Incremental Consistent Updating of Incomplete Databases

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

Incremental Consistent Updating of Incomplete Databases (Extended Version - Technical Report)

Jacques Chabin, Mirian Halfeld Ferrari
LIFO – Université d'Orléans, INSA CVL – Orléans, France

Nicolas Hiot
LIFO – Université d'Orléans, INSA CVL – Orléans, France
EnnovLabs – Ennov – Paris, France

Dominique Laurent
ETIS – CNRS, ENSEA, CY Université – Cergy-Pontoise, France

February 10, 2023

Abstract

Efficient consistency maintenance of incomplete and dynamic real-life databases is a quality label for further data analysis. In prior work, we tackled the generic problem of database updating in the presence of tuple generating constraints from a theoretical viewpoint. The current paper considers the usability of our approach by (a) introducing incremental update routines (instead of the previous from-scratch versions) and (b) removing the restriction that limits the contents of the database to fit in the main memory. In doing so, this paper offers new algorithms, proposes queries and data models inviting discussions on the representation of incompleteness on databases. We also propose implementations under a graph database model and the traditional relational database model. Our experiments show that computation times are similar globally but point to discrepancies in some steps.

1 Introduction

Incremental update algorithms are essential for incomplete real-life databases, often large and constantly updated. Modern applications usually involve the analysis of large amounts of data with missing and changing values. The quality of this analysis depends on the consistency of the data, the maintenance of which requires calculations whose cost needs to be reduced.

To address this problem, we build upon our prior work [9] where the generic problem of database updating in the presence of constraints was tackled from a theoretical point of view, and under the restriction that the database content was meant to fit in main memory. Hence, for missing values, we follow Reiter [28] who provides FOL (First-Order Logic) semantics to null values of type 'value exists but is currently unknown'. Constraints are expressed as tuple-generating dependencies (tgd), *i.e.,* implications of the form $(\forall X, Y)(B(X, Y) \Rightarrow (\exists Z)L(X, Z))$ where X, Y, Z are vectors of variables, $B(X, Y)$ is the conjunction of atoms of the form $L_i(X_i, Y_i)$ where X_i and Y_i are sub-vectors of X and Y, respectively, and $L(X, Z)$ is an atom.

In [9], our purpose was to allow for the insertions or the deletions of sets of tuples under the following hypotheses:

- A fixed set $\mathbb{C}$ of constraints as specified just above is assumed over a given set of predicates.
- The database $\mathfrak{D}$ to be updated is a set of instantiated atoms, in which marked nulls may occur. Moreover, the database contains no redundancies caused by these nulls.
- $\mathfrak{D}$ satisfies the constraints in $\mathbb{C}$, meaning that, for every c in $\mathbb{C}$, whenever there exists an instantiation h of X and Y such that $\mathfrak{D}$ contains all atoms in $h(B(X,Y))$ then h can be extended to Z so as $\mathfrak{D}$ also contains $h(L(X,Z))$.
- The updated database $\mathfrak{D}'$ satisfies the constraints in $\mathbb{C}$ and is not redundant, while being such that the updates are performed, that is, all atoms to be inserted are present in $\mathfrak{D}'$ and no atoms to be deleted is present in $\mathfrak{D}'$.

In this paper, we improve our work in [9] in two main aspects: (1) we propose an *incremental* version of the approach and (2) we deal with data stored in *database systems*, contrary to the in-memory version of [9].

Given an update u over a database instance $\mathfrak{D}$, our approach consists in generating (by the activation of constraints in $\mathbb{C}$) a set of new updates, U', as necessary side-effects to maintain the database consistent. Contrary to *from scratch* algorithms, whereby *the whole* database instance and *the whole* set $\mathbb{C}$ are involved in the generation of U', *incremental* algorithms minimize the amount of data to be accessed and the constraints to be triggered.

Paper Organisation. We overview our approach and its evolution (*from scratch* towards an *incremental* approach) through a motivating example in Section 2. Section 3 provides some background. Section 4 introduces the operations over which the incremental core computation is built. Incremental update algorithms, their implementation aspects and experimental results are introduced, respectively in Sections 5, 6 and 7. After presenting related work in Section 8, Section 9 concludes the paper.

2 Motivating Example

Figure 1 shows a set of constraints in the context of a university, researchers and students. Although the intuitive meaning of these constraints should be clear, we point out that in the constraints c_6, c_9, c_{10}, c_{11} and c_{12}, the right hand-side involves a variable not present in the left hand-side. Due to such contraints, known in the literature as tuple generating dependencies (tgd), nulls values may appear in the database instance, as explained below. Their intuitive meaning is as follows:

- the variable Z in $PhDPaper(Y,P,Z)$ of c_6 stands for the year the paper has been published;
- the variable Y in $Cites(X,Y)$ of c_9 stands for a publication cited by publication X;
- the variable Y in $Enrolled(X,Y)$ of c_{10} stands for a course student X is enrolled in;
- the variable Z in $Degree(Y,Z)$ of c_{11} stands for the degree course Y is part of;
- the variable Z in $Language(X,Y,Z)$ of c_{12} stands for the language in which is taught course X of degree Y.

c_1 : $Supervises(X,Y) \rightarrow Researcher(X)$

c_2 : $Supervises(X,Y) \rightarrow Student(Y)$

c_3 : $Authors(X,Y) \rightarrow Researcher(X)$

c_4 : $Authors(X,Y) \rightarrow Publication(Y)$

c_5 : $Researcher(X) \rightarrow Authors(X,Y)$

c_6 : $Supervises(X,Y), Authors(X,P), Authors(Y,P) \rightarrow PhDPaper(Y,P,Z)$

c_7 : $Cites(X,Y) \rightarrow Publication(X)$

c_8 : $Cites(X,Y) \rightarrow Publication(Y)$

c_9 : $Publication(X) \rightarrow Cites(X,Y)$

c_{10} : $Student(X) \rightarrow Enrolled(X,Y)$

c_{11} : $Enrolled(X,Y) \rightarrow Degree(Y,Z)$

c_{12} : $Degree(X,Y) \rightarrow Language(X,Y,Z)$

c_{13} : $Enrolled(X,Y) \rightarrow GrantEligible(X)$

Figure 1: Set of (general) constraints

Constraints from this set are used in subsequent examples to illustrate our proposes throughout the paper. Let us start with $\mathbb{C} = \{c_1, \ldots, c_6\}$ over the following database instance $\mathfrak{D}$:

$\mathfrak{D} = \{\ Researcher(Elin), Authors(Elin, P_{269}), Publication(P_{235}), Authors(Sten, P_{269}), Publication(P_{269}), Student(Sten), Supervises(Elin, Sten), Researcher(Nils), PhDPaper(Sten, P_{269}, 2022)\ \}$

(A) Constraint satisfaction. First, constraints c_1 and c_2 are satisfied, because $Supervises(Elin, Sten)$, $Researcher(Elin)$ and $Sudent(Sten)$ are in $\mathfrak{D}$. Constraint c_4 is satisfied as well because $Publication(P_{269})$ is in $\mathfrak{D}$. Similarly, c_6 is satisfied because $\mathfrak{D}$ contains $PhDPaper(Sten, P_{269}, 2022)$.

However, c_3 is *not* satisfied because $Authors(Sten, P_{269})$ is in $\mathfrak{D}$ and $Researcher(Sten)$ is not. Similarly, c_5 is not satisfied because $Researcher(Nils)$ has no matching fact over $Authors$ in $\mathfrak{D}$. Constraint satisfaction is obtained by adding facts, generated by a process called *chase*:

1. $Researcher(Sten)$ is added to satisfy c_3.
2. In order to satisfy c_5 one fact over $Authors$ must be added, but the value of the second argument (*i.e.,* the publication) is unknown. Despite that publications are present in $\mathfrak{D}$, those authored by $Nils$ are unknown. *Marked nulls* are used to account for this situation: $Authors(Nils, N_1)$ is added, which is read as '$Nils$ authored a publication, currently unknown but recognized as N_1'. The atom $Publication(N_1)$ is then inserted in order to satisfy c_4.

In this case, we obtain $\mathfrak{D}' = \mathfrak{D} \cup \{Researcher(Sten),\ Authors(Nils, N_1),\ Publication(N_1))\}$.

In the following items, we show differences between the *from-scratch* and incremental approaches to updates.

(B) Updates. Updates are insertions or deletions.

(B.1) **Insertions.** Given a database instance $\mathfrak{D}$ satisfying a set of constraints $\mathbb{C}$, an updated database is the result of inserting facts in $\mathfrak{D}$ while maintaining constraint satisfaction.

Let $\mathbb{C}_1 = \{c_1, c_6, c_7, c_8\}$, $\mathfrak{D}_1 = \{Researcher(Elin), Supervises(Elin, Sten), Authors(Elin, P_{269})\}$, and the set of required insertions $\mathsf{iRequest} = \{Authors(Sten, P_{269})\}$.

From-scratch approach. To reinforce constraints, side-effects are computed through a process called *chase*, that applies the constraints in $\mathbb{C}_1$ on $\mathfrak{D}_1 \cup \mathsf{iRequest}$, to generate a new set $chase(\mathfrak{D}_1)$. In doing so, facts already in $\mathfrak{D}_1$ might be generated again.

Incremental approach. Side-effects for insertions are computed based on iRequest as follows. In our example, the only constraint to be triggered when inserting $Authors(Sten, P_{269})$ is c_6 because (i)

a query on $\mathfrak{D}_1$ informs that the atoms $Supervises(Elin, Sten)$ and $Authors(Elin, P_{269})$ already exist in $\mathfrak{D}_1$, and (ii) c_6 is impacted by the insertion. Then, the result of the insertion is $\mathfrak{D}'_1 = \mathfrak{D}_1 \cup \{Authors(Sten, P_{269}),\ PhDPaper(Sten, P_{269}, N_2)\}$.

(B.2) **Deletions.** Consider $\mathfrak{D}'$ (from item (A) above) along with the constraints in $\mathbb{C} = \{c_1, \ldots, c_6\}$.

From-scratch approach. After removing from $\mathfrak{D}'$ a given set of facts dRequest, the deletion process includes the computation of $chase(\mathfrak{D}' \setminus \mathsf{dRequest})$ to check constraint satisfaction. Then, if one atom to be deleted is re-generated (up to null renaming), a backward chase is activated to identify the side-effects of the deletion. This is illustrated through the following two cases.

Case 1. Let $\mathsf{dRequest} = \{PhDPaper(Sten, P_{269}, 2022)\}$. First the fact $PhDPaper(Sten, P_{269}, 2022)$ is removed from $\mathfrak{D}'$ and then, constraint satisfaction is checked as done for insertions. Here, $\mathfrak{D}' \setminus \{PhDPaper(Sten, P_{269}, 2022)\}$ does not satisfy c_6 because Sten is still present as an author of paper P_{269}. As above, consistency is restored by inserting $PhDPaper(Sten, P_{269}, N_2)$. The resulting database $\mathfrak{D}'' = (\mathfrak{D}' \setminus \{PhDPaper(Sten, P_{269}, 2022)\}) \cup \{PhDPaper(Sten, P_{269}, N_2)\}$ is consistent and implements the deletion because it does not contain the atom to be deleted.

Case 2. Consider now $\mathsf{dRequest} = \{PhDPaper(Sten, P_{269}, N_2)\}$ on $\mathfrak{D}''$. A processing similar to the previous one would first remove the atom from $\mathfrak{D}''$ and then, insert $PhDPaper(Sten, P_{269}, N_3)$ to restore consistency with respect to the constraints. This result is not acceptable because the generated set is equal to $\mathfrak{D}''$ (up to a null renaming), meaning that the deletion has *not* been implemented. In this case, the processing is carried on by deleting all atoms responsible of the generation of the atom to be deleted.

This amounts to apply the constraints *backward* (from the head to the body), removing **one** atom from the body, to prevent the constraint from being triggered. To this end, for every constraint c, a literal in its body is marked as the one to be deleted in such a situation (for the sake of simplicity, let the leftmost literal in be the marked one). Here, $Authors(Elin, P_{269})$ has to be deleted, due to c_6. Then, we proceed following the ideas already presented: the latter deletion, because of c_5, requires the insertion of $Authors(Elin, N_4)$ which in turn, because of c_4, requires the insertion $Publication(N_4)$, returning the database

$$\mathfrak{D}_2 = (\mathfrak{D}'' \setminus \{PhDPaper(Sten, P_{269}, N_2), Authors(Elin, P_{269})\}) \cup \{Authors(Elin, N_4), Publication(N_4)\}.$$

Incremental approach. First, a backward chase is used to find the constraints impacted by the required update. In our example, constraint c_6, and then c_5 are the only ones concerned by deletions. Second, the chase is applied *only* on the rules detected just above and its result is analyzed as done in the *from-scratch* approach. It is worth noting that, as data is stored in a database, queries are used to detect whether constraints can be applied for chasing (backward or forward).

(C) Avoiding too many nulls. An important issue regarding side effects is making sure that the processing terminates. Considering the set of constraints $\mathbb{C}_1 = \mathbb{C} \cup \{c_7, c_8, c_9\}$ on $\mathfrak{D}'$, it is clear that $\mathfrak{D}''$ does *not* satisfy c_9, because $Publication(P_{269})$ and $Publication(P_{235})$ belong to $\mathfrak{D}''$ with no associated citation. In order to satisfy c_9, $Cites(P_{269}, N_6)$ and $Cites(P_{235}, N_7)$ must be inserted, which triggers the insertions of $Publication(N_6)$ and of $Publication(N_7)$, due to c_7. Then, to satisfy c_9, $Cites(N_6, N_8)$ and $Cites(N_7, N_9)$ have to be inserted, and we are clearly entering an infinite loop, which is not acceptable.

To cope with this difficulty, every null N is associated with an integer called *the degree of* N and denoted by $\delta(N)$. At each insertion, the degree of all nulls occurring in $\mathfrak{D}$ are set to 0, and when a constraint c is applied during the processing, all generated nulls are assigned a degree equal to $\delta + 1$ where δ is the maximal degree of the nulls in the atoms of the constraint body, or 0 if no null occurs in the constraint body. Moreover, assuming a fixed maximal null degree $\delta_{\max}$, insertion processing

is *stopped* as soon as a null N is such that $\delta(N) \geq \delta_{\max}$, and the insertion is *rejected*, that is $\mathfrak{D}$ is not changed.

For example, in the case just above, we have $\delta(N_6) = \delta(N_7) = 0$, $\delta(N_8) = \delta(N_6) + 1 = 1$ and $\delta(N_9) = \delta(N_7) + 1 = 1$, etc. If, for example $\delta_{\max}$ is set to 2, the generation of nulls will stop at the next round and the insertion will be **rejected**. The verification of null degree is similar in both *From-scratch* and incremental approaches (it was proven in [9] that by using δ_{max} we accept only consistent insertions).

(D) Avoiding redundancies (core). Side effects have to be computed in a *minimal* way to reflect as much as possible the so-called *minimal change* requirement. To illustrate this point consider the insertion in $\mathfrak{D}'$ of $Authors(Nils, P_{235})$. Adding this fact in $\mathfrak{D}'$ provokes redundancies, because the presence of $Authors(Nils, N_1)$ and of $Publication(N_1)$ is no longer required to ensure constraint satisfaction. The result of this insertion is the set $\mathfrak{D}'''$ defined by:

$$\mathfrak{D}''' = (\mathfrak{D}' \setminus \{Authors(Nils, N_1), Publication(N_1)\}) \cup \{Authors(Nils, P_{235})\}$$

In our implementations, redundancies in $\mathfrak{D}$ are eliminated through the computation of the *core*, seeking for mapping nulls to constants or nulls so as to detect redundant atoms. In our example, for h such that $h(N_1) = P_{235}$, we have:
– $h(Authors(Nils, N_1)) = Authors(Nils, P_{235})$ and
– $h(Publication(N_1)) = Publication(P_{235})$,
showing that $Authors(Nils, N_1)$ and $Publication(N_1)$ are redundant.

From-scratch approach. Once the updates are performed on the database, the whole instance is considered for simplifications.

Incremental approach. This new proposal aims to retrieve only the facts involved in the update operation. For instance, for $\mathfrak{D}'$ as in our example, in the incremental approach a query detects that N_1 is the *only* null value concerned by the update. No need to work with the whole instance $\mathfrak{D}'$.

From-scratch and incremental approaches at a glance. Consider the update process that includes the general ideas explained in items (B) and (C) above. Denote, respectively, by *upd* and $upd_{|U}$, its *from-scratch* and *incremental* versions. More precisely, when using the $upd_{|U}$ policy, only the database portion impacted by U is concerned, while the whole database is concerned by *upd* policy. The expression $\mathfrak{D} \diamondsuit U$ indicates the insertion/deletion of the required updates U in/from $\mathfrak{D}$.

In the *from-scratch* approach the new instance is denoted by $\mathfrak{D}' = core(upd(\mathfrak{D} \diamondsuit U))$, while in the incremental approach, the new instance is denoted by $\mathfrak{D}' = core_{|NullBucket}(upd_{|U}(\mathfrak{D} \diamondsuit U))$, where *NullBucket*, is the set of nulls impacted by the update policy ($upd_{|U}$) applied to $\mathfrak{D} \diamondsuit U$.

3 Preliminaries

We recall some formal definitions already used in [9]. We assume a standard FOL alphabet composed of three pairwise disjoint sets, namely: const, a set of constants, var, a set of variables and pred, a set of predicates, every predicate being associated with a positive integer called its arity. In this setting, a *term* is a constant or a variable and an atomic formula, or an atom, is a formula of the form $P(t_1, \ldots, t_n)$ where P is a predicate of arity n and $t_1, \ldots, t_n$ are terms. Every atom in which no variables occur is called a *fact*.

A *homomorphism* from a set of atoms A_1 to a set of atoms A_2 is a mapping h from the terms of A_1 to the terms of A_2 such that: (i) if $t \in$ const, then $h(t) = t$, and (ii) if $P(t_1, ..., t_n)$ is in A_1, then $P(h(t_1), ..., h(t_n))$ is in A_2. The set A_1 is *isomorphic* to the set A_2 if there exists a homomorphism h_1 from A_1 to A_2 which admits an inverse homomorphism (from A_2 to A_1).

We denote by Φ the set of all formulas of the form $(\exists \mathbf{X})(\varphi_1(\mathbf{X}_1) \wedge \ldots \wedge \varphi_n(\mathbf{X}_n))$ where $\mathbf{X}$ is a vector of variables made of all variables occurring in $\mathbf{X}_i$ $(i = 1, \ldots, n)$, and where for every $i = 1, \ldots, n$, $\varphi_i(\mathbf{X}_i)$ is an atomic formula in which the free variables are those in $\mathbf{X}_i$. If ϕ denotes such a formula in Φ, the set $\{\varphi_1(\mathbf{X}_1), \ldots, \varphi_n(\mathbf{X}_n)\}$ is denoted by $atoms(\phi)$.

Given ϕ in Φ, a *model* M of ϕ is a set of facts such that there exists a homomorphism from $atoms(\phi)$ to M. In such a setting, for all ϕ_1 and ϕ_2 in Φ, $\phi_1 \Rightarrow \phi_2$ holds if each model of ϕ_1 is a model of ϕ_2, and as usual, ϕ_1 and ϕ_2 in Φ are said to be *equivalent*, denoted by $\phi_1 \Leftrightarrow \phi_2$, if $\phi_1 \Rightarrow \phi_2$ and $\phi_2 \Rightarrow \phi_1$ both hold, that is if ϕ_1 and ϕ_2 have the same models.

For all ϕ_1 and ϕ_2 in Φ, ϕ_1 is said to be *simpler than* ϕ_2, denoted by $\phi_1 \preceq \phi_2$, if (i) $\phi_1 \Leftrightarrow \phi_2$ holds, and (ii) $atoms(\phi_1) \subseteq atoms(\phi_2)$. ϕ_1 is also said to be a *simplification* of ϕ_2. A simplification ϕ_1 of ϕ_2 is said to be *minimal* if $\phi_1 \preceq \phi_2$ and there is no ϕ_1' such that $\phi_1' \prec \phi_1$. For instance, let ϕ be the formula $(\exists x, y)(P(a, x) \wedge P(a, y))$; then $(\exists x)(P(a, x))$ and $(\exists y)(P(a, y))$ are two distinct but *equivalent* simplifications of ϕ.

It is shown in [9] that if ϕ is in Φ and ϕ_1 and ϕ_2 two *minimal* simplifications of ϕ, then $atoms(\phi_1)$ and $atoms(\phi_2)$ are isomorphic (in the literature, we find a similar result for graphs[22]). Minimal simplifications are also called *cores* and the core of a given formula ϕ is denoted by $core(\phi)$.

Basically, a *database instance* is a formula ϕ in Φ that cannot be simplified, *i.e.,* such that $core(\phi) = \phi$. Formulas in Φ are 'skolemized' by replacing the variables with specific constants referred to as *Skolem constants* or as *(marked) nulls* and by omitting the existential quantifier. We thus assume an additional set of symbols in our alphabet, denoted by NULL, disjoint from the sets CONST and VAR. Now a term can be of one of the following types: either a constant, or a null, or a variable. Any atom of the form $P(t_1, \ldots, t_n)$ where for every $i = 1, \ldots, n$, t_i is in CONST $\cup$ NULL, is called an *instantiated atom*. Given an instantiated atom A, denote by $null(A)$ the set of nulls appearing in A. Moreover, as usual, the transformed conjunctive formula is written as the *set* of its conjuncts. In other words, *a database instance is a set of instantiated atoms that can be written as* $atoms(Sk(\phi))$ *where* $Sk(\phi)$ *is the Skolem version of a formula* ϕ *in* Φ *such that* $core(\phi) = \phi$.

4 Simplification with Respect to Nulls: a Basic Operation

In our approach, a database $\mathfrak{D}$ is expected to be equal to its core to avoid data redundancy. It is thus of paramount importance to enforce this property when updating. To this end, we propose *incremental* algorithms, so as to deal with as few nulls as possible, based on those involved in the update processing.

More formally, given a set of atoms I and a set of nulls ν occurring in I, we look for a homomorphism h such that for every N not in ν, $h(N) = N$ and $h(I)$ is minimal so as $h(I) \subseteq I$. However, the following example shows that the choice of ν cannot be arbitrary. Indeed, given a set of nulls ν_0, with respect to which I is to be simplified, the set ν_0 has to be expanded to the set ν of all nulls 'linked' (directly or indirectly) to a null in ν_0 in some atom of I.

Example 1 Let $\nu_0 = \{N_1\}$ and I defined by:

$I = \{\ Student(Alice), Enrolled(Alice, N_1), Degree(N_1, N_2), Enrolled(Alice, Math),$
$Degree(Math, N_3), Degree(CS, N_4), Degree(CS, BSc)\ \}$

To simplify I with respect to ν_0, we should eliminate redundancies in I involving N_1. As N_1 occurs in $Degree(N_1, N_2)$ with the other null N_2, the simplification should deal with N_1 *and* N_2. Since N_1 and N_2 are not linked with any other null in the atoms of I, we have $\nu = \{N_1, N_2\}$. For h such that $h_1(N_1) = Math$ and $h_1(N_2) = N_3$, we obtain a non-redundant instance $I' = h(I)$ defined by

$I' = \{\ Student(Alice), Enrolled(Alice, Math), Degree(Math, N_3),$
$Degree(CS, N_4), Degree(CS, BSc)\ \}.$

Notice however that simplifications involving N_3 or N_4 have *not* to be considered. □

As shown by the above example, given I and ν_0, nulls 'linked' in I to nulls in ν_0 have to identified. We do so through the computation for every N in ν_0, of the set $\mathsf{LinkedNull}_{I,N}$ as explained next. We first define the sequence $\left(\mathsf{LinkedNull}^k_{I,N}\right)_{k\geq 0}$ by:

(i) $\mathsf{LinkedNull}^0_{I,N} = \{A_i \in I \mid N \in null(A_i)\}$

(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 easy to see that for every $k \geq 0$, we have $\mathsf{LinkedNull}^k_{I,N} \subseteq \mathsf{LinkedNull}^{k+1}_{I,N}$ and $\mathsf{LinkedNull}^k_{I,N} \subseteq I$. Thus, the sequence $\left(\mathsf{LinkedNull}^k_{I,N}\right)_{k\geq 0}$ is bounded by I and is monotonic. As I is finite, the 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 all nulls occurring in this set. Algorithm 1 shows how redundancies are dealt with in this context.

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

```
1: PSet := {LinkedNull_{I,N} | N ∈ ν0}
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 receives as input a set I of instantiated atoms, and a set of nulls ν_0. For each N in ν_0, the algorithm computes the set $\mathsf{LinkedNull}_{I,N}$ (line 1), which is stored in a set called $PSet$. Therefore, the nulls occurring 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 x_i. That is, $A_i(X_i)$ is obtained from A_i in P by replacing the nulls in A_i by the corresponding variables.

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 define a possible instantiation of the nulls in P. In this case, some atoms in P are redundant, and thus can be removed. 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 $h_m(A)$.

Example 2 Considering I as in Example 1 and $\nu_0 = \{N_1\}$, $\mathsf{LinkedNull}_{I,N_1}$ consists of the atoms $Enrolled(Alice, N_1)$ and $Degree(N_1, N_2)$. Thus, the query q_{core} is defined by:

$ans(x_1, x_2) \leftarrow Enrolled(Alice, x_1), Degree(x_1, x_2)$

returning the answer $\{(N_1, N_2), (Math, N_3)\}$ with more than one tuple. Hence, h_m such that $h_m(N_1) = Math$ and $h_m(N_2) = N_3$ is returned line 5, and I is simplified as illustrated in Example 1. □

To explain our method for computing the most specific homomorphism h_m we introduce the notion of P-homomorphism.

Definition 1 *Given I a set of instantiated atoms and N a null occurring in I, let $P = \mathsf{LinkedNull}_{I,N}$. A P-homomorphism is a homomorphism h such that $h(I) \subseteq I$ and for every null N' in $null(I) \setminus null(P)$, $h(N') = N'$.*

I is said to be P-reducible if there exists a P-homomorphism h such that $h(I)$ is a strict subset of I. □

In the following proposition, given a set I of instantiated atoms and a null N in ν_0, we use the following notation:

- P denotes the set of atoms $\mathsf{LinkedNull}_{I,N}$, and $null(P) = \{N_1, \dots N_p\}$ denotes the set of nulls occurring in P;
- $q_{core}(I)$ is the answer to q_{core} computed against I. That is, $q_{core}(I)$ is the set $\{h_1, \dots, h_q\}$ of all possible P-homomorphisms defined over $null(P)$. We suppose that h_1 is the identity, *i.e.,* for every $j = 1, \dots, p$, $h_1(N_j) = N_j$;
- H_P denotes the table with p columns and q rows such that $H_P[i, j] = h_i(N_j)$.
- Given a set of atoms Q, we denote by $cons\_null(Q)$ the set of all symbols σ such that σ is a constant or a null *not* in $null(Q)$.

We recall that given two homomorphisms 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 $h \circ h_1 = h_2$. Using these notation, the following proposition holds.

Proposition 1 *Given h_i and $h_{i'}$ in $q_{core}(I)$, $h_i \preceq h_{i'}$ holds if and only if, for every $j = 1, \dots, p$, we have:*

1. *If $H_P[i, j]$ is in $cons\_null(P)$, then $H_P[i, j] = H_P[i', j]$;*
2. *If $H_P[i, j]$ is a null N in $null(P)$, then for every $j' \neq j$ such that $H_P[i, j] = H_P[i, j']$ it holds that $H_P[i', j] = H_P[i', j']$.*

Proof. Let us first assume that $h_i \preceq h_{i'}$ holds. In this case, there exists h such that $h \circ h_i = h_{i'}$. If N_j is such that $h_i(N_j)$ is a constant or a null not in $null(P)$, then for every P-homomorphism h_P, $h_P(h_i(N_j)) = h_i(N_j)$. Hence, $h_{i'}(N_j) = h \circ h_i(N_j) = h_i(N_j)$, which shows item (1). If j and j' are such that $h_i(N_j) = h_i(N_{j'})$, then $h_{i'}(N_j) = h_{i'}(N_{j'})$ also holds, showing item (2).

Conversely, assume that for h_i and $h_{i'}$, items (1) and (2) hold. Let h be defined for every $j = 1, \dots, p$ as follows: if there exists N_k such that $h_i(N_k) = N_j$ then $h(N_j) = h_{i'}(N_k)$, otherwise $h(N_j) = N_j$. We first notice that h is well defined. Indeed, if k and k' are such that $h_i(N_k) = h_i(N_{k'})$, then we have two expressions defining $h(N_j)$, namely $h(N_j) = h_{i'}(N_k)$ and $h(N_j) = h_{i'}(N_{k'})$. However, by item (2) we have $h_{i'}(N_k) = h_{i'}(N_{k'})$, and thus, these two expressions yield the same value. We now prove that $h_i \preceq h_{i'}$, that is, that for every $k = 1, \dots, p$, then $h_{i'}(N_k) = h(h_i(N_k))$. If $h_i(N_k)$ is not in $null(P)$, then, we have $h_i(N_k) = h_{i'}(N_k) = N_k$, and by construction of h we also have $h(N_K) = N_k$. Therefore $h_{i'}(N_k) = h(h_i(N_k)) = N_k$. On the other hand, if $h_i(N_k) = N_j$, by definition of h, we have $h_{i'}(N_k) = h(N_j)$. Hence, $h_{i'}(N_k) = h(N_j) = h(h_i(N_k))$. Since for every $k = 1, \dots, p$, we have $h_{i'}(N_k) = h(N_j) = h(h_i(N_k))$, it follows that $h_i \preceq h_{i'}$, and the proof is complete. □

Example 3 Let $I = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a), C(N_3, a)\}$ and $\nu_0 = \{N_1\}$.

In this case, $P = \{\{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a)\}\}$ and thus $null(P) = \{N_1, N_2\}$ and $cons\_null(P) = \{a, N_3\}$. This implies that P-homomorphisms should *not* change N_3, or in other words, N_3 should be treated as constant. The query q_{core} is thus written as follows:

$q_{core} : ans(x_1, x_2) \leftarrow B(x_1, x_2), B(x_2, x_1), C(x_1, a), C(x_2, a)$

and the table H_P representing the answer $q_{core}(I)$ is shown below.

H_P	x_1	x_2
1	N_1	N_2
2	N_2	N_1

H_P has 2 columns (because $null(P)$ contains two nulls), and 2 rows due to two answers in $q_{core}(I)$. It is easy to see that $h_1 \preceq h_2$, and $h_2 \preceq h_1$ meaning that there is no advantage in trying to simplify the database instance in this case. Indeed, we have $h_1(I) = I$, where h_1 is the identity. We have $h_2(I) = I$ as well, although h_2 is not the identity. Remark that h_2 does not satisfy $h_2 = h_2 \circ h_2$ (*i.e.,* h_2 is not idempotent) because $h_2(h_2(N_1)) = h_2(N_2) = N_1$, whereas $h_2(N_1) = N_2$. As will be seen shortly, detecting such homomorphisms allows for computational optimizations. □

The following corollary shows how to find one most specific homomorphism, based on Proposition 1. To state the corollary, we use the following notation for $i = 1, \ldots, q$:

- γ_i is the number of nulls N in $null(P)$ such that $h_i(N)$ is in $cons\_null(P)$;
- $\mu_i = \{k \in \{1, \ldots, q\} \mid (\forall j = 1, \ldots, p)(h_i(N_j) \in cons\_null(P) \Rightarrow h_k(N_j) = h_i(N_j))\}$;
- π_i is the number of distinct nulls in $null(P)$ in the set $h_i(null(P))$.

Intuitively speaking, considering that H_P is the tableau, then γ_i is the number of columns that, at row i, contain a symbol in $cons\_null(P)$. On the other hand, μ_i is the set of all rows in H_P containing the same symbols of $cons\_null(P)$ in the same columns as row i does (*i.e.,* if $h_i(N_j) = c$ then $h_k(N_j) = c$). Then π_i is the number of distinct nulls in $nulls(P)$ occurring in row i.

The corollary below formalizes the following informal remarks:

1. If $h_i \neq h_i \circ h_i$, then h_i cannot be one of the most specific homomorphisms, because in this case, $h_i \prec h_i \circ h_i$. For instance, in Example 3, we have $h_2 \neq h_2 \circ h_2$.
2. Most specific homomorphisms are among the rows of H_P with the largest number of symbols in $cons\_null(P)$. Indeed, let h_i and h_j be such that row i contains strictly more symbols in $cons\_null(P)$ than row j and $h_i \prec h_j$. Then, there exists h such that $h \circ h_i = h_j$, and so, if N in $null(P)$ is such that $h_i(N)$ is in $cons\_null(P)$, we have $h(h_i(N)) = h_i(N)$, and so $h_i(N) = h_j(N)$. Thus, row j has at least as many symbols in $cons\_null(P)$ as row i, which implies a contradiction. Hence, for every N in $null(P)$, $h_i(N)$ is also in $null(P)$, in which case rows i and j have no symbols in $cons\_null(P)$, which is another contradiction.
3. Considering one of the rows defined just above, say row i, among all rows having the same symbols in $cons\_null(P)$ in the same columns as row i, we argue that a row with as few distinct nulls in $null(P)$ defines one most specific homomorphism.

Corollary 1 *Given I and P as above, denoting by $\{h_1, \ldots, h_q\}$ the set $q_{core}(I)$, the following holds:*

1. *If h_i is one of the most specific P-homomorphisms in $q_{core}(I)$ then h_i is idempotent, that is, $h_i \circ h_i = h_i$.*
2. *h_i is one of the most specific P-homomorphisms in $q_{core}(I)$ if (a) $\gamma_i = \max_{1 \leq j \leq q}(\gamma_j)$, and (b) $\pi_i = \min_{k \in \mu_i}(\pi_k)$.*

PROOF. First, Proposition 1 implies that $h_i \preceq h_i \circ h_i$ holds for every h_i. Moreover, as h_i is a P-homomorphism, we have $h_i(I) \subseteq I$. Thus $h_i \circ h_i(I) \subseteq I$, which implies that $h_i \circ h_i$ is a P-homomorphism as well. The proof of item (1) is therefore complete.

Assume that h_i satisfies (2) and let h_k be a P-homomorphism such that $h_i \preceq h_k$. By Proposition 1, if $h_i(N_j)$ is in $cons\_null(P)$, then $h_i(N_j) = h_k(N_j)$. Therefore, $\gamma_i \leq \gamma_k$, and as $\gamma_k \leq \gamma_i$, this implies $\gamma_i = \gamma_k$. Thus, k is in μ_i, which implies that $h_i(N_j)$ is in $null(P)$ if and only if so is $h_k(N_j)$. By Proposition 1, if j and j' are such that $h_i(N_j) = h_i(N_{j'})$ then we also have $h_k(N_j) = h_k(N_{j'})$. It therefore follows that less nulls in $null(P)$ occur for h_k, that is $\pi_k \leq \pi_i$. As $\pi_i \leq \pi_k$ must hold, we obtain that $\pi_i = \pi_k$, meaning that h_i and h_k are equal up to a null renaming. The proof is therefore complete. □

Algorithm 2: $ChooseMostSpecific(q_{core}(I))$

```
1: Build H_P as explained in Proposition 1
   {H_P has q rows and p columns}
2: row_max := 1 ; count_max := 0 ; i := 2
3: for all i = 2, ..., q do
4:     idemPot := true
5:     count_curr := 0 ; j := 1
6:     while idemPot = true and j ≤ p do
7:         if H_P[i, j] is in cons_null(P) then
8:             count_curr := count_curr + 1
9:         else
10:            Let N_k = H_P[i, j] {N_k is in null(P)}
11:            if H_P[i, k] ≠ N_k then
12:                idemPot := false {h_i is not idem-potent}
13:                Mark row H_P[i]
14:        j := j + 1
15:    if idemPot = true then
16:        if count_curr > count_max then
17:            row_max := i
18:            count_max := count_curr
19: row_spec := row_max
20: Let count_min be the number of distinct nulls in null(P) occurring in H_P[row_max]
21: for all i = 2, ..., q do
22:    if row H_P[i] is not marked and i ≠ row_max then
23:        match := true ; j := 1
24:        while match = true and j ≤ p do
25:            if H_P[row_max, j] is in cons_null(P) and H_P[row_max, j] ≠ H_P[i, j] then
26:                match := false
27:            j := j + 1
28:        if match = true then
29:            Let count_null be the number of distinct nulls in null(P) occurring in H_P[i]
30:            if count_null < count_min then
31:                row_spec := i
32:                count_min := count_null
33: return h_m, the homomorphism defined by H_P[row_spec]
```

As a consequence, finding a most specific P-homomorphism in $q_{core}(I)$ amounts to (i) discard any row not defining an idem-potent homomorphism and (ii) among the remaining rows, identify one homomorphism satisfying item 2 of Corollary 1. Algorithm 2 shows how to compute such a most specific homomorphism, and we notice that this does *not* require data access. To end the section, we illustrate Algorithm 2 as follows.

Example 4 We first consider the context of Example 3, where $I = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a), C(N_3, a)\}$ and $P = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a)\}$.

In this case, $null(P) = \{N_1, N_2\}$, $cons\_null(P) = \{N_3\}$, and the associated table H_P has been shown already. Applying Algorithm 2 based on the table H_P, the following computations are achieved.

The first loop line 3 aims at marking rows defining non idempotent P-homomorphisms (that is, such that $h \circ h \neq h$) and mean-while to find one unmarked row with as many symbols in $cons\_null(P)$ as possible, in reference to Corollary 1(2). These computations return the following:

- When processing row 2 of H_P, we have $H_P[2, 1] = N_2$ where N_2 is in $null(P)$, and $H_P[i, 2] = N_1$. Since $N_1 \neq N_2$, *idemPot* is set to `false` and row 2 is marked on line 13.
- Since there is no other row to process, the loop line 3 returns $row\_max = 1$ and $count\_curr = 0$.

Hence, Algorithm 2 returns $row\_spec = 1$ and so, h_m is defined by $h_m(N_1) = N_1$ and $h_m(N_2) = N_2$. In other words, I is not simplified, which is indeed the expected result.

We now illustrate further Algorithm 2, using two more sophisticated cases. First, let $\nu_0 = \{N_1\}$ and $I_1 = \{B(N_1, N_2), B(a, N_2), B(a, N_3), B(N_4, N_3), C(N_2, N_2), C(N_3, N_3)\}$. In this case, $\mathsf{LinkedNull}_{I,N_1} = \{N_1, N_2\}$ and thus, $PSet = \{P\}$ where $P = \{B(N_1, N_2), B(a, N_2), C(N_2, N_2)\}$. Moreover, the query

$q_{core} : ans(x_1, x_2) \leftarrow B(x_1, x_2), B(a, x_2), C(x_2, x_2)$

is generated and its answer against I_1, $q_{core}(I_1)$, is defined in the following table H^1_P:

H^1_P	x_1	x_2
1	N_1	N_2
2	a	N_2
3	a	N_3
4	N_4	N_3

H^1_P has 2 columns and 4 rows due to four possible answers in $q_{core}(I_1)$. Moreover, $h_1 \preceq h_2$, $h_2 \preceq h_3$ and $h_2 \preceq h_4$. Notice that h_3 and h_4 are not comparable because a, N_3 and N_4 are in $cons\_null(P)$. Applying Algorithm 2 based on the table H^1_P, the first loop line 3 achieves the following:

- No row is marked as non-idempotent on line 13. This is so because for every $i = 1, \ldots, 4$, and every $j = 1, 2$, if $H^1_P[i, j] = N_k$ where N_k is N_1 or N_2, $H^1_P[i, k] = N_k$.
- Regarding the value of $count\_curr$, the computed value is 0 for the first row, 1 for row 2, and 2 for rows 3 and 4 (because a, N_3 and N_4 are in $cons\_null(P)$). Thus, applying the test line 16, $count\_curr$ to set to 2, and on line 17, row_max is set to 3. Indeed, although for row 4, we have $count\_curr = 2$, the test line 16 fails, and thus the value of row_max is not changed. Then, row_spec is set to 3 on line 19 and $cont\_min$ is set to 0 on line 20.

Therefore, processing the loop line 21 yields no change and Algorithm 2 returns h_m defined by $h_m(N_1) = a$ and $h_m(N_2) = N_3$, in which case, $h_m(I_1) = \{B(a, N_3), B(N_4, N_3), C(N_3, N_3)\}$, which is not redundant, when considering N_3 and N_4 as particular 'constants'.

As a more sophisticated illustration of Algorithm 2, let $\nu_0 = \{N_1\}$ and $I_2 = \{B(N_1, N_2),\ B(a, N_2),\ C(N_2, N_2),\ C(N_2, N_3)\}$. Here, $\mathsf{LinkedNull}_{I,N_1} = \{N_1, N_2, N_3\}$ and thus, $PSet = \{P\}$ where $P = \{B(N_1, N_2),\ B(a, N_2),\ C(N_2, N_2),\ C(N_2, N_3)\}$. Moreover, the query:

$q_{core} : ans(x_1, x_2, x_3) \leftarrow B(x_1, x_2), B(a, x_2), C(x_2, x_2), C(x_2, x_3)$

is generated and $q_{core}(I_2)$, is defined in the following table H^2_P:

H^2_P	x_1	x_2	x_3
1	N_1	N_2	N_3
2	N_1	N_2	N_2
3	a	N_2	N_3
4	a	N_2	N_2

H^2_P has 3 columns and 4 rows due to four possible answers in $q_{core}(I_2)$. Moreover, $h_1 \prec h_2$, $h_1 \prec h_3$, $h_2 \prec h_4$ and $h_3 \prec h_4$. Applying Algorithm 2 based on the table H^2_P, the loop line 3 achieves the following:

- As above, no row is marked as non idempotent on line 13. This is so because for $i = 1, \ldots, 4$, and $j = 1, 2, 3$, if $H^2_P[i, j] = N_k$ where N_k is N_1, N_2 or N_3, $H^2_P[i, k] = N_k$.
- As above, on line 17, *row_max* is set to 3 and thus, *row_spec* is set to 3 on line 19. Here, *count_min* is set to 2 on line 20 because N_2 and N_3 are in $null(P)$.

When processing the loop line 21, the only row to be considered is row 4, for which *match* is `true`, thus implying that the test on line 28 succeeds. Since for row 4, the value of *count_null* is 1 (because row 4 contains the only null N_2), the value of *row_spec* is set to 4, line 31. Hence, Algorithm 2 returns h_m defined by $h_m(N_1) = a$, $h_m(N_2) = N_2$ and $h_m(N_3) = N_2$. In this case, $h_m(I_2) = \{B(a, N_2), C(N_2, N_2)\}$, which is not redundant. □

Homomorphisms have been used in database theory during the last decades, in the field of query optimization [4, 10] (we refer to [2] for an overview). We notice in this respect that, in [4], a partial pre-ordering between homomorphisms is defined using the same criteria as in Proposition 1, showing that our approach to simplification is closely related to the field of query optimization. Roughly, in our approach, we compare all the answers $(h_1, \ldots h_q)$ for q_{core} and chose one (h_m) among the most specific ones (which are incomparable). From another point of view, if we consider queries Q_1, Q_2, ..., Q_q as the instantiations of q_{core} by $h_1, \ldots h_q$, respectively, then h_m is a homomorphism such that $h_m(body(Q_i)) = body(Q)$ for all $Q \subseteq Q_i$. Actually, our simplification technique is based on tableau optimization, as done in [4] for query optimization, where the sets of variables and of distinguished variables are, respectively, called, in our approach, the $null(P)$ and *cons_null*. However, the contexts and the expectations in our approach are fundamentally different from those summarized in [2]. Indeed:

- In [2], the tableau is built up from the query *body*, whereas in our approach, the tableau is built up from the answer to a given query.
- Our approach generates *one* most specific homomorphism, where as the approach shown in [2] aims at discarding all non most specific.

As a result, the problem we deal with can be seen as *more specific* than the general case considered in [2, 4], thus resulting in a specific algorithm.

Query	Algo	Purpose
$q_{bucket}(I)_{[S]}$	3, 5	retrieves all nulls in I appearing in an atom $p(...)$ such that p is a predicate in a given set S
$q_{degree}(I)_{[S,\delta_{max}]}$	3	for each N in S, checks if N is in I and if $\delta(N) < \delta_{max}$
$q_{\delta}(I)_{[S,d]}$	3	for each N in S, if N is in I, sets $\delta(N)$ to d
$q_{Iso}(I)_{[S]}$	5	retrieves in I all atoms isomorphic to those in S

Figure 2: Queries used in our algorithms

5 Incremental Updating

In [9], update algorithms work on in-memory data, using no DataBase Management System (DBMS). This version considers a DBMS, based on which data access is implemented through queries. In this section, we show how to implement updates by restricting data access as much as possible.

5.1 Insertion

Algorithm 3 describes the insertion in $\mathfrak{D}$ of the atoms in the set iRequest. On line 1, the side-effects of the insertion are computed and stored in the set *ToIns*, and then the instance $\mathfrak{D}' = \mathfrak{D} \cup ToIns$ is simplified on line 3 through the computation of its core. If all nulls in the simplified instance have a degree less than the specified maximal degree δ_{max} (on line 4), null degrees are all set to 0 (on line 5) and $\mathfrak{D}'$ is returned since, as shown in [9], it is always consistent; otherwise, the database is not modified.

Algorithm 3: Insert($\mathfrak{D}, \mathbb{C}, \delta_{max}$, iRequest)

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

Contrary to the algorithms in [9], the main steps in Algorithm 3 are designed in an *incremental* manner. First, to avoid generating any non necessary side effect atoms, an *incremental* version of the chase procedure considered. According to this procedure, a constraint c is activated *only* when the following conditions hold:

(i) $body(c)$ contains at least one atom that maps to one being inserted, and
(ii) atoms in $body(c)$ that do not respect (i) map to atoms in the database instance $\mathfrak{D}$.

This new chase differs from the one in [9] in the following aspects: (a) only the rules c concerned by insertions are triggered and (b) queries are built to find in $\mathfrak{D}$ instantiations for atoms in $body(c)$.

Algorithm 4, called on line 1, implements our incremental chase procedure. The set $ToIns$ initially stores iRequest (line 1) and then, stores the generated side-effects (line 3) through the while loop on line 2, defined by the following conditions:

Algorithm 4: Chase4Insert($\mathfrak{D}, \mathbb{C}, \delta_{max}$, iRequest)

1: $ToIns :=$ iRequest
2: **while** $\exists c \in \mathbb{C}$ and $\exists h$ such that $h(body(c)) \subseteq \mathfrak{D} \cup ToIns$ and $h(body(c)) \cap ToIns \neq \emptyset$ and $\delta(h'(head(c)) \leq \delta_{max}$, where $h' \supseteq h$ maps to new nulls all existential variables in $head(c)$, and there does not exist h'' such that $h''(h'(head(c)) \in \mathfrak{D} \cup ToIns$ **do**
3: $ToIns := ToIns \cup \{h'(head(c))\}$
{Degrees of new nulls in $h'(head(c))$ are set to $d_{\max} + 1$, where $d_{\max}$ is the maximal degree in $h(body(c))$, or 0 if $h(body(c))$ contains no null}
4: **return** $ToIns$

- A constraint c is triggered *only* if at least one atom in $body(c)$ is instantiated to an atom in $ToIns$.
- The condition $\delta(h'(head(c)) \leq \delta_{max}$ ensures that only (side-effect) atoms whose degree is less than the maximum null degree are kept. The instantiation h' extends h by assigning new null values to existential variables in $head(c)$. When performing a chase step, the degree of new nulls are also computed.
- The last condition ensures termination along with a simplification. Indeed, if an instantiation of $h'(head(c))$, referred to as $h''(h'(head(c)))$, exists in $\mathfrak{D} \cup ToIns$, the constraint is satisfied, and no atom is inserted in $ToIns$. For instance, suppose $\mathfrak{D}_1 = \{Authors(Elin, P_2)\}$, $\mathbb{C} = \{c_5\}$ (Figure 1) and iRequest $= \{Researcher(Elin)\}$. The atom $Authors(Elin, N_1)$, generated by c_5, is not inserted since it maps to $Authors(Elin, P_2)$.

Another difference between the algorithm in [9] and Algorithm 3, is the simplification step on line 3 to maintain the database instance irredundant. Indeed, based on our earlier discussion in Section 4, $\mathfrak{D} \cup ToIns$ is simplified with respect to the nulls in *NullBucket*, computed through the query q_{bucket} on line 2. Thus, only the nulls in *NullBucket* and their 'linked' nulls are considered, thus optimizing the computation of the core of $\mathfrak{D} \cup ToIns$.

Example 5 Let $\mathbb{C} = \{c_1, c_3, c_4, c_{10}, c_{11}, c_{12}\}$, $\delta_{max} = 3$ and the following database instance:

$\mathfrak{D} = \{\ Authors(N_1, P_2), Authors(Alice, N_2), Publication(P_2), Publication(N_2),$
$Researcher(N_1), Researcher(Alice), Supervises(N_1, N_3)\ \}$

Let iRequest $= \{Authors(Alice, P_5), Student(Bob)\}$. Running Algorithm 3 in this case is as follows. Constraint c_4 is triggered due to the insertion of $Authors(Alice, P_5)$ and constraints c_{10}, c_{11}, c_{12} are triggered due to the insertion of $Student(Bob)$. Line 1 returns the following set $ToIns$, where null degrees are shown as exponents:

$ToIns = \{\ Authors(Alice, P_5), Publication(P_5), Student(Bob), Enrolled(Bob, N_5^0),$
$Degree(N_4^0, N_5^1), Language(N_4^0, N_5^1, N_6^2)\ \}.$

To simplify $\mathfrak{D} \cup ToIns$, the query q_{Bucket} retrieves in $\mathfrak{D}$ the nulls concerning *Authors* (*i.e.,* N_1 and N_2), *Publication* (*i.e.,* N_2), *Enrolled* (*i.e.,* N_4), *Degree* (*i.e.,* N_4, N_5) and *Language* (*i.e.,* N_4, N_5, N_6). Therefore, $NullBucket = \{N_1, N_2, N_4, N_5, N_6\}$, and by Algorithm 1, we obtain that $\mathsf{LinkedNulls}_{\mathfrak{D},N_1} = \{N_1, N_3\}$, $\mathsf{LinkedNulls}_{\mathfrak{D},N_2} = \{N_2\}$, and for $i = 4, 5, 6$, $\mathsf{LinkedNulls}_{\mathfrak{D},N_i} = \{N_4, N_5, N_6\}$. The simplification of $\mathfrak{D} \cup ToIns$ (line 3 of Algorithm 3) results in:

$\mathfrak{D}' = \{\ Authors(N_1, P_2), Authors(Alice, P_5), Publication(P_2), Publication(P_5), Researcher(N_1),$
$Researcher(Alice), Supervises(N_1, N_3), Student(Bob), Enrolled(Bob, N_4),$
$Degree(N_4, N_5), Language(N_4, N_5, N_6)\ \}.$

Notice also that, since the degree of nulls is checked only during the chase, before returning the updated instance, the degrees of all nulls are set to 0 on line 5 of Algorithm 3. □

5.2 Deletion

Algorithm 5: $Delete(\mathfrak{D}, \mathbb{C}, \delta_{max}, \mathsf{dRequest})$

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

Our incremental algorithm for the deletion from $\mathfrak{D}$ of atoms in dRequest is displayed as Algorithm 5. 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 $\mathsf{dRequest} = \{P(a, N_1)\}$ and $\mathfrak{D}_1 = \{P(a, N_5)\}$, then query q_{iso} returns $\{P(a, N_5)\}$. The side-effects are then computed on line 6, recalling from Section 2 that the side effects involve not only atoms to be deleted, but also atoms to be *inserted* as side-effects. In Algorithm 5, the corresponding sets are respectively denoted by $ToDel$ and $ToIns$.

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.

As for insertions, side effects are computed *incrementally* through Algorithm 6. First, it may happen that the deletion of an instantiated atom A makes the database inconsistent when it is a consequence of a constraint c. To find all such constraints c, we reason backward on $\mathbb{C}$ to find an instantiation h such that $h(head(c)) = A$. Then h is extended to verify, in a forward reasoning, whether $body(c)$ can be triggered and generate A again.

The idea is to check whether c generates an atom isomorphic to an atom being deleted (Algorithm 6, line 3). If so, at least one atom in $h(body(c))$ should be deleted in order to prevent c from being triggered. This atom is then inserted in $ToDel$ (line 4). Notice that, to avoid non-determinism, it is assumed that the atom to be deleted has been marked as '$-$' during rule design.

If no atom isomorphic to an atom to be deleted is generated, a new set called $NewToIns$ is generated as the side-effects of inserting the new instance of $head(c)$ and all atoms in $ToIns$ (line 6). If no atom in $NewToIns$ meets an atom to be deleted and if the degrees of the involved nulls are less that δ_{max}, then these atoms are inserted in $ToIns$ (line 8). Otherwise, the marked atom from $h(body(c))$ is inserted in $ToDel$ (line 10).

Example 6 Let $\mathfrak{D}_0 = \{GrantEligible(Sten), Student(Sten), Enrolled(Sten, CS)\}$, $\mathbb{C} = \{c_{10}, c_{13}\}$ and $\mathsf{dRequest} = \{GrantEligible(Sten)\}$.

On line 2, Algorithm 5 calls Algorithm 6 to perform an incremental chase. $ToIns$ and $ToDel$ are respectively initialized to $\emptyset$ and $\{GrantEligible(Sten)\}$, and a first iteration of the loop on line 2 is run. Constraint c_{13} is concerned by the deletion, because for h such that $h(head(c_{13})) = GrantEligible(Sten)$, as $Enrolled(Sten, CS)$ is in $\mathfrak{D}$, c_{13} generates $GrantEligible(Sten)$ (line 3). Therefore, $ToDel$ is set to $\{GrantEligible(Sten), Enrolled\ (Sten, CS)\}$ and $ToIns$ remains empty.

In the second iteration of the loop, c_{10} is detected to be concerned by the deletion of the atom $Enrolled(Sten, CS)$. With $Student(Sten)$ in $\mathfrak{D}$, c_{10} generates $Enrolled(Sten, N_1)$, which

Algorithm 6: Chase4Delete($\mathfrak{D}, \mathbb{C}, \delta_{max}$, isoDel)

1: $ToIns := \emptyset$ and $ToDel :=$ isoDel
2: **while** there exist $c \in \mathbb{C}$ and h such that $h(head(c)) \in ToDel$ and $h(body(c)) \subset (\mathfrak{D} \setminus ToDel) \cup ToIns$ **do**
3: **if** $\exists h'$ such that $h'(body(c)) = h(body(c))$ and $h'(head(c))$ is isomorphic to $h(head(c))$ **then**
4: $ToDel := ToDel \cup \{h'(body^-(c))\}$
5: **else**
6: $NewToIns :=$ Chase4Insert($\mathfrak{D}, \mathbb{C}, \delta_{max}, ToIns \cup \{h'(head(c))\}$)
7: **if** $NewToIns = Del = \emptyset$ and $\delta(N) < \delta_{max}$ for all nulls N in $NewToIns$ **then**
8: $ToIns = ToIns \cup NewToIns$
9: **else**
10: $ToDel := ToDel \cup \{h'(body^-(c))\}$
11: **return** $ToDel, ToIns$

is not isomorphic to $Enrolled(Sten, CS)$ (line 3). The next step consists in testing whether the atom $Enrolled(Sten, N_1)$ should be added to $ToIns$. To this end, Algorithm 6 chases forward, starting with $Enrolled(Sten, N_1)$ (line 6) to generate $GrantEligible(Sten)$. This atom being in $ToDel$ (line 7), $Student(Sten)$ is added to $ToDel$, and nothing is added in $ToIns$. Algorithm 6 returns $ToDel = \{Student(Sten), GrantEligible(Sten), Enrolled(Stem, CS), Enrolled(Sten, N_1)\}$, and $ToIns = \emptyset$. Algorithm 5 then performs the deletions and the resulting database instance is empty. □

6 Queries for Incremental Processing

By implementing our method using graph and relational database models, our goal is to study performance aspects, and to raise issues concerning the database design regarding queries.

6.1 Graph Data Model

The DBMS considered in this work is Neo4J, which deals with attributed graphs. Cypher is a well-established language for querying and updating *property graph databases*. As explained in [16], 'a Cypher query takes as input a property graph and outputs a table. These tables can be thought of as providing bindings for parameters that witness some patterns in a graph, with some additional processing done on them'. The central concept in Cypher queries is pattern matching. The MATCH clause searches for homomorphisms identifying a given pattern in the queried graph. The returned result is an instance over a table where attributes correspond to the variables in the Cypher query.

Our approach involves managing null values that have to be retrieved based on their co-occurrences as arguments of atoms (Section 4). Given a null N_1 we need to efficiently detect atoms having N_1 as one of its arguments, and then for every N occurring with N_1, to recursively access the atoms having N as argument. In doing so, the set of nulls is partitioned into blocks whose elements are those nulls that have to be considered in the simplification steps. To make such retrieval efficient, we adopt a model close to the logical formalism used in our previous explanations, composed of three types of nodes. Given an atom $P(t_1, \ldots, t_n)$ our graph database represents P as a node, linked to other nodes representing the terms $t_1, \ldots, t_n$. In this context, nodes in our graph database are

of three possible types distinguished by *labels*, and all nodes have *properties*, among which one is **symbol**. More precisely:

- Nodes of type *Atom* have one label **:Atom** representing the predicate symbol in an atom. This predicate symbol is the value of property **symbol** of such a node.
- Nodes of type *Constant* representing constant values. Such nodes have two labels, **:Element** and **:Constant**, and the value of their property *symbol* is the constant itself.
- Nodes of type *Null* reprensenting nulls. Such nodes have two labels, **:Element** and **:Null**, and the value of their property **symbol** is the name of the null prefixed with '_'.

An edge links nodes with label **:Atom** to nodes with label **:Element**. Moreover, an edge has the property *rank*, allowing to refer to the terms of an atom by their positions.

Figure 3 illustrates the schema of the atom $P(t_1, \ldots, t_n)$ by representing constant terms by t_i and nulls by t_j. Notation below edges indicates the cardinality of the relationship between an atom and its terms: an element is connected to at least one atom and atoms may have no terms.

Figure 4 illustrates part of our database instance (rectangular nodes are atoms and circular nodes are elements).

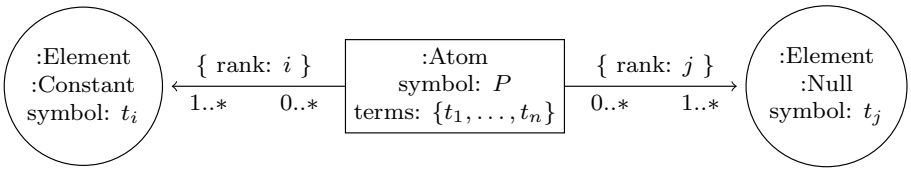


Figure 3: Graph database schema.

As explained before, our model benefits certain operations. However, it increases the cost of conversions between the graph-format and the logic-format for an atom. Such conversions are essential for the communication between the database and the procedures performing some computations locally. To optimize these conversions and graph traversals, we introduce the following redundancies in our database model, which have significantly improved our implementation (see Section 7).

- *To avoid edge traversal.*

  For each node **:Atom**, we store, as its attribute, an ordered list containing all its terms. In Figure 3, the rectangular node shows this new attribute: *terms*. For example, to obtain atom $Authors(Elin, P_{269})$ from the instance in Figure 4 starting with the node n_{117}, instead of traversing edges r_{19} and r_{20}, we just have to retrieve the attributes *terms* of node n_{117}.

- *To allow efficient access to nodes.*

  (a) A uniqueness constraint is added on the **Element** *symbol* (implying, *e.g.*, that there is a unique node in the database to represent *Elin*).

  (b) An index is built on the *symbol* of each atom, and a uniqueness constraint is defined on the couple *symbol/terms* (implying, *e.g.*, that there is a unique node in the database to represent atom $Authors(Elin, P_{269})$).

The algorithms presented in the previous sections involve the construction of queries in Cypher to be evaluated on our Neo4J database. We now focus on two of them: one needed when chasing and one that computes the set LinkedNull.

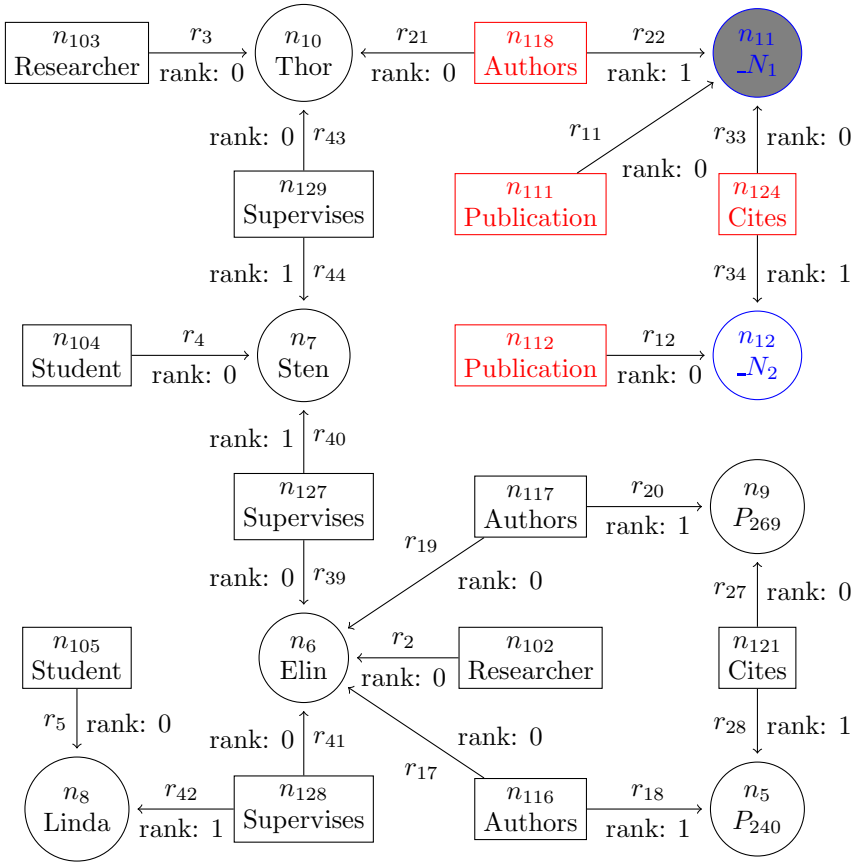


Figure 4: Graph database instance (extract). Optimization labels and attributes are omitted.

Query for chasing. Chasing means applying constraints. The application of a constraint happens when its body can be instantiated by facts in the database. Thus, to decide on the application of a constraint c, we need a query capable of :

(1) Verifying whether the database instance contains the facts necessary for the instantiation of $body(c)$ and

(2) returning a non-empty answer only if a corresponding instantiation for $head(c)$ does not already exist in the database.

In a logic formalism, if c is of the form $c : L_1(\alpha_1), \ldots, L_m(\alpha_m) \rightarrow L_0(\alpha_0)$, we should write the query $q_{ch} : Q(\alpha) \leftarrow L_1(\alpha_1), \ldots, L_m(\alpha_m), not\ L_0(\alpha_0)$, where α is the list of variables corresponding to variables in $body(c)$, that is, variables universally quantified variables of c. The idea here is: if h_t is an instantiation such that $h_t(body(c)) \subseteq \mathfrak{D}$, the query 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.

Figure 5 shows the Cypher template of query q_{ch}. We first look for atoms that match $body(c)$. On the line 3 in Figure 5, the `WHERE NOT EXISTS` clause is used to check that no instance of the $head(c)$ exists. Two expressions are built (`expr1` and `expr2`). Terms in α are treated orderly. Notice that `expr1` is built for dealing with atoms in $body(c)$ and `expr2` is built for dealing with the

```
1  MATCH (x_k:Element {value: t_i})
2  MATCH (a_1:Atom {symbol: 'L_1'}), ..., (a_m:Atom {symbol: 'L_m'})
3  WHERE expr1 and NOT EXISTS {
4        MATCH (a:Atom { symbol: 'L_0'})
5        WHERE expr2
6  } RETURN {α|_1 : x_1, α|_2 : x_2, ..., α|_k : x_k}}
```

Figure 5: Cypher template for chasing

```
1  MATCH (x0:Element:Constant {value: 'Bob'}),
2        (x1:Element:Constant {value: 'P_1'})
3  MATCH (a0:Atom {symbol: 'Authors'  }),
4        (a1:Atom {symbol: 'Authors'  }),
5        (a2:Atom {symbol: 'Supervise'})
6  WHERE (a0)-[:Authors   {rank: 0}]->(x2),
7        (a0)-[:Authors   {rank: 1}]->(x1),
8        (a1)-[:Authors   {rank: 0}]->(x0),
9        (a1)-[:Authors   {rank: 1}]->(x1),
10       (a2)-[:Supervise {rank: 0}]->(x2),
11       (a2)-[:Supervise {rank: 1}]->(x0)
12 and NOT EXISTS {
13       MATCH (a:Atom {symbol: 'PhDPaper'})
14       WHERE (a)-[:PhDPaper {rank: 0}]->(x0),
15             (a)-[:PhDPaper {rank: 1}]->(x1)
16 } RETURN {'X': x2.value, 'Y': x0.value, 'P': x1.value} AS sub
```

Figure 6: Cypher template for Example 7

atom in $head(c)$. The first `MATCH` acts as a starting point of the graph traversal. It is built with constants or nulls (*e.g.*, `(`x_k`:Element:Constant {symbol:`t_i`})`) as we usually consider constraints instantiated by insertions. Then the pattern, built with the second `MATCH` and the `WHERE` clause, links the constants to the positions in the atoms of the body.

Separating the two MATCH allows us to guide the query planner to first search the constants (called node seeking) and then look for the connected nodes to find the atoms. This is important because, in doing so we drastically reduce the search space, because constants are unique values retrieved in $O(1)$, and only **:Atom** connected nodes are searched, thus avoiding to visit all instance nodes of the predicate.

Example 7 Considering the insertion of $Authors(Bob, P_1)$ in the database instance

$\mathfrak{D} = \{Supervises(Alice, Bob), Authors(Alice, P_1)\}$

with the only constraint c_6 defined by:

$c_6 : Authors(X, P), Authors(Y, P), Supervises(X, Y) \rightarrow PhDPaper(Y, P, Z)$

Two instantiations h and h' should be checked: one on the first atom $Authors$ ($h(X) = Bob$, $h(P) = P_1$) and one for the second atom $Authors$ ($h'(Y) = Bob$, $h'(P) = P_1$). Figure 6 shows the chase query for the instantiation h' of the constraint c_6. □

Query to find LinkedNull sets. Figure 7 presents the Cypher query that implements the LinkedNull definition (Section 4) for building partitions of atoms. The clause `UNWIND` can transform any list into individual rows. For instance, if we consider a list `['Elin', 'Sten']` of constant symbols, the clause `UNWIND` over such a list gives a table with one column c and two rows whose values are `'Elin'` and `'Sten'`. In Figure 7, the clause `UNWIND` (line 1) is used to set nulls from a given list to our initial table with one row for each null. The goal of the first `MATCH` (line 2) is to select sub-graphs with

atoms sharing the same null. On the line 3, the range of the relationship (`*1..`) indicates that node `nullValueNode` can be connected to a node `endNode` by a path `pathP` of arbitrary length. Moreover, the direction is $\leftrightarrow$ indicates that `pathP` can be composed by edges having any orientation.

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 the line 6, the `WITH` clause performs a 'group by'. It allows to structure our working table with tuples where each null `nullValueNode` is associated to a list of `endNode`s (the nodes reached by paths `pathP`). On the 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. The resulting table partitions the atoms: each atom is associated to a list of nulls (those it is concerned by). In the worst case, the former list contains all atoms having a null in the database.

```
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
8  [n IN linkedNodes WHERE (n:Atom)] AS linkedAtoms,
9  [nullValueNode] + [n IN linkedNodes WHERE (n:Null)] AS linkedNulls
10 UNWIND linkedAtoms AS a
11 RETURN a.symbol as a, a.terms as e, linkedNulls
```

Figure 7: Cypher template to find LinkedNull sets

Example 8 Considering the graph of Figure 4, if we search for atoms whose nulls are linked to $\_N_1$, *i.e.,* `$nulls = ['_N1']`, after the first MATCH in Figure 7, we have:

nullValueNode	endNode	pathP
n_{11}	n_{111}	$[n_{11}, r_{11}, n_{111}]$
n_{11}	n_{118}	$[n_{11}, r_{22}, n_{118}]$
n_{11}	n_{124}	$[n_{11}, r_{33}, n_{124}]$
n_{11}	n_{12}	$[n_{11}, r_{33}, n_{124}, r_{34}, n_{12}]$
n_{11}	n_{112}	$[n_{11}, r_{33}, n_{124}, r_{34}, n_{12}, r_{12}, n_{112}]$

After the first WITH line 6, we have:

nullValueNode	linkedNodes
n_{11}	$[n_{111}, n_{118}, n_{124}, n_{12}, n_{112}]$

After the second WITH line 7, we have:

linkedAtoms	linkedNulls
$[n_{111}, n_{118}, n_{124}, n_{112}]$	$[n_{11}, n_{12}]$

□

6.2 Relational Data Model

Given an instantiated atom $P(t_1, \ldots, t_n)$ in the logical representation oinf a database, our relational model consists in defining a table whose schema is $R_P[A_1, \ldots A_n]$ where all attributes are of type

text. Notice that $P(t_1, \ldots, t_n)$ represents a tuple on R_P and, thus, $(t_1, \ldots, t_n)$ are values that can be constants or nulls (nulls have the symbol _ as a prefix). The translation of logical queries into SQL is straightforward. However, some operations require the construction of procedures to implement recursive queries. Algorithm 7 shows the implementation of LinkedNull in the relational context.

We argue in this respect that implementing Algorithm 7 using a recursive SQL query is not efficient. Indeed, to do so an additional table for storing the pairs of linked nulls is needed, and the following steps are necessary: (a) a recursive SQL query to compute the transitive closure and (b) a scan of the whole database to retrieve all corresponding atoms. Moreover, the additional table needs to be maintained up to date after each update, which requires further processing.

Algorithm 7: FindLinkedNull($\mathfrak{D}, NullBucket$)

```
1:  newNull := ∅, allNulls := ∅, linkedNullSet := ∅
2:  while NullBucket ≠ ∅ do
3:      allNulls := allNulls ∪ NullBucket
4:      for all table R_P in the database schema do
5:          for all tuple u in (select * from R_P where (A_1 in NullBucket) or ...
            or (A_n in NullBucket)) do
6:              build atom P(u); add P(u) in linkedNullSet
7:              for all null value _N ∈ null(u) do
8:                  if _N ∉ allNulls then
9:                      add _N in newNull
10:     NullBucket := newNull
11:     newNull := ∅
12: return linkedNullSet
```

We also notice that the implementation of an incremental chase in the relational model follows the idea of setting up query q_{ch} (as explained in Section 6.1) which can be written as an SQL query involving a `NOT EXISTS` clause.

6.3 Discussion

Querying graph database is significantly impacted by graph schema design. The schema we have chosen transforms nulls into first-citizen elements and facilitates operations where, by 'picking' a null, we can easily detect all atoms connected (directly or indirectly) to it. For instance, in Figure 4, if we 'pick' the null $\_N_1$ (the gray node n_{11}), we detect the atoms connected to it together with other nulls (*i.e.,* $\_N_2$, the blue node n_{12}). In other words, this model optimizes queries looking for linked nulls. However, it may not be appropriate for other kinds of queries. For instance, in the chase query, our model generates complex patterns that can be costly. The relational model is less flexible than graph models, and thus its impact on querying is weaker. However, relational model is not appropriate for the implementation of recursion, and nulls cannot be set as first-citizen element (identical null values appear repeatedly in the database instance). Algorithm 7 shows that to implement LinkedNull we have to check null values for *each* table, compromising the idea of an incremental approach. On the other hand, the graph model is well suited for implementing incremental algorithms, because as seen in Section 6.1, this model allows implementing LinkedNull by visiting *only* the atoms linked to nulls in $NullBucket$, as expected when considering an incremental computation.

7 Experimental Results

We gauge the performance of our incremental updating approach by analysing experiment results over a benchmark working on a graph (Neo4J) and a relational (MySQL) DBMS. A benchmark run executes an update on a database instance.

To build our database instances, we firstly view the original data sets from a FOL point of view. Roughly speaking, a node or a relationship in the original data sets corresponds to a predicate symbol, while their properties are the terms. The conversion to our database models is straightforward, as presented in Section 6. Nulls are inferred from already missing properties. Constraints are handcrafted, created from data observation and added to the databases we use for experiments. The following three data sets are the basis of our instances:

- *Movie*[1], available as a Neo4J instance, is a collection of data concerning movies, actors, directors. This data set contains 7 predicate symbols (with arity 2-4).
- *GOT*[2], available as a Neo4J instance, deals with the interactions between different characters in the book *Game of Throne*. This data set contains 19 predicate symbols (artiy 2-14).
- *LDBC*[3], available as a data set of the Linked Data Benchmark Council, offers synthetic data sets for benchmarking. This data set contains 23 predicate symbols (artiy 1-2).

From the LDBC data sets we build several instances, by varying their size or the number of nulls. To control the size of instances, their construction is the result of: (i) randomly selecting k facts, respecting the distribution of the original data set and, then, (ii) applying the 39 hand-made constraints on them. The result is a consistent database instance with nulls. Figure 8 presents a summary of our database instances (or samples). It is worth noting that, for example, an instance denoted as LDBC 1K, comes from a random selection of 1000 facts which evolves to 2248 after the chase and core processing. To control the number of nulls, we proceed as follows: we take the largest LDBC instance, *i.e.,* with 10 000 facts, and replace all nulls with constants. Then, we choose, randomly, some constants that are replaced by linked nulls. Figure 8 presents database instances used in our runs, eight having nulls, and one non-null instance. All the database instances are generated just once. By following this creation process, they are consistent and minimal.

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

Figure 8: Database instances (our samples).

[1] `https://github.com/neo4j-graph-examples/movies`
[2] `https://github.com/neo4j-graph-examples/graph-data-science`
[3] `https://ldbcouncil.org/benchmarks/graphalytics/`

Runs are built from instances in Figure 8 by (i) varying the update type (insertion or deletion); (ii) altering the size of the update (1, 5, 10 and 20 atoms) and (iii) augmenting artificially the number of facts in an instance. This latter step is done through the duplication of data n-times (1, 2 and 5), together with the renaming of the constants and the null names at each copy.

Each run performs 10 iterations plus 3 warm-up iterations (*i.e.,* an ordinary iteration used to preload the system and database cache) not counted in the execution time. Between each iteration, the original database instance is restored, and the Java garbage collector is triggered for consistent time measuring. The benchmarks are implemented in Java 16 with MySQL 8 and Neo4J 4.1 and executed on a Rocky Linux 8.7 virtual server with 4 vCPU and 16 GB of memory (8 reserved for the database and 5 for the Java program) through docker 20.10.21. In the docker container of a database instance, the average of read/write on disk is $1\,\mathrm{GB\,s^{-1}}$. The same server hosts: (i) one database server at the time and (ii) the benchmarks with only 4 vCPU.

Notice that, even if this configuration allows us to assess our implementations over different DBMS, our experiment performances are not representative of real world situations, where more powerful and dedicated hardware is available.

We first compare the incremental approach presented in this paper to the from-scratch in-memory approach in [9]. For this aspect, comparisons are performed only on the database *Movie* because the from-scratch in-memory version requires a huge amount of memory for its computation. We have an average of 9017 ms for an update with the in-memory version and scale of 1 (initial size of the instance). MySQL has an average of 151 ms and Neo4J has 2380 ms. For this small instance, the incremental approach is comparable with the from-scratch approach. 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. Thus, it should be clear that using a DBMS in which an incremental version of update processing is implemented, allows for efficiently updating large databases that do not fit in main memory.

Next, we analyse the performance of incremental updating with respect to the number of atoms (database size) and nulls of an instance. We denote by *incompleteness degree* the number of distinct LinkedNull sets on a database. We also investigate the number of *queries* generated to interact with the DBMS. Figure 10 presents our experiment results. On each plot, the right axis, indicates the total number of facts in the instance. The curves show the average of resulting values for all runs corresponding to the displayed abscissa.

We first note that the update type (insertion or deletion) has no real impact on the performance of our approach. Figure 10a shows that the number of queries is linear on the number of nulls, except for three down spikes when the degree of incompleteness of the database instance is low. This is the case for the database *Movies*, and the down spikes coincide to a situation where only this database is concerned. Indeed, thanks to the use of multiple data sets, we observe here that the predicate arity (*i.e.,* the number of edges per node or the number of columns in a table) may have an impact on our results. Linearity with respect to the number of nulls is explained by the fact that consistency preservation implies the generation of new data linked by their nulls. Thus, due to our construction method, bigger databases imply more linked nulls (*i.e.,* bigger LinkedNull sets). Incremental updates generate q_{Bucket} queries to retrieve impacted nulls. Bigger databases likely have more impacted nulls willing to be simplified during the core computation, increasing the number of necessary q_{core} queries.

Consequences of bigger LinkedNull sets are:

(i) in MySQL, Algorithm 7 generates a large amount of queries and

(ii) in Neo4J, the unique query needed to retrieve a LinkedNull set is more complex and, thus, more time-consuming.

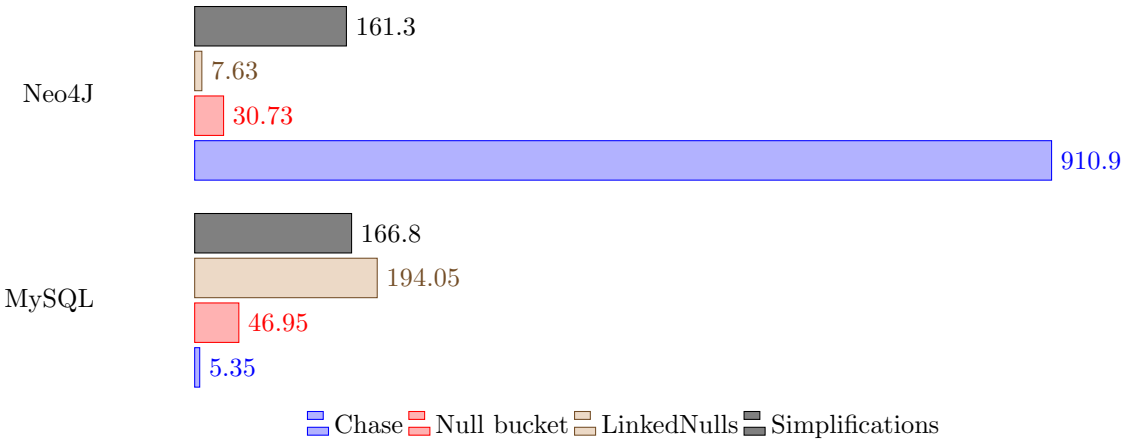


Figure 9: Average time of each operation per DBMS (ms) removing outsiders with more than 30 s differences

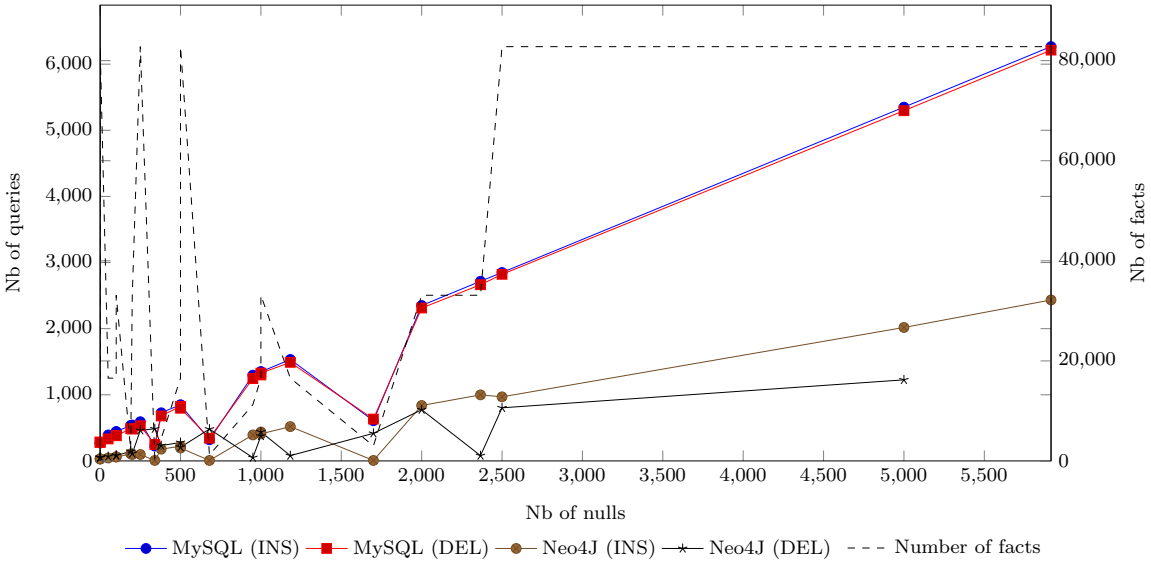
However, this augmentation is negligible as our model is designed to optimize such a query (Figure 10d).

Experimental results in terms of execution time of our updating approach is shown in Figures 10b and 10c. In MySQL (Figure 10b), update execution time is linear in the number of nulls while the database size has little impact. Indeed, as the number of queries increases with the number of nulls, update execution time in MySQL increases accordingly. In Neo4J (Figure 10c), update execution time is more significantly impacted by the size of the instance.

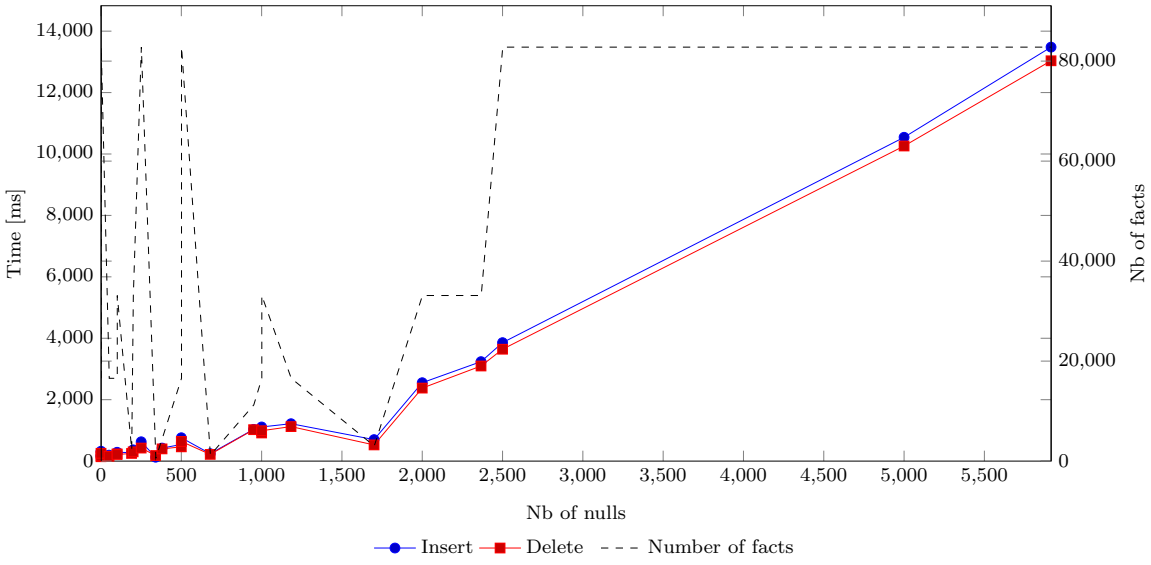
The explanation of this discrepancy comes from the separate analysis of the performance of the main operations of our approach (Figures 9 and 10d). The data model chosen in the Neo4J version optimizes the retrieval of LinkedNull, but is not appropriate to operations involving simplification (Section 4). Such operations involve complex pattern matching which are known to be expensive. The chase (Figure 9) is the most expensive operation for Neo4J, mainly due to the fact that it includes a simplification step (*e.g.*, if $A(a, b) \in \mathfrak{D}$ and $A(a, N_1)$ is generated by a constraint, then the insertion of $A(a, N_1)$ is canceled).

For the sake of readability, plots do not show results on GOT instances with more than 17 000 nulls. The results on this data set are similar: execution time evolves linearly with respect to nulls in MySQL and follows the size of the database in Neo4J. With the GOT runs, we achieve a mean execution time of 14 634 ms with MySQL and 5216 ms with Neo4J for 24 818 facts and 17 232 nulls. Increasing the size to 124 090 facts and 86 160 nulls rises run time to 156 132 ms with MySQL and to 203 140 ms with Neo4J.

Reproducibility. Results obtained by our experiments are reproducible through the use of the benchmarks and implementation available in `https://gitlab.com/jacques-chabin/UpdateChase`.

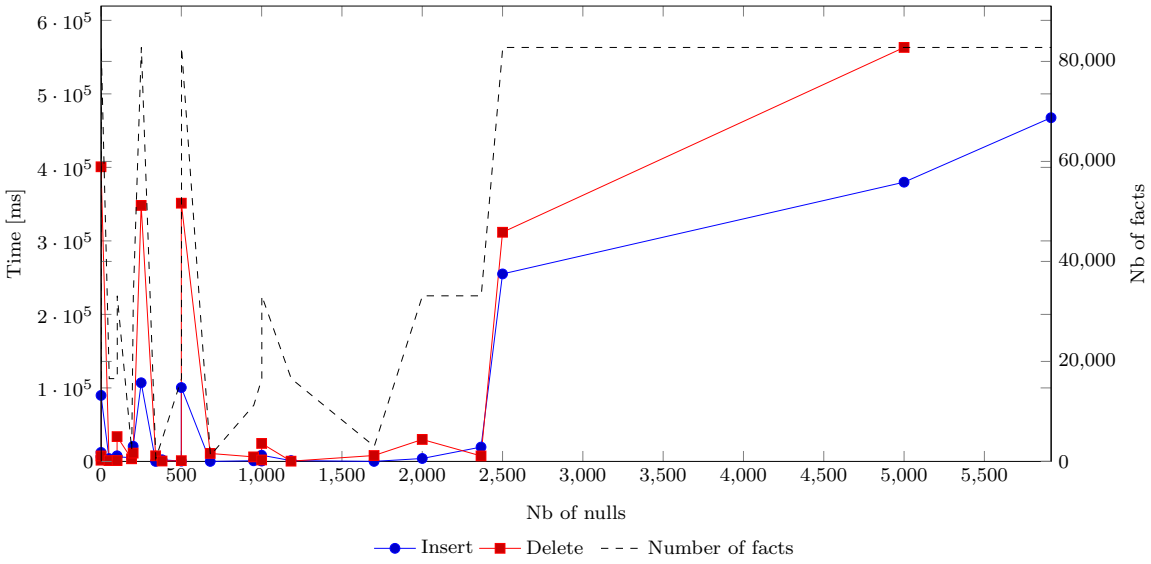


(a) Number of queries per null

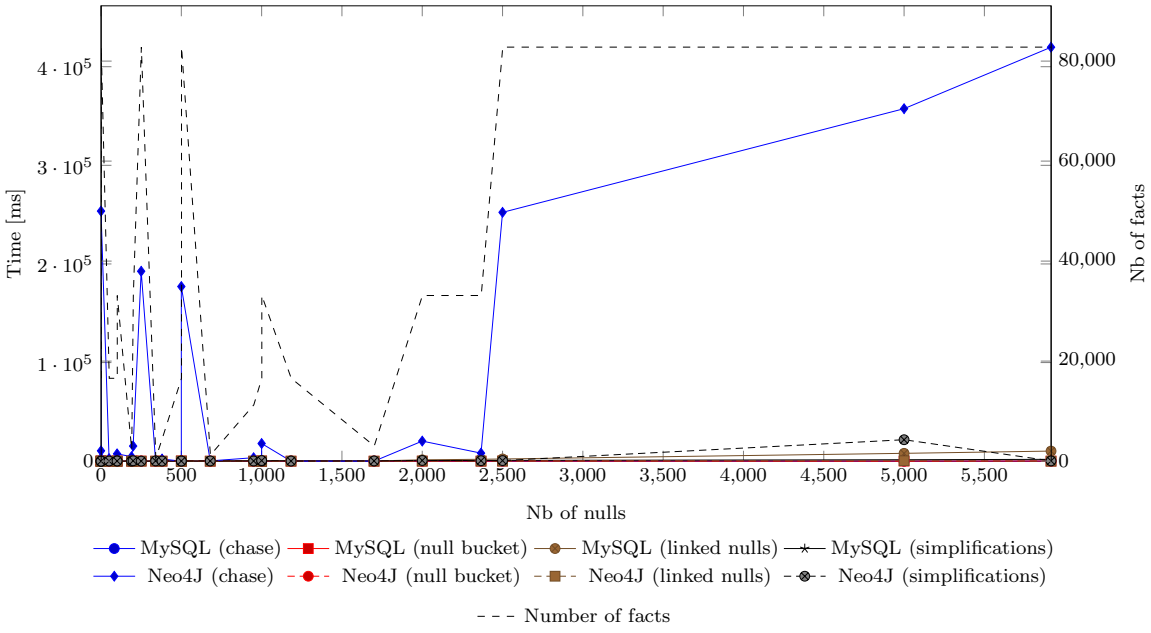


(b) Time per null for MySQL

Figure 10: Benchmarks results of 540 scenarios, average over 10 runs



(c) Time per null for Neo4J



(d) Time of operations per null

Figure 10: Benchmarks results of 540 scenarios, average over 10 runs

8 Related Works

Our work goals include four important features of modern applications: incompleteness, consistency as a measure of quality, incremental tools for efficient data processing and adaptability to graph data models.

Solid basis have been established for treating incompleteness of relational databases [12, 18, 23, 28, 32], particularly for querying. Much less attention has been given to updates on incomplete databases, although important work, such as [1, 13, 31] can be cited. Today, integrating and exchanging data are very common, leading to the proliferation of applications involving *dynamic incomplete data* on emerging data models that deal with more general graph-structured data. Incompleteness beyond the relational data model has received much less attention [30], and, in this context, updating with respect to constraints is rarely considered. Indeed, consistency maintenance is usually left aside in favour of efficiency, which can prove costly when we are concerned with the quality of analytical results. Work such as [21, 25, 29] witnesses the complexity of the problem of keeping a database consistent with respect to constraints in a dynamic environment. In [15, 19, 20] we find newer proposals, adapted to the RDF world, that considers constraints in our traditional database viewpoint (*i.e.,* not in the web semantic standard way, where constraints are just inference rules [17, 24, 27]). It is worth noting that the use of tuple generating constraints (TGD) increases expressiveness at the cost of difficulties that involve a chase procedure (cf. a survey in [26], a benchmark in [7]) to compute semantics and the generation of side effects in an update context - imposing extra insertions or deletions (with respect to those required by the user) to preserve consistency. The literature offers sufficient conditions to avoid a non-terminating chase which consist in limiting the format of constraints. We instead introduce δ_{max}, keeping the possibility of dealing with any kind of constraints while avoiding infinite processing. Furthermore, we use simplifications to keep the database instance as small as possible and to avoid the presence of useless nulls, *i.e.,* database maintenance consists in keeping its *core* (which follows the ideas in [11]) whose implementation is ensured by a simplification routine performed in association to update routines.

In brief, data analytic tools become essential in different application domains and their quality relies on data consistency. But in order to deal with huge scale applications, we must aim at efficient data processing solutions [30], bringing incremental solutions to the front of the stage, particularly when working with new data models (as done in the XML context [3, 6, 8]). In the context of graph databases, the approach in [14] proposes a method for 'incrementalizing' graph algorithms abstracted in a fix-point model. Our approach cannot be summarized by that proposal. As seen before, we can outline our method in the expression $\mathfrak{D}' = core_{|NullBucket}(upd_{|U}((\mathfrak{D} \diamondsuit U))$ where U is the set of user's required updates - this set is *increased* through an inference process that generates side-effects. The proposal in [14] needs a 'complete' set of updates as input. In other words, our fix-point operation involves changes on the update set while in [14] the update set is fixed. Their goal is to incrementally compute new answers on an updated graph and not to incrementally update the graph. As the *core* computation is not a fix-point one, it is not in the scope of [14].

Finally, our experiments reinforce the idea that graph schema design has a significant impact on query performance. Our graph schema is designed to optimize one type of query and performs badly to those that differ widely. Schema optimization may be a solution: as in [5], in this paper, it is done through techniques that reduce edge transversal.

9 Conclusions

This paper contributes to improve the maintenance of consistent incomplete databases by proposing incremental routines that interact with database systems. It extends prior work in [9] where a *from-*

scratch in-memory method was proposed. Two implementations of our approach, one under a graph database model and one under the traditional relational database model, are presented. Experiment results raise questions about the representation of nulls in a graph database. Indeed, this work is also a step towards incremental updating attributed graphs with incomplete data. It illustrates the impact of schema graph design in querying and, consequently, in the performance of an incremental updating approach that relies on two main queries: one that looks for linked nulls and another that looks for redundant atoms willing to be simplified. Property graph model has an increasingly important role today, the handling of nulls in such a model is related to schema definition and query optimization issues that need to be further explored.

References

[1] Abiteboul, S., and 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] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of databases*, vol. 8. Addison-Wesley Reading, 1995.

[3] Abrão, M. A., Bouchou, B., Halfeld Ferrari, M., Laurent, D., and Musicante, M. A. Incremental constraint checking for XML documents. In *XSym* (2004), no. 3186 in LNCS, pp. 112–127.

[4] Aho, A. V., Sagiv, Y., and Ullman, J. D. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst. 4*, 4 (1979), 435–454.

[5] Alotaibi, R., Lei, C., Quamar, A., Efthymiou, V., and Özcan, F. Property graph schema optimization for domain-specific knowledge graphs. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021* (2021), IEEE, pp. 924–935.

[6] Balmin, A., Papakonstantinou, Y., and Vianu, V. Incremental validation of xml documents. *ACM Trans. Database Syst. 29*, 4 (2004), 710–751.

[7] Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., and Tsamoura, E. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017* (2017), pp. 37–52.

[8] Bouchou, B., and Halfeld Ferrari Alves, M. Updates and incremental validation of XML documents. In *The 9th International Workshop on Data Base Programming Languages (DBPL)* (2003), Springer, Ed., no. 2921 in LNCS.

[9] Chabin, J., Halfeld Ferrari, M., and Laurent, D. Consistent updating of databases with marked nulls. *Knowl. Inf. Syst. 62*, 4 (2020), 1571–1609.

[10] Chandra, A. K., and Merlin, P. M. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on the Theory of Computing* (1977).

[11] Fagin, R., Kolaitis, P. G., and Popa, L. Data exchange: getting to the core. *ACM Trans. Database Syst. 30*, 1 (2005), 174–210.

[12] Fagin, R., Kuper, G. M., Ullman, J. D., and Vardi, M. Y. Updating logical databases. *Advances in Computing Research 3* (1986), 1–18.

[13] Fagin, R., Ullman, J. D., and Vardi, M. Y. On the semantics of updates in databases. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Colony Square Hotel, Atlanta, Georgia, USA* (1983), pp. 352–365.

[14] Fan, W., Tian, C., Xu, R., Yin, Q., Yu, W., and Zhou, J. Incrementalizing graph algorithms. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, pp. 459–471.

[15] Flouris, G., Konstantinidis, G., Antoniou, G., and Christophides, V. Formal foundations for RDF/S KB evolution. *Knowl. Inf. Syst. 35*, 1 (2013), 153–191.

[16] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 1433–1445.

[17] Gottlob, G., Orsi, G., and Pieris, A. Ontological queries: Rewriting and optimization. In *Proceedings of the 27th International Conference on Data Engineering, ICDE, Germany* (2011), pp. 2–13.

[18] Grahne, G. *The Problem of Incomplete Information in Relational Databases*, vol. 554 of *Lecture Notes in Computer Science*. Springer, 1991.

[19] Halfeld Ferrari, M., Hara, C. S., and Uber, F. R. RDF updates with constraints. In *Knowledge Engineering and Semantic Web - 8th International Conference, KESW, Szczecin, Poland, Proceedings* (2017), pp. 229–245.

[20] Halfeld Ferrari, M., and Laurent, D. Updating RDF/S databases under constraints. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS, Nicosia, Cyprus, Proceedings* (2017), pp. 357–371.

[21] Halfeld Ferrari Alves, M., Laurent, D., and Spyratos, N. Update rules in datalog programs. *J. Log. Comput. 8*, 6 (1998), 745–775.

[22] Hell, P., and Nesetril, J. The core of a graph. *Discrete Mathematics 109*, 1-3 (1992), 117–126.

[23] Imielinski, T., and Lipski Jr., W. Incomplete information in relational databases. *J. ACM 31*, 4 (1984), 761–791.

[24] Lausen, G., Meier, M., and Schmidt, M. Sparqling constraints for RDF. In *EDBT, 11th International Conference on Extending Database Technology, France, Proceedings* (2008), pp. 499–509.

[25] Link, S., and Schewe, K. An arithmetic theory of consistency enforcement. *Acta Cybern. 15*, 3 (2002), 379–416.

[26] Onet, A. The chase procedure and its applications in data exchange. In *Data Exchange, Integration, and Streams*. 2013, pp. 1–37.

[27] Patel-Schneider, P. F. Using description logics for RDF constraint checking and closed-world recognition. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, USA.* (2015), pp. 247–253.

[28] Reiter, R. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *J. ACM 33*, 2 (1986), 349–370.

[29] Schewe, K., and Thalheim, B. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybern. 13*, 3 (1998), 277–304.

[30] Sirangelo, C. *Representing and Querying Incomplete Information: a Data Interoperability Perspective*. 2014.

[31] Winslett, M. *Updating Logical Databases*. Cambridge University Press, New York, NY, USA, 1990.

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