A Data Quality Framework for Graph-Based Virtual Data Integration Systems

Yalei Li, Sergi Nadal(✉), and Oscar Romero

Universitat Politècnica de Catalunya (BarcelonaTech), Barcelona, Spain
yalei.li@estudiantat.upc.edu, {snadal,oromero}@essi.upc.edu

Abstract. Data Quality (DQ) plays a critical role in data integration. Up to now, DQ has mostly been addressed from a single database perspective. Popular DQ frameworks rely on Integrity Constraints (IC) to enforce valid application semantics, which lead to the Denial Constraint (DC) formalism which models a broad range of ICs in real-world applications. Yet, current approaches are rather monolithic, considering a single database and do not suit data integration scenarios. In this paper, we address DQ for data integration systems. Specifically, we extend virtual data integration systems to elicit DCs from disparate data sources to be integrated, using DC-related state-of-the-art, and propagate them to the integrated schema (global DCs). Then, we propose a method to manage global DCs and identify (i) minimal DCs and (ii) potential clashes between them.

Keywords: Data Quality · Data integration · Denial constraints

1 Introduction

We are nowadays witnessing an unprecedented growth in the volume of data that organizations are collecting as part of their decision making processes. With the proliferation of large-scale repositories of heterogeneous data, such as *data lakes* or open-data related initiatives, the ability to perform cross-analysis with high data quality deems a competitive advantage. Indeed, data quality is essential for the decision making process, where poor data quality can lead to wrong decision making, poor model performance, and operational instability [9,13,15]. Yet, in such large-scale data repositories, there coexists data generated by different providers who independently maintain them adhering to their own business rules and needs. Hence, the presence of missing, erroneous, out-of-date, or conflicting data is the norm rather than the exception [3,19].

Data integration systems, which have the main objective of providing an integrated view over an evolving and heterogeneous set of data sources [8], have mostly addressed data quality aspects from a *warehousing* perspective as part of their Extract-Transform-Load (ETL) processes [11]. This is, quality rules and

S. Chiusano et al. (Eds.): ADBIS 2022, LNCS 13389, pp. 104–117, 2022.
https://doi.org/10.1007/978-3-031-15740-0_9

constraints are enforced when materializing source data into the target schema. Yet, in those scenarios that require fresh query results, which are implemented via *virtual* integration systems that rewrite queries posed over the global schema in terms of queries over the data sources leveraging declarative mappings, the management of data quality remains a challenge [2,21]. Indeed, the kind of mappings adopted in this settings, represented as logical expressions, focus on specifying relations between source and target schemata but not how quality in the target schema must be enforced [12].

The state of the art on data quality management is focused on the automatic derivation of quality rules. This is, from a particular database instance, a set of rules are inferred via *rule mining* techniques and then implemented to detect errors (i.e., violations) [1]. To that end, the formalism of *denial constraints* has been widely adopted, as it is expressive enough to represent most data dependencies found in the literature such as key dependencies, functional dependencies, or order dependencies [5]. Succinctly, denial constraints are first-order formulae that express that a set of predicates cannot be all true for any combination of tuples in a relation, which are expressed as relationships between pairs of tuples of that relation. Despite the wide success of such model, which has given the rise to systems for denial constraint discovery (e.g., FastDC [6], Hydra [4], DynFD [20], or DCfinder [17]), or error detection and data repairing (e.g., Llunatic [7], HoloClean [18], or HoloDetect [10]), to the best of our knowledge they have not been studied in the context of virtual data integration systems.

In order to overcome the previously identified gap (i.e., manage data quality via denial constraints in a virtual data integration system), we present an approach that leverages related work on denial constraint discovery in order to synthesize rules from different data sources and manage them at the global (i.e., integration) level. To that end, our approach builds and extends a graph-based data integration system, which enables expressive visual query paradigms to non-expert users [16]. Precisely, we perform a bottom-up approach propagating the rules discovered at the sources to the target graph. In terms of DQ management, we take advantage of the techniques based on DC to express DQ rules. Two main phases are identified. The first step in this process is to elicit DCs at the source level and then propagate DCs to the target schema. Global rules consolidation is enabled in the integration graph, where users can actively manage and verify the rules even before propagating them to the integrated system.

Contributions. We summarize our contributions as follows:

- We define a data quality management framework that identifies quality rules (as denial constraints) per source, model them into a graph-based representation, and incrementally propagate them into the integrated schema.
- We globally conciliate rules automatically, which facilitates the identification of data cleaning tasks in the form of User Defined Functions.
- We extend a query rewriting algorithm to consider global DCs and enforce them over the underlying data sources.

Outline. The paper is structured as follows. Sections 2 and 3 discuss related work and introduce background concepts. Section 4, presents our approach, while Sect. 6 validates it. Section 7, concludes the paper and outlines future work.

2 Related Work

In this section, we review the state of the art in discovery and management of DCs. As previously stated, our objective is to leverage and benefit of from such methods on a graph-based data integration system. Thus, we narrow the scope to these projects that are openly available and guarantee reproducibility.

FASTDC [6]**.** It defines the syntax and fundamental semantics for DCs, and derives sound and complete inference rules. The implication testing algorithm checks whether a DC is implied by a set of DCs linearly, which effectively reduces the number of DCs in the output. The main DC discovery algorithm first builds the predicate space by comparing every tuple pair in the instance set, which contains all the possible predicates that can be formed into DCs. To overcome overfitting, FASTDC introduces an approximation parameter in A-FASTDC to allow flexibility in DCs satisfiability requirement. A DC stays valid if the percentage of violations on a instance over the total number of tuple pairs is below a given approximation threshold.

HYDRA [4]**.** In the spirit of FASTDC, aims to address the quadratic complexity in predicate evaluations and accelerate the DCs generation from the evidence sets. HYDRA devises a sampling technique to quickly approximate the DCs by processing only a small fraction of all tuples, providing adaptability to scale up with the number of tuples. It proposes to first samples tuple pairs to build an initial set of DCs for a dataset. The algorithm corrects the tuple pair samples from its focused sampling process and determines the complete evidence set to avoid the expensive comparison of all tuple pairs in FASTDC.

DCFINDER [17]**.** It follows FASTDC's approach with improvement on building evidence sets. DCFINDER generates a predicate space from the input database, and builds a data structure from the data records. It utilizes attribute value indexing to avoid the expensive tuple pair comparison of FASTDC. DCFINDER introduces predicate selectivity to drive efficiency even further to avoid the unnecessarily large number of logical operations when generating evidence sets. In an approach comparable to FASTDC, DCFINDER uses the DFS procedure to discover all minimal DCs based on evidence set coverage of DC candidates.

ADCMiner [14]**.** Focuses on mining approximate DCs, which discovers constraints in inconsistent databases and obtains more general and less contrived constraints. ADCMiner defines a novel approximation function that does not assume any specific definition of an approximate DC but takes the semantics as input. The function consists of two properties called *monotonicity* and *indifference*. ADCMiner generates all minimal ADCs if the approximation score is under

a given threshold. Unlike AFASTDC, ADCMiner reduces the process of finding ADCs by avoiding the post-process after detecting valid exact constraints. In general, the algorithm involves four parts: a predicate space generator, a sampler, an evidence set constructor, and an enumeration algorithm. Similar to HYDRA, ADCMiner includes a sampling process to reduce the running time significantly.

3 Preliminaries

In this section, we introduce the running example, which will be used to illustrate our approach, and later, discuss the formal background, which is also exemplified.

3.1 Running Example

We consider a (simplified) data integration scenario on the finance domain, related to organizations and their performance in the stock market. Table 1 presents exemplary data generated from three independent data sources. D_1 (see Table 1a) provides information about companies and their standard industrial classification of economic activities (SIC). D_2 (see Table 1b) yields contextual information related to the history about companies (i.e., founded year and founder/s). Finally, D_3 (see Table 1c) maintains information about the stock prices per company and date. In all cases, we consider the stock symbol to be the attribute used to join the different data sources.

Table 1. Three independent datasets providing information about companies, their history and stock prices

(a) D_1 – SIC

Symb	Comp	SIC
AAPL	Apple	3571
PYPL	Paypal	7389
V	Visa Inc	7389
GOOGL	Google	3571
...	...	...

(b) D_2 – History

S	N	Y	F
GOOGL	Alphabet Inc.	2015	LP&SB
F	Ford Motor C.	1903	HF
APPLE	Apple	1976	SJ&SW&RW
...	...	...	...

(c) D_3 – Stock

Code	Date	Price
AAPL	20220406	171.28
MMM	01/01/2001	100
V	20220406	220.86
...	...	...

As shown in the exemplary data, there exist data quality issues when considering each dataset individually. D_2's attribute names are coded and non-descriptive, while D_3 presents dates encoded in different formats and contains erroneous data (i.e., *MMM* is not a valid stock symbol). Note, however, additional data quality problems arise when considering their integration. If, as expected from the domain, we consider the stock symbol to be a company's primary key, then we can assume the existence of a functional dependency stating the symbol determines the company name. This, however, does not hold in the running example, where the symbol *GOOGL* has associated different names in different data sources. To manage such kind of situations (i.e., quality problems at both the local and the global level), the remainder of this section is devoted to present the formal background that our approach will build upon.

3.2 Formal Background

3.2.1 Graph-Based Virtual Data Integration

Here, we present the core components of our graph-based virtual data integration system. We refer the reader to [16] for further details on how queries are processed over such constructs.

Relations and Wrappers. A schema R is composed of a finite nonempty set of relational symbols $\{r_1, \ldots, r_m\}$, where each r_i has a fixed arity n_i. Let A be a set of attribute names, then each $r_i \in R$ is associated to a tuple of attributes denoted by $att(r_i)$. Let D be a set of values, a tuple t in r_i is a function $t : att(r_i) \rightarrow D$. For any relation r_i, $tuples(r_i)$ denotes the set of all possible tuples for r_i. We define the set of wrappers W as those elements in R that contain a function $exec(w)$ that returns a set of tuples $T \subseteq tuples(w)$. In practice, wrappers can be implemented via any language as long as there exists a mapping function from their data model to *first normal form* (1NF).

Global Graph. The global graph $\mathcal{G} = \langle V_\mathcal{G}, E_\mathcal{G} \rangle$ is an unweighted, directed, connected graph with no self loops. The vertex set $V_\mathcal{G}$ is partitioned into two disjoint sets C and F, respectively concepts and features. The set F itself is further partitioned into two disjoint subsets F_{id} and F_{id}^-, consisting of *id* features and *non-id* features, respectively. Next, labels in $E_\mathcal{G}$ contain the analyst's domain $\mathcal{L}$ as well as the set of *semantic annotations* $\mathcal{A}$. Semantic annotations are system specific labels and have a special treatment (e.g., `hasFeature`). Hence, we formalize the edge set $E_\mathcal{G}$ as the union of the sets $(C \times \mathcal{L} \times C)$ and $(C \times \{\texttt{hasFeature}\} \times F)$, the former assigning labels in $\mathcal{L}$ between concepts and the latter linking concepts and their features.

Source Graph. A source graph $\mathcal{S}$ is analogous to $\mathcal{G}$. However, here the vertex set $V_\mathcal{S}$ is composed of $(W \cup A)$, respectively the set of wrappers and attributes from the previous definition (note that $\mathcal{S}$ is a graph-based representation of the wrappers and their attributes). We use $wrap(\mathcal{S})$ to denote the set of wrappers in $V_\mathcal{S}$. Here, we introduce the semantic annotation `hasAttribute`, meant to connect a wrapper with its attributes. Thus, in $\mathcal{S}$ the edge set $E_\mathcal{S}$ is composed of $(W \times \{\texttt{hasAttribute}\} \times A)$.

Schema Mappings. A LAV schema mapping for a wrapper w is a pair $\mathcal{M}(w) = \langle \mathcal{F}, \gamma \rangle$, where $\mathcal{F}$ is an injective function $\mathcal{F} : att(w) \rightarrow F$; and γ is a subgraph of $\mathcal{G}$. Consequently, we define the functions $\mathcal{F}(w)$ and $\gamma(w)$ respectively denoting, for w, the mapping from attributes to features $\mathcal{F}$ and the subgraph γ. Recall that we encode mappings as part of the graph, precisely $\mathcal{M}$ contains $\mathcal{F}$ and φ. Thus, to encode $\mathcal{F}$ we extend the set of semantic annotations $\mathcal{A}$ with the `sameAs` label, linking attributes in $\mathcal{S}$ to features in $\mathcal{G}$.

Example 1. Figure 1 depicts the complete integration graph based on the running example depicted in Sect. 3.1.

3.2.2 Data Quality Management

Denial Constraints. A *predicate* P is a comparison unit in the form $v_1 \phi v_2$ or $v_1 \phi c$ where v_1, v_2 are values, respectively from the tuples t_x, t_y, ϕ is a comparison operator and c is a constant.

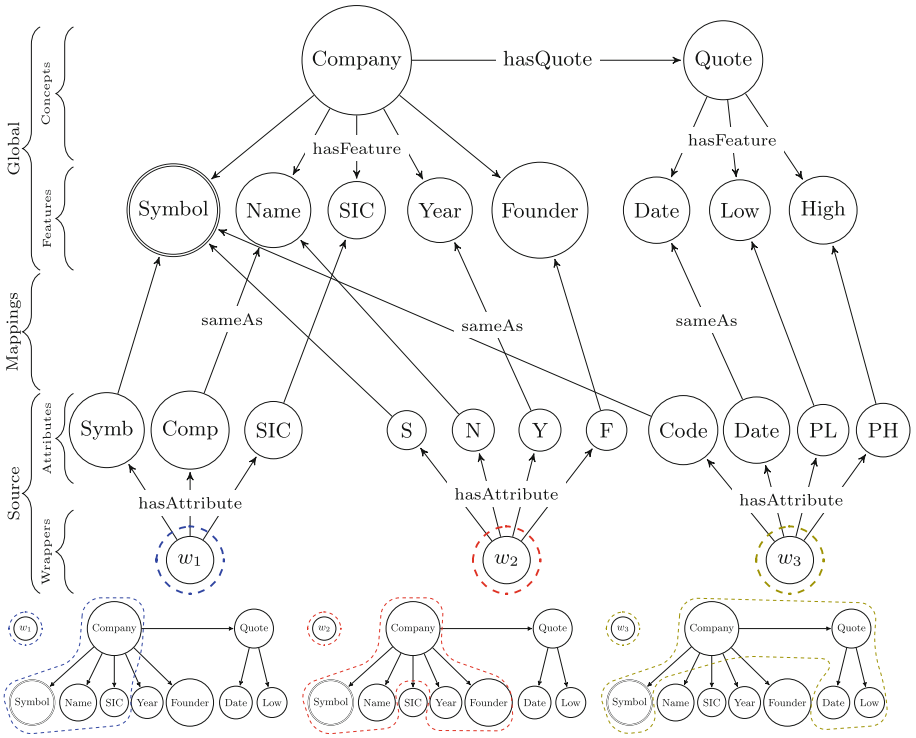

Fig. 1. An example integration graph. Doubly circled features denote IDs. The bottom colored graphs represent mappings (i.e., subgraphs of $\mathcal{G}$) for each wrapper dashed with the same color. There, some features have been omitted for clarity.

Definition 1 (Denial constraint). *A DC φ over a set of tuples T is an expression of the form $\forall t_x, t_y \in T, \neg(P_1 \wedge ... \wedge P_m)$ where φ is satisfied by T if and only if for any pair $t_x, t_y \in T$, at lease one of the predicates $P_1, ..., P_m$ is false.*

$r \models \varphi$ denotes a valid DC φ over a set of tuples T. This is, all predicates cannot be true for any tuple pair, otherwise, there is a violation. $\varphi.Pred$ denotes the set of predicates in φ. Then, we say a DC φ_1 is *minimal* if there does not exist a φ_2 such that $r \models \varphi_1, r \models \varphi_2$, and $\varphi_2.Pred \subset \varphi_1.Pred$.

Ranking DCs. In order to reduce the search space of valid DCs, a scoring function is defined to rank them. The *interestingness score* of each DC is calculated based on its succinctness and support from data. *Succinctness* models how overfitting a constraint rule is. This definition follows Occam's razor principle, where the competing hypothesis making fewer assumptions is preferred [6].

Definition 2 (Succinctness). *The succinctness of a DC φ, denoted $Succ(\varphi)$, is the minimal possible length of a DC divided by its own length $Len(\varphi)$. This is defined as $Succ(\varphi) = Min(\{Len(\phi)|\forall\phi\})$ / $Len(\varphi)$.*

Coverage determines the interestingness of a DC and measures its statistical significance. By definition, a valid DC φ needs to violate at least one predicate from the evidence. The higher number of satisfied predicates from the evidence, the more support it gives to φ. A pair of tuples satisfying k predicates is a *k-evidence* (kE). In the best case, the maximum k for a tuple pair in a DC φ is equal to $|\varphi.Pred| - 1$, otherwise it violates φ. A weight parameter is introduced to reflect a higher score to high values of k, from 0 to 1.

Definition 3 (Coverage). *A k-evidence (kE) for φ is a tuple pair $\langle t_x, t_y \rangle$, where k is the number of predicates in φ that are satisfied by $\langle t_x, t_y \rangle$ and $k \leq |\varphi.Pres| - 1$. The weight for a kE for φ is $w(k) = (k + 1) \,/\, |\varphi.Pres|$. The $Coverage(\varphi)$ is then defined as:*

$$Coverage(\varphi) = \frac{\sum_{k=0}^{|\varphi.Pred|-1} |kE| * w(k)}{\sum_{k=0}^{|\varphi.Pred|-1} |kE|}$$

4 Managing Data Quality in Virtual Data Integration

In this section, we present our proposed system structure and algorithms in detail. As depicted in Fig. 2, rectangles state the solutions we propose in the following sections. We first generate local DCs for each wrapper (i.e., source), and then propagate the DCs to the global graph. There, we establish algorithms to resolve the potential conflicts. Precisely, we address (a) minimal DCs maintenance to prune the redundancy of DCs at the global level; and (b) potential conflicts between DCs derived from contradictory predicates.

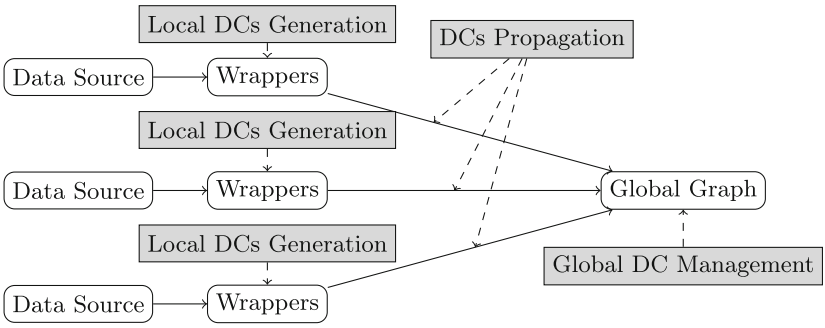

Fig. 2. Overview of the proposed solution process.

4.1 DC Generation and Graph-Based Representation

For each wrapper, we first utilize DCFINDER to produce DC rules at the local level and model them into its equivalent graph representation. For the sake of simplicity and ease of presentation, we narrow the kind of DCs we deal with to

those comparing attributes from the same wrapper, and thus no cross-attribute predicates are involved. Hence, DCFINDER generates DCs from each wrapper in the form of: $\{t.w.A_i\ \phi\ t.w.A_j \wedge ... \wedge t.w.A_x\ \phi\ t.w.A_y\}$. Additionally, due to the fact that DCFINDER tends to generate a large amount of DCs, we keep the interestingness score for each DC and filter top-k DCs based on the ranking.

Next, we describe how we model DC rules in an integration graph. To that end, we extend the vertex set of the source graph $\mathcal{S}$ with the set D of DCs. A DC $d \in D$ can be connected to a wrapper w via the semantic annotation `hasDC`. For each DC, we identify its predicates (via `hasPred`), which must be connected to two attributes from the same wrapper via `hasAtt1` and `hasAtt2`, and to an operator via `hasOp`. Additionally, we encode as nodes of $\mathcal{S}$ the confidence values of DCs, and link them via `hasScore`. Such model is likewise for $\mathcal{G}$, however here we consider the concept of *global DC*, which identifies a DC that must hold for all tuples at the global level (i.e., those generated from any of the wrappers via rewriting in the integration graph). To guarantee traceability and maintenance of the framework, DCs at the source level are connected to DCs at the global level via the `sameAs` semantic annotation.

Example 2. Consider a local DC expressed over w_3 stating that the high price of a stock quote should always be greater than the low one at any given date, expressed as the predicate: $\{t.PL \geq t.PH\}$. Figure 3, depicts the integration graph modeling it at the source and global graphs.

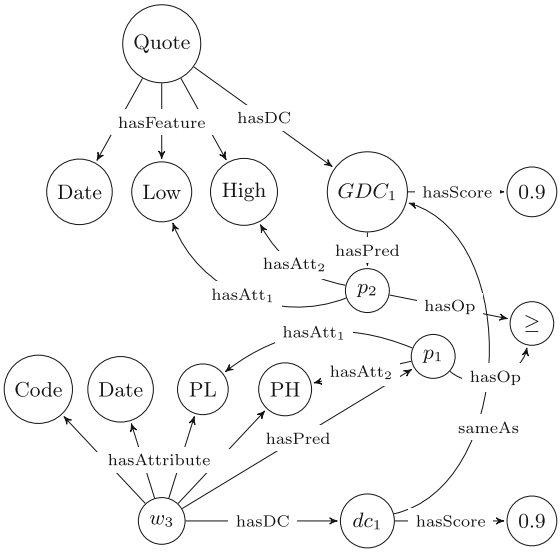

Fig. 3. Integration graph with DCs. Some edges, such as `sameAs` edges from attributes to features have been omitted for clarity.

4.2 Global DC Management

Once all local DCs have been propagated to the global graph, we perform two tasks to globally manage DCs: minimal DC maintenance and clash management.

Minimal DC Maintenance. It is essential to maintain minimal DCs in the global graph in order to reduce the number of valid constraint rules (as any minimal DC is still a valid DC if we add any other predicate). When a new DC φ is propagated from a source to the global graph and it happens to subsume a minimal global DC ϕ then, ϕ is considered redundant. Algorithm 1.1 updates the set of global minimal DCs when a new DC is propagated to the global schema. To that end, we initialize the set of DCs Σ_c that covers the same attributes c as the new global DC φ does (Line 1–2). For all the minimal DCs in Σ_c, we check whether the predicate sets composing the new global DC φ is a subset of any existing minimal DCs ϕ. If $\varphi.Pred \subset \phi.Pred$, we replace the ϕ with the new minimal DC φ. Otherwise, the new global DC is recognized as a valid DC but not labelled as minimal one (Line 3–9).

Algorithm 1.1: Minimal DC management.

```
   Input  : Set of all GDCs Σ, new GDC φ
   Output: Updated set of all GDCs Σ_i
   /* If the new GDC is a valid DC and minimal, replace the previous
      GDC containing the same predicates.                          */
 1 c ← φ.atts
 2 Σ_c ← Σ.contain(c)
   // check minimal GDCs from the impacted features.
 3 for φ ∈ Σ_c.minimal do
 4 |   if φ.Pred ⊂ φ.Pred then
 5 |   |   Σ_c.minimal ← Σ_c.minimal − φ
 6 |   |   Σ_c.minimal ← Σ_c.minimal + φ
 7 |   else
 8 |   |   Σ ← Σ + φ
 9 |   end
10 end
11 return Σ
```

Example 3. Consider the following two DCs c_1, c_2 from the running example: $c_1 : \forall t_\alpha, t_\beta \in w_3, \neg(t_\alpha.Code = t_\beta.Code \wedge t_\alpha.Date = t_\beta.Date \wedge t_\alpha.PL = t_\beta.PL)$, and $c_2 : \forall t_\alpha, t_\beta \in w_3, \neg(t_\alpha.Code = t_\beta.Code \wedge t_\alpha.Date = t_\beta.Date)$. c_1 indicates that the combination of company *Symbol*, *Date* and *Low Price* can identify a stock quote. c_2 expresses the same constraint omitting the *Low Price*. Since $c_2.Preds \subset c_1.Preds$, the minimal DC will be updated to c_2.

DC Clash Management. Two global DCs clash if they refer to common features and contradict each other. Clashes happen when there is no instance that may satisfy both DCs. Given the definition of denial constraint, clashing DCs

must be single-predicated over the same attribute and the logical conjunction of their predicates must be empty on the set of available instances. Algorithm 1.2 details how we detect clashes between two global DCs, which considers clashing and partially-clashing DCs.

Definition 4 (Clashing DCs). *Given two DCs c_1 c_2, they are clashing in T, if there does not exist any pair of tuples $\langle t_x, t_y \rangle \in T$, that can satisfy both DCs at the same time.*

Multi-predicated DCs can only partially clash. The partial clash would happen when there are contradictory tuple pairs in the sets of all satisfying tuple pairs from the two DCs due to conflictive predicates. Such predicates must hold on the same attribute and their logical conjunction must be empty.

Definition 5 (Partially clashing DCs). *Given two DCs c_i, c_j, they are partially clashing in T, if the set of all tuple pairs satisfying c_i contains the set of tuple pairs violating c_j.*

Algorithm 1.2: DC clash management

Input : Two global DCs φ_i, φ_j
Output: Updated set of all GDCs Σ_i

$c_i \leftarrow \varphi_i.att$, $c_j \leftarrow \varphi_j.att$
// check if the two GDCs are bounded to the same features
if $c_i = c_j$ **then**
  // check if the predicate sets contains contradictory pairs
  for $p_i \in \varphi_i$, $p_j \in \varphi_j$ **do**
    /* if the pair is contradictory, then a clash is found; T(p_i) is the set of tuples satisfying predicate p_i */
    if $T(p_i) \in T(\overline{p_j})$ *or* $T(p_j) \in T(\overline{p_i})$ **then**
      // compare interestingness scores and choose the highest
      if $\varphi_i.score > \varphi_j.score$ **then**
        $\Sigma_i \leftarrow \{\varphi_i\}$
      else
        if $\varphi_j.score \geq \varphi_i.score$ **then**
          $\Sigma_i \leftarrow \{\varphi_j\}$
        end
      end
    else
      $\Sigma_i \leftarrow \{\varphi_i, \varphi_j\}$
    end
  end
else
  $\Sigma_i \leftarrow \{\varphi_i, \varphi_j\}$
end
return Σ_i

5 Global Query Rewriting with Global DCs

Local DCs propagated to the target and accepted as global DCs may not hold in some sources. For example, consider Fig. 1. w_1, w_2 and w_3 contain attributes (e.g., *symb*, *s*, *code*) referring to the same global feature (*symbol*). Suppose a DC φ from w_1 on *symb*, which is then propagated to the global graph as DC ϕ on *symbol*. When querying the global graph, we must guarantee DC ϕ is guaranteed when performing query answering. This means we need to propagate ϕ to those sources where it originally did not hold. Algorithm 1.3 presents the method to construct DCs in the source graphs from a global DC. For a given global DC φ expressed in the graph, we denote $att(\varphi)$ and $op(\varphi)$ to distinguish the attributes and operators in φ. Following the rewriting algorithm in [16], we map the DC-linked features $feat(att(\vartheta))$ to the attributes in the wrappers. Given the relations from the global DCs E_ϑ, we form the DCs Σ_S in the source graph (the same attribute set can exist in various wrappers, where we form the DC for each wrapper). $dc(A_\theta, E_\vartheta, op(\vartheta))$ denotes the function to form valid DCs given the components.

Algorithm 1.3: Reconstruct a global DC

Input : Global DC ϑ
Output: Set of DCs in the source graph Σ_S
// Get nodes and edges from the global DC
$\langle V_\vartheta, E_\vartheta \rangle \leftarrow \vartheta$
// Map features to source attributes
$A_{dc} \leftarrow map(feat(att\vartheta))$
// Get the wrappers for the attributes
$W \leftarrow wrap(A_{dc})$
// Reconstruct DCs in source graph for each wrapper based on the edges and mapped attributes
for $w \in W$ **do**
  for $a_{dc} \in A_{dc}$ **do**
    if $a_{dc} \in att(w)$ **then**
      $A_M \cup = a_{dc}$
    end
    $\Sigma_S \cup = dc(A_M, E_{dc}, op(\vartheta))$
  end
end
return Σ_S

5.1 Query with DCs

In [16], a query rewriting algorithm is presented for query answering over the global graph in terms of queries over the wrappers. Here, we extend the rewriting algorithm to enable the enforcement of global DCs within a global query. When rewriting a query Q, the method produces sets of rewritings for each concept in

the query to further create a union of conjunctive queries. Then, here, we define $gdc(Q)$ to retrieve the global DCs covered by Q. Σ_G denotes the resulted set of $gdc(\varphi)$. V_Σ and E_Σ denote the composition of Σ, which are vertices and edges respectively. For each global DC in Σ_G, we form the set of DCs for each wrapper and generate the result set of DCs without any duplication. Algorithm 1.4 demonstrates the full method to derive the local DCs from a global query.

Algorithm 1.4: Reconstruct global DCs from a global query to the source graph.

Input : Global query Q
Output: Set of DCs in the source graph Σ_S
// Get the set of all global DCs covered by Q
$\Sigma_G \leftarrow gdc(Q)$
for $\vartheta \in \Sigma_G$ **do**
  $\Sigma_S \cup = ReconstructGDC(\vartheta)$
end
return Σ_S

This way, when querying a global graph with DCs, we apply the constraint rules to all wrappers containing the restricted attributes. For example, consider an integration graph with feature Gender G and FirstName FN, below shows the SQL-like query to retrieve all the records with the global DCs $gdc(G)$: $\forall t_\alpha, t_\beta \in G, \neg(t_\alpha.FN = t_\beta.FN \wedge t_\alpha.G \neq t_\beta.G)$:

```
SELECT G.Gender, G.FirstName
FROM global_graph G WHERE gdc(G)
```

Consider the same integration graph with two wrappers w_1 and w_2 containing the mapped attributes G and FN. The LAV mapping will apply the $gdc(G)$ to both wrappers and join the results, as shown below.

```
SELECT w1.GD, w1.FN FROM wrapper_1 w1
UNION
SELECT w2.GD, w2.FN FROM wrapper_2 w2
WHERE gdc(G) /* Apply the gdc as the filter for all wrappers */
```

6 Validation

We implemented a case study based on financial data. We modeled the SEC Edgar database which releases the XBRL (eXtensible Business Reporting Language) taxonomies every year, given the annual update of U.S. GAAP (Generally Accepted Accounting Principles). Thus, we build wrappers for Edgar based on each release year (i.e., one wrapper per year). For each wrapper, DCFINDER produces sets of DCs. Given the high frequency of the Edgar schema version updating, the wrappers of Edgar share a large portion of

attributes, which leads to the overlap of DCs for different wrappers. Following the global DC management algorithms, we are able to conciliate the different versions of local DCs and prune the total number of global DCs to a meaningful level. Following the minimal DC maintenance strategy, we are able to avoid redundant DCs at the global level and derive meaningful local DCs such as $dc_1 : \neg(t_0.Assets \leq t_1.Assets \wedge t_0.Equity \geq t_1.Equity)$ and $dc_2 : \neg(t_0.Liabilities \geq t_1.Liabilities \wedge t_0.Assets \leq t_1.Assets)$. Note dc_1 and dc_2 are valid minimal DCs since there does not exist valid DCs that can be derived from their subsets. Interestingly, the DC $dc_3 : \neg(t_0.Assets \leq t_1.Assets \wedge t_0.Equity \geq t_1.Equity \wedge t_0.Liabilities \geq t_1.Liabilities)$ implies the rule of financial reporting (assets equals to the sum of equities and liabilities).

We also apply the clash management algorithm to resolve contradicting DCs. For instance, the DC $dc_4 : t_0.PeriodEnding \neq t_1.PeriodEnding$ is generated due to the standardized release date in 2012. Then, $dc_5 : \neg(t_0.PeriodEnding = t_1.PeriodEnding)$ states the possible variance of release date for different companies in 2016. This is due to the U.S. GAAP update to allow flexibility for companies to define their own financial year. In this scenario, dc_4 and dc_5 are partially clashing because of the complementary predicates of *PeriodEnding* feature. We first try to resolve this conflict by the interestingness scores, but both shown high statistical significance in each wrapper. Then, we applied each DC to the global graph and detect the #violations from all sources. dc_5 was valid in all wrappers, while dc_4 generated multiple violations. Thus, we rejected dc_4 and propagated dc_5 to the global schema. Overall, with the global DC management algorithms, we were able to prune the total number of global DCs to 21, which we manually validated as valuable business rules.

7 Conclusions and Future Work

We addressed the DQ problem for virtual data integration systems. The novelty of our approach lies on the consideration of a (potentially conflicting) set of data sources, as opposite to the traditional methods on data quality management that consider a single database instance. To that end, we first elicit DC rules from the data sources and express them in the integration graph to then define DC management methods that enable the global conciliation of DCs. We modified our query rewriting algorithm to guarantee global DCs, while query answering, in all data sources regardless of where they were generated. As future work, we aim to fully automate the process of generating, propagating and enforcing DCs. This requires the extension of traditional knowledge graph bootstrapping methods for quality rules. Another interesting line of work is that of automatically generating data flow operators such that they repair the data errors identified in some sources (instead of fixing them at query time).

Acknowledgements. This work was partly supported by the DOGO4ML project, funded by the Spanish Ministerio de Ciencia e Innovación under project PID2020-117191RB-I00. Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union - NextGenerationEU, under project FJC2020-045809-I.

References

1. Abedjan, Z., et al.: Detecting data errors: where are we and what needs to be done? Proc. VLDB Endow. **9**(12), 993–1004 (2016)
2. Batini, C., Rula, A.: From data quality to big data quality: a data integration scenario. In: SEBD, Volume 2994 of CEUR Workshop Proceedings, pp. 36–47. CEUR-WS.org (2021)
3. Batini, C., Rula, A., Scannapieco, M., Viscusi, G.: From data quality to big data quality. J. Database Manag. **26**(1), 60–82 (2015)
4. Bleifuß, T., Kruse, S., Naumann, F.: Efficient denial constraint discovery with hydra. Proc. VLDB Endow. **11**(3), 311–323 (2017)
5. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Inf. Comput. **197**(1–2), 90–121 (2005)
6. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. Proc. VLDB Endow. **6**(13), 1498–1509 (2013)
7. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: Cleaning data with Llunatic. VLDB J. **29**(4), 867–892 (2020). https://doi.org/10.1007/s00778-019-00586-5
8. Halevy, A.Y.: Answering queries using views: a survey. VLDB J. **10**(4), 270–294 (2001). https://doi.org/10.1007/s007780100054
9. Haug, A., Zachariassen, F., Van Liempd, D.: The costs of poor data quality. J. Ind. Eng. Manag. (JIEM) **4**(2), 168–193 (2011)
10. Heidari, A., McGrath, J., Ilyas, I.F., Rekatsinas, T.: HoloDetect: few-shot learning for error detection. In: SIGMOD Conference, pp. 829–846. ACM (2019)
11. Jarke, M., Jeusfeld, M.A., Quix, C., Vassiliadis, P.: Architecture and quality in data warehouses. In: Pernici, B., Thanos, C. (eds.) CAiSE 1998. LNCS, vol. 1413, pp. 93–113. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054221
12. Kolaitis, P.G.: Schema mappings, data exchange, and metadata management. In: PODS, pp. 61–75. ACM (2005)
13. Laranjeiro, N., Soydemir, S.N., Bernardino, J.: A survey on data quality: classifying poor data. In: PRDC, pp. 179–188. IEEE Computer Society (2015)
14. Livshits, E., Heidari, A., Ilyas, I.F., Kimelfeld, B.: Approximate denial constraints. Proc. VLDB Endow. **13**(10), 1682–1695 (2020)
15. Loshin, D.: Evaluating the business impacts of poor data quality. Inf. Qual. J. (2011)
16. Nadal, S., Abello, A., Romero, O., Vansummeren, S., Vassiliadis, P.: Graph-driven federated data management. IEEE Trans. Knowl. Data Eng. (2021)
17. Pena, E.H.M., de Almeida, E.C., Naumann, F.: Discovery of approximate (and exact) denial constraints. Proc. VLDB Endow. **13**(3), 266–278 (2019)
18. Rekatsinas, T., Chu, X., Ilyas, I.F., Ré, C.: HoloClean: holistic data repairs with probabilistic inference. Proc. VLDB Endow. **10**(11), 1190–1201 (2017)
19. Sadiq, S.W., Papotti, P.: Big data quality - whose problem is it? In: ICDE, pp. 1446–1447. IEEE Computer Society (2016)
20. Schirmer, P., et al.: DynFD: functional dependency discovery in dynamic datasets. In: EDBT, pp. 253–264. OpenProceedings.org (2019)
21. Xiao, G., et al.: Ontology-based data access: a survey. In: IJCAI, pp. 5511–5519. ijcai.org (2018)