Undefined 1 (2013) 1–26
IOS Press

Data Access over Large Semi-Structured Databases

A Generic Approach Towards Rule-Based Systems

Editor(s): Name Surname, University, Country
Solicited review(s): Name Surname, University, Country
Open review(s): Name Surname, University, Country

Bruno Paiva Lima da Silva [a], Jean-François Baget [b] and Madalina Croitoru [a,*]
[a] *University of Montpellier 2, 2 Place Eugene Bataillon, 34095 Montpellier Cedex 5, France*
E-mail: {bplsilva,croitoru}@lirmm.fr
[b] *INRIA, 2004 Route des Lucioles, 06902 Sophia-Antipolis, France*
E-mail: baget@lirmm.fr

Abstract. Ontology-Based Data Access is a problem aiming at answering conjunctive queries over facts enriched by ontological information. There are today two manners of encoding such ontological content: Description Logics and rule-bases languages. The emergence of very large knowledge bases, often with unstructured information has provided an additional challenge to the problem. In this work, we will study the elementary operations needed in order to set up the storage and querying foundations of a rule-based reasoning system. The study of different storage solutions have led us to develop ALASKA, a generic and logic-based architecture regrouping different storage methods and enabling one to write reasoning programs generically. This paper features the design and construction of such architecture, and the use of it in order to verify the efficiency of storage methods for the key operations for RBDA, storage on disk and entailment computing.

Keywords: Ontology-Based Data Access, Knowledge Representation

1. Introduction

Knowledge Representation (KR) is one of the basic issues in Artificial Intelligence (AI) research. In order to create applications that are capable of intelligent reasoning, human knowledge about an application domain has to be encoded in a way that can be handled by a problem-solving computing process. Representing knowledge inside the machine has proved to be a non-trivial task. The main difficulty is to have a way to constrain and to make *explicit* the intended conceptual models of a KR formalism, in order to facilitate large-scale knowledge integration and to limit the possibility of stating something that is reasonable for the system but not reasonable in the real world [16].

In this paper we are interested in a particular subset of positive, existential fragment of First Order Logic (FOL) expressed using a rule-based language [11]. This language can be encoded in several manners and the encoding will impact the efficiency of storage and querying mechanisms of the language. Despite the importance of the task, a throughout analysis of how the language encoding affects storage and querying is non existing in the literature.

The problem addressed in this paper is the ONTOLOGY-BASED DATA ACCESS (OBDA) problem. The problem consists in, given a knowledge base containing facts and ontological data, and a conjunctive query, to

[*]Corresponding author. E-mail: croitoru@lirmm.fr.

determine whether there is an answer to the conjunctive query in the knowledge base. There are currently two distinct manners to represent ontological data: description logics languages and rule-based languages.

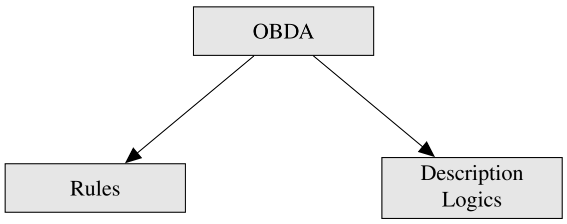

Fig. 1. Approaches for the OBDA problem.

The contribution of this paper is the integration of different implementation approaches under the same framework unified by the means of a common logical vision, under a generic architecture (ALASKA) that communicates to different storage systems through an unified language and to use such generic software architecture to provide practical means for enabling the storage and querying of large knowledge bases on disk avoiding out of memory limitations.

1.1. Paper Structure

The paper is organized as follows. Section 2 presents the problems and the motivation of the paper. Section 3 describes the ALASKA platform and the systems integrated within it for experimental work. Section 4 illustrates how ALASKA works through the means of an example. Section 5 features the use of ALASKA to investigate storage efficiency of different stores, while Section 6 features the use of ALASKA for querying purposes. Section 7 closes the paper with a discussion about future work on querying large knowledge bases using constraint satisfaction techniques.

1.2. Problem overview

The RULE-BASED DATA ACCESS (RBDA) problem, derived from ONTOLOGY-BASED DATA ACCESS [22] knows today an interest in knowledge systems allowing for expressive inferences. In its basic form, its input consists in a set of facts, an ontology and a conjunctive query, and the problem consists of finding if answers to the query can be deduced from the facts, eventually using inferences allowed by the ontology. This deduction mechanism could either be done (1) previous to query answering by fact saturation using the ontology (forward chaining) or (2) by rewriting the query according to the ontology and finding a match of the rewritten query in the facts (backwards chaining).

Let us consider a knowledge base F that consists of a set of logical atoms, a set of rules $\mathcal{R}$ written in some (first-order logic) language, and a conjunctive query Q. The RBDA problem stated in reference to the classical *forward chaining* scheme is the following: “Can we find an answer to Q in a database F' that is built from F by adding atoms that can be logically deduced from F and $\mathcal{R}$?”

Since forward chaining schemes can unreasonably increase the size of the database (and thus cannot be relied upon when considering very large knowledge bases), some algorithms use a *backward chaining scheme* (or query rewriting) for query answering. In that case, the set of rules $\mathcal{R}$ (and sometimes the database itself F, leading to a complexity increase) is used to build from Q a rewritten query Q' (that is often a disjunction of conjunctive queries), such that there exists an answer to Q in F' if and only if there exists an answer to Q' in F.

There are today two major approaches in order to represent an ontology for the OBDA problem. The first one are the Description Logics. Description Logics are families of languages used for defining concepts. Based upon constructors, the expressivity of the languages comes from the combination of these constructors. Description Logics allow today for very large expressivity. However, such expressivity is responsible for exponential blow-up when answering conjunctive queries. In order to be able to answer conjunctive queries, people have defined and studied "lite" description logics, which are less expressive but in which conjunctive query answering is decidable (e.g. $\mathcal{EL}$([3]) and DL-Lite [13] families).

The second method is to represent the ontology via inference rules. Recent works consider the Datalog$^+$ [11] language to encode a generalization of Datalog that allows for existentially quantified variables in the head of the rules. Such capacity is also responsible for undecidability when answering conjunctive queries. Works have focused on identifying and developing algorithms for particular fragments of Datalog$^+$ that are decidable [12,8]. For this reason we will focus on this problem in this work.

While above work focuses on logical properties of the investigated languages, existing approaches employ a less principled approach when implementing such frameworks. It is well known [2,14] that encoding such languages can be equivalently done using (hyper)graphs or relational databases. However, none of the existing systems make this possibility of different encodings explicit. The choice of the appropriate encoding is left to the knowledge engineer and it proves to be a crafty task.

2. Context

2.1. Fundamental Notions

2.1.1. Facts

Syntax The syntax of the logical language we use is the following: we consider constants but no other functional symbols. In order to represent a knowledge base, a vocabulary has to be defined. A vocabulary W is composed of a set of predicates P and a set of constants C. Constants are tokens that identify the individuals in the knowledge base, while predicates represent relations between such individuals. We also consider X, a set of variables in the knowledge base.

Definition 2.1 (Vocabulary) *Let C be a set of constants and P a set of predicates. A vocabulary is a pair $W = (P, C)$ and $arity$ is a function from P to $\mathbb{N}$. For all $p \in P$, $arity(p) = i$ means that the predicate p has arity i.*

We will also consider an infinite set X of variables, disjoint from P and C. Hence, an atom on W is of form $p(t_1,...,t_k)$, where p is a predicate of arity k in W and the t_i are constants in W or variables. A term is an element of $C \cup X$. For a given atom A, we note $terms(A)$, $csts(A)$ and $vars(A)$ respectively the terms, constants and variables occurring in A.

Definition 2.2 (Fact) *A fact is a finite, but possibly empty, set of atoms on a vocabulary. For a given fact F, we note $atoms(F)$ the atoms occurring in F.*

Example Let us consider a vocabulary $W = (P, C)$. P = {man,woman}, $C = \{Bob, Alice\}$ and $arity$ = {(man,1),(woman,1)}. $man(Bob)$ and $woman(Alice)$ are two distinct atoms on W, and $F = \{man(Bob), woman(Alice)\}$ is a fact on W.

Semantics

Definition 2.3 (Interpretation) *Let $W = (P, C)$ be a vocabulary. An interpretation of W is a pair $I = (\Delta, .^I)$ where Δ is the domain of the interpretation, and $.^I$ a function where: $\forall$ c in C, $c^I \in \Delta$ and $\forall$ p in P, $p^I \subseteq \Delta^{arity(p)}$.*

An interpretation is non empty and can be possibly infinite.

Definition 2.4 (Model) *Let F be a fact on W, and $I = (\Delta, .^I)$ be an interpretation of W. We say that I is a model of F iff there exists an application v : $terms(F) \rightarrow \Delta$ (called a justification of F in I) such that:*

- *$\forall$ $c \in csts(F)$, $v(c) = c^I$ and*
- *$\forall$ $p(t_1,...,t_k) \in atoms(F)$, $(v(t_1),...,v(t_k)) \in p^I$.*

Definition 2.5 (Fact to logical formula) *Let F be a fact. $\phi(F)$ is the logical formula that corresponds to the conjunction of atoms in F. And $\Phi(F)$ corresponds to the existential closure of $\phi(F)$.*

Example Let us consider a fact $F = \{person(x), name(x, Bob), age(x, 25)\}$.

- $\phi(F) = person(x) \wedge name(x, Bob) \wedge age(x, 25)$.
- $\Phi(F) = \exists x\, person(x) \wedge name(x, Bob) \wedge age(x, 25)$.

Property 2.1 (Model equivalence) *Let F be a fact and I be an interpretation of W. Then I is a model of F iff I is a model (in the FOL sense) of $\Phi(F)$.*

Definition 2.6 (Entailment) *Let F and G be two facts, F entails G if every model of F is also a model of G. The entailment relation is then noted $F \models G$.*

Computing

Definition 2.7 (Homomorphism) *Let F and F' be facts. Let σ: $terms(F) \rightarrow terms(F')$ be a substitution, i.e. a mapping that preserves constants (if $c \in C$, then $\sigma(c) = c$). We then note $\sigma(F)$ the fact obtained from F by substituting each term t of F by $\sigma(t)$. Then σ is a homomorphism from F to F' iff the set of atoms in $\sigma(F) \subseteq F'$.*

Example Let $F = \{man(x_1)\}$ and $F' = \{man(Bob), woman(Alice)\}$. Let $\sigma : terms(F) \rightarrow terms(F')$ be a substitution such that $\sigma(x_1) = Bob$. Then σ is a homomorphism from F to F' since the atoms in $\sigma(F)$ are $\{man(Bob)\}$ and the atoms in F' are $\{man(bob), woman(Alice)\}$.

Property 2.2 (Entailment) *Let F and Q be facts. $F \models Q$ iff there exists Π an homomorphism from Q to F.*

Complexity The entailment problem is a NP-Complete problem. However, there exist polynomial cases of the problem, such as when Q has a tree structure. See [14,17] for polynomial subclasses.

2.1.2. Rules

Syntax Rules are objects used to express that some new information can be inferred from another information. Rules are built upon two different facts, and such facts correspond to the two different parts of a rule, called head and body. Once the body of a rule can be deduced from a fact, then the information in the head should also be considered when accessing information.

Definition 2.8 (Rule) *Let H and B be facts. A rule is a pair R = (H, B) of facts where H is called the head of the rule and B is called the body of the rule. A rule is commonly noted $B \rightarrow H$.*

Semantics

Definition 2.9 (Rule model) *Let W be a vocabulary, I an interpretation on W, and R a rule on W. We say that I is a model of R iff for every justification V_B of B in I there exists a justification V_H of H in I such that $\forall t \in vars(B) \cap vars(H)$, $V_B(t)$ = $V_H(t)$.*

Definition 2.10 (Rule to logical formula) *Let R = (H, B) be a rule. Let b_x be the variables from B, and h_x be the variables from H that are not in B, the logical formula corresponding to R is the following: $\Phi(R)$ = $\forall b_x$ ($\phi(B) \rightarrow \exists h_x$ $\phi(H)$).*

Example Let us consider a rule R = $\{person(x), person(y), sibling(x, y)\} \rightarrow \{person(z), parent(x, z), parent(y, z)\}$. $\Phi(R) = \forall x, y($ $person(x) \wedge person(y) \wedge sibling(x, y) \rightarrow \exists z$ $person(z) \wedge parent(x, z) \wedge parent(y, z)$).

Property 2.3 (Model equivalence) *Let R be a rule and I be an interpretation of W. Then I is a model of R iff I is a model (in the FOL sense) of $\Phi(R)$.*

Definition 2.11 (Knowledge base) *Let W be a vocabulary. A knowledge base (KB) is a pair K = $(F,\mathcal{R})$ where F is a fact on W and $\mathcal{R}$ is a set of rules on W.*

Definition 2.12 (KB model) *Let K = $(F,\mathcal{R})$ be a knowledge base and I be an interpretation. I is a model of K iff I is a model of F and also a model of every rule R_i in R.*

Definition 2.13 (Entailment) *Let K be a knowledge base and Q be a fact. K entails Q iff all models of K are also models of Q.*

```
Algorithm 1: Rule-Deduction
  Input: K a knowledge base, Q a fact
  Output: TRUE if all the models of K are also models of Q
```

The RULE-BASED DATA ACCESS is defined as the following:

Definition 2.14 (Logical representation) *Let K = $(F, \mathcal{R})$ be a knowledge base. $\Phi(K)$ = $(\Phi(F),\Phi((R)))$ is the logical representation of K. $\Phi(F)$ is the logical formula of F and $\Phi(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \Phi(r)$.*

Property 2.4 (Model equivalence) *Let K be a knowledge base and I be an interpretation. Then I is a model of K iff I is a model (in the FOL sense) of $\Phi(K)$.*

In other words, given a knowledge base K and a conjunctive query Q, the RBDA problem consists in answering if Q can be deduced from K, denoted $K \models Q$.

Algorithms Rule application can be performed of two different methods, called FORWARD CHAINING and BACKWARDS CHAINING.

Forward chaining

Definition 2.15 (Applicable rule) *Let R = (H, B) be a rule and F be a fact. R is applicable to F if there exists an homomorphism $\Pi : B \rightarrow F$. In this case, the application of R to F according to Π is a fact $\alpha(F, R, \Pi)$ = $F \cup \Pi^{safe}(H)$.*

Please note the use of Π^{safe} instead of Π. Π^{safe} is an application that converts existential variables into fresh ones at the moment of joining new information with the initial fact. Such process is important in order to avoid unnecessary specializations. A derivation is the result of a finite sequence of rules application.

Definition 2.16 (Derivation) *Let F be a fact. F' is a derivation of F iff there exists a finite sequence of facts F = F_0, ..., F_k = F' (called the derivation sequence) such that for every i there exists R and Π such that F_i = $\alpha(F_{i-1}, R, \Pi)$.*

Definition 2.17 (Saturation) *Let F be a fact and R be a set of rules. $\Pi_R(F)$ = $\{\Pi{:}B_R \rightarrow F\}$ is the set of homomorphisms of the body of applicable rules to F. $\alpha(F, R)$ = $F \bigcup_{\pi \in \Pi_R(F)} \pi^{safe}(H_R)$ is the result of the application of all those rules. The saturation of a fact is the process of applying rules from the initial fact until no more new information can be added to the fact via rule application. Let the initial fact F_0 = F, and F_i = $\alpha(F_{i-1}, R)$, a fact is saturated when $F_i \equiv F_{i+1}$.*

Theorem 2.1 (Equivalence) *Let $K = (F, R)$ be a knowledge base and Q be a fact. The following assertions are all equivalent:*

- $K \models Q$
- *there exists a derivation F ... F' such that $F' \models Q$*
- *there exists an $n \in \mathbb{N}$ such that $F_n, R \models Q$*

Backwards chaining As opposed to forward chaining which enriches the facts with the rule application, the backwards chaining rewrites the initial query in a union of several new queries. The decomposition is obtained by applying rules on the query, i.e. by seeing which rule could have generated the query and from which fact. All possibilities are kept and further decomposed until the initial set of facts is reached or all possibilities are examined. The backwards chaining approach does not enrich the facts but works on the query.

One of the motivations of the ALASKA features is the big addition of facts due to forward chaining rule application. This is the reason why in this work we do not focus on backwards chaining. In the following we will simply define what a backwards chaining decomposition is and then the reader is invited to further consult works cited below on backwards chaining.

Definition 2.18 (Backwards chaining) *Let Q be a fact and R a set of rules. We denote $B(Q, R) = \{Q_i \mid \forall F,\ (F, R) \models Q$ iff $\exists\ Q_i \in B(Q, R)$ such that $F \models Q_i\}$.*

The work of [23], corrected by [9], and adapted to First Order Logic in [7] provides such a rewriting.

Complexity and decidability The complexity of RBDA may vary according to the set of rules present in the ontology. When there are no rules in the ontology, the problem is then equivalent to homomorphism computation, which is a NP-complete problem.

In the presence of rules, the problem is undecidable. Both the forward chaining and backwards chaining mechanisms are not certain of halting. This is easy to verify through the means of very simple examples.

Forward chaining Let K = (F,R) be a knowledge base, $F = person(Bob)$, and $R = \{\{person(x)\} \rightarrow \{parent(y, x), person(y)\}\}$.

Let $Q = \{parent(x, Tom)\}$ be a fact. Asking a forward chaining mechanism if Q can be deduced from K may eventually never stop. The mechanism will first verify if Q can be deduced from F, if there is an x having Tom as parent in F. As it is not the case, rules will be applied and F will be enriched into F' = $\{person(Bob)$,$parent(p_1, Bob)$,$person(p_1)\}$. The mechanism will then verify if Q can be deduced from F'. As it is still not the case, it will once again apply rules and enrich F' into F". And it will do it infinitely as in this case, no answer will be ever found to the query.

Backwards chaining Let K = (F,R) be a knowledge base, and $R = \{\{p(x, y)$,$p(y, z)\} \rightarrow \{p(x, z)\}\}$.

Let $Q = \{p(a, b)\}$ be a fact. Asking a backwards chaining mechanism if Q can be deduced from K may also eventually never stop. The mechanism will first verify if $\{p(a, b)\}$ can be deduced from F. If that is the case, the mechanism will stop. Otherwise, it will rewrite the initial query Q into a new query $Q_1 = \{p(a, x_0)$,$p(x_0, b)\}$. Q is deduced from K if Q_1 is deduced from K. If Q_1 can not be deduced from F, the mechanism will rewrite the query again, for example with $Q_2 = \{p(a, x_0)$,$p(x_0, x_1)$,$p(x_1, b)\}$. Such sequence of rewritings may never end. Any finite rewriting corresponds to a finite sequence, for example, of length k of form $\{p(a, x_0) \ldots p(x_k, b)\}$. The facts could always contain a sequence of length $k + 1$.

Works [6] have identified classes of rules in which the RBDA problem is decidable. One of those is the class of Range-Restricted rules. Such class does only contain rules in which no existential variable is present in the head of the rule, in other words, when no new variable is created upon rule application. They are part of the Finite Expansion Set class, a class in which the saturation process is finite. Another classes of rules in which the problem is decidable are the FUS (Finite Unification Sets) and BTS (Bounded Treewidth Sets) classes. The rules of BTS class have the particularity of generating and infinite fact with an arborescent structure, while the ones in FUS have the particularity of generating a finite sequence of query rewritings. See [25] for further details. Those classes will appear once again in the next section, where a comparison between RBDA and other languages will be established.

2.2. *Problem Evolution*

Derived from the ONTOLOGY-BASED DATA ACCESS problem, RBDA knows today a totally renewed interest due to the recent evolutions in the KR and in the Information Technology (IT) field in general. Information sources have became larger and larger, reaching sizes that one can not load entirely in a system's main memory. Information has also became more and more semi-structured [1], which leads to a different challenge according to the manner one intends to query a knowledge base. Such emergence of semi-structured knowledge bases has led to the emergence of non-relational database models that fit best such kind of information structure.

2.2.1. *Large KBs*

It is very difficult not to see in the current context the continuous growth of the size of datasets. Big-Data popularity has now became important, not only in the industrial scope but also in the academic scope. Domains such as social networks and Semantic Web are responsible for the emergence and treatment of a very large quantity of data. It is sometimes difficult to quantify the sizes of certain of those knowledge bases. In academia, projects such as DBPedia [1], UniProt [2] and GeoNames [3] are well known for having very large amount of information. A consequence of such increase in the size of the data sources is also the emergence of the XLDB acronym, for eXtremely Large Databases, in order to replace/complete the former VLDB one (for Very Large Databases).

In the whole document, we consider as large, every knowledge base that can not be entirely loaded in main memory when an application is run. Our work will focus only in knowledge bases stored on disk and accessible via reading and writing interfaces.

2.2.2. *Semi-structured*

Semi-structured data is data that is not raw data, neither data structured according to well defined schema. In [1], several types of semi-structured data are described. Most of the large data sources cited above contain a large amount of semi-structured data, known for their evolutive or undetermined schema. Are also part of semi-structured data the data in which it is impossible (or at least very hard) to dissociate the schema from the data itself.

This kind of data is currently very widely spread inside Open Data, Linked Data [10] and data mining fields, where the integration of heterogeneous data sources is often needed. Classical relational databases have already shown limited efficiency when dealing with those kind of data. The emergence of databases based upon different data models introduces a new interest in verifying whether those new databases are most suited for semi-structured data than the classic (relational) one.

2.2.3. *NoSQL*

The idea of NoSQL was born around 1996 when Carlo Strozzi, unhappy with the performances of relational databases has started developing a database management system that would be based upon the relational model of Codd, but in which the querying would not be performed via an SQL interface (hence the NoSQL name). This project was finally a failure, and, when it was discontinued, Strozzi stated that the performance issues he blamed on the databases at that time was not related to the querying interface itself, but rather to Codd's relational model [15].

A few years later, several databases appeared using different data models. The NoSQL movement is considered today as the regroupment of many database management systems using a data model other than Codd's relational model. Because of that, the question of the use of SQL as querying interface is not even major anymore, as some of those management systems does not feature a SQL interface, and other feature it along with another querying mechanism. In the list of the most known elements of NoSQL, one will find the column-oriented databases, document-based databases, XML databases and graph databases.

2.3. *Novelty and motivation*

After having defined the technical characteristics and challenges of the problem, we are able to set as a goal to obtain a system that would be able to perform conjunctive queries over knowledge bases of any size and of any structure. Those knowledge bases could be stored in main memory but the case we intend to focus on is when the knowledge base is located in a secondary device. In order to reach that goal, we will first study the approaches and methods that already exist for storing and querying information. The study of those systems will be based on their ability to answer the following questions:

1. - Can it be used for representing a knowledge base in our formalism?
2. - Can it be used for computing entailment?

[1]http://dbpedia.org
[2]http://www.uniprot.org/
[3]http://www.geonames.org/

3. - Can it be used for computing entailment with rule application?

Figure 2 summarizes the capacities of each approach.

-	KB Rep.	$F \models Q$	$F, \mathcal{R} \models Q$	Sec. Mem.
Prolog	Yes	Native	Native	No
CoGITaNT	Yes	Native	Native	No
Rel. Databases	Yes	Native (SQL)	Not native	Yes
Triple Stores	Yes	Native (SPARQL)	Not native	Yes
Graph Databases	Yes	Not native	Not native	Yes

Fig. 2. Table comparing the features of the studied methods.

Prolog and CoGITaNT are the only systems previously described that integrate a native support of rules. Unfortunately, both systems require the load of the whole knowledge base in main memory prior to executing reasoning processes (a characteristic that we agreed to avoid once we focus on studying the efficiency of reasoning activity of systems/algorithms when dealing with very large amount of data). For those reasons, we will not consider both systems for the future.

On the other hand, relational databases are widely known for their ability of managing information stored in secondary memory. They also feature a native SQL interface which is able to perform conjunctive queries over the facts. Rules are not natively featured but can be introduced upon it. An interesting subject of study will be the efficiency of the SQL interface when performing queries over semi-structured data. As querying a relational database is not restricted to SQL, we will also be able to write a custom algorithm for computing homomorphisms and verify its efficiency against the SQL interface. Those aspects will be covered and detailed later on in this paper.

The case of triples stores is similar to the one of relational databases. Most implementations of those stores feature a native SPARQL interface in order to perform conjunctive queries. Graph databases however do not always feature a native querying interface and in those cases writing a querying algorithm is required in order to perform conjunctive queries over the data. Graph homomorphism algorithms have already been used before on data stored in main memory. The focus of this work will then be to check their efficiency when dealing with data stored on disk.

Based on these conclusions, we propose a software platform that will first serve as a testing suite for all the systems that were not discarded after this first analysis. Our work will focus in transforming and translating information in order to have a common data language which is compatible to the formalism presented in Section 2.1 and shared by all the connected storage systems. To this end, in the long term, we aim creating a multi-layered architecture featuring an abstract layer composed of classes and interfaces that would act as a physical representation of the positive subset of First Order Logic we work with. Such layer would ensure the equivalence between the operations on the data no matter which is the data model of the storage system that holds the information.

3. ALASKA

3.1. Introduction to ALASKA

As explained in Section 2.3, different approaches have led to different methods to implement software systems able to perform conjunctive query answering over a knowledge base. Such approaches being with, or without, the presence of an ontology to enrich such knowledge base. Although the description of the problem itself has remained the same, several new factors have altered its current nature and, consequently, the manner to proceed in order to address it. In the list of the main factors that have led to this situation, the emergence of very large and unstructured knowledge bases. Setting the threshold that defines what is a very large knowledge base may be a tricky task. We have defined by "very large" a knowledge base containing an amount of information that cannot be entirely stored in one single machine main memory at any part of the process. We tend to consider large knowledge bases containing information going from approximatively 10 million triples up to 1 billion triples (and more...). This list of factors also features the emergence of different database management systems using different data models than the traditional relational one (See the NoSQL movement in Section 2.2.3).

As previously mentioned, such factors have led the existing methods to fail, or at least to become obsolete due to inefficiency. Although the idea of having a working software suite to allow users to perform conjunctive queries over knowledge bases stored in secondary memory is not new, there is still no tool able to give a more in-depth analysis of the previous failures while, at the same time, integrate and test the latest algorithms and storage technologies. The lack of such a tool has has led us to study how to proceed in order to develop a tool that would enable a better study of the RBDA problem. The study has became the starting point of ALASKA [18], a project that would enable users to store pieces of information in predefined encodings directly into secondary memory, without having to perform any manual manipulations. ALASKA, an acronym for an "**A**bstract and **L**ogic-based **A**rchitecture for **S**torage and **K**nowledge bases **A**nalysis" would realize such task of translating information from one encoding to another in a totally transparent manner for the user, using First Order Logic as intermediary language. Once the information is stored, ALASKA would also work as a querying interface, allowing one to query the stored information using queries presented in different querying languages.

Although efficient storage and querying is the aim of the work that has resulted in the birth of ALASKA, it is also fundamental to state that its functionalities and features should not be limited to the study of RBDA. The ability of using logical representations internally and having an easy and simple connection to the stores on disk could directly enhance performances of other studies in the KR field. Some applications, such as rule application and record linkage studies or any other application requiring logical operations over large amounts of data may benefit from the features of ALASKA.

In order to adapt to the different types of applications aiming to manage or manipulate large amount of information, ALASKA will not only feature a range of options of storage methods an user can choose, but also a range of readers and parsers, able to transform different types of data as input into its internal logical representation before being stored on disk. By doing this, ALASKA can be considered a software enabling users and programmers to store and manage their information into different aspects without having to reach and manipulate the data itself.

The choice of using JAVA as language for the platform is based on several aspects. Even if the fact of being run inside a virtual machine (VM) makes code execution less efficient, JAVA has become more and more popular and is today the best choice when one wants to easily integrate libraries and other pieces of code into another project (such as ALASKA does). Also, a lot of storage systems are currently written in JAVA, either come with a JAVA client API. In this trade-off where a *wide* investigation of existing storage systems (integrating many relevant systems) is opposed to a *in-depth* one (enhancing the efficiency of ALASKA in certain circumstances), we have favored the *wide* choice. This choice is also motivated by the priority of our research group to have a fully functional system able to compare and address heterogeneous sources, and not necessarily to create the fastest system for RBDA.

3.2. Foundations of ALASKA

The ALASKA core (data structures and functions) is written independently of any language used by storage systems it will access. The advantage of using a subset of First Order Logic to maintain this genericity is to be able to access and retrieve data stored in any system by the means of a logically **sound** common data structure. **Local encodings** will be transformed and translated into any other **representation language** at any time. The operations that have to be implemented are the following: (1) retrieve all the terms of a fact, (2) retrieve all the atoms of a fact, (3) add a new atom to a fact, (4) verify if a given atom is already present in the facts.

Basically, the abstract layer of ALASKA is a logical layer. Storage systems are used to store data that can be seen as sets of logical atoms of form $p(t_1, \ldots, t_k)$. Wrappers are used to encode this atom according to the storage system paradigm. For instance, this atom will be encoded as the line $(t_1, \ldots, t_k)$ in the table p in a relational database, and as a directed hyperedge labeled p whose incident nodes are the ones encoding respectively $t_1, \ldots, t_k$ in a graph-based storage system.

Whatever this storage system, ALASKA only reads and writes atoms or sets of atoms. It is thus entirely possible, for instance, to read the RDF triples (s, p, o) stored in Jena (they will be seen as atoms $pred(s, p, o)$), and write them in a SQL database, where they will be stored as lines (s, o) in the table p.

This abstract layer is not only used to read and write in an uniform manner into various storage systems, but also to process queries. A conjunctive query can also be seen a set of atoms. ALASKA is able to transform them into, for example, SQL or SPARQL queries, to benefit from the native querying mechanism of specific storage systems. Moreover, a generic backtracking algorithm has been designed, that allows to process these queries on any of these storage systems. This backtrack relies upon elementary queries, that check whether or not a grounded atom is stored in the system, or enumerate all atoms that specialize a given one. This backtrack does not incorporate powerful optimizations and pruning features, since it is designed to process simple queries. For more difficult queries, a constraint solver, based upon Choco [21], relies upon the same elementary queries (this constraint solver is currently under evaluation).

Though ALASKA is designed as a generic platform for RBDA (Rule-Based Data Access), it does not yet integrate any ontological reasoning features. We have designed this platform to be fully compatible with existential rules (also known as Datalog+/-) [11]. These rules are powerful enough to encode the semantics of RDF(S), or "lite" description logics families such as DL-Lite or $\mathcal{EL}$ families. There is an ongoing work aiming to integrate such families in ALASKA.

The platform architecture is multi-layered. Figure 3 represents its class diagram, highlighting the different layers.

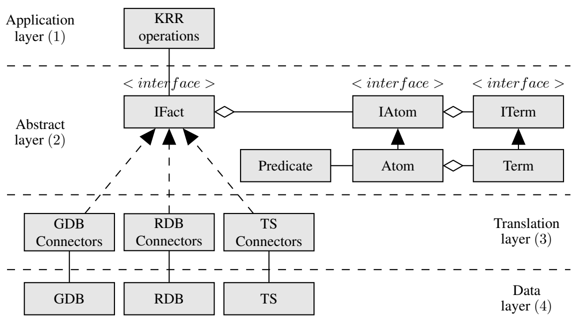

Fig. 3. Class diagram representing the software architecture.

The first layer is (1) the **application** layer. Programs in this layer use data structures and call methods defined in the (2) **abstract** layer. Under the abstract layer, the (3) **translation** layer contains pieces of code in which logical expressions are translated into the languages of several storage systems. Those systems, when connected to the rest of the architecture, compose the (4) **data** layer. Performing higher level KRR operations within this architecture consists of writing programs and functions that use exclusively the formalism defined in the abstract layer. Once this is done, every program becomes compatible to any storage system connected to architecture.

3.3. Storage systems

By having several different stores connected to ALASKA, one of our main contributions of the software is to **help make explicit different storage system choices for the knowledge engineer in charge of manipulating the data**. More precisely, ALASKA can encode a knowledge base expressed in the positive existential subset of first order logic in different data structures (graphs, relational databases, 3Stores etc.). This allows for the transparent performance comparison of different storage systems.

The performance comparison is done according to the (1) storage time relative to the size of knowledge bases and (2) query time to a representative set of queries. On a generic level the storage time need is due to forward chaining rule mechanism when fast insertion of new atoms generated is needed in an application (described in [5]). We also needed to interact with a server of ABES, the French Higher Education Bibliographic Agency, for retrieving their RDF(S) data. The server answer set is limited to a given size. When retrieving extra information we needed to know how fast we could insert incoming data into the storage system. Second, the query time is also important. The chosen queries used for this study are due to their expressiveness and structure. Please note that this does not affect the fact that we are in a semi-structured data scenario. Indeed, the nature of ABES bibliographical data with many information missing is fully semi-structured. The generic querying allowing for comparison is done across storage using a generic backtrack algorithm to implement the backtracking algorithm for subsumption. ALASKA also allows for direct querying using native data structure engines (SQL, SPARQL).

In this work, we have given the priority to the integration of embeddable storage systems within ALASKA. By embeddable, we mean that the storage system core is packed and is distributed inside the main application using the store. This kind of system is opposed to the stores featuring the client-server architecture, where the client connects to the server through a port and asks for the server to perform the operations it wants, while the server takes care of the physical management of the data requested. Client-server stores is the most appropriated choice when one is interested in deploying the content of the database to several clients or throughout a network. Very often, an embeddable system is a simple solution for an application in need of simple and direct access (non-secured very often) to the data. In this case, the system features an API with the functions needed to manage the data directly without calling a third-party server.

In some cases it may happen that a store is deployed as a client-server architecture, but both client and server are located on the same host. In this case, no network communication is needed. Such solution may bring an efficiency loss when interacting with the storage system. This is the case for instance when one executes multiple operations that can not be regrouped over a database via a JDBC driver. The communication with the driver, then with the server before having access to the information is responsible for slowdowns when executing the program.

Next section will feature short descriptions of the current embeddable stores that were connected to ALASKA and were also used for the experimental aspects of this paper, featured in Sections 5.2 and 6.2. This list is not final. Others systems were also connected and tested but are not part of the work in this paper. Also, there are also other stores in which the writing of a connector is needed in order to use it within ALASKA. The stores featured in this work are the following:

→ **Relational databases:** Sqlite
→ **Graph databases:** Neo4J, DEX
→ **Triples stores:** Jena TDB

3.3.1. *Relational databases*

SQLite

Sqlite SQLite[4] is the embedded relational database used within ALASKA. It is a very lightweight relational database management system written in C. SQLite is ACID-compliant, which means that it supports transactions natively. and implements most of the SQL standard. The fact that it is an embedded store has made SQLite be very popular in the world of mobile and desktop applications. Plenty of different programs embed SQLite and use it for internal storage of data and options (cf. Mozilla Firefox). This way, it differs itself from the "larger" relational database management systems, which are deployed on servers and are very widely known for their use on business and industrial applications (MySQL, Oracle, etc.). It was possible to connect SQLite to ALASKA thanks to the existence of a JDBC driver for SQLite. Such driver was not provided by the SQLite developers themselves but by a third-party project.

3.3.2. *Graph databases*

The following graph databases have been used within ALASKA:

Neo4j
the graph database

Neo4J Neo4J[5] is a graph database, and is by far the most popular graph database at the moment, at least in the industrial world. Started as a small project, Neo4J has seen its popularity being multiplied by a huge factor in the recent years. It is now distributed under two distinct versions: Neo Technologies offers now a commercial version of the database along with support, while the former open source version of the database is still available via a "Community" version still maintained by the core developers of Neo and using the help of a large community of developers that have embraced the Neo4j project. It is fully implemented in JAVA. It is an embedded store but it is absolutely possible to deploy the store on a server dedicated to store data and handle queries from distinct users. The internal structure of Neo4j is based on the Property Graph model. It is also ACID-compliant, which means that it ensures the properties defined by Codd for relational databases (Atomicity Consistency Isolation Durability), that ensures the reliability of data transactions. For that, it features a fully transactional persistence engine that secure transactions and data manipulation throughout time.

*dex

[4]http://www.sqlite.org/

[5]http://www.neo4j.org/

DEX DEX[6] is a graph database management system written in Java and C++. It was originated by the research carried out at the DAMA-UPC research group in Barcelona. Since 2010 is maintained, upgraded and distributed by Sparsity Technologies. DEX is licensed under a proprietary license of Sparsity Technologies, but is free for academic use. It is another graph database that has been seeing a strong success both in academia and business world. DEX internal model of a graph is fully compatible with the Property Graph. The internal representation of a graph in DEX is very particular, as the graph is often partitioned according to the graph structure, and also because of the fact that information such as nodes and edges identifiers are regrouped and encoded into large bitmaps. The major strength of DEX is the ability of having very fast reading and writing operators at lower level able to manipulate those large bitmaps with great efficiency. At higher level, the system features all the operations needed in order to enable the user to manage a graph stored by DEX as any Property Graph. Unlike Neo4J, DEX is not ACID-compliant.

3.3.3. Triples stores

The following triples stores have been used within ALASKA:

Jena Jena[7] is a Semantic Web framework for Java. It provides an API to read data from RDF content, to manage it, query it and then to write such content in different formats. The project also provides an internal embedded Triple Store, which is called TDB. This is the part of the framework we will be connecting to ALASKA and thus comparing it to the other kind of stores already connected to the platform. Other features of the framework such as parsing and output will not be used in ALASKA for now. This is due to one of the major drawbacks of the Jena framework which is that it keeps an internal representation of the workbench as an abstract model in main memory. Which means for instance that parsing a very large knowledge base located on disk could make the machine where the parser is located to run out of memory very fast. More details on those issues will be related on Section 5.2. Jena is an Open Source project, started by HP and now maintained by the Apache Foundation.

[6]http://www.sparsity-technologies.com/dex
[7]http://jena.sourceforge.net/

3.4. ALASKA operations

As previously mentioned in Section 3, the genericity of ALASKA relies directly on the use of the generic methods specified in the abstract layer of the platform. Writing programs, such as the backtracking algorithm presented, that have no communication with the store except using those methods ensure that the program is automatically compatible to any store integrated to ALASKA. Along with the $addAtom$ function, that stores an atom to disk and has been already presented in the last section, this section will list and detail the principal functions of that abstract layer with respect to RBDA.

A short explanation with an example of how does each of those functions is translated in each type of storage system will also be given, as it differs from one data model to another. The SQL statements given below assume that the information has been stored to the database with the storage procedure detailed in 5.1.1, where all the column names are known (col_1 to col_N). Also, one should note that the triples store used within ALASKA, Jena TDB, does provide internal functions for triples search and filtering, thus making the use of SPARQL statements for reading the knowledge base not necessary.

3.4.1. Retrieving all the atoms with a given predicate

Retrieving all the atoms with a given predicate is one of the functions that might be useful to a higher-level application using ALASKA. In this example, we show how to retrieve all the atoms with the p predicate in different systems.

– In a graph database, the procedure of obtaining all the atoms with a given predicate starts with the search for all the predicate nodes with such predicate. The function then asks the database whether there is a node in the database with the following information in its key-value table: $\{$type:$predicate$, label: $p\}$. The function does not return any atom if no predicates nodes are found. If there is a match, however, the function first reads the $arity$ information of the predicate node. We assume that the verification that the arities are respected for all atoms in the knowledge base. For each positive result, the program searches for all the terms connected to the predicate node, from

1 to the arity of p. Atom objects, as defined in ALASKA are created with the searches result. The complexity of the operation is $\mathcal{O}(n_p)$, n_p being the number of predicate nodes of label p, for a graph database without edge type indexing, and $\mathcal{O}(n)$ otherwise, n being the size of the knowledge base.

– In a relational database, obtaining all the atoms with a certain predicate is made through a $SELECT$ statement. One should notice that such statement is very straight-forward if the table is stored with one table per predicate in the knowledge base. The SQL statements are the following:
SELECT * FROM p (for the first algorithm presented).
SELECT * FROM $tuples$ WHERE col_p = p (for the second one).
In both cases, the result needs to be fetched and returned as terms in ALASKA model. In the first case, the rows returned will be read, and for each term, the $vars$ table will have to be accessed in order to check if a returned term is variable or not. Same thing for the second case, where the information will be obtained by checking the prefix of the term.
The complexity of the operation in a database with an index on predicates is $\mathcal{O}(m)$, m being the number of tuples in table p, $\mathcal{O}(n)$ otherwise.

3.4.2. Listing the terms connected to a given term with a given predicate

Listing the terms connected to a given term requires, besides a term, a predicate and two terms positions to be passed as input. The first term position indicates the position of the given term in the atom, while the second one indicates which term of the atom in the knowledge base to return. There is always the possibility of returning the whole atom, however this is not our aim. This function corresponds to the $enumerate$ function previously presented in Section 3. As example, we will show how to retrieve all the terms connected to a, the second term of the atom, via the p predicate that are at first position in the atom. The function focus in the position of the term to retrieve and thus does not consider the arity of the predicate (of course, the requested position must be lower or equal to the arity of the predicate). In the case that p is a binary predicate, the function returns all the X such that $p(X, a)$ exist in the knowledge base. If for instance, p is a predicate of arity 4, it would return all the X such that $p(X, a, ?, ?)$ is in the knowledge base.

– In a graph database, the listing of all the X terms connected to a via the p predicate, at first position of the atom is computed by the following program. It first selects the term node corresponding to the a term. If it is found, the program will now check if there is an edge of label 2 between the selected node and a predicate node with label p. For all results, the program will search for the term connected to the predicate node by an edge of label 1. The label will be memorized, inserted into a Term object as defined by ALASKA, and pushed to a collection. The list of candidates is then returned by the program.
The complexity of the operation in a graph database is $\mathcal{O}(|a_n|)$, a_n being the size of the neighbourhood of a.

– In a relational database, finding such a list of candidates is made through a $SELECT$ statement where one element of the row is instantiated, and only one is wanted in return. In order to find all the X for the $p(X, a)$ atom, the SQL statements are the following (according to the manner the knowledge base is stored in the database):
SELECT col_1 FROM p WHERE col_1 = 'a' (for the first algorithm presented).
SELECT col_1 FROM $tuples$ WHERE col_p = p AND col_2 = 'c : a' (for the second one).
The result also needs to be fetched and returned as terms in ALASKA model, and the procedure is done exactly as for the previous operation.
The complexity of the operation in a database with an index on predicates is $\mathcal{O}(m)$, m being the number of tuples in table p, $\mathcal{O}(n)$ otherwise.

3.4.3. Verifying the existence of a given atom

Verifying the existence of a given atom corresponds to the other elementary operation of the backtracking algorithm for computing homomorphisms presented in 3. This function is also referred as the $check$ function. We will use the atom $p(a, b, c)$ as example, where a, b and c are constants.

– In a graph database, the procedure of verifying if $p(a, b, c)$ starts with the search for the predicate node. The function first asks the database whether there is a node in the database with the following information in its key-value table: $\{$type:$predicate$, label: p, arity: $3\}$. For each positive result (as there can be more than one), the program will now check if all the nodes corresponding to the terms are correctly connected to this predicate node. In the case of the atom $p(a, b, c)$, the program asks the database if there is a node of type $term$ and label a connected to the predicate node, and if such edge has its label 1. If that is not the case, the predicate node is discarded and the program tests another predicate node. If yes, the program verifies the b and c term nodes. If the edges between all the terms nodes and the predicate node are correct, the atom is found and the program returns a positive answer. If no positive match is found for all the predicates nodes tested, the atom was not found and the program returns a negative answer. The complexity of the operation in a database without an index on the predicate is $\mathcal{O}(n_p)$.

– In a relational database, the verification of an atom is made through a $SELECT$ statement where all the elements of the query are instantiated. For the $p(a, b, c)$ atom, the SQL statements are the following (according to the manner the knowledge base is stored in the database):
  SELECT * FROM p WHERE col_1 = 'a' AND col_2 = 'b' AND col_3 = 'c' (for the first algorithm presented).
  SELECT * FROM $tuples$ WHERE $col_p = p$ AND col_1 = '$c : a$' AND col_2 = '$c : b$' AND col_3 = '$c : c$' (for the second one).
  In both cases, there is no need to fetch and read the results obtained. Executing the SQL operation will return an iterator, and the answer of whether the atom is found or not can be given by verifying if the iterator is empty or not.
  The complexity of the operation in a database with an index on predicates is $\mathcal{O}(m)$, m being the number of tuples in table p, $\mathcal{O}(n)$ otherwise.

4. Illustrating Example

The translations detailed above will now be explained through the means of an example. In this example, we will use a knowledge base represented by an image. Such knowledge base contains a fact and no ontology. The fact corresponds to the information contained in the image. This information will first be extracted (manually) from the image and represented as a text. From the text, a knowledge base will be created, and the information of the text will then be transformed into logics. Once the logical expression of the fact is obtained, it will be then transformed in order to be stored in any of the storage methods supported by ALASKA.

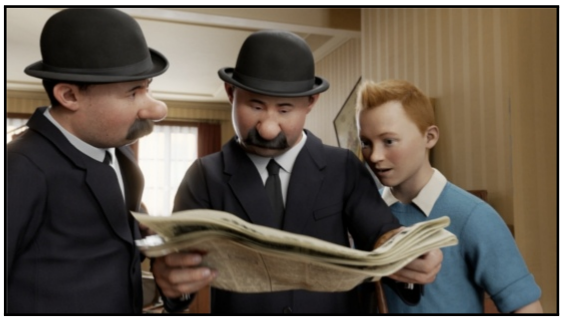
Fig. 4. Example: Scene cut from Tintin movie, featuring Tintin and the Duponts.

Figure 4 is the image selected for the examples. It presents a scene from the latest Tintin movie. The information we have extracted from the image is the following:

The picture features three men. Two of them are twins. Both of them are wearing a suit. Both suits are black. One of the twins is holding and reading a newspaper. The other man is also reading the newspaper. He wears a blue shirt.

From this paragraph, we will manually create the knowledge bases for the example. Two different knowledge bases, K_1 and K_2 will be created: one featuring only predicates of arity 2, and the other one without any restriction in the arities of predicates.

For K_1, the predicates of the example are: $type$, $twins$, $wears$, $reads$, $holds$, $color$, all of arity 2. The variables of the example are $m1$, $m2$ and $m3$ for the three men, $s1$ and $s2$ to represent the suits, and $s3$ for the shirt. n will represent the newspaper. Colors and object types are represented as constants: $Black$, $Blue$, Man, $Suit$, $Shirt$, $Newspaper$.

For K_2, the predicates (with their arities) of the example are: man (1), $suit$ (1), $shirt$ (1), $newspaper$ (1), $twins$ (2), $wears$ (2), $reads$ (2), $holds$ (2), $color$ (2). The variables of the example are: $m1$, $m2$ and $m3$ for the three men. $s1$ and $s2$ represent the suits, $s3$ the shirt and n will represent the newspaper. Now, only the colors are represented by constants, $Black$ and $Blue$.

Figure 5 summarizes the vocabularies of both knowledge bases:

K_1		
Predicates (6)	Variables (6)	Constants (5)
$type$	$m1$	$Black$
$color$	$m2$	$Blue$
$twins$	$m3$	Man
$wears$	$s1$	$Suit$
$reads$	$s2$	$Newspaper$
$holds$	$s3$	
	n	

K_2		
Predicates (11)	Variables (7)	Constants (2)
man (1)	$m1$	$Black$
$suit$ (1)	$m2$	$Blue$
hat (1)	$m3$	
$shirt$ (1)	$s1$	
$newspaper$ (1)	$s2$	
$same\text{-}as$ (2)	$s3$	
$twins$ (2)	n	
$wears$ (2)		
$reads$ (2)		
$holds$ (2)		
$color$ (2)		

Fig. 5. Listing of the K_1 and K_2 vocabularies.

The logical expression of the fact in K_1 is:

$\exists m1, m2, m3, s1, s2, s3, n \ (type(s1, Suit) \wedge type(s2, Suit) \wedge type(s3, Shirt) \wedge type(h1, Hat) \wedge type(h2, Hat) \wedge type(m1, Man) \wedge type(m2, Man) \wedge type(m3, Man) \wedge type(n, Newspaper) \wedge twins(m1, m2) \wedge wears(m1, s1) \wedge wears(m1, h1) \wedge wears(m2, s2) \wedge wears(m2, h2) \wedge wears(m3, s3) \wedge reads(m2, n) \wedge reads(m3, n) \wedge holds(m2, n) \wedge color(s1, Black) \wedge color(s2, Black) \wedge color(s3, Blue)$

While the logical expression of the fact in K_2 is:

$\exists m1, m2, m3, s1, s2, s3, n \ (man(m1) \wedge man(m2) \wedge man(m3) \wedge suit(s1) \wedge suit(s2) \wedge shirt(s3) \wedge newspaper(n) \wedge twins(m1, m2) \wedge wears(m1, s1) \wedge wears(m1, h1) \wedge wears(m2, s2) \wedge wears(m2, h2) \wedge wears(m3, s3) \wedge reads(m2, n) \wedge reads(m3, n) \wedge holds(m2, n) \wedge color(s1, Black) \wedge color(s2, Black) \wedge color(s3, Blue)$

ALASKA is now able to store such information in any storage system connected. The transformations used within ALASKA are explained in Section 5.1. In the case of a relational database, different schemas are possible as the user has to choose if he wants to have one table per predicate, or one single table (when it is possible), and also how to keep track of which terms are variables or not. In this example, the atoms of K_1 will all be stored in one single table, as K_1 only features predicates of arity 2, and the terms will be renamed according to the fact that they are constants or variables. K_2 will be stored in a database with one table per predicate, with an extra table containing the list of variables in the knowledge base. One should not forget that the schema definition and the tuples insertion is still independent of the chosen RDBMS.

$triples$		
col_p	col_1	col_2
$type$	v:s1	c:Suit
$type$	v:s2	c:Suit
$type$	v:s3	c:Shirt
$type$	v:m1	c:Man
$type$	v:m2	c:Man
$type$	v:m3	c:Man
$type$	v:n	c:Newspaper
$twins$	v:m1	v:m2
$wears$	v:m1	v:s1
$wears$	v:m2	v:s2
$wears$	v:m3	v:s3
$reads$	v:m2	v:n
$reads$	v:m3	v:n
$holds$	v:m2	v:n
$color$	v:s1	c:Black
$color$	v:s2	c:Black
$color$	v:s3	c:Blue

Fig. 6. Storing K_1 in the relational database.

Figures 6 and 7 show how K_1 and K_2 will be stored in a relational database. The storage process is detailed in Section 5.1.

In the case of graph databases, the transformation is not unique and straight-forward as it is for relational databases, it depends on the data model of the chosen store. As seen previously, in this work we will only focus on the graph databases using the Property Graph model. For K_1, which has only predicates of arity 2, the transformation is straight-forward by representing the terms of the knowledge base by nodes and the atoms of the logical formula by edges between the nodes of the terms of each atom. K_2 will need, however, a different transformation as it contains predicates with an arity different than 2. In this case,

man
col_1
m1
m2
m3

$suit$
col_1
s1
s2

$shirt$
col_1
s3

$newspaper$
col_1
n

$twins$	
col_1	col_2
m1	m2

$wears$	
col_1	col_2
m1	s1
m2	s2
m3	s3

$reads$	
col_1	col_2
m2	n
m3	n

$vars$
col_1
m1
m2
m3
s1
s2
s3
n

$color$	
col_1	col_2
s1	Black
s2	Black
s3	Blue

$holds$	
col_1	col_2
m2	n

Fig. 7. Storing K_2 in the relational database.

ALASKA will use the transformation that represent the atoms predicates by nodes, connecting each term node to its predicate node and precising the position of such term in the atom.

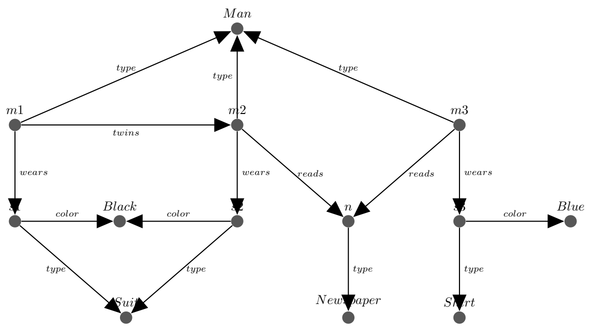

Fig. 8. Storing K_1 in a graph database using the Property Graph Model.

Figures 8 and 9 illustrate K_1 and K_2 properly encoded in the Property Graph model, using the two distinct transformations, ready for storage in a graph database.

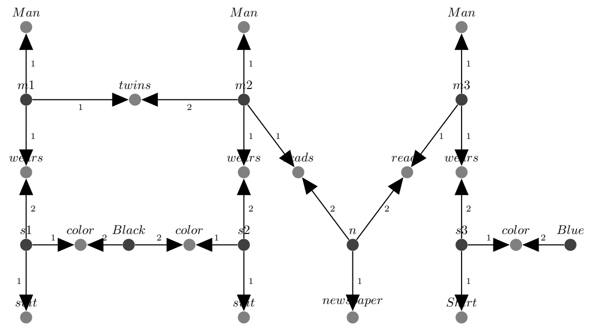

Fig. 9. Storing K_2 in a graph database using the Property Graph Model.

5. Storage

5.1. Knowledge base storage

In this section, we explain how a knowledge base is stored in each of the storage systems connected to ALASKA.

5.1.1. Relational databases

Storing facts in a relational database needs to be performed in two distinct steps. First, the relational database schema has to be defined according to the vocabulary of the knowledge base. The information concerning the individuals in the knowledge base can only be inserted once this is done. According to the arities of the predicates given in the vocabulary, there are two distinct manners to build the schema of the relational database: in the classic case, one relation is created for each predicate in the vocabulary. The second case occurs only when all the predicates in the vocabulary share the same arity (cf. RDF [19]). In this case, it is possible to define one single relation and to include the whole knowledge content in this relation. Such encoding is very similar to the ones used for Triples Stores [20].

One should not forget that there are no variables in a relational database. Indeed, variables are frozen into fresh constants before being inserted into a table. In order not to lose such information upon storing a fact, two alternatives exist: in the first method, a prefix is added to the label of every term of the database. A term t would then be renamed to $c{:}t$ if it is a constant or $v{:}t$ if it is a variable. In the second case, no changes are made to the label of the terms, but an extra unary relation, R_{vars} is created. Tuples containing the label of the variables in the database would then be added to the relation. Both alternatives have advantages and drawbacks: the first one does not alter the schema of the database in any case, but does rename every single term in it. On the other hand the second one does not rename any term of the database but does add a relation to the database schema. By adding a new table containing the variables information, a $SELECT$ call is then needed to answer whether a term is a constant or variable in the second case, while it can be directly answered by reading the term's label if the first method is used. For that reason, the first method will be preferred to the second one for our experimental processes. However, both solutions are available in ALASKA and the platform does leave the choice of use to the user, according to what he seems more appropriate.

Four different manners for storing a knowledge base in a relational database are possible, combining the two different options to keep track of the variables and the two different options according to the number of tables in the database. Below, we will present the algorithms of two of those four manners, in a way that every technical option is covered.

Algorithm 3: KB to Relational Database algorithm

```
Input: K a knowledge base
Output: a boolean value
begin
    create table vars ($col_1$);
    foreach Atom a in A do
        p ⟵ a.predicate;
        if !exists table with label p then
            n ⟵ p.arity;
            create table p ($col_1$,...,$col_n$);
        foreach Term t in a.terms do
            if t is a variable then insert into vars (t);
        insert into p ($t_1$,...,$t_n$);
    return true;
end
```

Algorithms 5.1.1, 5.1.1 are used when storing a fact in a relational database. In the first algorithm, one table is created in the database for each predicate in the knowledge base, while the second one is only for cases when all the predicates share the same arity. Also, in the first algorithm, an extra table named $vars$ with a single column is created at the beginning in order to store variables. That does not happen in the second, as the system keeps track of the variables by renaming the terms inside the tables before insertion.

The relational database management system used within ALASKA is SQLite. Versions 3.0 or higher of the RDBMS use a separate B+tree per table and a B-tree per index in the database. One can see that the algorithms defined above does not define any particular index and also does not define any primary or foreign keys. Indeed, the behaviour of ALASKA is not to make any supposition about the data that it is given for storage/management. Based on this idea, indexing the knowledge base in order to ensure a better efficiency upon reading stage would require to index all the tables by all its attributes, which would have a significant additional cost in space. As for tables, the fact that they do not have a primary key suggests that the information is stored in the B+tree using the $rowid$, an identifier for each row of a table kept by the system. A B+tree is a variant of the B-tree in which the sequential access has been improved by the use of pointers between leaves nodes of the tree.

Algorithm 4: KB to Relational Database algorithm (v2)

```
Input: K a knowledge base,n the arity of all predicates of K
Output: a boolean value
begin
    create table tuples ($col_p$,$col_1$,...,$col_n$); foreach Atom a in A do
        foreach Term t in a.terms do
            if t is a variable then t ⟵ v:t;
            else t ⟵ c:t;
        insert into tuples (a.predicate,$t_1$,...,$t_n$);
    return true;
end
```

5.1.2. Graph databases

Storing a knowledge base into a Property Graph-based database can also be done in two different manners, and as for relational database, the two manners differ according to the arities of the predicates in the knowledge base. In the first case, when all the predicates in the knowledge base are of arity 2, the transformation is straight-forward, with a node for each term, and an edge for each atom, connecting the nodes corresponding to the terms position in the atom. In the second case, it is needed to encode the bipartite graph with different nodes for terms and predicates corresponding to the knowledge base fact. The bipartite graph, being only composed of binary relations, is then easily stored to the database.

In both cases, we have chosen to have limited use of the key-value tables associated to the terms and edges of the graph. For terms, the term label and the information of whether it is a variable or not is stored in the table. In the case of edges, the label is the only information stored in the table.

```
Algorithm 5: KB to Property Graph algorithm
  Input: K a knowledge base
  Output: a boolean value
begin
    g ⟵ empty graph;
    foreach Atom a in A do
        if exists node with label a.terms[0].label then
            head ⟵ node.id;
        else
            create new node with id newId;
            newId.put(label,a.terms[0].label);
            newId.put(variable,a.terms[0].isVariable);
            head ⟵ newId;
        if exists node with label a.terms[1].label then
            tail ⟵ node.id;
        else
            create new node with id newId;
            newId.put(label,a.terms[1].label);
            newId.put(variable,a.terms[1].isVariable);
            tail ⟵ newId;
        create new edge with id edgeId from head to tail;
        edgeId.put(label,a.predicate);
    return true;
end
```

Fig. 10. KB to Property Graph algorithm

Algorithm 10 displays the algorithm for storing a fact in a graph database for the first case, when all the predicates are binary. Algorithm 5.1.2 displays the version for storing any knowledge base into a property graph-based database.

```
Algorithm 6: KB to Property Graph algorithm (v2)
  Input: K a knowledge base
  Output: a boolean value
begin
    g ⟵ empty graph;
    foreach Atom a in A do
        create new node with id predId;
        predId.put(label,a.predicate);
        predId.put(arity,a.predicate.arity);
        predId.put(type,predicate);
        foreach Term t in a.terms do
            if exists node with label t.label then
                nodeId ⟵ node.id;
            else
                create new node with id newId;
                newId.put(label,t.label);
                newId.put(variable,t.isVariable);
                newId.put(type,term);
                nodeId ⟵ newId;
            create new edge with id edgeId from nodeId to predId;
            edgeId.put(label,pos);
    return true;
end
```

In the first one, the graph creation process is very simple. For each atom of the fact, the algorithm creates an edge from the node corresponding to the first term of the atom to the node corresponding to the second one. The algorithm verifies first if both nodes already exist in the graph. If that is not the case, such nodes are created prior to the edge creation.

In the second case, the graph representing the knowledge base will feature two types of nodes: terms and predicates. For each atom of the fact, a new predicate node will be created. Then, for each term of the atom, an edge from the node corresponding to the term to the newly created predicate node is created. The label of such node corresponds to the position of the term in the atom. Once again, terms nodes are verified prior to the edge creation and created if needed. In this case, the arity of the predicate is introduced in the key-value table of each predicate node.

Two graph databases are currently used within ALASKA, Neo4J and DEX. Neo4J storage model is basically build upon pointers and linked lists. Every node has an ID in Neo4J, and the database provides a simple mapping from IDs to nodes. An edge is internally represented as a linked list, containing the IDs of the starting and ending nodes (as every edge in the graph database is binary), and the relationship type of the edge (which corresponds to our predicates). The list contains then 5 pointers: a pointer to the previous edge leading from the start node, a pointer to the next edge leading from the start node, a pointer to the previous edge leading to the end node, a pointer to the next edge leading to the end node, and a pointer to the first pair (key,value) in the key-value table of the edge. This table is implemented through a double-linked list. Searching for a node or an edge in Neo4j is done by a search algorithm that navigates through the pointers to find the requested information. As other databases, Neo4J supports external indices, such as B-trees and text-based indices for edges. Adding those indices to the core of the database has not been the priority of the developers, however.

As for DEX, the internal storage of a graph is the following. The graph is split into a combination of links and bitmaps. Every object in the graph (node or edge) has an unique ID in the database. The key-value table of nodes and edges is represented via attributes and values. A link is an internal data structure in DEX that is the combination of a map with multiple bitmaps. It ensures a bidirectional association between values and the IDs. Given a value, the link allows for example obtaining a bitmap containing the IDs of all objects containing such value. A graph in DEX features a bitmap that indexes nodes and edges by their type. As all attributes in the key-value stores must be declared prior to added to the graph, those are also indexed via a link structure. Two more links are used in order to index the incoming and outgoing edges of each node. No details on the efficiency of the bitmap compression and decompression have been given by the DEX developing staff.

5.1.3. Triples Stores

Storing a knowledge base in a triples store is very similar to performing it in a property graph-based database, as both only support binary relations natively. As for graph databases, a different encoding must be introduced in order to store knowledge bases with predicates with arities bigger than 2. No major differences should be highlighted, excepted from the fact that as terms are only designed by URIs in a triples stores, the information of the arity of a predicate must be entered to the store by the means of a new triple containing the arity of a given predicate.

Algorithm 5.1.3 displays how a knowledge base with no restrictions on the arities of the predicates is stored in a triple store.

Algorithm 7: KB to Triples Store algorithm

```
Input: K a knowledge base
Output: a boolean value
begin
    g ⟵ empty store;
    foreach Atom a in A do
        foreach Term t in a.terms do
            pos ⟵ the position of t in a;
            create new triple (t, alaska:pos, a.predicate);
        create new triple (a.predicate, alaska:arity, a.predicate.arity);
    return true;
end
```

The triples store used within ALASKA is Jena TDB. A TDB database corresponds to a single folder on disk. The database is composed of a table for nodes, indexes on the triples, and a table for prefixes. The table of nodes stores all the RDF terms in the database. As each RDF term is internally represented by an ID, the node table provides two mappings in order to easily obtain the ID from a node, or a node from an ID. Such mappings are particularly helpful on storage and querying processes. The triples of the database are indexed by subject, property and objects. Each of these indices contains all the information about all the triples. If this may be helpful when processing queries, this is also the cause of a certain redundancy on the data stored on disk, using more disk space than other stores. Indices and mappings are implemented with a custom implementation of the B+tree. Such implementation is very similar to relational databases, with the advantage of having a native index on terms and atoms.

5.2. Experimental Work

Storing information on disk may come into play in two different steps of RBDA. The first is when one has to load a knowledge base and decides to store it locally in a particular system. The second is when a rule application process is launched and the process generates brand new information to be added to the current fact. While in the first case speed is not that relevant, as it could be considered a pre-processing step and often is only needed once, it is crucial for the second. In this section we present in detail our experimental work on storage efficiency.

5.2.1. Workflow

Let us consider the workflow used by ALASKA in order to store new facts. The fact will first be parsed in the application layer (1) into the set of atoms corresponding to its logical formula, as defined in the abstract layer (2). Then, from this set, a connector located in layer (3) translates the fact into a specific representation language in (4). The set of atoms obtained from the fact will be translated into different data models.

In this workflow, the RDF file is given as input to the Input Manager in the application (layer 1). Information is then forwarded according to the selected output system. The fact from file is first transformed in an IFact object (layer 2). It is then translated (layer 3) to the language of the system of choice (graph, relational database, or triple store) before being stored onto disk (layer 4). This workflow is visualised in Figure 11 where a RDF file is stored into different storage systems.

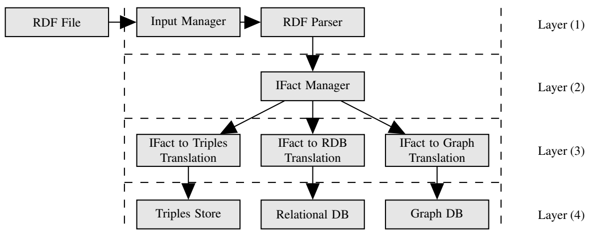

Fig. 11. Workflow for storing a knowledge base in RDF using ALASKA.

5.2.2. Input Data

In order to perform our experimental work, knowledge bases had to be selected for our protocol. The knowledge base we have used has been introduced by the SP2B project [24]. The SP2B project supplies a generator that creates arbitrarily large knowledge bases maintaining a similar structure to the original DBLP[8] knowledge base, which they have studied. The argument of the project members for choosing DBLP is that it reflects the social network character of the Semantic Web, where many small pieces of information are put together, creating a global network of data. The generated knowledge bases are in RDF format (N-TRIPLES format), although our testing protocol within the ALASKA platform uses logical expressions as input to the system. Using those generated knowledge bases then require an initial translation from RDF into first order logic expressions. The use of the SP2B knowledge bases is relevant to this work since:

- $\rightarrow$ logical knowledge bases are not easily available throughout the web.
- $\rightarrow$ this kind of knowledge bases seem to be very similar to all the emergent knowledge bases that have appeared with Social Networks and the Semantic Web, in which it would be highly recommended to perform RBDA.

5.2.3. Challenges

Storing large knowledge bases using a straightforward implementation of the testing protocol has highlighted different issues. We have distinguished three different issues that have appeared during the tests: (1) memory consumption at parsing level, (2) use of transactions, and (3) garbage collecting time.

Memory consumption at parsing level depends directly of the parsing method chosen. A few experiences have shown that some parsers/methods use more memory resources than others while accessing the information of a knowledge base and transforming it into logic. We have initially chosen the Jena framework parsing functions in order to parse RDF content, but we have identified that it loads almost the whole input file in memory at reading step. We have thus replaced the RDF parser to a different one, which works only with N-Triples encoded RDF files, that does not store the facts in main memory but feeds them one at a time to the ALASKA.

Garbage collecting (GC) issues have also appeared as soon as preliminary tests were performed. Several times, storing not very large knowledge bases resulted in a GC overhead limit exception thrown by the Java Virtual Machine. The exception indicates that at least 98% of the running time of a program is consumed by garbage collecting.

Managing **transactions** also became necessary in order to reduce the loss of efficiency obtained in the preliminary tests. As we work with different storage methods, their transaction management systems also differ. While it is needed to manage transactions manually in certain systems, transactions are enabled by default in others, thus needing to be explicitly handled in order to obtain success. Tests have shown that trying to store all the atoms of a knowledge base at once in a single transaction was effective up to a certain point, and inefficient beyond this point as the transaction content is kept in memory until the transaction is committed or discarded. In order to avoid too much memory consumption, we have decided to run multiple transactions while storing a large knowledge base on disk.

In order to address both transaction and garbage collection issues, a buffer of atoms was set up. The buffer is filled with freshly parsed atoms at parsing level. At the beginning, the buffer is full and then every parsed atom is pushed into the buffer before being stored. Once the buffer is full, parsing is interrupted and the atoms in the buffer are sent to the storage system for being stored. Once all atoms are stored, instead of cleaning the buffer by destroying all the objects, the first atom of the buffer is moved from the buffer into a stack of atoms to be recycled. Different stacks are created for each arity of predicates. In order to replace this atom, a new atom is only created if there is no atom to be recycled from the stack of the arity of the parsed atom. If there is an atom to be recycled, then it is then put back in the buffer, with its predicate and terms changed by attribute setters. The buffer is then filled once again, until it is full and the atoms in it are sent to storage system.

5.2.4. Contribution

In order to store large knowledge bases on disk using a single machine, preventing the issues described above, we have implemented the storage algorithm of Figure 5.2.4. The algorithm is run by an InputManager class within ALASKA that handles all the storage calls from users.

Algorithm 5.2.4 illustrates the manner the Input Manager handles a stream of atoms received as input. Other parameters passed as input are the fact where

[8]http://www.informatik.uni-trier.de/ ley/db/

```
Algorithm 8: Input Manager storage method
Input: S a stream of atoms,f an IFact,bSize an integer
Output: a boolean value
begin
    buffer ⟵ an empty array of size bSize;
    counter ⟵ 0;
    foreach Atom a in S do
        if counter = bSize then
            f.addAtoms(buffer,null);
            counter ⟵ 0;
        buffer[counter] = a;
        counter++;
    f.addAtoms(buffer,counter); return true;
end
```

the stream of atoms must be stored, and an integer representing the size of the buffer of atoms that will be instantiated. The buffer along with a counter are created at the beginning of the procedure. The procedure is very simple, as it puts the atoms of the stream in the buffer until its capacity is reached. The buffer, full of atoms is then sent to the storage system that manages the fact f. This is made through the $addAtoms$ method, implemented in each store connected to ALASKA.

The solution of using a buffer has been chosen after solutions storing atoms one-by-one has shown to be very inefficient, and all-at-once solutions would load the whole transaction content in memory, going beyond the limit of memory usage for such process.

5.2.5. Results

As previously mentioned, the SP2B project supplies a generator that creates knowledge bases with a certain parametrised quantity of triples maintaining a similar structure to the original DBLP knowledge base. The generator was used to create knowledge bases of increasing sizes (5 million triples, 20, 40, 75 and respectively 100). Each of the knowledge bases has been stored in Jena, DEX, SQLite and Neo4J. In Figure 12 we show the time for storing the knowledge bases and their respective sizes on disk.

The user can see that the behaviour of Jena is worse than the other storage systems. Let us also note that DEX behaves much better than Neo4J and this is due to the fact that ACID transactions are not required for DEX (while being respected by Neo4J). Second, the size of storage is also available to the user. One can see, for instance, that the size of the knowledge base stored in DEX and Neo4J is well under the size of initial RDF file. However, the size of the file stored in Jena is bigger than the one stored in SQLite and bigger than the initial size of the RDF file.

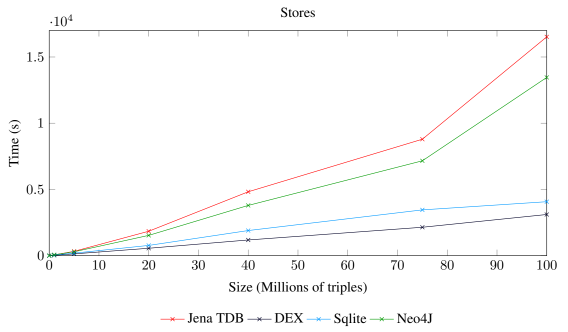

Fig. 12. Storage time and KB sizes in different systems

Size of the stored knowledge bases					
System	5M	20M	40M	75M	100M
DEX	55 Mb	214.2 Mb	421.7 Mb	785.1 Mb	1.0 Gb
Neo4J	157.4 Mb	629.5 Mb	1.2 Gb	2.3 Gb	3.1 Gb
Sqlite	767.4 Mb	2.9 Gb	6.0 Gb	11.6 Gb	15.5 Gb
Jena TDB	1.1 Gb	3.9 Gb	7.5 Gb	13.9 Gb	18.1 Gb
RDF File	533.2 Mb	2.1 Gb	4.2 Gb	7.8 Gb	10.4 Gb

Fig. 13. Storage time and KB sizes in different systems

The storage tests were performed on a dedicated server with the following characteristics: 64-bit Quad-core AMD Opteron 8384 with 512 Kb of cache size and 64 Gb of RAM. Please note that this second server is shared between multiple processes, therefore the tests will only use part of all this computing power. The memory size of the JAVA Virtual Machines created for executing the testing processes was of 4Gb.

6. Querying

Before performing any querying activity, we also need to define the purpose of using ALASKA as a querying interface instead of using any native solution. We have agreed that ALASKA would help us to define which are the most efficient solution for querying a large knowledge base, by comparing systems and querying engines when answering conjunctive queries. Two different factors come into play when performing the queries: memory consumption and execution time. In the test we will present, we are only considering execution time to define efficiency. We acknowledge the fact that memory consumption is certainly important, especially when performing complex queries over very large bases stored in secondary memory, but we do not consider this aspect in our tests. Please note that this is configurable and possible to control within ALASKA. The reason is twofold. First handling memory consumption for each system (and optimizing it) is not part of our comparing tests (or of the generic functionality offered by ALASKA). Second, intuitively, comparing a generic algorithm over different storage systems (and thus calling the elementary operations of each) should not lead to important differences between the systems at hand.

The generic backtrack algorithm used in ALASKA takes a query and for each term explores the candidate answers in the KB. The backtrack is due to the fact that the terms have to be correctly included in the right atoms (i.e. by the right predicates). The algorithm relies on the state of the "$level$" and "$goingUp$" variables in order to proceed to the exploration of the knowledge base. The algorithm starts by ordering the terms of the query. The algorithm does not require any particular order to work properly, but ordering strategies can enhance the efficiency of the algorithm. In the tests performed, our ordering was very simple: it consisted in sorting the terms list by letting the constant terms ahead of the variable terms. Inside the constant and variable parts of the list, the terms are ordered as they appear in the query. More elaborate orderings can be implemented as it relies on a custom function for ordering terms. $level$ corresponds to the index of the term the algorithm is trying to match, while $goingUp$ indicates if the algorithm continues its exploration, has reached a dead end or has found a match for all the terms. In the later case, a successful answer to the query has been found.

Algorithm 2: Backtrack algorithm

```
Input: A conjunctive query Q, and a fact F
Output: lists all the answers of Q in F
begin
    Q ⟵ order(Q);
    level ⟵ 0;
    goingUp ⟵ false;
    if H = ∅ then
        answerFound(Q);
    else
        while level ≠ 0 do
            if level = |Q| then
                answerFound(Q);
                goingUp ⟵ true;
            else currentTerm = Q[level];
            if goingUp then
                if otherCandidates(currentTerm,F) then
                    goingUp ⟵ false;
                    level ⟵ level + 1;
                else level ⟵ level - 1;
            else
                if findCandidates(currentTerm,F) then
                    level ⟵ level + 1;
                else
                    goingUp = true;
                    level ⟵ level - 1;
end
```

6.1. Abstract Procedures

As the mechanism of the algorithm is quite simple, its efficiency is then very tightly linked to the efficiency of the sub-functions $findCandidates$ and $otherCandidates$. $findCandidates$ is called once the algorithm goes a level below and has to search all the terms in the knowledge base that can be a match for a given term of the query. The second one is called every time the algorithm goes back up to a level previously visited, modifying the current matching of a term to a different candidate, and going back down to search for new answers. If there are not any new candidates for such term of the query, the function then returns $false$, and the algorithm goes back up again. The algorithm stops once there are no candidates left for the first term of the query, which means that there is nothing left to explore.

Considering that the $otherCandidates$ function only modifies a value of the matchings set currently built, and that all the matching candidates for a given term are previously computed using the $findCandidates$ function, the $otherCandidates$ function has a very small impact to the time and memory usage of the algorithm. Technically, it only consists of moving forward an iterator, thus the efficiency of the algorithm relies mainly on the efficiency of the $findCandidates$ function. We will explain in the following how this function can be quite costly in certain cases.

Indeed the number of calls to its sub-functions ($enumerate$ and $check$) depends on the number of times a term appears in the query. $enumerate$ and $check$ may be considered as the elementary operations of the backtrack algorithm. When called, $enumerate$ returns a list containing all the terms in the knowledge base that has exactly for neighbours the matchings of the neighbours of a given term of the query. While the $check$ function asks the system managing the knowledge base whether a given atom can be found in the base or not. The number of calls to the $check$ function depends directly on the quantity of results returned by the calls to the $enumerate$ function. As the $enumerate$ function computes the potential candidates for a match, the $check$ function verifies if a given candidate has to be maintained or discarded as a candidate. The number of calls to both functions generally increases as the size of the knowledge base also grows.

It is very important to state that during the execution of the backtracking algorithm the entire communication between the algorithm and the storage system in which the knowledge base is stored is performed through the *enumerate* and *check* function calls. No external communication between them is allowed, and that is what maintains the complete genericity of the algorithm. It is then mandatory to implement both functions in every system one wants to integrate to ALASKA. Querying the knowledge base with the generic algorithm is impossible if such implementation is not performed. Those are however not the only functions required for every storage system integrated to the platform. The *enumerate* and *check* functions require knowing from a knowledge base whether a term exists in the base or not, whether a predicate exists in the base or not, and whether two given terms belong to an atom with a given predicate or not. These are what we call the consulting, or reading functions required by ALASKA. Along with the writing functions needed to store a knowledge base, these functions compose the core functions that ALASKA needs to manage a knowledge base stored in a given storage system. More details about those functions are given in Section 3.4.

For each query, according to its structure and the degree of the terms of the query, the number of calls to *enumerate* and *check* will differ. As the efficiency of the algorithm relies on the efficiency of those two functions, we have added intermediary timers in order to enable ALASKA to measure the time the algorithm spends in each of those calls. This way, one performing a query is not only allowed to retrieve the total time of the execution of a query, but also how much of that time was spent enumerating candidates or retrieving atoms in the knowledge base.

6.2. *Experimental Work*

Performing queries is at the heart of the RBDA problem. It is needed when no ontological content is present to enrich facts, and also present in both methods of rule application we have explained in Section 2.1. Unlike storage, querying efficiency does not rely only on the storage systems, but in a triplet composed of storage systems, querying method and also the queries chosen. After a first battery of tests, in which neat conclusions were difficult to obtain, we have focused in the adaptation of our problem into a CSP problem and the integration of a CSP solving program in order to address conjunctive query answering. In this section we present in detail our experimental work on querying.

6.2.1. *Workflow*

Querying tests within our architecture takes place as indicated in Figure 14. Queries entered in ALASKA are processed and handled by the generic algorithms present in ALASKA, or translated to different querying languages according to the user's choice. As discussed in 2.3 and seen in the picture, a fact stored in a property graph-based database can only be queried in ALASKA using a generic querying algorithm. In addition to the backtracking algorithm described in Section 6, a CSP solving algorithm was also designed for answering conjunctive queries. More details about this solution are given in Section 7. Facts stored in a relational database can of course still be queried via the native SQL interface of the database, and fact stored in a triples store can also be queried using the native SPARQL interface of the store.

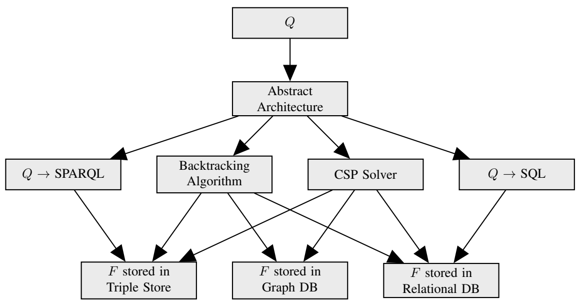

Fig. 14. ALASKA storage and querying workflow.

6.2.2. *Results*

We have tested all the systems previously listed in Section 3.3 with the generic backtracking algorithm detailed in Section 6 as well as the native query engines when available for each system. The knowledge bases used for the tests were generated using the same data generator already used for the storage tests. The queries used for the tests are the following:

1. `type(X,Article)`
   Returns all the elements which are of type article.
2. `creator(X,PaulErdoes)` $\wedge$ `creator(X,Y)`
   Returns the persons and the papers that were written with Paul Erdoes.
3. `type(X,Article)` $\wedge$ `journal(X,Journal1-1940)` $\wedge$ `creator(X,Y)`

Returns the creators of all the elements that are articles and were published in Journal 1 (1940).

4. `type(X,Article) ∧ creator(X, PaulErdoes)`
Returns all the articles created by Paul Erdoes.

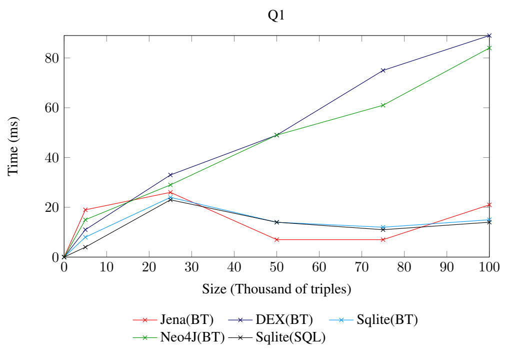

Fig. 15. Querying efficiency results for query 1.

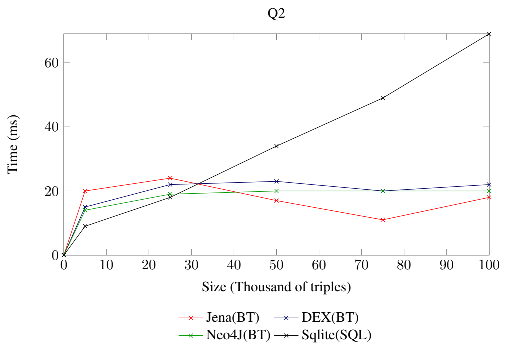

Fig. 16. Querying efficiency results for query 2.

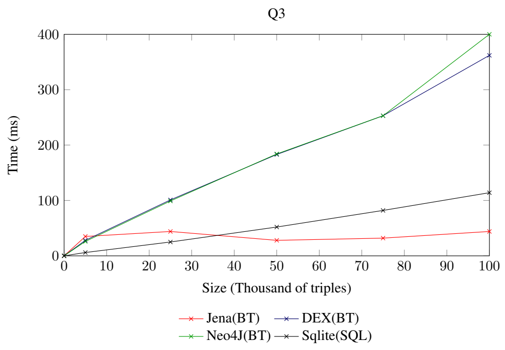

Fig. 17. Querying efficiency results for query 3.

The queries in the set were slightly inspired from the queries featured in the SP2B project [24]. However, the queries featured in the paper were designed with the purpose of covering all the features of the SPARQL query language, which is not our goal here. As we only deal with conjunctive queries, we have removed and also modified some queries in the set in order to have a set of queries that would be relevant for our purposes. We also state that such queries could have been executed in real-use case such as querying for articles and their properties in a bibliography management system such as DBLP. The queries were executed over knowledge bases of 5, 25, 50, 75, and 100 thousand triples.

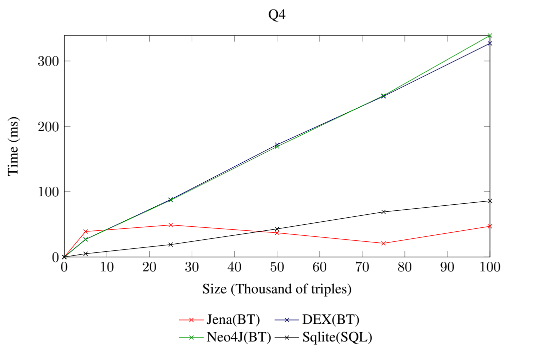

Fig. 18. Querying efficiency results for query 4.

The results presented above have shown the behaviour of the different storage systems integrated to ALASKA against a set of queries manually input. We notice that systems behave differently against different queries. This is due to the fact that each query differs from the others according to its structure, number of calls to the elementary operations ($enumerate$,$check$) and number of answers in the knowledge base.

We observe in the results that Jena TDB has shown to be efficient in all cases. For the only query it has not been the fastest system to answer, its response time was still very close from the fastest ones. We note that the querying efficiency of Jena TDB is linked to the efficiency of the index structures the system automatically builds at storage. Improving its querying efficiency by increasing the disk usage might be a good solution for an use case in which disk usage is not a constraint.

We also observe that independently of the internal data structure, both graph databases have almost the same efficiency for all the queries. Using a SQL engine has shown to be less efficient than using a backtracking algorithm for Q2. In this query, the number of answers is fixed, as no new papers from Paul Erdoes appear after a certain time. As the backtracking algorithm over a graph database explores the neighbourhood of the Paul Erdoes term, the answering time of the query in this case will remain constant even on a larger knowledge base. On the other hand, the SQL engine has to join the creator table with itself, and as the size of the table grows fast, the answering time of the engine for this query also grows. Using a backtracking algorithm has however shown to be less efficient for the other queries.

We have seen that a backtracking algorithm can be a good solution on instances in which a join algorithm does not perform well. However, the behaviour of the backtracking algorithm we have implemented on knowledge bases under a million triples has raised the question on how to optimize this backtracking algorithm. As using an SQL engine has shown to be efficient enough for simple queries, we look forward optimizing the backtracking algorithm for complex queries.

7. Discussion and Future Work

This section is structured as follows:

- In Section 7.1 we give an overview of the work presented and discuss the current limitations of ALASKA.
- Sections 7.2 and 7.3 provide pointers to current and future work. We discuss two lines of improvements and namely the optimizing of the backtracking algorithm using Constraint Satisfaction techniques and indexing issues.

7.1. Conclusion

In this paper, we have presented ALASKA, the software architecture we have designed and we have used to study the efficiency of existing storage systems for the elementary operations of conjunctive query answering. As previously stated, the ALASKA framework allows to build the first layer of an ambitious research aim, a generic platform for RULE-BASED DATA ACCESS. This first layer concerned the storage and querying operations that such generic platform must use during the reasoning process.

One current limitation is the lack of comparison with SPARQL. While this is an immediate next step, more importantly we also need improve the backtracking algorithm in order to deal with larger datasets and make our work competitive with respect to Linked and Open Data projects. This opens the way to two main future work directions: large scale optimization and indexing. We discuss the two in the remainder of the paper.

7.2. Constraint Satisfaction Programming

Optimizing the backtracking algorithm can be done in two different manners. First, in a technical point of view, it is possible to verify whether internal data structures used by the algorithm are adequate and offer a good complexity for the operations required by the program. Also, there are today libraries that propose new and optimized implementations of the native data structures in JAVA. Second, at algorithmic level, several proposals for optimizing the backtracking algorithm are available in the literature [4]. Most of those have been studied in the constraint domain.

Instead of implementing all the available algorithmic optimizations, we have rather preferred to adapt our problem into a constraint satisfaction problem. The reason of this choice is twofold. First, in an algorithmic point of view, is that most of these existing optimizations are already available in the CSP solvers. Second, the integration of a CSP solver within ALASKA, added to a well written adaptation of our data access problem to a constraint satisfaction problem would ensure that the updates and optimizations to the solver program would make our instances of the problem benefit of such updates without having to change our implementation.

A constraint satisfaction problem is a triple (X, D, C), where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint $c \in C$ is in turn a pair (t, R) where t is a n-tuple of variables and R is an n-ary relation on D. An evaluation of the variables is a function from the set of variables to the domain of values, $v : X \to D$. An evaluation v satisfies a constraint $((x_1, \ldots, x_n), R)$ if $(v(x_1), \ldots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints.

In order to integrate a CSP solver within ALASKA, we have chosen the Choco [21] solver. The fact that it has a complete documentation available on the web and that it is written in JAVA have guided our choice. Finally, for our problem, which can be considered particular in the CSP domain, the availability of the developers of Choco have helped confirming our choice. We have defined two different manners to adapt our problem into a CSP problem. The first manner concerns knowledge bases that can be entirely loaded in main memory. It has been later modified and extended to large knowledge bases, which is the second manner we have defined.

7.2.1. Simple transformation to CSP

Transforming the entailment problem into a CSP problem is quite simple when the knowledge base one wants to deduce a fact from is small enough to be entirely loaded in main memory. The procedure of transformation is the following:

- The network is composed of variables and constraints between the variables. Each variable has a domain. The domain of a variable contains the possible values for that variable. In the representation of our problem, the variables of the network correspond to the terms of the query Q, while the constraints will feature the list of tuples that satisfy the given constraint. Each constraint corresponds to an atom of Q.

- No information about the values in the variables' domains is known at start. Hence, all the variables are instantiated with all the available integer values. (It is always possible to filter the domain of a variable manually during the solver execution, but it is impossible to add a value to the domain.)

- Constraints are added to the network once all the variables have been defined. One constraint is created for each atom of Q. For each constraint, the variables connected to this new constraint are all the variables corresponding to the terms of the atom the constraint represents. The list of authorized tuples for this constraint is then read in the knowledge base. Such proceeding can not be used when dealing with a larger knowledge base, as computing all the authorized tuples may lead the computer to run out of memory.

- The solver can now be run once the variables and constraints have been properly instantiated.

Note that as the Choco library only manipulates integer values, each term in the knowledge base has to be represented by an integer. Key-value stores will then be used by the system to retrieve a term by its integer identifier and vice-versa. Filling such key-value stores depends on the size of the knowledge base. In the case of a small knowledge base, it can be done as a preprocessing step prior to creating the network.

7.2.2. Transformation with large KBs

As previously mentioned, some things presented in the previous section changed, and the problem receives an additional difficulty when dealing with large knowledge bases. One of the important changes is that it becomes forbidden to perform any operation having its complexity depending on the size of the knowledge base. This is the reason why a different transformation of our problem into a CSP problem had to be designed, since it is now needed to be able to indicate that a variable, at instantiation time, contains all the possible values in the knowledge base without having to compute them all.

The method chosen to address this problem came with the technical limitations of the Choco library itself. Indeed, Choco maintains the information concerning the variables' domains in memory, and keeps track of the evolution of those values for the eventuality of backtracking during the solver execution. The fact of keeping track of all this information in memory introduces a physical limit to the size of a domain. According to our preliminary tests with Choco, the limit of values in the domain of a variable in Choco is around 35000 values. As we have not found any mature idea on how to bypass such technical constraint, we will remain under such technical limit for this work.

One should not forget that this number of 35000 values in a domain does not mean that we will be limited to knowledge bases with 35000 terms or less (as such size of knowledge base fits perfectly in main memory) but rather that it will restrain every term of the query (each variable in the network) to explore only 35000 terms of the knowledge base during the execution of the solver. Indeed, the knowledge base size forbids us to precompute all the lists of authorized tuples for each constraint in the network. Such procedure will now be performing at solving stage, and only in certain conditions, in order to avoid performing too many reading operations in the knowledge base.

In this case, instead of having the key-values stores shared between all the variables of the network, each variable will have its own key-values tables. The values in this tables will be affected at runtime, and, this, a term in the knowledge base may have two different integer identifiers in two different variables. It will be the task of the propagation function inside the constraints to maintain the coherence between terms and the integer identifiers of such terms in all the variables connected to a constraint. The procedure of transforming a conjunctive query over a large knowledge base in a CSP problem is the following:

- To begin with, the terms of the knowledge base will not be read prior to the solver execution, excepted the constant terms in the query. This is due to the fact that a constant term in the query can only be matched to the same constant in the facts. Looking for the constants in the knowledge base may save time in the case one of them does not exist in the knowledge base. For each constant in the query, its matching in the knowledge base will be given the first integer value in the domain of the variable. A constraint of equality is linked to the variable, to indicate that this is the unique possible value for the variable.

- Once this is done, all the variables of the network corresponding to the non-constant terms of the query are initialized, with a domain going from 1 to 35000. This is performed without reading the knowledge base. The constraints are then created and attached to the variables. As for the variables, they will also be created but no knowledge base information will be read at this time. The only information they will carry upon initialization are the predicate of the corresponding atom and an integer value that will serve as a threshold for triggering a propagation sequence.

- The solver is ready to be run once all the variables and constraints of the network are properly created and "instantiated". In this version, as no information from the knowledge base was read into the network, the solver will not find any answer to the query when launched.

- Finding an answer to the query will only be possible through a propagation mechanism, that will enable the solver to filter the domains of the variables. The propagation is a function located in the constraint class that indicates to the solver how to proceed for finding answers. The behaviour of the propagation method we have implemented is to consult the knowledge base for retrieving information once the size of the domain of a variable attached to the constraint is lower than the threshold value set for this constraint. The bigger this value, the bigger are the chances for triggering the propagation. As the domains of all the variables of the network are not filtered at launch (and consequently higher than any threshold), this solution only covers conjunctive queries with at least one constant for now. The presence of the constant in the query ensures that at least one propagation call is performed, as the size of the domain of the variable is reduced to 1 at launch. This is a temporary solution that will be upgraded later with the use of more efficient indexation techniques.

- The knowledge base is read once the propagation function of a constraint of the network is triggered. The authorized tuples for this constraint will be computed then added to constraint information. This will filter the domains of the variables connected to the constraint. The efficiency of this solution comes from the fact that the propagation function does not read the whole knowledge base, but only looks for the neighbourhood of certain terms. This is due to the $enumerate$ function, already used in the generic backtracking algorithm and described in Section 3.4.

- Filtering the domain of the variables connected to a constraint should trigger the propagation function in another constraint, until all the lists of authorized tuples needed are computed. Once this happens, the solver can find the answers of the query in the knowledge base. If the propagation sequence stops before all the constraints have their list of tuples computed, this means that all the previous propagations have not filtered the remaining domains enough to reach the threshold value.

Despite the fact that the propagation mechanism for answering conjunctive queries using a constraint satisfaction solver when the knowledge base is very large has been successfully implemented, we have not succeeded in adding it to the list of querying methods in our testing protocols. The reason for this comes to the fact of the CSP solver not knowing how to proceed once the propagation sequence stops. The larger the knowledge base, more are the number of terms in the domain of each variable, making the threshold value more difficult to reach. Of course, it is always possible to increase such value manually prior to the solver execution, however not only it is impossible to know in advance which value to choose, but it also increases the amount of information read from the knowledge base and loaded in memory by the network. Another strategy for this cases is to pause the solver and to filter manually the domains of the variables of a given constraint, according to information of the knowledge base. If one has information about the frequency of predicates or some index on the knowledge base, he could use it in order to help the solver once his propagation sequence does not help it anymore to filter the domain of the variables in the network. Such ideas will be discussed in Section 7.3.

7.3. Indexing

As previously stated in Section 7, the propagation strategy of the CSP when dealing with large knowledge bases is the following: reading the knowledge base in order to look for a specific information only happens once a variable in the network has its domain size under a certain threshold. This value is set initially and can easily be modified. Once the size of the domain of a variable gets under the threshold, then the code contained inside the constraint classes calls the $enumerate$ method, which filters the domain of the neighbouring variables.

Such strategy has a major known drawbacks, which is the fact that the solver does not have an idea on how to proceed when there is no propagation to execute. This happens for example when launching the solver, as all the domains of the variables are full at that moment. For this reason we have been restricted to performing queries containing at least one constant term. This reduces the domain of the variable of the constant term to a single value, which is lower than any threshold value set, and launches the propagation.

However, if one wants to avoid the solver to be stuck during its execution or to handle queries with no bootstrap (no constant terms in the query), a better solution would need to be designed. The major idea in order to have this solved would be to add indexing features to the CSP solver, enabling it to read an index or some knowledge base statistics about terms or predicates once the propagation sequence is not enough for finding an answer.

References

[1] S. Abiteboul. Semi-structured data. In *Encyclopedia of Database Systems*, pages 2599–2601. 2009.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] F. Baader, S. Brandt, and C. Lutz. Pushing the EL envelope. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 364–369, 2005.

[4] J.-F. Baget. *ReprÃl'senter des connaissances et raisonner avec des hypergraphes : de la projection Ãą la dÃl'rivation sous contraintes*. PhD thesis, 2001.

[5] J.-F. Baget, M. Croitoru, and B. P. L. da Silva. Alaska for ontology based data access. In *ESWC (Satellite Events)*, pages 157–161, 2013.

[6] J.-F. Baget, M. Leclère, and M.-L. Mugnier. Walking the decidability line for rules with existential variables. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2010.

[7] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

[8] J.-F. Baget, M.-L. Mugnier, S. Rudolph, and M. Thomazo. Walking the complexity lines for generalized guarded existential rules. In T. Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-2011)*, pages 712–717. IJCAI / AAAI, 2011.

[9] J.-F. Baget and E. Salvat. Rules dependencies in backward chaining of conceptual graphs rules. In *International Conference on Conceptual Structures (ICCS)*, pages 102–116, 2006.

[10] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[11] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 77–86. ACM, 2009.

[12] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *LICS*, pages 228–242, 2010.

[13] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[14] M. Chein and M.-L. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Advanced Information and Knowledge Processing. Springer, 2009.

[15] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[16] M. Croitoru. *Conceptual Graphs at Work: Efficient Reasoning and Applications*. PhD thesis, University of Aberdeen, 2006.

[17] M. Croitoru and E. Compatangelo. A tree decomposition algorithm for conceptual graph projection. In *Tenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 271–276. AAAI Press, 2006.

[18] B. P. L. da Silva, J.-F. Baget, and M. Croitoru. A generic platform for ontological query answering. In *SGAI International Conference on Artificial Intelligence (AI-2012)*, pages 151–164, 2012.

[19] P. Hayes, editor. *RDF Semantics*. W3C Recommendation. W3C, 2004. http://www.w3.org/TR/rdf-mt/.

[20] A. Hertel, J. Broekstra, and H. Stuckenschmidt. Rdf storage and retrieval systems. In *Handbook on Ontologies*, pages 489–508. Springer, 2009.

[21] N. Jussien, G. Rochart, X. Lorca, et al. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

[22] M. Lenzerini. Data integration: A theoretical perspective. In *21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[23] E. Salvat and M.-L. Mugnier. Sound and complete forward and backward chaining of graph rules. In *International Conference on Conceptual Structures (ICCS)*, pages 248–262, 1996.

[24] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.

[25] M. Thomazo. Ontology based query answering with existential rules. In *Proceedings of the 23rnd International Joint Conference on Artificial Intelligence (IJCAI-13)*, 2013.