Validating Data from Semantic Web Providers

Jacques Chabin, Mírian Halfeld Ferrari Alves, Béatrice Markhoff, Thanh Binh Nguyen

▶ **To cite this version:**

Jacques Chabin, Mírian Halfeld Ferrari Alves, Béatrice Markhoff, Thanh Binh Nguyen. Validating Data from Semantic Web Providers. 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2018), 2018, Krems, Australia. pp.682–695, 10.1007/978-3-319-73117-9_48 . hal-01836993

HAL Id: hal-01836993
https://hal.science/hal-01836993

Submitted on 18 Feb 2022

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

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

Validating Data from Semantic Web Providers

Jacques Chabin[1], Mirian Halfeld-Ferrari[1], Béatrice Markhoff[2], Thanh Binh Nguyen[1⋆]

[1] Université d'Orléans, INSA CVL, LIFO, France
[2] Université Francois Rabelais de Tours, LI, France

Abstract. As the Linked Open Data and the number of semantic web data providers hugely increase, so does the critical importance of the following question: how to get usable results, in particular for data mining and data analysis tasks? We propose a query framework equiped with integrity constraints that the user wants to be verified on the results coming from semantic web data providers. We precise the syntax and semantics of those user quality constraints. We give algorithms for their dynamic verification during the query computation, we evaluate their performance with experimental results, and discuss related works.

Keywords: Semantic Web Data, User Quality Constraint, Query Rewriting

1 Introduction

There exist now very large knowledge bases on the web of Linked Open Data, as DBpedia, Yago or BabelNet. The largest ones contain millions of entities and billions of facts about them (attribute values and relationships with other entities) [20]. Applications are needed to help humans exploring this huge knowledge network, performing data analysis and data mining tasks. Promising recent proposals are currently experimented on only one semantic web data source [7, 10], and these processes can be expected to be even more helpful when they will deal with several linked open data sets. One crucial point for such applications, and in particular for data mining algorithms, is that the data collection and pre-processing steps have to be *safe and sound*.

In order to help semantic web data mining tool designers for performing the data collection and pre-processing steps, we propose a semantic web data validator [4]. The idea is to extend a query environment over semantic graph

⋆ This work is supported by Girafon Project, funded by Region Centre Val de Loire.

databases with a mechanism for filtering answers according to a user customized context. In this paper, we use the term "user" for the query-writer. The *user context* is composed of *(i)* the *view* she/he has defined on the needed semantic web data and *(ii)* a set of *personalization tools*, such as integrity constraints, confidence degrees, etc. In this paper, we only deal with integrity constraints, that we call *user quality constraints*, leaving the other kinds of personalization tools for other discussions (see [4, 6]).

User quality constraints are restrictions imposed on query results. Both the constraints and the queries are expressed in terms of the user's view of data. The constraint verification is triggered by a query and consists in filtering its answers. In this way, there may be some inconsistencies within sources, but the answers given to the user are filtered to ensure their consistency w.r.t. her/his constraints. The following example illustrates the kind of constraints a user can define and what are their effects on query answers.

Example 1. Let us consider a query $q_1(X) \leftarrow teacherOf(X, Y)$ in a context with two constraints:

$$c_p : teacherOf(X, Y) \rightarrow professor(X).$$
$$c_n : teacherOf(X, Y), takesCourse(X, Y) \rightarrow \bot.$$

The first constraint is to verify that each teacher of a course is a professor. The second constraint disallows to accept, in the query answers, a person who teaches a course while she/he is enrolled in the same course. Suppose the database is as in Figure 1. Although $\{Bob, Tom, Alice, Ann\}$ are answers to query q_1, $\{Alice, Ann\}$ are invalid w.r.t. c_p, while $\{Tom\}$ causes a violation of constraint c_n. Only $\{Bob\}$ satisfies all constraints. Thus, the answer to q_1 in the user context consisting of $\{c_p, c_n\}$ is $\{Bob\}$.

teacherOf(Bob, DB)	professor(Bob)
teacherOf(Bob, Java)	professor(Tom)
teacherOf(Tom, Java)	takesCourse(Tom, Java)
teacherOf(Alice, DB)	takesCourse(Bob, Java)
teacherOf(Ann, DB)	reasearchesIn(Ann, DB)
	reasearchesIn(Bob, DB)

Fig. 1: Database instance

From Example 1, it can be noticed that when a constraint is triggered by instantiated atoms in the query's body, it requires auxiliary appropriate queries to verify its side effect. For instance, the fact $teacherOf(Bob, DB)$ triggers both c_p and c_n, thus queries like $q_{11}() \leftarrow professor(Bob)$ and $q_{12}() \leftarrow takesCourse(Bob, DB)$ are produced to verify whether Bob is a professor and whether Bob is registered in the Database course. It is easy to see that, when dealing with a big amount of data, the impact of such auxiliary queries may be important. Even though most of them are simple queries, they can lead to a system overloading. A solution to avoid such issue is *to integrate as much as possible the constraints into the query*, in such a way that the answers will not only satisfy the initial query, but they will also respect all integrated constraints.

This paper is organized as follows: in Section 2, we present the overall query framework with user context, and precise the syntax and semantics of user quality constraints. In Section 3 we give algorithms for their dynamic verification during the query computation. In Section 4 we evaluate their performance with experimental results, and discuss related works.

2 A Querying Framework with Constraints

2.1 Querying environment

Our query processing framework is depicted in Figure 2. It comprises two distinct parts which communicate: *Data validation*, responsible for checking constraints satisfaction, and *Data providers* for computing answers to the queries issued from the data validation part. The later may actually integrate several end-data-providers, or it may connect only one provider. For ensuring that the final answers to the user's queries satisfy all user constraints, a dialogue between the two parts is established, for getting intermediate results and sending subsidiary queries.

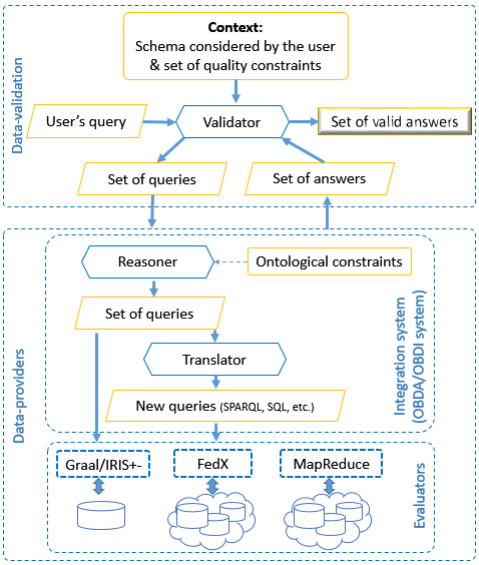

Fig. 2: Query system overview

The user defines her/his *context* by setting her/his view on the queried sources, a set of datalog predicates as explained in next section, and a set of quality constraints involving these predicates. The user's *query* involves these predicates, so quality constraints can be used as rewriting-rules to reformulate each query q, resulting in a union of conjunctive queries whose answers, contained in q's answers, are valid w.r.t. the user quality constraints.

Afterwards, these conjunctive queries are sent to the *Data providers* part, which evaluates them against data stored on sources. The query evaluation process is transparent to the validation step, in particular, answers that are entailed are treated in the same way as those that actually exist in sources. We respect the potential ontological dimension of semantic web sources, while interpreting the *user constraints* using the closed-world assumption. Indeed, as it deals with semantic data, the evaluating process performed by the *Data providers* part relies on the open-world assumption, where ontological constraints are used to deduce new information. Ontological constraints are used as rewriting-rules to reformulate a query into a set of new conjunctive queries, for taking into account integration information (OBDA/OBDI Systems [18, 3]), or for dealing with incomplete information issues [3, 11, 15, 12]. But such rewritings are performed by the *Data providers* part, independently from the *Data validation* part.

As our system may be deployed with various data management systems, a module will translate datalog+- queries [5] (used by Graal[3]) into SPARQL for FedX [19], and HIVE-SQL for MapReduce (as proposed in [4]).

[3] `https://graphik-team.github.io/graal/`

2.2 Constraints

Our constraints are expressed in a first-order logic formalism. We consider an alphabet made up of three disjoint sets const, var and pred, of constants, variables and predicate names, respectively. A term t is either a variable or a constant and an *atom* is a formula $p(x_1, x_2, \ldots, x_n)$ where p is a predicate name and each x_i is a term. A *substitution* is a total mapping $\sigma : \mathsf{var} \to T$ from variables to terms. A *homomorphism* from a set of atoms A_1 to a set of atoms A_2, all over the same schema R, is a substitution h from the set of terms of A_1 to the set of terms of A_2 such that: (i) if t is a constant, then $h(t) = t$, and (ii) if $r(t_1, ..., t_n)$ is in A_1, then $h(r(t_1, ..., t_n)) = r(h(t_1), ..., h(t_n))$ is in A_2. The notion of homomorphism naturally extends to conjunctions of atoms. Two atoms A_1 and A_2 are *unifiable* if and only if there exists a substitution σ s.t. $\sigma(A_1) = \sigma(A_2)$. Furthermore, if two atoms A_1 and A_2 are unifiable then there exists a *most general unifier* (mgu) θ s.t. $\theta(A_1) = \theta(A_2)$.

A *conjunctive query* (CQ) q of arity n over a given schema is a logical rule of the form $q(\mathbf{X}) \leftarrow \phi(\mathbf{X},\mathbf{Y})$, where $\phi(\mathbf{X},\mathbf{Y})$ is a conjunction of atoms over the schema, q is a n-ary predicate and $\mathbf{X}, \mathbf{Y}$ are sequences of terms. Given a logical rule r, we denote by $body(r)$ the rule's antecedent by $head(r)$ its consequent.

Our *user quality constraints* [4] are also logical rules. We define a user *context* as a set $\mathcal{C}$ of constraints, composed of three subsets, as follows:

Positive constraints ($\mathcal{C}_P$): Each positive constraint has the form

$$\forall \mathbf{X}, \mathbf{Y} \quad L_1(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} \quad L_2(\mathbf{X}, \mathbf{Z})$$

$L_1(\mathbf{X}, \mathbf{Y})$ and $L_2(\mathbf{X}, \mathbf{Z})$ are atoms and $\mathbf{Z}$ are existential variables.

Negative constraints ($\mathcal{C}_N$): each negative constraint is a rule having the form

$$\forall \mathbf{X} \ \phi(\mathbf{X}) \rightarrow \bot$$

where $\phi(\mathbf{X})$ is an atom $L_1(\mathbf{X})$ or a conjunction of two atoms $L_1(\mathbf{X}_1), L_2(\mathbf{X}_2)$, which have a non-empty intersection between the terms in $\mathbf{X}_1$ and $\mathbf{X}_2$. We refer to $\mathcal{C}_{N1}$ and $\mathcal{C}_{N2}$ as sets of negative constraints having only one atom and two atoms, respectively. Negative constraint is a special case of denial constraint with at most two occurrences of database literals.

Equality-generating dependency constraints without nulls ($\mathcal{C}_K$): each EGD is a rule having the general form

$$\forall X_1, X_2, \mathbf{Y}, \mathbf{Z}_1, \mathbf{Z}_2 \quad L_1(\mathbf{Y}, X_1, \mathbf{Z}_1), L_2(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2.$$

where $\mathbf{Y}$ is a sequence of common terms of L_1 and L_2 that has at least one element. Notice that EGD include functional dependency (and thus, key constraints) having the form $L_1(\mathbf{Y}, X_1, \mathbf{Z}_1), L_1(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2$.

In the rest of this paper, for simplicity, we will omit the quantifiers. We say that a constraint c is triggered by an atom A when there is a homomorphism h from $body(c)$ to A. Positive constraints are a special case of linear tuple generating dependency (TGD [2]) which contain only one atom in the head. They cover both inclusion dependency class and join dependency class [2]. When $\mathbf{Z}$ is not empty, the homomorphism h is extended to h' such that, for each existential variable $z \in \mathbf{Z}$, $h'(z)$ is a new fresh variable. It is well-known that facts from a database instance may trigger such constraints, and the *chase* procedure ([17]) is the standard process for the generation of new facts from a database instance

and a set of dependencies (TGD or EGD) [2]. It can also be used to decide containment of conjunctive queries in the presence of constraints ([14]). In this paper, we consider that the set of positive constraints is weakly acyclic, which guarantees the decidability of query containment [9]. In Example 1, c_p and c_n are illustrations for the definitions of positive and negative constraints above. An example of EGD constraints can be as follows:

$$c_k : worksFor(X, Y, Z), headOf(X, W) \rightarrow Z = W.$$

It states that if a person X is the head of W and if she is working for organization Y in department Z then W must be the department Z.

3 Validating Semantic Web Query Outputs

Given a user's query q, the validation of its result on the basis of user's quality constraints in $\mathcal{C}$ can be performed in two ways: by rewriting q to take into account the constraints in $\mathcal{C}$, or by the evaluation of auxiliary queries, composed on the basis of initial results obtained for q. Even if the choice between these two processes depends on the query evaluation power of data providers, it is important to study their *costs and benefits* in a common framework. To do so, in this paper, we use Graal [1] as conjunctive query evaluator for both techniques. More precisely, we focus on testing and comparing the performance of our validation approach in the following two scenarios: (1) the rewriting of q on the basis of constraints in $\mathcal{C}$, followed by the rewritten-query evaluation, and (2) what we call the naive solution, *i.e.* evaluate q, then build and evaluate multiple auxiliary queries on the basis of q's answers. This section summarizes these two scenarios, and in Section 4 we analyse in details their respective validation performance.

3.1 Query Rewriting with Constraints

Given a CQ q and a set of constraints $\mathcal{C}$, let us consider examples to illustrate the situations our query rewriting algorithm tackles with.

Example 2. Query q_1 below looks for professors who were born in a foreign country. Constraints establish a user's context imposing a professor to be associated with a course (c_{p_a}) offered by a department (c_{p_b}). Moreover, the user is interested only in professors working in the public sector (c_{p_c}).

$$q_1(X_1) \leftarrow professor(X_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$
$$c_{p_a} : professor(X) \rightarrow teacherOf(X, Y).$$
$$c_{p_b} : teacherOf(X, Y) \rightarrow offeredCourseIn(Y, Z).$$
$$c_{p_c} : professor(X) \rightarrow employeeGov(X).$$

In this context, we see $body(q_1)$ as a set of atoms capable of triggering constraints and producing new atoms that should be added to the query's body. This operation corresponds to a chase computation ([17]), which starts with the atoms in $body(q_1)$. Special attention is required in the use of variable renaming. The new rewritten query, that the system should send to data providers, is:

$q'_1(X_1) \leftarrow professor(X_1), teacherOf(X_1, Y_1), offeredCourseIn(Y_1, Y_2),$
$employeeGov(X_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$ □

When the query, or the constraints, contain *constants*, the above rewriting technique should be revised, as illustrated by the following example.

Example 3. Consider query q_2, and constraint c_{p2} imposing restrictions on database teachers - they should do research in the database domain:

$$q_2(X) \leftarrow teacherOf(X, Y).$$
$$c_{p2} : teacherOf(Z, DB) \rightarrow researchesIn(Z, DB).$$

Notice that no restriction is imposed on teachers in other domains. Here we cannot apply the chase as in Example 2, because a query $q'_2(X) \leftarrow teacherOf(X, DB)$, $researchesIn(Z, DB)$ would ignore the teachers of all other domains. In this case, our proposal is to replace q_2 by the union of the two following queries:

$$q_{2.1}(X) \leftarrow teacherOf(X, Y), \neg teacherOf(X, DB).$$
$$q_{2.2}(X) \leftarrow teacherOf(X, Y), teacherOf(X, DB), researchesIn(X, DB). \quad \square$$

Algorithm 1 summarizes our rewriting solution. In this algorithm the input is composed of a conjunctive query, and positive and negative constraints. However, negative constraints in $\mathcal{C}_{N2}$, *i.e.*, those having the form $L_1(\mathbf{X}_1), L_2(\mathbf{X}_2) \rightarrow \bot$ are transformed into two equivalent formulas: $L_1(\mathbf{X}_1) \rightarrow \neg L_2(\mathbf{X}_2)$ and $L_2(\mathbf{X}_2) \rightarrow \neg L_1(\mathbf{X}_1)$. In this way, negative constraints receive a similar treatment as positive constraints. For instance, from Example 1, the constraint c_n can be written as $c_{n_1} : teacherOf(X, Y) \rightarrow \neg takesCourse(X, Y)$ and $c_{n_2} : takesCourse(X, Y) \rightarrow \neg teacherOf(X, Y)$. Query q_1 is then rewritten as $q_1(X) \leftarrow teacherOf(X, Y)$, $\neg takesCourse(X, Y)$.

In Algorithm 1, Function RewriteWithConstraints is the main program, which ensures that each query is rewritten by taking into account all positive and negative constraints in $\mathcal{C}$. It calls Function Integrate, the kernel of our rewriting method, which computes the new queries that replace the given query q, by integrating in q the restrictions imposed by the given constraint c.

The instantiation of constraints *w.r.t.* the atoms L in q's body is done on line 15 by using a *mgu* θ, and c' is the resulting constraint, instantiated with constants in q. Then, on line 18, we consider the cases where c' can be triggered by L. This happens when θ is a variable renaming, or, when it replaces variable in c by constants in L (afterwards, there may still exists a homomorphism ν from $body(c')$ to L). For instance, consider query q_3 and constraint c_3 as follows:

$$q_3(X) \leftarrow professor(Bob), teacherOf(Bob, X)$$
$$c_3 : professor(X) \rightarrow inDept(X, Y)$$

With $L = professor(Bob)$ and $\theta = \{X/Bob\}$, we obtain $c'_3 : professor(Bob) \rightarrow inDept(Bob, Y_1)$, where Y_1 is a new variable resulting from variable renaming performed by *createRule* (line 19). Similarly, in Example 2, for $L = professor(X_1)$ and $\theta = \{X/X_1\}$ we obtain $c'_{p_a} : professor(X_1) \rightarrow teacherOf(X_1, Y_1)$.

When the homomorphism ν exists, the query's body is completed with the head of c' (line 19). The loop on line 5 ensures that the query's body will be completed with all the atoms obtained by triggered constraints. Notice that the idea here is to use a chase procedure applied to rules that respect some syntactic restrictions. Indeed, our current implementation deals with a set of weakly acyclic TGD ([9]). Roughly, a set of TGD is acyclic if it does not allow

for cascading of labelled null creation during the chase. Example 2 illustrates a rewritten query obtained by following the above steps.

Algorithm 1: *RewriteWithConstraint*

Input : A conjunctive query q and a set of constraints $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_N$
Output: A set of queries Q s.t. each $q' \in Q$ does not contain explicit contradictions and the answers of q' respect $\mathcal{C}_P \cup \mathcal{C}_N$. Notice that we can get $Q = \emptyset$ as output.

1 **Function** `RewriteWithConstraint(`q, $\mathcal{C}$`):`
2 $Q = \{q\}$;
3 **repeat**
4 $hasChanged$ = false;
5 **foreach** $c \in \mathcal{C}$ **do**
6 **foreach** $q \in Q$ **do**
7 $Q' = \mathsf{Integrate}(q, c)$;
8 **if** *(*$|Q'| = 1$ *and* $q' \in Q'$ *is more restricted than* q*) or (*$|Q'| > 1$*)* **then**
9 $Q = Q \backslash \{q\} \cup Q'$;
10 $hasChanged$ = true;
11 **until** *not* $hasChanged$;
12 **return** Q;
13 **Function** `Integrate(`q, c`):`
14 $Q' = \{q\}$;
15 **foreach** $L \in body(q)$ *s.t.* $\exists mgu\ \theta : \theta(L) = \theta(body(c))$ *and not* $tested(L, c)$ **do**
16 $c' = createRule(\theta(head(c)), \theta(body(c)))$;
17 **foreach** $q' \in Q'$ **do**
18 **if** $\exists$ *homomorphism* ν *from* $body(c')$ *to* L **then**
19 $q_1 = createRule(head(q'), body(q') \wedge \nu(head(c')))$;
20 $Q'' = \{q_1\}$;
21 **else**
22 $q_1 = createRule(head(q'), body(q') \wedge \neg\theta(body(c')))$;
23 $q_2 = createRule(head(q'), body(q') \wedge \theta(body(c')) \wedge \theta(head(c')))$;
24 $Q'' = \{q_1, q_2\}$;
25 $Q'' = Simplify\_Verify(Q'')$;
26 **if** *(*$|Q''| = 1$ *and* $q'' \in Q''$ *is more restricted than* q'*) or (*$|Q''| > 1$*)* **then**
27 $Q' = Q' \backslash \{q'\} \cup Q''$;
28 markTested(L,c);
29 //Mark L as already tested *w.r.t.* c, *i.e.* $tested(L, c)$ = true
30 **return** Q';

When the homomorphism ν does not exist, we are dealing with constants that cannot map to variables or with different constants. Let us consider Example 3, after executing line 15 of Algorithm 1 with $L = teacherOf(X, Y)$. We have $c'_{p2} : teacherOf(Z, DB) \rightarrow researchesIn(Z, DB)$ (no changes *w.r.t.* c_{p2}). No homomorphism from $body(c')$ to L is possible. Line 22 deals with results that are *not* concerned by the constraint. In this case, the query body is completed with the negation of the constraint's body. Thus, in our Example 3, $q_{2.1}$ selects people who do not teach DB. With the database instance of Figure 1, the answer for $q_{2.1}$ is Tom. Then, on line 23, we deal with results *concerned* by the constraint. In

Example 3, $q_{2.2}$ selects two kinds of people: (i) those who are database researchers and *only* teach DB and (ii) those who teach and do research in the database domain but also teach other subjects. Continuing with our example, the desired answers for q_2 are Bob, Ann and Tom. With our algorithm, Bob and Ann are not answers for $q_{2.1}$, but they are answers to $q_{2.2}$. The result of q_2 is the union of the answers for $q_{2.1}$ and $q_{2.2}$.

Rewritten queries, put in the set Q'', are sent to function *Simplify_Verify* (line 25) that, for each query, removes redundant atoms. This function also ensures that Q'' does not contain queries with explicit contradiction. In other words, the function checks whether: (i) there is no two atoms having the form $L(\mathbf{X})$ and $\neg L(\mathbf{X})$ in the query body and (ii) atoms in the query body cannot trigger a negative constraint.

We use query containment (see, for instance [2] for a revision on the subject) to decide whether a rewritten query replaces a given one. On line 27, notice that at each iteration step, the set Q' contains the most restricted rewritten queries obtained so far. Each iteration step considers an atom in the query body and one single constraint. The output of the Integrate function is the set Q', which contains the most restricted rewritten queries obtained for one query *w.r.t.* one constraint c. Then, on line 9, the replacement of the original query q is considered. If only one query q' results from Integrate, q is replaced by q' only when q' is more restricted than q. Otherwise, when more than one rewritten queries result from Integrate, q is replaced by them.

The query obtained after only chasing the original query *w.r.t.* positive constraints corresponds to the universal plan of [8]. However, when dealing with negative constraints, even when Integrate performs only lines 17-20 to rewrite a given query, the rewritten query may contain negative atoms.

3.2 Building Auxiliary Queries

Given a query q, to ensure its answer consistency *w.r.t.* user's quality constraints, instead of dealing with query rewriting, one can consider the generation of subqueries from the initial answers obtained from q. Let h_t be the homomorphism used to produce tuple t as an answer to the query q. We want to check whether t is valid *w.r.t.* constraints. Tuple t is considered valid only when *all* constraints triggered during the validation process are satisfied.

Let $L(\mathbf{X})$ be an atom of $body(q)$. The instantiated atom $h_t(L(\mathbf{X}))$ may trigger a constraint c. According to the type of c, an auxiliary query q' is created:

- For $c \in \mathcal{C}_P$ the auxiliary boolean query is $q'() \leftarrow h_t(L_0(\mathbf{X}_0))$ where $L_0(\mathbf{X}_0) = head(c)$. The resulting tuple t is valid *w.r.t.* c if the answer of q' is positive. Notice however that each fact f resulting from the instantiation of $h_t(L_0(\mathbf{X}_0))$ on the database may trigger another constraint. The validation process continues until no constraint is triggered and corresponds to a chase procedure, establishing a dialogue between the validator and the providers.
- For $c \in \mathcal{C}_N$ and assuming that c has the form $L(\mathbf{X}), L_0(\mathbf{X}_0) \rightarrow \bot$ the auxiliary boolean query is $q'() \leftarrow h_t(L_0(\mathbf{X}_0))$. Tuple t is valid *w.r.t.* c if the

answer of q' is negative. Clearly, if c has the form $L(\mathbf{X}) \rightarrow \bot$, the verification is straightforward.
- For $c \in \mathcal{C}_K$, assuming that c has form $L(\mathbf{Y}, X_1, \mathbf{Z}_1), L_0(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2$ and $\mathbf{X} = \mathbf{Y} \cup X_1 \cup \mathbf{Z}_1$, the auxiliary query is $q'(X_2) \leftarrow h_t(L_0(\mathbf{Y}, X_2, \mathbf{Z}_2))$. Tuple t is valid *w.r.t.* c if the answer set is a singleton containing the tuple value $h_t(X_1)$.

3.3 Complete Validation

Finally, Algorithm 2 is responsible for validating the result of a query q *w.r.t.* a set of constraints $\mathcal{C}$. Algorithm 1 rewrites the query only *w.r.t.* positive and negative constraints. Then it must be completed by the generation of auxiliary queries, from the answers of the rewritten queries, at least for dealing with EGD constraints in $\mathcal{C}$. On line 2 of Algorithm 2, Function *RewriteWithConstraint* returns a set Q of rewritten queries. Afterwards, Function *Eval* evaluates all queries in Q (line 3), and answers are stored in the set Solutions. On line 5, $\mathcal{C}_{check}$ is the set of the constraints which are not addressed by Algorithm 1. Function *Valid* verifies whether an answer *sol* is valid *w.r.t.* $\mathcal{C}_{check}$ by generating corresponding auxiliary queries, as sketched in Section 3.2.

Algorithm 2:

Input : A conjunctive query q and a set of constraints $\mathcal{C}$.
Output: Answers of q respecting $\mathcal{C}$.

```
1 AnsSet = ∅;
2 Q = RewriteWithConstraint(q, C);
3 Solutions = Eval(Q);
4 Cache = CreateCache();
5 C_check = remainingConstraints(C);
6 foreach sol ∈ Solutions where sol = (t, h_t) do
7     if Valid(sol, C_check, Cache) then
8         AnsSet := AnsSet ∪ {t};
9 return AnsSet;
```

4 Experimental Results and Related Works

Our main goal is to compare the overall performance between (*i*) our first scenario, *i.e.* the query rewriting approach performed by Algorithm 2 when only the EGD constraints are not considered by Function *RewriteWithConstraint*, and (*ii*) our second scenario, the naive approach, performed by Algorithm 2 when Function *RewriteWithConstraint* is simply not applied. Both approaches compute the same *valid* answers (whose number is given in column 5 and 6 in Table 1(a) for the given conjunctive query, *i.e.* answers that satisfy the given set of quality constraints. Another important goal of experiments is to analyze features that affect the computation efficiency, such as the size of datasets, the size of queries, the number and type of constraints, etc.

We performed experiments using a HP ZBook laptop equipped with a quad-core Intel i7-4800MQ processors at 2.7GHz and 16Gb of RAM. We developed

	Trig.	Num.Rew.Que.		Max num.	Valid answers	
	cons.	w.opt.	wo.opt.	atoms	1 univ.	5 univ.
Q1	4	1	4	7	523	3331
Q2	1	1	1	2	7861	36682
Q3	2	2	2	5	3599	23749
Q4	0	1	1	2	10735	67702
Q5	6	6	8	14	50	59
Q6	8	2	8	13	6631	36538
Q7	6	2	8	13	21	220

(a) Queries and Rewritten Queries

		1 university		5 universities	
	RewTime	EvalTime	Total	EvalTime	Total
Q1	0.043	0.372	0.415	0.492	0.535
Q2	0.001	0.429	0.430	6.388	6.389
Q3	0.007	0.124	0.131	0.804	0.811
Q4	0	0.111	0.111	0.692	0.692
Q5	0.048	0.702	0.75	0.773	0.821
Q6	0.011	20.522	20.533	122.285	122.296
Q7	0.01	3.193	3.203	162.105	162.115

(b) Rewriting, Evaluation-Verification (s)

Table 1: Rewriting Approach

	1 university					5 universities				
	Eval.	Verif.	Total	Init.ans.	Num.Que.	Eval.	Verif.	Total	Init.ans.	Num.Que.
Q1	0.965	1.172	2.137	1548	2072	1.191	7.14	8.331	10095	13426
Q2	0.153	49.952	50.105	7861	7861	1.038	t/o	t/o	36682	-
Q3	0.041	1.515	1.556	3599	3599	2.59	10.709	13.299	23749	23749
Q4	0.026	0.072	0.098	10735	0	0.166	0.43	0.596	67702	0
Q5	0.227	1.704	1.931	50	200	0.735	1.363	2.098	59	236
Q6	9.205	57.948	67.153	6631	39786	16.108	t/o	t/o	36538	-
Q7	4.772	0.535	5.307	96	159	292.216	0.712	292.928	645	1305

Table 2: Evaluation and Verification in the Naive Approach (s)

Java programs using Graal, a Java toolkit dedicated to knowledge-base querying within the framework of existential rules (*e.g.* Datalog+-). We used the LUBM[4] benchmark, which describes the organizational structure of universities with 43 classes and 32 properties, and provides a generator of synthetic data with varying size. For analyzing the impact of the size of databases on the tested solutions, we created two versions of datasets containing data of 1 and 5 universities, containing 86,165 and 515,064 triples, respectively. These datasets are loaded and managed directly by Graal, which converts them from RDF/XML to Dlgp, its supported data format. Inspired by the 14 test queries of LUBM, we devised 7 queries and 12 constraints written in Dlgp (4 positive, 5 negative, and 3 keys) [5].The queries spread from simple queries with few atoms ($Q1$, $Q2$) to more complex queries ($Q6$, $Q7$), and may contain constants ($Q5$). Some constraints also involve constants ($Cp2$, $Cp3$, $Cp4$). Column 1 in Table 1(a) contains the number of constraints triggered by each query. The second and third columns present the number of rewritten queries either applying the simplification query-containment test (Function $Simplify\_Verify$), or not. Theoretically, a query that involves n constraints can be rewritten into 2^n reformulations in the worst case. Experimental results show that in some cases ($Q1$, $Q6$, $Q7$), Function $Simplify\_Verify$ significantly reduces the number of rewritings. Column 4 shows the maximum number of atoms in rewritten queries, which demonstrates that the more constraints are used in the rewriting procedure, the more complex are the rewritings (number atoms or joins).

[4] Lehigh University: `http://swat.cse.lehigh.edu/projects/lubm/`

[5] Details in the techinical report: `http://www.univ-orleans.fr/lifo/rapports.php?annee=2017`

We now turn our attention to the time of rewriting and complete evaluation-verification, reported in Table 1(b), which contains the following information: (*i*) the time needed for rewriting, indicated in Column $RewTime$; (*ii*) the time needed for evaluating all queries obtained from the rewriting step, shown in column $EvalTime$, for the two tested datasets; (*iii*) the total time for performing these two steps (Column $Total$). Rewritings are very fast and the evaluation time is clearly the major part in the total time, in all cases. Furthermore, the evaluation time is directly proportional to the size of the tested dataset. Moreover, the rewritten-query complexity affects the evaluation time, for instance, $Q6$ and $Q7$ have 13 atoms in their body and their evaluation times on 5 universities are the biggest ones. Interestingly, $Q5$ has 14 atoms and does not need so much time for the evaluation. The reason is that $Q5$ contains a constant, which highly reduces its querying space. In summary, these first experiments demonstrate how the dataset size, the query complexity, the number of involved constraints and the presence of constants in initial and rewritten queries, impact the overall time of the rewriting-and-evaluating approach for processing a query with user-constraints.

Concerning now the experimental results for the naive approach, shown in Table 2, we have, for each dataset: (*i*) the time needed for evaluating the initial query in Column $Eval.$; (*ii*) the time necessary for generating and executing auxiliary queries to verify all answers obtained from the previous evaluation step, in Column $Verif.$; (*iii*) the overall processing time in Column $Total$. (*iv*) the number of answers before constraint verification in Column $Init.Ans.$; and (*v*) the number of auxiliary queries generated, in Column $Num.\ Queries$. Naturally, the dataset size has a similar effect as in the rewriting approach. However, the number of generated auxiliary queries plays an even more significant role in the total processing time. Intuitively, this number depends (*i*) on the size of the initial answer set and (*ii*) on the number of involved constraints. We can notice that, contrary to the rewriting approach, the complexity of the query has little effects on the total execution time in the naive approach. See, for instance, $Q6$ and $Q7$ which have similar complexity. However, $Q6$ has many answers, provoking the generation of many sub-queries. Indeed, the verification step is carried out by generating simple sub-queries for each answer *w.r.t.* each constraint.

Perhaps one of the most meaningful observation provided by our experiments is that, when the dataset size increases, the rewriting approach is clearly far more efficient than the naive approach. This is specially the case when the initial query gives a large number of answers, no matter if it is a simple or a complex query, and these answers trigger a lot of constraints: $Q2$ and $Q6$ are typical examples of such cases, which induce a time-out for 5 universities. For $Q4$, which triggers no constraint, the naive approach is better or similar to the rewriting one.

Related Works We already mentioned the main works related to our proposal in Section 2.1. Firstly, ontological-constraints-based query-rewritings in Ontology-Based Data Access (OBDA) systems [18, 3] and rewritings in incomplete information querying systems [11, 15, 12] inspired our solution. In [16] we also find different semantics for query answering over inconsistent Datalog$^{\pm}$

ontologies. Their goal is to propose corrections to the database, while ours is to avoid answering on the basis of inconsistent data. Indeed, we designed our solution with traditional database constraints that must be verified, while in those works ontological constraints are seen as inference rules. Our user constraints allows us to verify answer sets and eleminate those answers that do not comply with the user needs. For instance, coming back to c_p given in Introduction, which enforces that all person who teaches is a professor, the answer $teacherOf(Bob, DB)$ is valid only if $professor(Bob)$ is true in the provided answers, *i.e.* the fact $professor(Bob)$ is not inferred from the user constraints.

For this reason, our rewriting algorithm is based on traditional results in the database domain already cited in Section 2.2 [2, 17, 14]. We are currently studying to what extent our proposed user-context is covered by the traditional framework of answering queries using views, for which a general rewriting algorithm is presented in [8], and further improved in [13]. We already mentioned this algorithm, called C&B for its two phases (Chase and BackChase), at the end of Section 3.1. It first constructs a canonical rewriting called $UniversalPlan$ by using TGDs rules, which play the same role as our positive constraints, and then it searches minimal reformulations among the candidates in the $UniversalPlan$, using EGDs rules. But how it could apply to our context is not obvious, because we already mentioned that, in general, the Chase can not be directly used with constraints containing constants, excepted when there exists a homomorphism from the constraint's atoms to the query's atom (see Lines 17-20 in Algorithm 1).

5 Conclusion

We presented a solution for validating a set of user quality constraints when performing query evaluation, in the semantic web context. A naive way to verify them is to generate auxiliary queries after having got the result set from the evaluation of the user query. Our experiments have put in evidence that these auxiliary queries, generally simple but performed on huge data sets, sometimes lead to overload the system. Integrating as much as possible the constraints into the original user query can help to overcome this drawback. We presented an algorithm for such a constraint-query integration, and provided experimental results that demonstrate its benefits regarding total query-with-constraints processing time. Both techniques are correct and complete. In other words, given the query Q and the constraints C, (i) there is no answer to Q that satisfies C, but is not in the answer set of both methods (completeness); (ii) all the answers produced by both algorithms are answers to Q that respect C (correction). Our immediate future works will concern extending our experiments to take into account the data provider features and capabilities (*e.g.* not all of them can evaluate complex queries).

References

1. Graal. At https://graphik-team.github.io/graal/.

2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.
3. S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web Data Management*. Cambridge University Press, New York, NY, USA, 2011.
4. M. Bamha, J. Chabin, M. Halfeld-Ferrari, B. Markhoff, and T. B. Nguyen. Personalized environment for querying semantic knowledge graphs: a mapreduce solution. Technical report, LIFO- Université d'Orléans, RR-2017-06, 2017.
5. A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.
6. J. Chabin, M. Halfeld-Ferrari, and T. B. Nguyen. Querying Semantic Graph Databases in View of Constraints and Provenance. Technical report, LIFO- Université d'Orléans, RR-2016-02, 2016.
7. C. d'Amato, A. G. B. Tettamanzi, and T. D. Minh. Evolutionary discovery of multi-relational association rules from ontological knowledge bases. In *EKAW*, pages 113–128. Springer International Publishing, 2016.
8. A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Rec.*, 35(1):65–73, Mar. 2006.
9. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89 – 124, 2005. Database Theory.
10. L. Galárraga, S. Razniewski, A. Amarilli, and F. M. Suchanek. Predicting Completeness in Knowledge Bases. In *WSDM*, pages 375–383, 2017.
11. G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, pages 2–13, 2011.
12. G. Gottlob, G. Orsi, and A. Pieris. Query rewriting and optimization for ontological databases. *CoRR*, abs/1405.2848, 2014.
13. I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1015–1026, New York, NY, USA, 2014. ACM.
14. D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984.
15. D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo. Inconsistency-tolerant query answering in ontology-based data access. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:3 – 29, 2015.
16. T. Lukasiewicz, M. V. Martinez, and G. I. Simari. Inconsistency handling in datalog+/- ontologies. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pages 558–563, 2012.
17. D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, Dec. 1979.
18. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.
19. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, pages 601–616, 2011.
20. G. Weikum, J. Hoffart, and F. M. Suchanek. Ten years of knowledge harvesting: Lessons and challenges. *IEEE Data Eng. Bull.*, 39(3):41–50, 2016.