Managing Linked Nulls in Property Graphs: Tools to Ensure Consistency and Reduce Redundancy

Dominique Laurent, Jacques Chabin, Mirian Halfeld Ferrari, Nicolas Hiot

▶ **To cite this version:**

Dominique Laurent, Jacques Chabin, Mirian Halfeld Ferrari, Nicolas Hiot. Managing Linked Nulls in Property Graphs: Tools to Ensure Consistency and Reduce Redundancy. ADBIS 2023, Sep 2023, Barcelogne, Spain. pp.180–194, 10.1007/978-3-031-42914-9_13 . hal-04310353

HAL Id: hal-04310353
https://hal.science/hal-04310353v1

Submitted on 17 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Managing Linked Nulls in Property Graphs: Tools to Ensure Consistency and Reduce Redundancy

Jacques Chabin[1], Mirian Halfeld-Ferrari[1], Nicolas Hiot[1,2], and Dominique Laurent[3]

[1] LIFO – Université d'Orléans, INSA CVL, Orléans, France
[2] EnnovLabs – Ennov, France
[3] ETIS – CNRS, ENSEA – CY Université, Cergy-Pontoise, France

Abstract. Ensuring the provision of consistent and irredundant data sets remains essential to minimize bugs, promote maintainable application code and obtain dependable results in data analytics. A major challenge in achieving consistency is handling incomplete data, *i.e.,* missing information that may be provided later, comes from the fact that, the use of marked (or linked) nulls is required in many applications to express unknown but connected information. In this context, it is well known that maintaining the data consistent and irredundant is not an easy task. This paper proposes a query-driven incremental maintenance approach for consistent and irredundant incomplete databases. Can graph databases improve the efficiency of this operation? How can graph databases manipulate linked nulls? What is the impact of using graph databases on other essential maintenance operations? This paper presents an innovative approach to answering these questions, highlighting the proposal's strengths and weaknesses and offering avenues for further research.

Keywords: graph databases · incremental maintenance · incomplete data · constraints · tuple generating dependencies · updates.

1 Introduction

Consistent data stores lead to fewer bugs, easier maintenance, and reliable analytics. Developers can focus on their tasks without being bogged down by data consistency issues. Our current research projects focus on data extracted from clinical cases. We deal with databases which are incomplete but consistent with respect to some given integrity constraints expressed as tuple-generating dependencies (tgd). Consistency is our first concern. For example, let $c1 : Pat(x)$, $SOSY(x, y) \rightarrow PrescExam(x, z)$ be a constraint which implies that if a patient x exhibits a symptom y, then they should pass an exam z. If we consider the fact that Lea is a patient with pain in her hands, *i.e.,* $Pat(Lea)$ and $SOSY(Lea,$ *pain on hands*$)$, the database must also include the atom $PrescExam(Lea, N_1)$ for consistency. This atom indicates that an exam has been prescribed to Lea, but we do not yet know which one.

It is also crucial to address incompleteness, which can take various forms. Our focus in this work is on a database perspective, where incompleteness arises when values are missing. We adopt Reiter's approach [20] that provides First-Order Logic (FOL) semantics to null values of type 'value exists but is currently unknown'. If null values are linked, then the missing data is linked as well. For instance, the set $\mathfrak{D}_1 = \{PrescExam(Lea, N_1), ExamResult(Lea, N_1, N_2)\}$ seen as a database instance, indicates that an examination is prescribed for Lea, with unknown type denoted by N_1, and with unknown result represented by N_2.

Several sources can provide cleaned and well-formatted data to our database. Integrating this new data while maintaining consistency and avoiding redundancy is a challenge - this also involves matching the new data with the missing data to avoid redundancy. For example, if we add the new information $PrescExam(Lea, x\text{-}ray)$ to $\mathfrak{D}_1$ above, we cannot replace N_1 by *x-ray* in $PrescExam(Lea, N_1)$, because N_1 also appears in $ExamResult(Lea, N_1, N_2)$, which indicates that N_1 is a linked null. If later we receive the new information $ExamResult(Lea, x\text{-}ray, join\ inflammation)$, then we replace N_1 by *x-ray* since the instantiation of N_1 is the same in all atoms where it appears. The resulting database instance in this case is $\mathfrak{D}_1'' = \{ExamResult(Lea,\ x\text{-}ray,\ join\ inflammation),\ PrescExam(Lea,\ x\text{-}ray)\}$.

Work in [6] is well-suited to tackle the aforementioned challenge. However, in this paper, we take it a step further by introducing an *incremental* version of the approach and applying it to a *graph database* system. Our proposal leverages the advantages of data access, manipulation, and management capabilities provided by database systems, contrary to the in-memory version of [6]. Moreover, it aims to improve the process of creating sets of atoms that are linked by their null values, which was an expensive step in the earlier approach.

To achieve our objective, we undertook a study to investigate the potential of using a graph database. Our research aims to overcome the limitations of the original approach by designing a graph database model that facilitates the exploration of relationships between null values, along with proposing incremental update routines. One type of graph model that we have been using in our projects is the labelled property graph (LPG), which includes named relationships and properties. We have been working with LPG-based graph databases like Neo4J, and Cypher is the most widely used query language for LPGs. It is also the basis for the development of GQL, an ISO standard in progress.

In summary, our paper introduces an innovative approach that leverages the advantages of graph databases to enhance the creation of linked atoms with null values. We also examine the impact of this approach on other essential maintenance operations, providing insights into its overall effectiveness. Our paper also presents a critical analysis of the use of graphs in this context, highlighting the strengths and weaknesses of this approach and offering suggestions for future research directions. Additionally, we provide an overview of the advantages of our incremental version. The rest of this section showcases the contribution of our work and discusses the representation of null information in LPG-graphs.

Incremental Database Evolution. Database evolution is initiated by update requests. Its maintenance involves two main actions: *chasing* and *simplification*. The *chase* procedure is used to ensure consistency with a set of integrity constraints, expressed as rules, and which may generate new null values. The simplification process eliminates redundancies by removing null values that can be instantiated without breaking their links. This action corresponds to the computation of the *core* [9]. We adopt the policy in [6] to define the evolution of our database, but the proposal in this paper differs from the in-memory version in [6] in the following aspects:

1. *Incrementality is the kernel of our approach:* (a) Chasing is performed only on constraints concerned by the update, whereas in [6], all constraints are checked. (b) Simplification is guided by the null values potentially simplifiable due to the update, in contrast to [6], where simplification considers all null values.

2. *Our approach is query-driven as it deals with data stored in database systems, unlike the in-memory version in [6]:* (a) A constraint is a rule with a conjunction of atoms in its body. It is triggered, to produce the atom in its head, only when its body is fully instantiated based on the update atoms and the database instance. A *chase query*, partially instantiated with update constants, searches the database to fully instantiate the body of that constraint. (b) Queries, denoted as q_{bucket}, retrieve the null values appearing in the database instance and connected to the update being performed. (c) The process relies on the results of query q_{bucket} to identify atoms with nulls linked to those in q_{bucket}'s results. Then, a conjunction of these atoms is used to construct a new query, denoted by q_{core}, which guides the simplification decisions.

3. *When working with database systems, we assess which database model is better suited for our approach.* (a) Relational database model is used as our baseline. Each atom $R(A)$ is represented as a tuple A in table R. In this context, it is observed that the most costly task is the extraction of sets of atoms that are linked through their null values. Indeed, this operation requires scanning all tables, since null values can be associated to any attribute in a table, and thus cannot be indexed. (b) Query engines typically assume that graphs in graph databases are complete, but this assumption is not valid in practice due to missing data. Neo4J's approach of treating nulls as 'value does not exist' is however not adequate in case of linked nulls. To address this challenge, we propose a novel database design that treats nulls as first-class citizens. (c) Our graph database model enables null values to be treated as first-class citizens and indexed. This model simplifies operations in which we can identify all atoms that are directly or indirectly linked to a null value by simply selecting that null value.

Our proposal follows the database evolution semantics from [6], which has been shown to be effective[4], deterministic, and adhere to a minimal change property. Let $\mathfrak{D}$ be an incomplete but consistent database instance and U be a set of updates. The notation $\mathfrak{D}\diamondsuit U$ represents the insertion or deletion of the

[4] If an update is not rejected, the updated database contains the inserted data and does not contain the deleted ones

required updates in or from $\mathfrak{D}$. In [6], a from-scratch approach generates a new database instance denoted as $\mathfrak{D}' = core(upd(\mathfrak{D}\diamondsuit U))$, where upd is the update process described in [6]. In contrast, our paper describes an *incremental* version of the update process denoted as $upd_{|U}$. Therefore, the new database instance is represented as $\mathfrak{D}' = core_{|NullBucket}(upd_{|U}(\mathfrak{D}\diamondsuit U))$, where $NullBucket$ refers to the set of nulls affected by the update ($upd_{|U}$) applied to $\mathfrak{D}\diamondsuit U$.

Paper Organization. Section 2 presents the background necessary to understand our approach described in Sections 3-4. Here, we recall the logic formalism used in our algorithms, while more practical considerations about the design of our graph database aim to highlight the implementation of the queries on Cypher. Section 5 presents our experimental study, and Section 6 overviews related work and presents future work.

2 Preliminary Considerations: Theory and Application

Theoretical Background. Assuming familiarity with FOL, we consider atoms as $P(t_1, \ldots, t_n)$ where P is a predicate of arity n and $t_1, \ldots, t_n$ are terms (constants, nulls, or variables). A fact is an atom with only constants and an instantiated atom has no variables. $null(A)$ denotes the set of nulls in an instantiated atom A. Homomorphisms between sets of atoms A_1 and A_2 map terms in A_1 to A_2, such that: (i) $h(t) = t$ if t is a constant, (ii) if $P(t_1, \ldots, t_n)$ is in A_1, then $P(h(t_1), \ldots, h(t_n))$ is in A_2. If h_1 is a homomorphism from A_1 to A_2 with an inverse homomorphism, then A_1 is isomorphic to A_2.

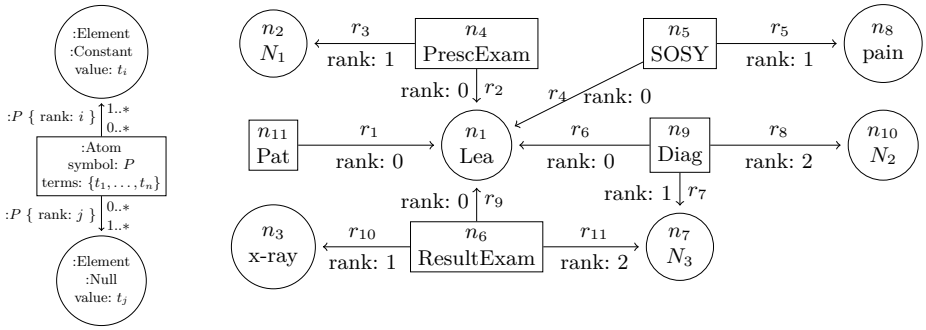
Φ denotes the set of existentially quantified formulas ϕ, which are conjunctions of atomic formulas. The set of atomic formulas in ϕ is denoted by $atoms(\phi)$. A *model* M of a formula ϕ in Φ is a set of facts that has a homomorphism from $atoms(\phi)$ to M. If each model of ϕ_1 is a model of ϕ_2, then we write $\phi_1 \Rightarrow \phi_2$, and ϕ_1 and ϕ_2 are said to be *equivalent*, denoted by $\phi_1 \Leftrightarrow \phi_2$, if $\phi_1 \Rightarrow \phi_2$ and $\phi_2 \Rightarrow \phi_1$ hold. ϕ_1 is said to be *simpler than* ϕ_2, denoted by $\phi_1 \sqsubseteq \phi_2$, if $\phi_1 \Leftrightarrow \phi_2$ and $atoms(\phi_1) \subseteq atoms(\phi_2)$. A simplification ϕ_1 of ϕ_2 is *minimal* if $\phi_1 \sqsubseteq \phi_2$ and there is no ϕ_1' such that $\phi_1' \sqsubset \phi_1$. For instance, if ϕ is $(\exists x, y)(P(a, x) \wedge P(a, y))$, then $(\exists x)(P(a, x))$ and $(\exists y)(P(a, y))$ are distinct but *equivalent* simplifications of ϕ. It is proven in [6] that if ϕ is in Φ and ϕ_1 and ϕ_2 are *minimal* simplifications of ϕ, then $atoms(\phi_1)$ and $atoms(\phi_2)$ are isomorphic. Minimal simplifications are called *cores* and the core of a given formula ϕ is denoted by $core(\phi)$.

A *database instance* is a set of instantiated atoms written as $atoms(Sk(\phi))$, where $Sk(\phi)$ is the Skolem version of a formula ϕ in Φ such that $core(\phi) = \phi$. A *constraint* (or rule) is a tuple-generating dependency (tgd) of the form $(\forall X, Y)(body(X, Y) \rightarrow (\exists Z)head(X, Z))$, more simply written $body(X, Y) \rightarrow head(X, Z)$, where X, Y, and Z are vectors of variables, $body(X, Y)$ is a conjunction of atoms and $head(X, Z)$ is an atom. Constraint satisfaction is defined as usual: given a set I of instantiated atoms, $I \models c$ if for every homomorphism h such that $h(body(c)) \subseteq I$, there is a homomorphism h' such that $h(body(c)) = h'(body(c))$ and $h'(head(c))$ belongs to I. Here, by *homomorphism*

we mean any mapping from the variables in c to constants or nulls. If $\mathbb{C}$ is a set of constraints, $I \models \mathbb{C}$ if for every c in $\mathbb{C}$, $I \models c$.

Graph Database Design in Neo4J. Our incremental approach is implemented through Cypher queries on a database, with the goal of optimizing the costly process of creating sets of atoms linked by null values. To make such retrieval efficient, given an atom $P(t_1, \ldots, t_n)$ our graph model represents P as a node, linked to other nodes representing the terms $t_1, \ldots, t_n$. All nodes have a *symbol* property and are classified into three types distinguished by their labels: *Atom* has the label **:Atom** and the value of their *symbol* property corresponds to the predicate symbol P; *Constant* represent constant values and have two labels, **:Element** and **:Constant**. The value of their *symbol* property is the constant itself, and *Null* represent nulls and have two labels, **:Element** and **:Null**. The *symbol* property of such nodes is the null name, which is prefixed with '_'

Atom are connected by an edge to *Constant* and *Null* having a *rank* property that identifies the position of the term within the atom. The model of the atom $P(t_1, \ldots, t_n)$ is illustrated in Figure 1a, where t_i represents constants and t_j represents nulls. The notation on the right of the edges indicates the relationship cardinality between an atom and its terms: each element is connected to at least one atom, while atoms may have no terms. Figure 1b illustrates part of our database instance (rectangular nodes are atoms and circular nodes are elements) where optimization labels and attributes are omitted.



(a) Model (b) Instance (example)

Fig. 1: Graph database model and an instance

Our model offers advantages for certain operations, but it increases the cost of converting between graph and logic formats for atoms. These conversions are essential for communication between the database and local computation procedures. To optimize conversions and graph traversals, we introduce the following design redundancies that significantly improve performance.

- *To avoid edge traversal:*

(a) Each *Atom* stores, as a property called *terms*, an ordered list containing the terms of the atom (rectangular node in Figure 1a); *e.g.*, to obtain atom $SOSY(Lea, pain)$ in Figure 1b starting with the node n_5, instead of traversing edges r_4 and r_5, we retrieve the attributes *terms* of node n_5.
(b) Each edge is assigned a label P named as the attribute *symbol* of its source *Atom* node (edges with label P in Figure 1a); *e.g.*, to obtain all atoms of the form

$SOSY(Lea, \_)$, starting from node n_1 in Figure 1b, we just have to traverse r_4. Edges r_1, r_2, r_9 and r_6 have not to be visited.

• *To allow efficient access to nodes:*

(a) A uniqueness constraint is added on the property *symbol* of nodes with label **:Element** (implying that, *e.g.*, there is a unique node representing *Lea*).

(b) An index is built on the property *symbol* of each node with label **:Atom**, and a uniqueness constraint is defined on the pair of properties *symbol/terms* (implying that, *e.g.*, there is a unique node representing atom $SOSY(Lea, pain)$).

3 Incremental Redundancy Reduction

In our approach, a database $\mathfrak{D}$ is expected to be equal to its core to avoid data redundancy. Formally, given a set of atoms I and a set of nulls ν occurring in I, we look for a homomorphism h such that $h(N) = N$ if $N \notin \nu$ and $h(I)$ is minimal so as $h(I) \subseteq I$. For instance, let $I_1 = \{\ PrescExam(Lea, N_1),\ ExamResult(Lea, N_1, N_2),\ PrescExam(Lea,\ x\text{-}ray),\ ExamResult(Lea, x\text{-}ray,\ N_3),\ ExamResult(Lea, scanner, N_4)\ \}$, and $\nu = \{N_1, N_2\}$. For h_1 such that $h_1(N_1) = x\text{-}ray$ and $h_1(N_2) = N_3$, we have $I'_1 = h(I_1) = \{\ PrescExam(Lea,\ x\text{-}ray),\ ExamResult(Lea, x\text{-}ray, N_3),\ ExamResult(Lea, scanner, N_4)\ \}$. Notice that N_4 is *not* involved in the simplification.

Given I and ν_0, nulls *linked* in I to nulls in ν_0 have to be identified. We do so by computing for every N in ν_0, the set $\mathsf{LinkedNull}_{I,N}$ obtained as the limit of the sequence $\left(\mathsf{LinkedNull}^k_{I,N}\right)_{k\geq 0}$ defined by: (i) $\mathsf{LinkedNull}^0_{I,N} = \{A_i \in I \mid N \in null(A_i)\}$ and (ii) $\mathsf{LinkedNull}^k_{I,N} = \{A_i \in I \mid (\exists A_j \in \mathsf{LinkedNull}^{k-1}_{I,N})(null(A_i) \cap null(A_j) \neq \emptyset)\}$. It is indeed easy to see that for every $k \geq 0$, $\mathsf{LinkedNull}^k_{I,N} \subseteq \mathsf{LinkedNull}^{k+1}_{I,N}$ and $\mathsf{LinkedNull}^k_{I,N} \subseteq I$. Thus the sequence $\left(\mathsf{LinkedNull}^k_{I,N}\right)_{k\geq 0}$ is monotonic and bounded by I. As I is finite, this sequence has a unique limit, which is precisely the sub-set of I denoted by $\mathsf{LinkedNull}_{I,N}$.

It therefore turns out that redundancy has only to be checked with respect to the atoms in $\bigcup_{N\in\nu_0} \mathsf{LinkedNull}_{I,N}$ and the set ν of nulls occurring in this set.

Algorithm 1: $Simplify(I, \nu_0)$

```
1: PSet := {LinkedNull_{I,N} | N ∈ ν_0}
2: for all P ∈ PSet do
3:     Build the query q_core and compute its answer q_core(I)
4:     if | (q_core(I)) | > 1 then
5:         h_m := ChooseMostSpec(q_core(I))
6:         I := (I \ P) ∪ h_m(P)
7: return I
```

Algorithm 1 shows how redundancies are dealt with, given input I and ν_0: $\mathsf{LinkedNull}_{I,N}$ is computed for each N in ν_0 (line 1), and so, the nulls in $PSet$ constitute the set ν with respect to which I is simplified. On line 3, for each P in

$PSet$, a query $q_{core} : ans(X) \leftarrow A_1(X_1), \ldots, A_n(X_n)$ is built by replacing each occurrence of N_i in P by a variable x_i. Thus, assuming that p nulls occur in P, when evaluating the answer $q_{core}(I)$ of q_{core}, the tuple $(N_1, \ldots, N_p)$ is obviously returned. However, it may happen that the answer contains other tuples, each of which defining a possible instantiation of the nulls in P, meaning that P is redundant. To implement these remarks, when the evaluation of q_{core} over I returns more than one tuple (line 4), one most specific tuple is chosen (line 5), and denoting by h_m the associated homomorphism, I is simplified (line 6) by replacing all atoms A in P by $h_m(A)$. Let us consider the set I_1 as above and $\nu_0 = \{N_1\}$. Then, $\mathsf{LinkedNull}_{I,N_1} = \{\ PrescExam(Lea, N_1)$, $ExamResult(Lea, N_1, N_2)\ \}$. Thus: $q_{core} : ans(x_1, x_2) \leftarrow PrescExam(Lea, x_1)$, $ExamResult(Lea, x_1, x_2)\}$ and $q_{core}(I_1) = \{(N_1, N_2), (x\text{-}ray, N_3)\}$. Hence, h_m such that $h_m(N_1) =$ *x-ray* and $h_m(N_2) = N_3$ is returned (line 5), and I_1 is simplified into I_1' as above.

To compute the most specific homomorphism h_m, we construct a table H_P with p columns and q rows, where p is the number of nulls in $P = \mathsf{LinkedNull}_{I,N}$, q is the number of answers returned by $q_{core}(I)$, and each h_i is an answer of $q_{core}(I)$ with h_1 being the identity. A cell $H_P[i, j]$ in H_P is set to $h_i(N_j)$. Given h_1 and h_2 over the same set of symbols Σ, h_1 is said to be *less specific than* h_2, denoted by $h_1 \preceq h_2$, if there exists a homomorphism h over Σ such that $h \circ h_1 = h_2$. We use H_P for identifying among the answers $h_1, \ldots h_q$, one most specific homomorphism h_m (see [5] for more details). Our approach is comparable to query optimization techniques [2,7] because we use tableau optimization. However, our approach differs from [2] in two fundamental ways: (1) our tableau is based on query answers rather than on the query body, and (2) we generate one most specific homomorphism, whereas in [2] non-most specific homomorphisms are discarded.

We compute LinkedNull with a Cypher query whose template is shown in Figure 2. The `UNWIND` clause is used to convert a list into individual rows. A `MATCH` clause is used to identify patterns through homomorphisms in the LPG graph and returns a table of variable instantiations. Here, the `MATCH` clause looks for paths starting with the null of the `nullValueNode` to any other node representing an atom which is not `nullValueNode` itself (condition imposed by the `WHERE` clause). On line 6, the `WITH` clause performs a 'group by' to structure the table with tuples where each null `nullValueNode` is associated to a list of `endNode`s (the nodes reached by paths `pathP`). On line 7 a new organisation is built: `linkedNodes` is divided into two lists, one containing nodes that represent predicate symbols (`linkedAtoms`) and one for those representing nulls (`linkedNulls`). Notice that we place the initial node `nullValueNode` in the first position of the latter list. In the resulting table, each atom is associated to a list of nulls.

4 Incremental Chase and Update Procedures

Insertions. Algorithm 2 inserts atoms from set iRequest into $\mathfrak{D}$. The side-effects are computed using a chase procedure (line 1), and the core of $\mathfrak{D} \cup ToIns$ is

```
1 UNWIND $nulls AS nullPredName
2 MATCH (nullValueNode:Element:Null {value: nullPredName}),
3     pathP = (nullValueNode)-[*1..maxPathLength]-(endNode)
4 WHERE endNode <> nullValueNode AND
5     ALL(n IN nodes(pathP) WHERE NOT (n:Constant))
6 WITH COLLECT(DISTINCT endNode) AS linkedNodes, nullValueNode
7 WITH [n IN linkedNodes WHERE (n:Atom)] AS linkedAtoms,
8    [nullValueNode] + [n IN linkedNodes WHERE (n:Null)] AS linkedNulls
9 UNWIND linkedAtoms AS a RETURN a.symbol as a, a.terms as e, linkedNulls
```

Fig. 2: Cypher template to find LinkedNull sets

computed by considering only the nulls in *NullBucket* (retrieved through q_{bucket} line 2) and their linked nulls. $q_{bucket}(I)_{[S]}$ searches for null values that appear in atoms that are less specific than one in S. Instead of imposing restriction on the constraint format, we use a maximal degree δ_{max} to control null value depth and avoid infinite chasing. At each insertion, null degrees are set to 0. When a constraint is applied, generated nulls are assigned a degree of $\delta + 1$, where δ is the maximal degree of nulls in the constraint body, or 0 if no null occurs. If $\delta(N) \geq \delta_{\max}$, insertion is stopped, and $\mathfrak{D}$ is not changed. In Algorithm 2, if all nulls in the simplified instance have degree less than $\delta_{\max}$ (check through q_{degree} line 4), null degrees are set to 0 (through q_δ line 5), and $\mathfrak{D}'$ is returned since it is always consistent (as proven in [6]); otherwise, the database is not modified.

Algorithm 2: Insert($\mathfrak{D}, \mathbb{C}, \delta_{max}$, iRequest)

1: $ToIns := Chase4Insert(\mathfrak{D}, \mathbb{C}, \delta_{max}, \mathsf{iRequest})$
2: $NullBucket := \{N_j \mid N_j$ is a null obtained by $q_{bucket}(\mathfrak{D} \cup ToIns)\}$
3: $\mathfrak{D}' := Simplify(\mathfrak{D} \cup ToIns, NullBucket)$
4: **if** $q_{degree}(\mathfrak{D}')_{[NullBucket, \delta_{max}]}$ **then**
5: $\quad q_\delta(\mathfrak{D}')_{[NullBucket,0]}$ {for each N in $NullBucket$, if in $\mathfrak{D}'$, sets $\delta(N)$ to 0}
6: $\quad$ **return** $\mathfrak{D}'$
7: **else**
8: $\quad$ **return** $\mathfrak{D}$

Deletions. Our incremental algorithm for the deletion from $\mathfrak{D}$ of atoms in dRequest is displayed in Algorithm 3. On line 1, all atoms in $\mathfrak{D}$ isomorphic to one in the set dRequest are retrieved through the query q_{Iso}. For instance, if dRequest $= \{P(a, N_1)\}$ and $\mathfrak{D}_1 = \{P(a, N_5)\}$, then query q_{Iso} returns $\{P(a, N_5)\}$. On line 2, the $Chase4Delete$ function incrementally computes the side-effects by generating two sets of atoms, $ToDel$ and $ToIns$, which represent atoms that should be deleted and inserted as side-effects, respectively. Once these side-effects have been incorporated in $\mathfrak{D}$ to produce $\mathfrak{D}'$ (line 3), this new instance is simplified as in the case of insertion: impacted nulls are generated on line 4 and the simplified instance is computed on line 5. We notice that, contrary to insertions, deletions are *never* rejected.

Algorithm 3: $Delete(\mathfrak{D}, \mathbb{C}, \delta_{max}, \mathsf{dRequest})$

1: $\mathsf{isoDel} := q_{Iso}(\mathfrak{D})_{[\mathsf{dRequest}]}$
2: $ToDel, ToIns := Chase4Delete(\mathfrak{D}, \mathbb{C}, \delta_{max}, \mathsf{isoDel})$
3: $\mathfrak{D}' := (\mathfrak{D} \cup ToIns) \setminus ToDel$
4: $NullBucket := \{N_j \mid N_j \text{ is a null obtained by } q_{bucket}(\mathfrak{D}')_{[ToIns \cup ToDel]}\}$
5: $\mathfrak{D}' := Simplify(\mathfrak{D}', NullBucket)$
6: **return** $\mathfrak{D}'$

Chasing. Different chasing versions have been suggested in literature [19]. Our method uses the parameter $\delta_{\max}$ to deal with constraints without any limitations. Our approach is an incremental version of the chasing method presented in [6], which is closely related to standard and core chase procedures. Two similar routines implement the chasing reasoning by using a query denoted by q_{ch}.

$Chase4Insert$ is called on line 1 of Algorithm 2. It avoids the generation of unnecessary side effect atoms by activating a constraint c only if: (i) $body(c)$ contains at least one atom being inserted, (ii) atoms in $body(c)$ that don't meet (i) map to atoms in the database instance $\mathfrak{D}$, and (iii) the instantiation of $head(c)$ is inserted only if no equivalent atom already exists in $\mathfrak{D}$. In logical terms, the query is $q_{ch} : Q(\alpha) \leftarrow L_1(\alpha_1), \ldots, L_m(\alpha_m)$, *not* $L_0(\alpha_0)$, where α is the list of variables corresponding to variables in $body(c)$. If $h_t(body(c)) \subseteq \mathfrak{D}$, then q_{ch} has a non-empty answer only if $h'_t(L_0(\alpha_0)) \notin \mathfrak{D}$ for any extension h'_t of h_t. Furthermore, in $Chase4Insert$, during a chase step, the degree of newly created nulls is computed and the condition $\delta(h'(head(c))) \leq \delta_{max}$ is verified *i.e.,* we do not trigger rules having nulls in $body(c)$ that do not meet this condition.

$c_1 : Pat(x), SOSY(x, y)^- \rightarrow PrescExam(x, z)$
$c_2 : PrescExam(x, z)^-, PlaceOfExam(z, w) \rightarrow ExamResult(x, z, y)$
$c_3 : ExamResult(x, y, z)^- \rightarrow Diag(x, y, w)$

Fig. 3: Set of (general) constraints

$Chase4Delete$ is called at line 2 of Algorithm 3. To determine the constraints affected by the deletion of atom A, a *backward chase* is performed. This chase identifies an instantiation h such that $h(head(c)) = A$. Then the homomorphism restriction $h|body(c)$ is applied to c to generate an extended instantiation h'. We check if $h'(head(c))$ is isomorphic to the atom $h(head(c))$ being deleted, using the following reasoning: (1) If an isomorphic atom is generated, we insert the atom marked as '$-$' from $h(body(c))$ into $ToDel$, as at least one atom in $h(body(c))$ must be deleted to prevent c from being triggered. Notice that, to avoid non-determinism, one atom in the body of the constraints is marked as '$-$' during constraint design (to indicate deletion priority). (2) If no isomorphic atom is generated, we add $h'(head(c))$ to $ToIns$ along with its side effects. However, we also check whether δ_{max} is respected when computing the side effects of $h'(head(c))$. If not, we insert the marked atom from $h(body(c))$ into $ToDel$.

Example 1. In the context of clinical cases, let $\mathbb{C}$ be the abridged version of constraints depicted in Figure 3. Constraint c_2 links prescribed exams, that have taken place in a medical center, to results, while constraint c_3 connects exam results to diagnoses.

Let $\mathfrak{D}_1 = \{Pat(Lea),\ SOSY(Lea, \textit{pain on hands}),\ PrescExam(Lea, N_1)\}$ be a database instance consistent with respect to $\mathbb{C}$ and $\delta_{max} = 3$. Given $\mathsf{iRequest} = \{PrescExam(N_2, testCovid),\ PlaceOfExam(testCovid, LabA),\ PrescExam(Lea, \textit{x-ray})\}$, only constraints c_2 and c_3 are triggered. Algorithm 2 (line 1) returns the set $ToIns = \{PrescExam(Lea, \textit{x-ray}),\ PrescExam(N_2^0, testCovid),\ Diag(N_2^0, covidTest, N_4^2),\ PlaceOfExam(testCovid, LabA),\ ExamResult(N_2^0, covidTest, N_3^1)\}$ where exponents represent null degree. Let $I = \mathfrak{D}_1 \cup ToIns$.

For core computation, the query q_{Bucket} find null values in I that concerns the predicates in $ToIns$. Thus, $NullBucket = \{N_1, N_2,\ N_3, N_4\}$ and by Algorithm 1, we obtain that $\mathsf{LinkedNulls}_{I,N_1} = \{N_1\}$ and $\mathsf{LinkedNulls}_{I,N_2} = \{N_2, N_3, N_4\}$. The simplification of I (line 3 of Algorithm 2) results in: $\mathfrak{D}_2 = \{Pat(Lea),\ PrescExam(Lea, \textit{x-ray}),\ SOSY(Lea, \textit{pain on hands}),\ PlaceOfExam(covidTest, LabA),\ Diag(N_2, covidTest, N_4),\ PrescExam(N_2, covidTest),\ ExamResult(N_2, covidTest, N_3)\}$.

Consider now Algorithm 3 applied to $\mathfrak{D}_3 = \{\ SOSY(Lea, \textit{pain on hands}),\ Pat(Lea),\ PrescExam(Lea, \textit{x-ray})\}$. The deletion of $PrescExam(Lea,\ \textit{x-ray})$ implies $ToDel = \{PrescExam(Lea, \textit{x-ray})\}$ and $ToIns = \{PrescExam(Lea,\ N_1)\}$, since re-applying c_1 on $\mathfrak{D}_3 \setminus \{PrescExam(Lea, \textit{x-ray})\}$ generates an atom with a null value. The new updated instance is $\mathfrak{D}_4 = \{PrescExam(Lea, N_1),\ Pat(Lea),\ SOSY(Lea, \textit{pain on hands})\}$. If we now require the deletion of $PrescExam(Lea, N_1)$ from $\mathfrak{D}_4$, re-applying c_1 on $\mathfrak{D}_4 \setminus \{PrescExam(Lea, N_1)\}$ generates an atom isomorphic to the one being deleted. Regarding side-effect deletions, $ToDel = \{PrescExam(Lea, N_1),\ SOSY(Lea, \textit{pain on hands})\}$ and $ToIns = \emptyset$. The updated instance is $\mathfrak{D}_5 = \{Pat(Lea)\}$. □

5 Experimental Study

We tested using three data sets: *Movie*[5] and *GOT*[6] are Neo4J instances with 7 and 19 predicate symbols, respectively, and *Social*[7] is a data set from the Linked Data Benchmark Council with 23 predicate symbols. We refer to [5] and our gitlab repository [17] for a detailed explanation of how we generated additional data from the original sources, ensured consistency in our database, and controlled the generation of nulls. We have 9 instances as illustrated in Table 1, where 8 of them contains nulls and 1 instance without nulls.

Database	Nb of facts	Nb of nulls	Nb of rules	Null/Facts (τ)
Movie	604	340	12	0.56
GOT	24818	17232	32	0.69
$Social_{1K}$	2248	190	39	0.08
$Social_{10K}$	16559	1183	39	0.07
$Social_{10K}^{0N}$	16559	0	39	0.00
$Social_{10K}^{50N}$	16559	50	39	0.00
$Social_{10K}^{100N}$	16559	100	39	0.01
$Social_{10K}^{500N}$	16559	500	39	0.03
$Social_{10K}^{1000N}$	16559	1000	39	0.06

Table 1: Datasets used in experiments

Runs are built from instances in Table 1 by varying the update type (insertion or deletion), adjusting the size of the update (1, 5, 10 and 20 atoms), and artificially increasing the number of facts by duplicating data n-times (scales 1, 2 and 5). During each run, 10 iterations were performed, with 3 warm-up iterations to preload the system and database cache. The database is restored between each iteration and the Java garbage collector is triggered

[5] https://github.com/neo4j-graph-examples/movies
[6] https://github.com/neo4j-graph-examples/graph-data-science
[7] https://ldbcouncil.org/benchmarks/graphalytics/

to ensure consistent timing. The benchmarks were implemented in Java 16 with MySQL 8 and Neo4J 4.4, running on a Rocky Linux 8.7 with 4 vCPU and 16 GB of memory, achieving an average of 1 GB s^{-1} read/write on disk, using docker 20.10.21.

Impact of our incremental maintenance approach. The results demonstrate the effectiveness of using a DBMS that implements incremental update processing for efficiently updating large databases. Due to the high memory requirements of the from-scratch method, the comparison is limited to the *Movie* database only. When tested on *Movie*-scale1, these updating approaches are comparable. We obtain, on average: 9017 ms for the in-memory, from-scratch version and, for the query-based, incremental ones, 151 ms for MySQL and 2380 ms for Neo4J. When considering an instance five times larger, we get an average of 888 966 ms for the in-memory version, 595 ms for MySQL and 2706 ms for Neo4J.

Impact of using a graph database. The primary goal of our experiments is to leverage the benefits of graph databases in enhancing the linked null search operation. An implementation of our incremental approach over a relational database (with nulls represented by special (Skolem) constants) is settled as our baseline. The results demonstrate that our graph data model significantly enhances retrieving LinkedNull sets, but its impact on other essential maintenance operations was more negative than expected. The retrieval of LinkedNull sets was found to be the most expensive operation of our updating policy in the relational model, as shown in Figure 4 where outsiders with more than 30 s differences are removed. Our graph model, discussed in Section 2, focuses on optimizing this aspect. The results are amazing with the retrieval of LinkedNull sets being 25 times less expensive in the graph model than in the relational model (a reduction of 96%). We expected our proposed model to have a bad performance for chasing, as pattern matching is known to be time-consuming, and our model generates queries with more complex patterns. However, the actual results were even worse than anticipated, with the chasing operation being 170 times more expensive in the graph model than in the relational model. As a result, the overall performance of the relational model outperforms the graph model.

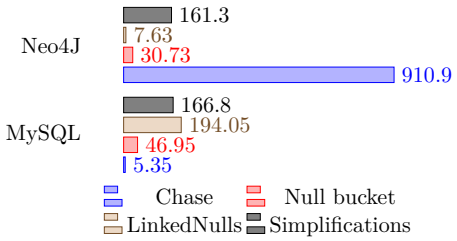
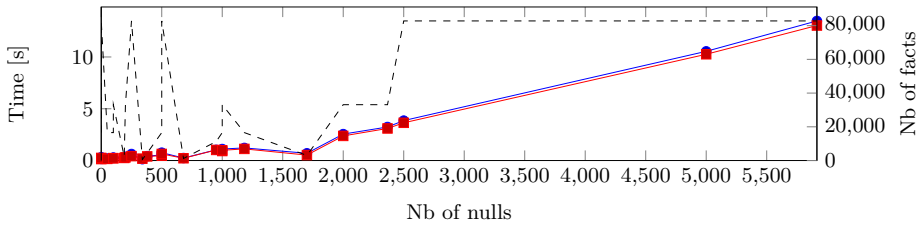


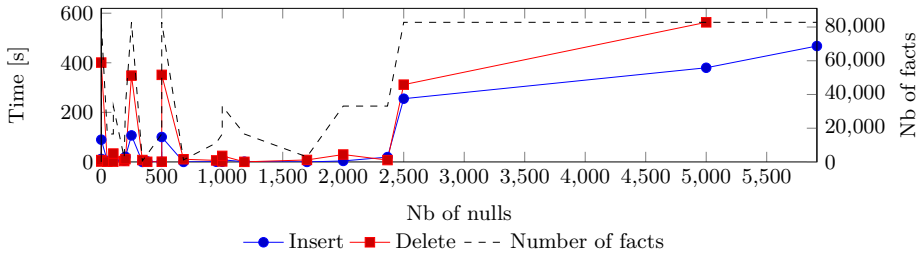
Fig. 4: Average time (ms) per DBMS

Detailed performance analysis. We analyze incremental updating performance with respect to database size, the number of nulls and the number of queries generated to interact with the DBMS. In Figure 5, each plot's right axis represents the total number of facts in the instance. The curves indicate the average resulting values for all runs corresponding to the displayed abscissa. To enhance readability, the plots disregard outcomes for *GOT* occurrences with over 17 000 nulls. Figures 5a and 5b show that for MySQL databases,

the updating runtime increases with the number of nulls, but it is slightly impacted by the database instance size. For Neo4J, the situation is the completely inverse of what was observed in MySQL. The type of the update (insert or delete) does not affect the performance of our approach, regardless of the database system. The number of generated queries increases with the increase of the number of nulls. The increase in nulls implies an augmentation in the number and size of LinkedNull sets. As a consequence, the MySQL version generates a large amount of queries that impacts its performance while the impact on the Neo4J version is negligible. Our approach is better suited for data sets with predicates of high arity, rather than those composed of binary atoms, such as the *Social* dataset.



(a) Time per null for MySQL



(b) Time per null for Neo4J

Fig. 5: Benchmarks results of 540 scenarios, average over 10 runs

Reproducibility. Results obtained by our experiments are reproducible through the use of the benchmarks and implementation available in [17].

6 Related and Future Work: conclusions

Given our experimental results, how can we address the questions posed in our abstract? Yes, utilizing a graph database and our data model can significantly improve the efficiency of retrieving linked null partitions. However, this improvement comes at a high cost, as the chasing operation is considerably less efficient on a graph compared to the relational model. A potential solution is to add design redundancy by overlaying different graph designs, such as connecting *Atoms* (e.g., connecting *patient* and *exam* nodes with an edge *prescribeTo*). Yet, the graph's complexity and update difficulty should be noted.

Our study confirms the pros and cons of using graph databases. They perform poorly when it comes to handling intricate pattern matching. The design of a graph database heavily relies on queries, unlike a relational database that presents a uniform data structure to work with. Consequently, queries that are not optimized for the database model may result in poor performance. However, they excel at path traversal queries, as nodes store information concerning their neighbourhood. As a result, they are an attractive option for exploring data relationships and for considering data analytics techniques such as predicting node connections. But how accurate are the results of data analytics on possibly inconsistent data? Constraints in graph databases are not widely used, but Neo4J has proposed some options. Starting with key for graphs [12], the graph entity dependencies (GEDs) have been proposed and their static analysis properties have been studied [13]. We refer to [4,3] for excellent overviews on the subject.

Updates are often less prioritized than queries in scientific research, despite evidence that maintaining database coherence in a dynamic environment can be complicated (see *e.g.*, [15,21]). Handling incomplete data in databases is also a difficult problem. Today, it deserves attention, especially in light of the increasing interest in *certain* answers [8]. While there is a solid foundation for addressing incompleteness in relational databases [10,14,18,20,24,1,11,23], incompleteness beyond the relational data model has received less attention [22]. As a result, updating with respect to constraints is rarely considered in this context.

Our approach is a step in this direction and raises the question of how representing linked nulls on graphs. It employs Reiter's semantics for unknown data to address the consistency maintenance problem from a logical standpoint. Our exclusively positive constraints allow for proven correction and completion of our updating policy (in [6]). It is possible to encounter null values in the query answers, which implies that, for the moment, nothing in our database allows us to provide their instantiation. They may also indicate a connection with other data. This aligns with the needs of our projects, but we must also consider modern applications that require data analytics. Which model should we adopt to meet these needs? A hybrid database model [16] could potentially be a solution, but it needs to be flexible enough to handle multiple representations of the data in each data model. Our incremental approach is a valuable tool in this context because it is designed to be independent of the data model.

Acknowledgements Work partially supported by projet SENDUP (ANR-18-CE23-0010) and developed in the context of the DOING action (MADICS and DIAMS). We express our gratitude to the interns who contributed to this project, in particular Lucas Moret-Bailly for his valuable suggestions.

References

1. Abiteboul, S., Grahne, G.: Mise-à-jour des bases de données contenant de l'information incomplète. In: Journées Bases de Données Avancés, 6-8 Mars 1985, St. Pierre de Chartreuse (Informal Proceedings). (1985)

2. Aho, A.V., Sagiv, Y., Ullman, J.D.: Efficient optimization of a class of relational expressions. ACM Trans. Database Syst. **4**(4), 435–454 (1979)
3. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Hare, K.W., Hidders, J., Lee, V.E., Li, B., Libkin, L., Martens, W., Murlak, F., Perryman, J., Savkovic, O., Schmidt, M., Sequeda, J.F., Staworko, S., Tomaszuk, D.: Pg-keys: Keys for property graphs. In: SIGMOD Conference. pp. 2423–2436. ACM (2021)
4. Bonifati, A., H. L. Fletcher, G., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2018)
5. Chabin, J., Halfeld Ferrari, M., Hiot, N., Laurent, D.: Incremental consistent updating of incomplete databases (extended version - technical report). Tech. rep., LIFO- Université d'Orléans, (2023), `https://hal.science/hal-03982841`
6. Chabin, J., Halfeld Ferrari, M., Laurent, D.: Consistent updating of databases with marked nulls. Knowl. Inf. Syst. **62**(4), 1571–1609 (2020)
7. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Symposium on the Theory of Computing (1977)
8. Console, M., Guagliardo, P., Libkin, L., Toussaint, E.: Coping with incomplete data: Recent advances. In: PODS. pp. 33–47. ACM (2020)
9. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. **30**(1), 174–210 (2005)
10. Fagin, R., Kuper, G.M., Ullman, J.D., Vardi, M.Y.: Updating logical databases. Advances in Computing Research **3**, 1–18 (1986)
11. Fagin, R., Ullman, J.D., Vardi, M.Y.: On the semantics of updates in databases. In: Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, USA. pp. 352–365 (1983)
12. Fan, W., Fan, Z., Tian, C., Dong, X.L.: Keys for graphs. Proc. VLDB Endow. **8**(12), 1590–1601 (2015)
13. Fan, W., Lu, P.: Dependencies for graphs. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS, Chicago, USA. pp. 403–416 (2017)
14. Grahne, G.: The Problem of Incomplete Information in Relational Databases, Lecture Notes in Computer Science, vol. 554. Springer (1991)
15. Halfeld Ferrari Alves, M., Laurent, D., Spyratos, N.: Update rules in datalog programs. J. Log. Comput. **8**(6), 745–775 (1998)
16. Hassan, M.S., Kuznetsova, T., Jeong, H.C., Aref, W.G., Sadoghi, M.: Grfusion: Graphs as first-class citizens in main-memory relational database systems. In: SIGMOD Conference. pp. 1789–1792. ACM (2018)
17. Hiot, N., Moret-Bailly, L., Chabin, J.: `https://gitlab.com/jacques-chabin/UpdateChase` (2023)
18. Imielinski, T., Lipski Jr., W.: Incomplete information in relational databases. J. ACM **31**(4), 761–791 (1984)
19. Onet, A.: The chase procedure and its applications in data exchange. In: Data Exchange, Integration, and Streams, pp. 1–37 (2013)
20. Reiter, R.: A sound and sometimes complete query evaluation algorithm for relational databases with null values. J. ACM **33**(2), 349–370 (1986)
21. Schewe, K., Thalheim, B.: Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. Acta Cybern. **13**(3), 277–304 (1998)
22. Sirangelo, C.: Representing and Querying Incomplete Information: a Data Interoperability Perspective (2014), `https://tel.archives-ouvertes.fr/tel-01092547`
23. Winslett, M.: Updating Logical Databases. Cambridge University Press, New York, NY, USA (1990)

24. Zaniolo, C.: Database relations with null values. J. Comput. Syst. Sci. **28**(1), 142–166 (1984)