Types	Neo4J	Barik et al.	Pokorny et al.	OrientDB	This work
Bidirecional edge					●
Cardinality of Edges		●			●
Class Disjunction		●	●	❍ (1)	❍ (1)
Edge Condition					●
Node Condition					●
Foreign Keys		●			
In/Out (EndPoint)		●			●
Primary Keys	❍	❍	❍	❍ (2)	❍ (2)
Property Type			●	❍	❍
Range of property		●	●	❍	❍
RegEx				❍	❍
Required Edge	❍	❍	●		●
Required Property	❍	❍	❍	❍	❍
Unique	❍	●	❍	❍	❍
Unique - Composed			●	❍	❍

(1): OrientDB enforces a single label (class) per node.
(2): The primary key constraint is equivalent to a composition of the required property and unique property.

tioned by some authors, can be replaced by the simultaneous definition of unique and required ICs. If primary keys are used to maintain referential integrity, one falls into the same anti-pattern of using foreign key constraints. Similarly, composite primary keys can be obtained by combining composite unique constraints with a required property constraint for each component property.

Class disjointness ICs disallow membership of a single node to two or more classes. In the case of OrientDB, this IC is a design decision of the DBMS itself and every node must belong to a single node class. Therefore, this IC does not apply to OrientDB in its literal sense. In contrast, one may use a node property to store the "category" or "class" of a node. If multiple classifications are desired, one may model the classes as nodes and model membership as an edge from the instance to the class. If a single extra classification is allowed, one may use a single property and limit its value range using maximum/minimum constraints or using RegEx ICs. A RegEx (short for Regular Expression) is a string that compactly describes a whole set of allowed values. RegExes can be used to describe also sets of allowed values, such as the expression `adult|minor` which accepts only two possible values: "adult" and "minor".

6 Conclusions

Due to the large volume of data produced continuously by computing systems, that resulted in the phenomena called *Big data*, new solutions for managing and storing data have been developed. The greatest motivation for such development comes from the fact that traditional technologies for data storage and management are unable to meet the performance and scalability requirements demanded by most of the novel data-intensive applications that have emerged recently. New data models have been adopted, allowing large volumes of data to be handled, resulting in a fast adoption of these technologies in the software market. In this context, graph DBs gained popularity due to their ability to easily represent data used by several applications, for which the graph data model suits perfectly. However, this category of DBMS still lacks effective mechanisms to enforce the integrity of the stored data. Therefore, this work proposed extending a graph DBMS, adding support for 6 new ICs: node condition, required edge, in/out, edge cardinality, bidirectional edge and edge condition. As future work, we plan to extend the number of supported constraints, including ICs that appear in the related work and in [7], and also to evaluate their efficiency and relevance in comparison with the constraints that are implemented by other works and in the current DBMSs.

An evaluation study compared the original OrientDB version, a modified version with added support for the new ICs and a third case in which data validation is done by the client application. Experiments demonstrated that the modified OrientDB presented a small increase in the execution time of data manipulation operations, having the original version as baseline. This increase, though, is not big enough to be considered statistically significant for five of the new ICs. Only in one case – the node condition constraint – there is a noticeable, but still small, performance loss. However, the modified version of OrientDB is significantly faster than performing validations at the application level. In addition, the resulting implementation also simplifies the development of client applications, which can skip data validation checks and leave them to be done by the DBMS.

The same strategy could also be adopted to add support for new ICs to other graph DBs. Furthermore, despite adding support for only six ICs, the proposed solution allows the easy addition of other constraints, with the aim of further extending the mechanisms that can guarantee the integrity of graph DBs.

References

1. Amin, M.S., Yan, B., Sriram, S., Bhasin, A., Posse, C.: Leveraging a social graph to deliver relevant recommendations (Aug 2019), https://patents.google.com/patent/US10380629B2/en
2. Angles, R.: The property graph database model. In: Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018. http://ceur-ws.org/Vol-2100/paper26.pdf
3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1:1–1:39 (Feb 2008), http://doi.acm.org/10.1145/1322432.1322433
4. Barik, M.S., Mazumdar, C., Gupta, A.: Network vulnerability analysis using a constrained graph data model. In: Ray, I., Gaur, M.S., Conti, M., Sanghi, D., Kamakoti, V. (eds.) Information Systems Security. pp. 263–282. Cham (2016)
5. Corbellini, A., Mateos, C., Zunino, A., Godoy, D., Schiaffino, S.: Persisting big-data: The NoSQL landscape. Information Systems **63**, 1 – 23 (2017), http://www.sciencedirect.com/science/article/pii/S0306437916303210
6. van Erven, G.C.G., Holanda, M., Carvalho, R.N.: Detecting evidence of fraud in the brazilian government using graph databases. In: Rocha, Á., Correia, A.M., Adeli, H., Reis, L.P., Costanzo, S. (eds.) Recent Advances in Information Systems and Technologies. pp. 464–473. Springer International Publishing, Cham (2017)

7. Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., Zimányi, E.: Grad: On graph database modeling (2016), https://arxiv.org/ftp/arxiv/papers/1602/1602.00503.pdf
8. Ip, H., Tang, H.: Parallel evidence combination on a SB-tree architecture. In: 1996 Australian New Zealand Conference on Intelligent Information Systems. Proceedings. ANZIIS 96. IEEE (1996), https://doi.org/10.1109/anziis.1996.573882
9. Navathe, S.B., Elmasri, R.: Fundamentals of Database Systems. Pearson, 7th edn. (2016)
10. Pokorný, J.: Graph Databases: Their Power and Limitations. In: Saeed, K., Homenda, W. (eds.) Computer Information Systems and Industrial Management. pp. 58–69. Springer Int Publishing, Cham (2015)
11. Pokorný, J.: Conceptual and Database Modelling of Graph Databases. In: Proceedings of the 20th International Database Engineering Applications Symposium. pp. 370–377. IDEAS '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2938503.2938547
12. Pokorný, J., Valenta, M., Kovačič, J.: Integrity Constraints in Graph Databases. Procedia Computer Science **109**, 975 – 981 (2017), http://www.sciencedirect.com/science/article/pii/S1877050917311390, 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal
13. Roy-Hubara, N., Rokach, L., Shapira, B., Shoval, P.: Modeling graph database schema. IT Professional **19**(6), 34–43 (Nov 2017). https://doi.org/10.1109/MITP.2017.4241458
14. Sandhu, R.S.: On five definitions of data integrity. In: Proceedings of the IFIP WG11.3 Working Conference on Database Security VII. pp. 257–267. North-Holland Publ., Amsterdam, The Netherlands (1994)
15. Shirasaki, Y., Kobayashi, Y., Aoyama, M.: A speech data-driven stakeholder analysis methodology based on the stakeholder graph models. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). vol. 2, pp. 213–220 (Jul 2019). https://doi.org/10.1109/COMPSAC.2019.10209
16. de Souza, V.C.O., dos Santos, M.V.C.: Maturing, consolidation and performance of nosql databases: Comparative study. In: Proceedings of the Annual Conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1. pp. 32:235–32:242. SBSI 2015, Brazilian Computer Society, Porto Alegre, Brazil, Brazil (2015), http://dl.acm.org/citation.cfm?id=2814058.2814097
17. Stokebraker, M.: SQL Databases v. NoSQL Databasesd. In: Communications of the ACM. vol. 53, pp. 10–11 (2010)
18. Vaz, R.V., de Oliveira, J.D.Q., Ribeiro, L.A.: Duplicate management using graph database systems: A case study. In: Proceedings of the XV Brazilian Symposium on Information Systems. pp. 50:1–50:8. SBSI'19, ACM, New York, NY, USA (2019), http://doi.acm.org/10.1145/3330204.3330260
19. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604 (May 2014). https://doi.org/10.1109/SP.2014.44