Personalized Environment for Querying Semantic Knowledge Graphs: a MapReduce Solution

Mostafa Bamha, Jacques Chabin, Mirian Halfeld-Ferrari, Béatrice Markhoff, Thanh Binh Nguyen

▶ **To cite this version:**

Mostafa Bamha, Jacques Chabin, Mirian Halfeld-Ferrari, Béatrice Markhoff, Thanh Binh Nguyen. Personalized Environment for Querying Semantic Knowledge Graphs: a MapReduce Solution. [Research Report] LIFO, Université d'Orléans. 2017. hal-01916736

HAL Id: hal-01916736
https://hal.science/hal-01916736

Submitted on 8 Nov 2018

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.

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Personalized Environment for Querying Semantic Knowledge Graphs: a MapReduce Solution

Mostafa Bamha, Jacques Chabin, Mirian Halfeld Ferrari, Béatrice Markhoff, Thanh Binh Nguyen
LIFO, Université d'Orléans

Rapport n^o **RR-2017-06**

Personalized Environment for Querying Semantic Knowledge Graphs: a MapReduce Solution*

Mostafa Bamha[1], Jacques Chabin[1], Mirian Halfeld-Ferrari[1], Béatrice Markhoff[2], and Thanh Binh Nguyen**[1]

[1] Université Orléans, INSA CVL, LIFO EA, Orléans, France
{bamha, jchabin, mirian, binh}@univ-orleans.fr
[2] Université François Rabelais de Tours, LI EA, Blois, France
markhoff@univ-tours.fr

Abstract. Querying according to a personalised context is an increasingly required feature on semantic graph databases. We define contexts by using constraints imposed on queries and not on data sources. No correction trial is performed on an inconsistent database but answers are ensured to be valid. Data confidence according to provenance is also taken into account. As constraint validation and query evaluation are two independent modules, our approach can be tested with different query evaluators. This paper focus on a MapReduce query environment.

Keywords: graph database, RDF, constraint, context, MapReduce.

1 Introduction

Constraint verification, usually neglected in modern scenarios due to velocity and volume exigencies, is a fundamental issue when *answer quality* and *context personalisation* are required. The Resource Description Framework (RDF) is a graph-based data model which is not only restricted to web semantics but increasingly popular for sharing data in different knowledge areas ([6,17]). Providing valid or context dependent answers to one working on distributed RDF data is a growing necessity.

This paper describes a query environment over semantic graph databases (or knowledge graphs) involving a mechanism for *filtering* answers according to a *customised context* that settles *global constraints* and *confidence degrees.* Inconsistency on sources is allowed but query answers are filtered to ensure consistency *w.r.t.* constraints. Figure 1 illustrates our data graph system. A *global system* offers a *graph schema* containing global predicates and constraints. A *local system* offers data access and tools to compute answers. It is composed by different data sources which stores a *distributed instance of the graph.* Queries are built over the global schema but evaluated over local sources.

* Work partially supported by APR-IA GIRAFON.
** Supported by a PhD grant Orléans-Tours.

Different query evaluation system can be adapted to this general model. In this paper, we focus on an organization that provides distributed sources over a MapReduce platform to store and query its data. In this context, we discuss the benefits of data distribution according to confidence degrees and we offer a first performance analysis of our constraint validation proposal.

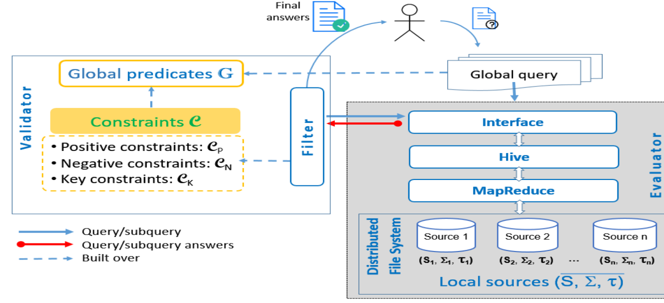

Fig. 1. Query system overview

We use the university domain ontology (LUBM[3]) to illustrate our approach.

Example 1. Let us suppose a context with only two constraints:

$$c_{P_0} : professor(Xprof) \rightarrow masters(Xprof, Xuniv)$$

$$c_{N_0} : masters(Xstud, Xuniv), undergrad(Xstud, Xuniv) \rightarrow \bot$$

The first positive constraint states that a professor must have a master degree. The second negative constraint does not accept students having an (undergraduate) degree and a master degree from the same university. We assume the following sources of a distributed database with their truth confidence degree (τ) and facts.

Source	τ	True facts
S_1	0.6	professor(Bob), professor(Lea)
S_2	0.8	professor(Alice), professor(Louise)
S_3	0.7	masters(Louise,Orleans), masters(Lea,Orleans), masters(Alice,Orleans)
S_4	0.6	undergrad(Alice,Orleans),undergrad(Bob,Orleans),undergrad(Lea,Tours)

In this scenario we consider query $q(Xprof) \leftarrow professor(Xprof)$. Let q's required confidence degree be $\tau_{in} = 0.6$, indicating that sources having a smaller confidence degree should not be taken into account. The answer is the set $\{(Louise), (Lea)\}$. Neither tuple (Bob) nor $(Alice)$ are answers due to violations to c_{P_0} and c_{N_0}, respectively. However, if $\tau_{in} = 0.7$ then (Lea) is no more an answer but $(Alice)$ becomes an answer. As $undergrad(Alice, Orleans)$ is no longer available (with $\tau = 0.6$), no reliable violation for c_{N_0} is detected. □

[3] LUBM: Lehigh University Benchmark developed to facilitate the evaluation of Semantic Web repositories; `http://swat.cse.lehigh.edu/projects/lubm/`

Validating RDF data deserves considerable attention and bright to light different ways of considering constraints: the so-called ontology constraints are seen as inference rules, contrary to classical database constraints ([9]). As our *global constraints* customize user's exigencies we deal with them in a traditional database viewpoint. In the above example, *Bob* is a professor but no information concerning his master degree is available from the data sources. Constraint c_{P_0} is not supposed to infer the missing information concerning *Bob*.

Paper organisation: Important definitions are summarized in Section 2. Section 3 focuses on the semantics of constrained queries over sources with different confidence degrees. Section 4 summarizes query evaluation over a MapReduce platform; while experimental results are discussed in Section 5. Section 6 presents some related work and Section 7 concludes the paper.

2 A Graph Database

2.1 Preliminaries

Alphabet and atomic formulas. Let $\mathbf{A}$ be an alphabet consisting of constants, variables, predicates, the equality symbol ($=$), quantifiers ($\forall$ and $\exists$) and the symbols $\top$ (true) and $\bot$ (false). We consider four mutually disjoint sets, namely: (1) $\mathbf{A}_C$, a countably infinite set of constants, called the underlying database domain; (2) $\mathbf{A}_N$, a countably infinite set of fresh labelled nulls which are placeholders for unknown values; (3) VAR an infinite set of variables used to range over elements of $\mathbf{A}_C \cup \mathbf{A}_N$ and (4) PRED, a *finite* set of predicates or relation names (each predicate is associated with a positive integer called its arity). The only possible terms are constants or variables. An atomic formula (or atom) has one of the forms: (*i*) $P(t_1, ..., t_n)$, where P is an n-ary predicate, and $t_1, ..., t_n$ are terms; (*ii*) expressions $\top$ (true) and $\bot$ (false) or (*iii*) $t_1 = t_2$ (where t_1 and t_2 are terms). A conjunction of atoms is often identified with the set of all its atoms. We denote by $\mathbf{X}$ sequences of terms $X_1 \dots X_k$ where $k \geq 0$ (in the context one can understand when only variables are used).

Substitution. A *substitution* from one set of symbols E_1 to another set of symbols E_2 is a function $h : E_1 \Rightarrow E_2$. A *homomorphism* from a set of atoms A_1 to a set of atoms A_2, both over the same schema R, is a substitution h from the set of terms of A_1 to the set of terms of A2 such that: (i) if $t \in$CONST, 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.

Database schema and instances. Databases are represented as a relational database in the logic programming perspective. A database schema $\mathbb{S}$ is composed by a finite set of predicate symbols S. A *fact* (or a ground atom) over $S \in \mathbb{S}$ is an atom of form $S(u)$ where $u \in \mathbf{A}_C^n$. Under the logic-programming perspective, an instance over S is a finite set of instantiated atoms over S while an instance over schema $\mathbb{S}$ is a finite set $\mathcal{S}$ which is the union of instances over S, for all $S \in \mathbb{S}$.

Queries. A *conjunctive query* (CQ) q of arity n over a given schema is a formula 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 and q is a n-ary predicate. A *boolean conjunctive* query (BCQ) is a CQ of arity zero. We denote by $body(q)$ (respect. $head(q)$) the set of atoms composing the body (respect. the head) of a given query q. Let I be an instance for the given schema. The *answer* to a CQ q of arity n over I, denoted as $q(I)$, is the set of all n-tuples $t \in \mathbf{A}_C^n$ for which there exists a homomorphism $h_t : X \cup Y \Rightarrow \mathbf{A}_C$ such that $h_t(\phi(\mathbf{X},\mathbf{Y})) \subseteq I$ and $h_t(\mathbf{X}) = t$. We denote by h_t *a homomorphism used to obtain an answer tuple* t. Technically, the answer *false* (*i.e.*, a negative answer) for a BCQ corresponds to the empty result set and the answer *true* (*i.e.*, a positive answer) corresponds to the result set containing the empty tuple. A positive answer over I is denoted by $I \models q$.

2.2 Graph database, constraints and provenance

Our query environment is composed by two independent modules (Figure 1): a validator and an evaluator. In the validator, a query q_1 is built from the initial query q and a part of the set of constraints and sent to the evaluator. The evaluator computes answers to q_1 by consulting distributed data sets and returns them to the validator, which starts a constraint verification step over the answer sets. This validation processes may require the generation of other subsidiary queries. Once this 'dialogue' between the validator and the evaluator is finished, *i.e.*, all constraint verifications are done, valid answers are available to the user. The evaluator integrates query translation and planning mechanisms.

In this context, the following aspects of our approach are worth underling:

1. A graph schema $\mathbb{G}$ is a finite set of predicate symbols or relation names $G_1 \dots G_m$ associated to a set of constraints $\mathcal{C}$ defined over $\mathbb{G}$.
2. Local data sources $(\mathcal{S}_1, \tau_1), \dots, (\mathcal{S}_n, \tau_n)$ store a distributed graph instance. For $0 \leq i \leq n$, each $\mathcal{S}_i$ is a data source instance respecting a local schema $\mathbb{S}_i$ and each τ_i is the source confidence degree, represented by a number in the interval $[0, 1]$.
3. We assume the existence of a mapping between global and local systems (*e.g.* via a LAV approach). In general, when a global query q has a non-empty set of answers then there exists at least one re-writing of q in terms of sub-queries $q'_1 \dots q'_m$ where each q'_j $(1 \leq j \leq m)$ is a sub-query expressed over local relations. These mapping aspects are out of the scope of this paper and to simplify explanations, our examples consider identical relation schemas for global and local systems.
4. Our proposal can be implemented over different evaluation mechanisms. This paper focus on a MapReduce solution. When confidence degrees are disregarded we write $\overline{\mathcal{S}} \models q$, a shorthand of $\mathcal{S}_1 \sqcup \dots \sqcup \mathcal{S}_n \models q$, to denote that the answer of a BCQ q is positive *w.r.t.* local databases. We see local databases as a whole, *i.e.*, a system (or distributed database) capable of answering our global query. We denote by $ans(q, \overline{\mathcal{S}})$ the set of tuples obtained as answers for a conjunctive global query q over the distributed database $\overline{\mathcal{S}}$.

5. When data provenance is considered, we associate a confidence degree to each source and a minimal confidence degree (τ_{in}) to the query. These measures are settled by the user, according to his knowledge of source accuracy or proposed by a recommendation system. Our system offers some flexibility in how to use veracity information (cf. Section 3). We write $\overline{(\mathcal{S}, \tau)} \models q : \tau_{in}$ and we denote by $ans(q : \tau_{in}, \overline{(\mathcal{S}, \tau)})$ the set of tuples obtained as answers for a conjunctive global query $q : \tau_{in}$ over a distributed database $\overline{(\mathcal{S}, \tau)}$.
6. Answers for a given query q are filtered according to quality restrictions settled for an application. An user may establish the context in which a certain number of queries is evaluated and choose another context for other queries. The customization of this quality context is provided by constraints, *i.e.*, restrictions imposed on queries. Only data respecting them are allowed as query answers. Inconsistencies on sources are allowed and our approach does not aim at correcting them, but at discarding them during query evaluation.

The following definition introduces the constraints used in this paper.

Definition 1 (Global Constraints). A constraint c is a rule and we denote by $body(c)$ and $head(c)$, the left hand-side and the right hand-side of c, respectively. A set $\mathcal{C}$ of constraints over $\mathbb{G}$ is composed by three subsets, as follows:
(1) **Positive constraints** ($\mathcal{C}_P$): Each positive constraint has the form

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

where L_1 and L_2 are atoms, such that for every constraint c whose head has a non-empty $\mathbf{Z}$, there is no other constraints $c_1 \in \mathcal{C}_P$ $(c_1 \neq c)$ for which $body(c1)$ and $head(c)$ are unifiable.
(2) **Negative constraints** ($\mathcal{C}_N$): Each negative constraint has 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 $L_1(\mathbf{X}_1), L_2(\mathbf{X}_2)$, having a non-empty intersection between the terms in $\mathbf{X}_1$ and in $\mathbf{X}_2$.
(3) **Equality-generating dependency constraints** without nulls ($\mathcal{C}_K$) (also called key constraints): Each EGD has the general form

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

where $\mathbf{Y}$ is a sequence having at least one term. Notice that EGD include functional dependency having the form $L_1(\mathbf{Y}, X_1, \mathbf{Z}_1), L_1(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2$ □

Notice that positive constraints are a special case of linear TGD (Tuple Generating Dependency ([4])): they contain only one atom in the head and do not allow a fresh null propagation (atoms with existential variables are not unifiable with another atom in a rule's body). Our constraints generalize the well known functional and inclusion dependencies in relational databases and extend the active rules introduced in [10]. When the body of a positive constraint matches a ground atom, the (now instantiated) constraint triggers side effects, setting other facts which should also be true in the distributed database. To formally define this behaviour, we introduce an immediate consequence operator.

Definition 2 (Immediate consequence operator). Let T be an operator over $\mathcal{C}$ and I be a set of facts. We define $T_{\mathcal{C}}(I) = I \cup \{\nu'(head(c)) \mid c \in \mathcal{C}$ and $\nu(body(c)) \subseteq I$ and $\nu' \subseteq \nu$ is an extension of ν such that , for each

existential variable $Z_i \in head(c)$, we have $h'(Z_i) = z_i$, where $z_i \in \Delta_N$ is a fresh labelled null not introduced before }. □

3 Querying Environment with Constraints and Confidence

This section presents our personalized querying environment over a distributed semantic graph. Original aspects of our approach are:
(*i*) A constraint violation implies discarding only invalid answers (not the whole database). However, no answer built on the basis of non-valid data is accepted.
(*ii*) The distributed database (local system) may contain non valid data which are filtered when involved in a global query.
(*iii*) Constraints are active rules imposing a dialogue between the global and the local levels: on the global level, they are triggered by facts in the query body (facts resulting from query instantiation, during evaluation). Triggered constraints usually impose extra verifications on the local level (*e.g.* testing if another fact is true in the distributed database).
(*iv*) Answer computation also takes into account the confidence degree of data provenance, discarding data considered as not faithful.
(*v*) The flexibility of our approach includes settling constraints appropriate to a given application, choosing confidence degrees of data sources and establishing how to use confidence information to compute answers.

3.1 Formal definition and properties

When data sources are associated to confidence degrees, a query q can have a required confidence degree τ_{in} and one can expect that only data coming from sources whose confidence degrees respect a given condition *w.r.t.* τ_{in} are taken into account to build answers for q.

Definition 3 (Local querying with confidence). Let $(\mathcal{S}, \tau)$ be a local source database where $\mathcal{S}$ is a database instance and τ is the truth or confidence degree of the database. Let q be a query over $(\mathcal{S}, \tau)$ with the minimum required truth degree τ_{in}. The answer of $q : \tau_{in}$ over $(\mathcal{S}, \tau)$ is the set $ans(q{:}\tau_{in}, (\mathcal{S}, \tau)) = \{(t : \tau_{out}) \mid \tau_{out} = \tau \text{ and } t \in q(\mathcal{S}) \text{ and } cond(\tau_{in}, \tau)\}$, where $cond(\tau_{in}, \tau)$ is a condition we may establish to avoid considering some sources. □

In this paper, $cond(\tau_{in}, \tau) = (\tau \geq \tau_{in})$ discarding all sources whose confidence is inferior to τ_{in}. In Example 1, if $\tau_{in} = 0.7$, the tuple (Lea) is no more an answer for q because $professor(Lea)$ comes from S_1 with $\tau = 0.6$.

The system can be parametrized with other conditions, and even no condition can be settled at this step (*i.e.* all sources are considered in the computation of τ_{out}). Details on how the confidence degrees are computed are out of the scope of the paper. One can suppose that they are given by the user or by a recommendation routine.

To compose the response for a query $q:\tau_{in}$, we put together answers produced by differently trusted database. Firstly, we need a set of possible candidate answers: tuples t that are trustable *w.r.t.* τ_{in}.

Definition 4 (Candidate answer over $\overline{(\mathcal{S},\tau)}$). Let $\overline{(\mathcal{S},\tau)}$ be a graph database instance composed of n local databases having different truth degrees. A couple $(t : \tau_{out})$ is a candidate answer for a global query $(q : \tau_{in}, \overline{(\mathcal{S},\tau)})$ if the following conditions hold: (1) tuple t is an answer obtained from local sources, *i.e.* $t \in ans(q, \overline{\mathcal{S}})$; (2) $\tau_{in} \leq \tau_{out}$ and the computation of τ_{out} is defined by $\tau_{out} = f(\tau_{in}, \{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$ where:
(i) each $\tau^k_{out_{S_j}}$ denotes the degree of the tuples in $ans(q_k : \tau^k_{in}, (S_j, \tau_j))$ for the sub-query q_k $(1 \leq k \leq m)$ generated to be evaluated on the local source (S_j, τ_j) during the evaluation process of q (where $i, j, l \in [1, n]$) and
(ii) function f computes a confidence degree taking as input the query confidence degree and the confidence degrees of data sources concerned by q. □

A user can parametrize the use of confidence degrees by choosing different functions f and by combining this choice with *cond* in Definition 3. For example, consider that $cond(\tau_{in}, \tau) = true$. In this case, the selection of t is based *only* on the confidence degree computed by f. If f is the average, the resulting τ_{out} computes the average of *all* data sources involved in the query answering. If however a condition such as $\tau_{out} \geq 0.5$ is used in Definition 3, only sources respecting it are used in the average computation. Our examples consider that $f(\tau_{in}, \{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$ corresponds to $min(\{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$, and as stated above, we disregard sources whose confidence is inferior to τ_{in}. Thus, in Example 1, the answer set for $q : 0.6$ is $\{((Lea) : 0.6), ((Louise) : 0.7)\}$.

Global query answers are restrained by constraints in $\mathcal{C}$. To find an answer t to a query q means to find an instantiation h_t (cf. paragraph *Queries*, Section 2) for the body of q that generates t. Verifying whether $h_t(body(q))$ is valid *w.r.t.* $\mathcal{C}$ ensures the validity of our answer. In the following definition we put together constraint and confidence degree verification to answer global queries. Let $q:\tau_{in}$ be a conjunctive global query and $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_N \cup \mathcal{C}_K$ be a set of constraints over $\mathbb{G}$. Valid candidate answers are those that respect constraints and are obtained by trusted databases.

Definition 5 (Valid candidate answers). The set of valid candidate answers of a query $q : \tau_{in}$, restrained by $\mathcal{C}$, over a database $\overline{(\mathcal{S},\tau)}$, denoted by *valCandAns* $(q:\tau_{in}, \mathcal{C}, \overline{(\mathcal{S},\tau)})$ is defined by the set $\{(t : \tau_{out})\}$ respecting the following conditions: (1) t is a candidate answer as in Definition 4 and h_t is a corresponding homomorphism (Section 2); (2) there exists h_1 such that for all $L \in h_1(T^*_{\mathcal{C}_P}(h_t(body(q)))$ the following conditions hold:

(i) there is a positive answer for $q() \leftarrow L : \tau_{in}$ on $\overline{(\mathcal{S},\tau)}$;

(ii) for each $c \in \mathcal{C}_N$ of the form $L_1, L_2 \rightarrow \bot$, if there is a homomorphism ν such that $\nu(L_i) = L$, then there is no homomorphism ν' that extends ν and for which there is a positive answer for $q'() \leftarrow \nu'(L_{\bar{i}}) : \tau_{in}$ (In our notation, if $i = 1$ then $\bar{i} = 2$ and vice-versa.) and

(iii) for each $c \in \mathcal{C}_K$ of the form $L_1(\mathbf{Y}, X_1, \mathbf{Z}_1), L_2(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2$ if there is a homomorphism ν such that $\nu(L_i(\mathbf{Y}, X_i, \mathbf{Z}_i)) = L$, then the answer of $q(X_{\bar{i}}) \leftarrow \nu(L_{\bar{i}}(\mathbf{Y}, X_{\bar{i}}, \mathbf{Z}_{\bar{i}})) : \tau_{in}$ is a singleton containing the tuple value $\nu(X_i)$. □

It is important to understand that constraints are triggered by atoms in the body of a query. Let us suppose a query q over database $\overline{\mathcal{S}}_1$, on a context defined by $\mathcal{C}$, where $c \in \mathcal{C}_P$ has an existential variable in its head. We assume that h_t is the homomorphism used to produce tuple t as an answer for q on $\overline{\mathcal{S}}_1$, that f is a fact in $h_t(body(q))$ and that there is an homomorphism ν for which $\nu(body(c)) = f$. Constraint c produces atom $h'_t(head(c))$ where h'_t is an extension of h_t where the existential variable is associated to a new fresh null in $\mathbf{A}_N$. Then, to decide whether the answer t is valid *w.r.t.* c, we need to check whether there is a homomorphism $h_1 : \mathbf{A}_N \rightarrow \mathbf{A}_C$ such that $h_1(h'_t(head(c)))$ is a fact in $\overline{\mathcal{S}}_1$. The example below illustrates this situation.

Example 2. We consider a new context defined by the following constraints:

- $c_{P_1} : AssistantProfessor(X) \longrightarrow \exists Y TeacherOf(X, Y)$
  Every *AssistantProfessor* is the teacher of a course.
- $c_{P_2} : GraduateStudent(X) \longrightarrow \exists Y TakesCourse(X, Y)$
  For every *GraduateStudent*, there is a *course* followed by him.
- $c_{N_1} : TeacherOf(X, Y), TakesCourse(X, Y) \longrightarrow \perp$
  One cannot be the teacher and a student of the same course.

In this scenario, query $q_1(X) \leftarrow AssistantProfessor(X), GraduateStudent(X)$ asks for people who are both *AssistantProfessor* and *GraduateStudent*. Let us suppose that a distributed database $\overline{\mathcal{S}}$ offers the following instantiations for $body(q)$ from which the answers are built. These instantiations are represented by two homomorphisms, namely:

- h_t which associates variables of atoms in $body(q)$ to constants in $\overline{\mathcal{S}}$.
  Here, we have the answer $h_t : \{X/Bob\}$ with
  $h_t(body(q)) = \{AssistantProfessor(Bob), GraduateStudent(Bob)\}$.
- h_1 which associates nulls in $\mathbf{A}_N$ to constants appearing in $\overline{\mathcal{S}}$.
  When positive constraints are triggered from facts in $h_t(body(q))$, we obtain $TeacherOf(Bob, N_1)$ and $TakesCourse(Bob, N_2)$ where N_1 and N_2 are fresh nulls. By instantiating these new fresh nulls, we assume that four answers to q_1 are possible:

$$h_{1_1} = \{ AssistantProfessor(Bob), GraduateStudent(Bob), TeacherOf(Bob, BDD), TakesCourse(Bob, BDD)\}$$
$$h_{1_2} = \{AssistantProfessor(Bob), GraduateStudent(Bob), TeacherOf(Bob, BDD), TakesCourse(Bob, Maths)\}$$
$$h_{1_3} = \{AssistantProfessor(Bob), GraduateStudent(Bob), TeacherOf(Bob, Algo), TakesCourse(Bob, BDD)\}$$
$$h_{1_4} = \{AssistantProfessor(Bob), GraduateStudent(Bob), TeacherOf(Bob, Algo), TakesCourse(Bob, Maths)\}$$

The three first instantiations are *not* valid due to negative constraint c_{N_1}. For instance, in h_{1_3}, $TakesCourse(Bob, BDD)$ triggers c_{N_1}. As $TeacherOf(Bob, BDD)$ is in $\overline{\mathcal{S}}$, the instantiation h_{1_3} is discarded. On the other hand, the last instantiation respects all the constraints (neither $TakesCourse(Bob, Algo)$ nor $TeacherOf(Bob, Maths)$ are true in $\overline{\mathcal{S}}$). Thus, in conclusion, Bob is a valid answer to our query. Indeed, in our approach, we are looking for an instantiation (built from data of $\overline{\mathcal{S}}$) for which the following condition holds:

$$h_1(T^*_{\mathcal{C}_P}(h_t(body(q))) \models \forall X \exists Y, \exists Y' AssistantProfessor(X) \wedge GraduateStudent(X) \wedge TeacherOf(X,Y) \wedge TakesCourse(X,Y') \wedge \neg TeacherOf(X,Y') \wedge \neg TakesCourse(X,Y) \quad \square$$

Given a query $q : \tau_{in}$, restrained by $\mathcal{C}$, over $\overline{(\mathcal{S}, \tau)}$, we stress that our method renders *all* and *only* valid answers.

Proposition 1. The set $valCandAns$ over $\overline{(\mathcal{S}, \tau)}$, for a query $q : \tau_{in}$, defined on $\mathbb{G}$ with constraints $\mathcal{C}$ is correct and complete. $\square$

Proof: straightforward from Definition 5.

3.2 Implementation issues

As mentioned in Section 2.2, to perform constraint validation, our validator (on the upper global level) establishes a dialogue with a query evaluator (on the lower, local, distributed level). This dialogue has two main steps. The first step concerns a rewritten version of the user's query on the basis of some selected constraints. The second step deals with several new simpler queries generated on the basis of the remaining constraints and the answers obtained during the first step. Algorithm 1 summarizes this role process.

On line 3 of Algorithm 1, function *RewritreWithConstraint* is called to perform the rewriting of $q : \tau_{in}$ into a new query q_1 whose body is q's body completed by positive or negative atoms, according to chosen constraints in $\mathcal{C}_{rew}$. For instance, suppose that in $\mathcal{C}_{rew}$ we have constraints $c_1 : A(X) \to B(X)$, $c_2 : A(X), D(X) \to \bot$, query $q(X) \leftarrow A(X)$ and a distributed database $\overline{(\mathcal{S}, \tau)}$. In the rewriting phase, q is translated into $q_1(X) \leftarrow A(X), B(X), \neg D(X)$ with minimum confidence degree τ_{in}. Clearly, answers for $q_1 : \tau_{in}$ will respect the given constraints[4]. The query q_1 is sent to the evaluator ($Eval1$, on line 4) and answers are stored in a set Solutions. They are valid *w.r.t.* C_{rew}.

The advantage of validating through rewriting is that validation and query evaluation are performed together, in just one step. The disadvantage concerns the need of a (lower-level) query evaluator capable of dealing with non conjunctive queries (and this is not always the case when working with new distributed query environments focusing on large amount of data). Another problem is that

[4] Notice that, during this re-writing, consistency of the query's body is verified *w.r.t.* $\mathcal{C}_N$ (*e.g.* a query such as $q_2(X) \leftarrow A(X), D(X)$ is detected as inconsistent and discarded).

Algorithm 1: $PersonalizedQuerying(q : \tau_{in}, \overline{(\mathcal{S}, \tau)}, \mathcal{C})$

Input :
- Global conjunctive query q with minimum confidence degree τ_{in}
- Distributed data source (*i.e.*, the corresponding links to data access)
- Global constraint set $\mathcal{C}$, settled for the application.

Output: Answers of q on $\overline{(\mathcal{S}, \tau)}$ respecting $\mathcal{C}$ and confidence degree τ_{in}.

```
1  AnsSet := ∅;
2  C_rew := Choose(C, (S,τ))
3  q1 := RewritreWithConstraint(q, C_rew);
4  Solutions := Eval1(q1 : τ_in, (S,τ));
5  Cache := CreateCache();
6  C_check := C \ C_rew
7  foreach (sol, τ_out) ∈ Solutions where sol = (t, h_t) do
8  |  if Valid(sol, C_check, τ_in, Cache) then
9  |  |  AnsSet := AnsSet ∪ {(t, τ_out)};
10 |  end
11 end
12 return AnsSet;
```

the evaluation of such a completed and more complex query is not necessarily more efficient than the evaluation of many simpler queries.

The set Solutions (line 4) contains pairs (sol, τ_{out}) where sol is the pair (t, h_t). Then, on line 8, function *Valid* performs two main actions:
(i) It generates auxiliary queries from the instantiation of each remaining constraint c (not in C_{rew}), triggered by a fact appearing in $h_t(body(q))$. For instance, from the example above, suppose now that $\mathcal{C}_{rew} = \emptyset$ and thus $\mathcal{C}_{check} = \{c_1, c_2\}$. If the evaluation of q on $\overline{(\mathcal{S}, \tau)}$ results in $h_t = \{X/a\}$, then auxiliary queries $q_a() \leftarrow B(a)$ and $q_b() \leftarrow D(a)$ (both with minimum confidence degree τ_{in}) are generated in other to verify constraints c_1 and c_2.
(ii) It returns only the valid answers, which are stored in the answer set. All answers produced with data which do not respect constraints are discarded.

Determining whether it is better to evaluate many simple queries instead of rewriting the user's query into a complex one is a tricky problem. Efficiency depends not only on the query evaluator, but also on the database schema and instances. For this reason, our approach offers flexibility. On line 2 of Algorithm 1, one can choose which constraints are going to be used in the rewriting perspective. This choice can be tuned up gradually according to the needs of database administrators and users. Feasibility tests were performed with GraalDatalog, MySQL and Virtuoso SPARQL query service on *DBpedia*.

4 Query Evaluation using MapReduce

MapReduce is a simple yet powerful framework for implementing applications in large scale distributed systems without having extensive prior knowledge of issues

related to data redistribution, or task allocation and fault tolerance ([5,11]). Most MapReduce frameworks includes *Distributed File Systems* (DFS) designed to store very large files with streaming data access patterns and data replication for fault tolerance while guaranteeing high disk I/O throughput. We use an open source version of MapReduce called Hadoop developed by *The Apache Software Foundation*. Hadoop framework includes a distributed file system called HDFS[5] designed to store very large files. In MapReduce frameworks the access to data requires a full scan of input data from DFS, which may increase disk/IO and communication costs in applications involving very large datasets. Many data management frameworks have been introduced to allow efficient access to large datasets stored in a DFS. In these frameworks, only relevant columns/data can be accessed and queried using "efficient" SQL-like query languages. *Apache Hive*, *Hbase* and *Pig* are examples of such data management frameworks.

In this paper, we use *Hive* with RDF data. *Hive* provides data indexing and partitioning for efficient data analysis. Queries in HiveQL (an SQL-like language) are converted to a sequence of MapReduce jobs. The main motivation to use *Hive* is its ability to manage huge amount of compressed datasets. Each table can be divided into partitions (each partition corresponds to one or more HDFS buckets/splits), providing a more efficient execution of queries involving large datasets with different confidence degrees. Communication costs, HDFS disk I/O and data analysis processing time decrease because only table'splits corresponding to, at least, a given confidence factor are selected for data analysis. Figure 1 places Hive and MapReduce in our general architecture.

5 Experimental Results

Our experimental results comprise two steps. We study the impact of data distribution guided by confidence degrees on the query evaluator. Then, we analyse the use of this query evaluator in conjunction with our constraint validator.

5.1 The impact of data distribution using confidence factors

To evaluate the performance of our table's partitioning using confidence factors, we compare the execution of different HiveQL queries using both partitioned and non-partitioned LUBM data sources (*w.r.t.* confidence factors). In partitioned LUBM data sources, records of each table are stored into blocks and each block contains only records corresponding to a unique confidence factor. For non-partitioned data, each block of data, of each table, may have different confidence factors.

To study the effect of confidence factor in partitioning our LUBM benchmark (about $100GB$ of source data corresponding to approximately $8GB$ of compressed tables), we consider confidence factors ranging from 25% to 95% (*i.e.* 0.25, 0.95). We have run a large series of experiments where 24 `Virtual`

[5] HDFS: Hadoop Distributed File System.

`Machines` (VMs) were randomly selected from our university cluster using OpenNubula software for VMs administration. Each VM has the following characteristics: 1 Intel(R) Xeon@2.53GHz CPU, 2 Cores, 8GB of Memory and 100GB of Disk. Setting up a Hadoop cluster consists of deploying each centralised entity (namenode and jobtracker) on a dedicated VM and co-deploying datanodes and tasktrackers on the rest of VMs. The data replication parameter was fixed to 3 in the HDFS configuration file.

Query	Total CPU Time	HDFS Read	Original table size
$\mathcal{Q}_1$	51.52 seconds	2709273139 Bytes (∼2.7 Gbytes)	∼19 Gbytes
$\mathcal{Q}_2$	478,13 seconds	19857918459 Bytes (∼19 Gbytes)	∼19 Gbytes

Table 1. Effect of partitioned data (*w.r.t.* condidence factor) on $\mathcal{Q}$'s execution time.

Table 1 shows the execution of query $\mathcal{Q}$: SELECT Count(*) FROM publicationAuthor WHERE cf_factor>=85; in two different situations. We denote by $\mathcal{Q}_1$ its execution using partitioned data (*w.r.t.* confidence degree), and by $\mathcal{Q}_2$ its execution using non partitioned data. In all tests, including those presented in Table 1, we notice that $\mathcal{Q}_1$ outperforms $\mathcal{Q}_2$. Execution time for $\mathcal{Q}_1$ is approximately 10 times smaller than for $\mathcal{Q}_2$. The ratio between these two execution time can be explained by the fact that in $\mathcal{Q}_1$ only relevant data (*e.g.* data corresponding a confidence factor higher that 85%) is read from HDFS whereas in $\mathcal{Q}_2$ all input data need to be read. In this scenario, HDFS disk I/O and the amount of data transmitted over the network diminish considerably, implying a reduction of the query processing time as well. However, queries with low τ_{in} are not really impacted by data partitioning according to confidence degrees because, in this case, almost all input data should be read from HDFS, anyway.

5.2 Querying under constraints on a MapReduce environment

Table 2 shows some experimental results on partitioned and compressed data (as described in Section 5.1) over 6 machines. Our performance is illustrated with a conjunctive query q having 5 joins ($6-8$ after rewriting). $Time_1$ expresses the maximal CPU time spent by one of our cluster machines. Lines $TRew$ and $TSubQuery$ indicate the time needed, respectively, to compose query q_1 (Algorithm 1, line 4) and to generate subsidiary queries (Algorithm 1, line 7). Both are negligible (at least when compared to the number of queries).

Test T_1 applies the two validation steps of Algorithm 1 (rewriting and subsidiary queries). The difference between T_{1_1} and T_{1_2} relies on the number of negative or key constraints. In T_{1_1}, 181 subsidiary queries are necessary for constraint validation, but only 35 are sent to the query evaluator (the others are validated by results in the cache). All other tests apply only the rewriting step (no subsidiary query is needed). As HiveQL does not allow queries with more than one embedded sub-query, tests T_2, T_3, T_4 use at most one negative or key constraint. Tests T_2 and T_3 cannot be done by using subsidiary queries - indeed, q_1 has more than 150000 answers!

	T_{1_1}	T_{1_2}	T_2	T_3	T_4
Number of C_P - C_N - C_K	1-2-2	1-1-1	2-0-0	3-0-0	3-0-1
Minimum confidence degree used	85	85	60	60	60
Time to rewrite ($TRew$)	0.003	0.004	0.003	0.003	0.003
Time for first evaluation ($Time_1$)	3875	3875	3577	4533	5834
Number of answers	46	46	177371	184188	0
Number of subsidiary queries	181	120			
Time to generate subsidiary queries ($TSubQuery$)	0.397	0.334			
Number of sub queries evaluated	35	24			
Time for total sub-queries evaluation ($Time_2$)	3910	2827			
Number of validated answers	4	4			

Table 2. Different steps of query validation and evaluation (time in seconds).

With our MapReduce evaluator, when the answers for q_1 (Algorithm 1) are numerous, the validation via subsidiary queries is not possible. In this situation the rewriting option should be adopted for the whole set of constraints – but this is not always possible due to HiveQL limited expression power. Excluding this situation, the feasibility of our approach with a MapReduce evaluator is proved, specially when only positive constraints are requested (in this case, $\mathcal{C} = \mathcal{C}_{rew}$ in Algorithm 1). In this latter scenario, we can deal with very large databases. Notice also that the expression power of positive constraints is considerable since they include TGD (present in tests T_2, T_3, T_4).

6 Related Work

The system presented in this paper is related to three active research domains, rarely related to each others: (i) the use of Datalog with RDF graphs [3], (ii) consistent query answering [14,13,15] and (iii) the query-rewriting in Ontology-Based Data Access (OBDA) systems [9,14].

Ontological queries (such as in [4,7]) have inspired our query language on graph databases. However, contrary to those work, our positive constraints are traditional database constraints and not the so-called ontological constraints, seen as inference rules. Although proposals such [16] employ the same constraint point of view, our approach is original since constraints are triggered by instantiated atoms in the query body (a step towards efficient treatment of large amount of data, since inconsistency source data are allowed but filtered out from final answers). Key constraints are treated in [13] but an answer is given if it is restricted to query positions not constrained by a key. For instance, the query $q(Z) \leftarrow r(X, Y, Z)$ in the presence of a key constraint $r(X, Y_1, Z_1), r(X, Y_2, Z_2) \rightarrow Y_1 = Y_2$ and data $r(a, b, c)$, $r(a, b', c)$ offers c as an answer. This result is discarded in our approach due to constraint violation. Indeed, for us, answers are issued only from facts (in the query body) not taking part in a constraint violation.

In [14], authors carry out an investigation to deal with inconsistencies for DL-$Lite_{A,id,den}$ by proposing a notion of database-repair that reaches a trade-off between the expressive power of the semantics and the computational complexity of inconsistency-tolerant query answering. Indeed, in DL-$Lite_{A,id,den}$ positive inclusion assertions, denial assertions and identification assertions correspond to our positive constraints, negative constraints and key constraints respectively. They define an AR-repair as a maximally consistent subset of the original ABox of the ontology, and an IAR-repair as the intersection of all AR-repair of a knowledge base. Then they propose an algorithm to rewrite the query under the IAR-repair semantics which allows to obtain the consistent answers to the query without actually computing the IAR-repair. In [15], we find a similar spirit but for query answering over inconsistent Datalog$^{\pm}$ ontologies. The common point between such approaches and ours relies on the idea of living with inconsistencies in databases, but trying to obtain consistent results during query answering. The difference is that our answers are not based on inferred facts and they are computed on the basis of several independent sources. We also take into account data confidence according to provenance which can be parametrized by users.

In [1,2,8,18] we find proposals to deal with unwilling data. Our approach is inspired in [1,2] where a fuzzy datalog is presented. Our originality is to offer a parametrized use of confidence information.

Lastly, experimentations using MapReduce solutions for querying RDF graphs have been performed as shown in [12,6]. The first survey focus on data warehousing context in the cloud, which is again different from our purposes. The second one classifies MapReduce-based solutions in two families: native and hybrid. As most existing systems, our solution belongs to the first class. Our originality is to use the confidence rates to distribute data among the nodes. This highly reduces the numbers of actually queries values.

7 Conclusions and Perspectives

Our query environment offers *personalised contexts* (defined by constraints) and a *declarative* query language over *distributed RDF data* sources not trusted equally. Our constraints generalize functional and inclusion dependencies. All (and only) valid and reliable answers are returned to the user.

Initial tests with data sets from *Berlin* or *DBPedia* proved the feasibility (in several seconds) and usefulness of our proposal. In this paper, for *large* data sets, experimental tests show that: (*i*) our MapReduce query evaluator is well adapted to our approach when only positive constraints are used; (*ii*) the impact of confidence factors on data distribution is considerable. When more than one constraint in sets $\mathcal{C}_N$ or $\mathcal{C}_K$ are needed, other evaluators should be envisaged to treat a big amount of data.

Our long-term goal is a *declarative* query language capable of dealing with standard database queries and graph analysis. This task, the current focus of our work, comprises new operator specification (aggregation, recursion, . . .) and integration to a query evaluator (adapted or extensible for graph analysis).

References

1. Á. Achs. Computed answer from uncertain knowledge: A model for handling uncertain information. *Computers and Artificial Intelligence*, 26(1):63–76, 2007.
2. Á. Achs and A. Kiss. Fuzzy extension of datalog. *Acta Cybern.*, 12(2):153–166, 1995.
3. M. Arenas, G. Gottlob, and A. Pieris. A datalog-based language for querying rdf graphs. In *AMW*, volume 1644, 2016.
4. 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.
5. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI ' 04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.
6. J. M. Giménez-Gárcia and M. A. M. Javier D. Fernández. Mapreduce-based solutions for scalable SPARQL querying. *Open Journal of Semantic Web*, 1(1):1–18, 2014.
7. F. Goasdoué, V. Lattès, and M. Rousset. The use of CARIN language and algorithms for information integration: The PICSEL system. *Int. J. Cooperative Inf. Syst.*, 9(4):383–401, 2000.
8. G. Gottlob, T. Lukasiewicz, M. V. Martinez, and G. I. Simari. Query answering under probabilistic uncertainty in datalog+ / - ontologies. *Ann. Math. Artif. Intell.*, 69(1):37–72, 2013.
9. G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 2–13, 2011.
10. M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in datalog programs. *J. Log. Comput.*, 8(6):745–775, 1998.
11. M. A. H. Hassan, M. Bamha, and F. Loulergue. Handling data-skew effects in join operations using mapreduce. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 145–158, 2014.
12. Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.
13. P. G. Kolaitis, E. Pema, and W. Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.
14. D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo. Inconsistency-tolerant query answering in ontology-based data access. *J. Web Sem.*, 33:3–29, 2015.
15. 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.
16. E. Prud'hommeaux, J. E. L. Gayo, and H. R. Solbrig. Shape expressions: an RDF validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems, SEMANTICS 2014, Leipzig, Germany, September 4-5, 2014*, pages 32–40, 2014.
17. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.
18. G. Stoilos, N. Simou, G. B. Stamou, and S. D. Kollias. Uncertainty and the semantic web. *IEEE Intelligent Systems*, 21(5):84–87, 2006.