A Generic Platform for Ontological Query Answering

Bruno Paiva Lima da Silva, Jean-François Baget, Madalina Croitoru

A Generic Platform for Ontological Query Answering

Bruno Paiva Lima da Silva and Jean-François Baget and Madalina Croitoru

Abstract The paper presents ALASKA, a multi-layered platform enabling to perform ontological conjunctive query answering (OCQA) over heterogeneously-stored knowledge bases in a generic, logic-based manner. While this problem knows today a renewed interest in knowledge-based systems with the semantic equivalence of different languages widely studied, from a practical view point this equivalence has been not made explicit. Moreover, the emergence of graph database provides competitive storage methods not yet addressed by existing literature.

1 Introduction

The ONTOLOGICAL CONJUNCTIVE QUERY ANSWERING problem (OCQA) (also known as ONTOLOGY-BASED DATA ACCESS (ODBA) [15]) knows today a renewed interest in knowledge-based systems allowing for expressive inferences. In its basic form the input consists a set of facts, an ontology and a conjunctive query. The aim is to find if there is an answer / all the answers of the query in the facts (eventually enriched by the ontology). Enriching the facts by the ontology could either be done (1) previous to query answering by fact saturation using the ontology (forward chaining) or (2) by rewriting the query cf. the ontology and finding a match of the rewritten query in the facts (backwards chaining).

While existing work focuses on logical properties of the representation languages and their equivalence [1, 9], the state of the art employs a less principled approach when implementing such frameworks. From a practical view point, this equivalence has not yet been put forward by the means of a platform allowing for a logical, uniform view to all such different kinds of paradigms. There is indeed existing work studying different cases for optimizing the reasoning efficiency [17, 5], however, none of them look at it as a principled approach where data structures are deeply

LIRMM, University Montpellier II, France, e-mail: bplsilva, croitoru@lirmm.fr
LIRMM, INRIA, France, e-mail: baget@lirmm.fr

investigated from a storage and querying retrieval viewpoint. The choice of the appropriate encoding is then left to the knowledge engineer and it proves to be a crafty task.

In this paper we propose a generic, logic-based architecture for OCQA allowing for transparent encoding in different data structures. The proposed platform, ALASKA (**A**bstract and **L**ogic-based **A**rchitecture for **S**torage systems and **K**nowledge bases **A**nalysis) allows for storage and query time comparison of relational databases, triple stores and graph databases.

1.1 Motivation and Use Case

We will motivate and explain the contribution of ALASKA by the means of a real world case from the ANR funded project Qualinca. Qualinca is aiming at the validation and manipulation of bibliographic / multimedia knowledge bases. Amongst the partners, the ABES (French Higher Education Bibliographic Agency) and INA (French National Broadcasting Institute) provide large sets of RDF(S) data containing referencing errors. Examples of referencing errors include oeuvres (books, videos, articles etc.) mistakenly associated with the wrong author, or authors associated with a wrong / incomplete subset of their oeuvres. In order to solve such referential errors Qualinca proposed a logic based approach (as opposed to the large existing body of work mostly employing numerical measures) [11]. However, when it comes to retrieving the RDF(S) based data from ABES (or INA) and storing this data for OCQA, no existing tool in the literature could help the knowledge engineer decide what is the best system to use. Furthermore, the recent work in graph databases has surprisingly not been yet investigated in the context of OCQA (while graphs give encouraging results when used in central memory as shown in [9]).

The main contribution of ALASKA is to help make explicit different storage system choices for the knowledge engineer. 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 Qualinca, we also needed to interact with the ABES server 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).

This paper will explain the ALASKA architecture and the data transformations needed for the generic logic-based view over heterogeneously stored data over the following systems:

→ **Relational databases:** Sqlite 3.3.6[1], MySQL 5.0.77[2]
→ **Graph databases:** Neo4J 1.5[3], DEX 4.3[4]
→ **Triples stores:** Jena TDB 0.8.10[5], Sesame 2.3.2[6]

Tests were also performed over Oracle 11g. A major drawback of using it for our tests is that the software requires the initial setting of the memory usage. Since the server used [7] is not as performant as needed, Oracle did not have a lot of resources available.

2 State of the Art

Let us consider a database F that consists of a set of logical atoms, an ontology $\mathscr{O}$ written in some (first-order logic) language, and a conjunctive query Q. The OCQA 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 $\mathscr{O}$?"

Recent works in databases consider the language Datalog$^+$ [6] to encode the ontology. This is a generalization of Datalog that allows for existentially quantified variables in the hypothesis (that do not appear in the conclusion). Hence the forward chaining scheme (called here *chase*) can generate new variables while building F'. This is an important feature for ontology languages (called *value invention*), but is the cause of the undecidability of Datalog$^+$. An important field of research today, both in knowledge representation and in databases, is to identify decidable fragments of Datalog$^+$, that form the Datalog$^+_-$ family [7, 4]. In description logics, the need to answer conjunctive queries has led to the definition / study of less expressive languages (e.g. $\mathscr{EL}$ [2] and DL-Lite families [8]). Properties of these languages were used to define profiles of the Semantic Web OWL 2 [8] language. In the following we briefly present the logical language we are interested in this paper and the notation conventions we use.

---

[1] http://www.sqlite.org/
[2] http://www.mysql.com/
[3] http://www.neo4j.org/
[4] http://www.sparsity-technologies.com/dex
[5] http://jena.sourceforge.net/
[6] http://www.openrdf.org/
[7] 64-bit Quadcore AMD Opteron 8384 with 512 Kb of cache size and 12 Gb of RAM
[8] www.w3.org/TR/owl-overview

A vocabulary W is composed of a set of predicates and a set of constants. In the vocabulary, constants are used to identify all the individuals one wants to represent, and predicates represent all the interactions between individuals.

Definition 1 (Vocabulary). Let C be a set of constants, and $P = P_1 \cup P_2 \ ... \cup P_n$ a set of predicates of arity $i = 1, ..., n$ (n being the maximum arity for a predicate). We define $W = (P,C)$ as a vocabulary.

An atom on W is of form $p(t_1,...,t_k)$, where p is a predicate of arity k in W and the $t_i \in T$ are terms (i.e. constants in W or variables). For a formula ϕ, we note $terms(\phi)$ and $vars(\phi)$ respectively the terms and variables occurring in ϕ. We use the classical notions of semantic consequence ($\models$), and equivalence ($\equiv$). A conjunct is a (possibly infinite) conjunction of atoms. A fact is the existential closure of a conjunct. We also see conjuncts and facts as sets of atoms.

The full fact w.r.t. a vocabulary W contains all ground atoms that can be built on W [9]. (thus any fact on W is a semantic consequence of it). A $\forall$-rule is a formula $\forall X(H \rightarrow C)$ where H and C are conjuncts and $vars(C) \subseteq vars(H) \subseteq X$. A $\forall\exists$-rule $R = (H,C)$ is a closed formula of form $\forall x_1 \ ... \ \forall x_p \ (H \rightarrow (\exists z_1 \ ... \ \exists z_q \ C))$ where H and C are two finite non empty conjuncts respectively called the hypothesis and the conclusion of R. In examples, we omit quantifiers and use the form $H \rightarrow C$.

Definition 2 (Facts, KB). Let F a set of facts, and $\mathscr{O}$ a set of rules, we define a knowledge base (KB) $K = (F,\mathscr{O})$.

The complexity of OCQA may vary according to the type of rules of $\mathscr{O}$. Also, according to the set of rules we choose, we retrieve a semantical equivalence from one problem onto others that are (or have already been) studied in the literature. Choosing $\mathscr{O}$ as an empty set makes the problem equivalent to ENTAILMENT in the RDF [13] language. Using a set of $\forall$-rules instead will get us into the RDFS and Datalog scope. Finally, if one chooses to define $\mathscr{O}$ as a set of $\forall\exists$-rules, the problem becomes similar to the ones we find in the Datalog$\pm$ [7] and Conceptual Graphs languages [9].

While the above semantic equivalences have been shown, storage structures implementing these languages are compared in an ad-hoc manner and only focus on relational databases and 3Stores [12]. Such methodology does not ensure for logical soundness and completeness because the knowledge is not viewed across platform in a logical manner. More importantly, graph databases have not yet been considered as a storage solution in the OCQA context. In the next section we present the ALASKA architecture: a generic, logic based platform for OCQA.

[9] In this paper we use the notion of fact for any subset of atoms, contrary to the Datalog notation. and justified by "historical" reasons [9].

3 Alaska Architecture

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.

The platform architecture is multi-layered. Figure 1 represents its class diagram, highlighting the different layers.

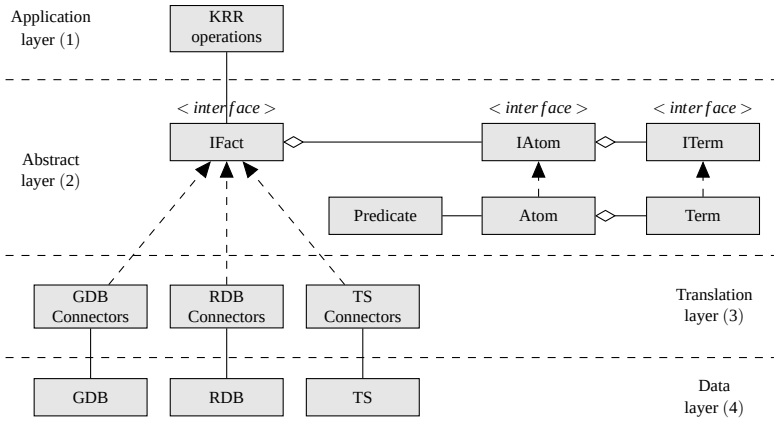

Fig. 1 Class diagram representing the software architecture.

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.

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 a graph database model and into the relational database model. Both models are detailed in

the next Subsection. Please note that the forward chaining rule application process generates new atoms and thus this process respects the insertion workflow detailed below.

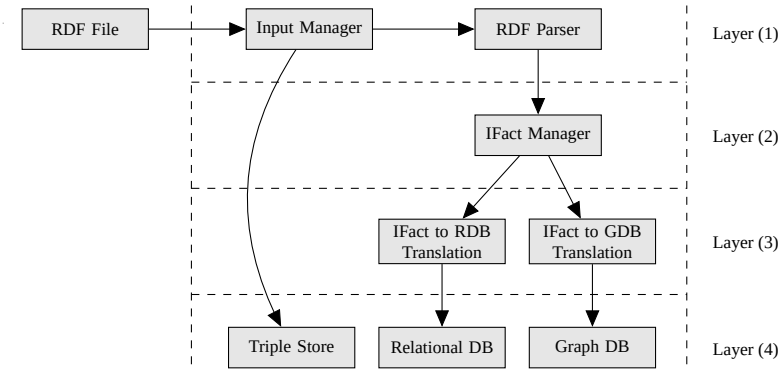

Fig. 2 Testing protocol workflow for storing a knowledge base in RDF with ALASKA.

This workflow is visualised in Figure 2 where a RDF file is stored into different storage systems. In this workflow, we have chosen not to translate the knowledge base into IFact when storing the knowledge base into a triples store since it would require the exact opposite translation afterwards. This way, when the chosen system of destination is a triples store, the Input Manager does not operate a translation but sends the information from the RDF file directly to the triples store. Such direct storage is, of course, available for every representation in ALASKA (relational databases or graphs). When evaluating the storage systems, for equity reasons, such "shortcuts" have not been used.

Finally, the querying in our architecture takes place exactly in the same manner as the storage workflow. In Figure 3, on the left hand side we show the storing workflow above and on the right hand side the querying workflow (generic algorithm or native querying mechanism).

3.1 Transformations

In this section we detail the transformations used in order to store the logical facts into the different encodings: relational databases (traditional or 3Stores) or graph databases.

Encoding facts for a relational database needs to be performed in two steps. First, the relational database schema has to be defined according to the knowledge base vocabulary. 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

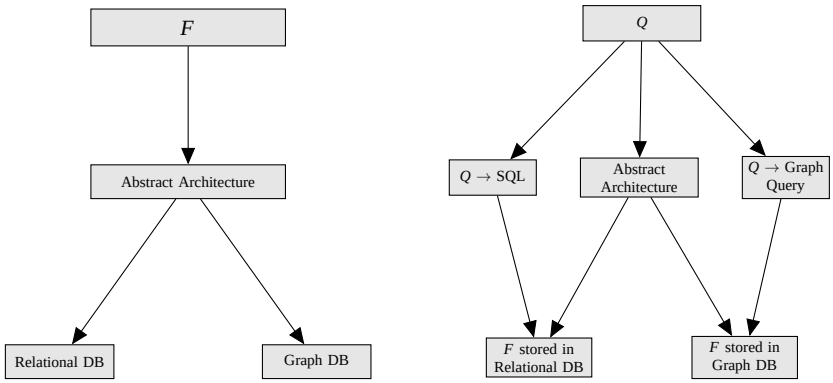

Fig. 3 ALASKA storage and querying workflow.

the vocabulary, there are two distincts manners to build the schema of the relational database: in the first case, one relation is created for each predicate in the vocabulary. The second case can only be used when all the predicates in the vocabulary share the same arity (cf. RDF [13]). In this case, only one relation is defined. This encoding is similar to the ones used for Triples Stores [14].

It has to be remembered that there are no variable terms in a relational database. Indeed, variables are frozen into fresh constants before being inserted into a table. In order to maintain this information within the base, two distinct methods exist: in the first method, every term t in the database is renamed and receives an identifier prefix: c:t or v:t according to its type. In the second case, no changes are made to the label of the terms, but an extra unary relation, R_{vars} is created and added to the schema. Variable terms are then stored inside this table. The schema of the database remains unchanged when using the first method.

Definition 3. Let R the set of relations of the relational database. $R = \bigcup_{i=1}^{n} R_i$ and $R_i = \{R_p(A_1,...,A_i) \mid \forall p \in P_i\}$. If variables are stored in an additional relation, then $R = \bigcup_{i=1}^{n} R_i \cup R_{vars}(A_1)$.

Once the database schema is defined, relation tuples are defined according to the atoms of the KB facts. $(x_1,...,x_j) \in R_{p_j}$ iff $p_j(x_1,...,x_j)$. In the particular case in which all predicates do share the same arity, the set of relations of the database $R = \{R_{k+1}(A1,...,A_{k+1})\}$. In this case, relation tuples are defined with the following: $(x_1,...,x_j,p_j) \in R_{k+1}$ iff $p_j(x_1,...,x_j)$.

Once the fact in the knowledge base is stored in the relational database, query answering can be done through SQL requests. By the definitions above, applying a SQL statement S_Q over the relational database corresponds to compute a labeled homomorphism from a conjunctive query Q into the fact [1].

Definition 4. Let $G = (N,E,l)$ a hypergraph: N is its set of vertices, E its set of hyperedges and l a labelling function.

When encoding a fact into a hypergraph, the nodes of the hypergraph correspond to the terms of the fact. Hence, $N = T$. The (hyper)edges that connect the nodes of the hypergraph correspond to the atoms in which are present the corresponding terms. $E = \bigcup_{p \in P} E_p$ with $E_p = \{(t_1,...,t_j) \mid p(t_1,...,t_j) \in A \text{ and } l(t_1,...,t_j) = p\}$.

Let us illustrate the processes detailed above by the means of an example. The knowledge base we take for example contains the following facts:

1. $\exists a,b,c,d,e\ (\ p(a,b) \wedge p(c,e) \wedge q(b,c) \wedge q(e,d) \wedge r(a,c) \wedge q(d,c)\ )$

R_p	
1	2
c:a	c:b
c:c	c:e

R_q	
1	2
c:b	c:c
c:e	c:d

R_r	
1	2
c:a	c:c
c:d	c:c

R_3		
1	2	3
c:a	c:b	p
c:c	c:e	p
c:b	c:c	q
c:e	c:d	q
c:a	c:c	r
c:d	c:c	r

Fig. 4 Encoding a fact in a relational database.

Figure 4 represents the encoding of the knowledge base facts in a relational database. As there are no variables in the example, no additional relation is added to the database schema. As all the predicates in the example share the same arity, both possibilities of schema for the relational database are represented on the figure. The three tables on the left list the content of the database when there is one table created per predicate. The table on the right shows the content of the database when one single table is created and an extra attribute is created to store the predicate of the atoms.

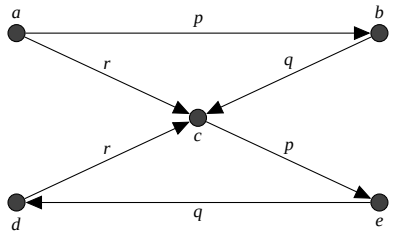

Fig. 5 Graph containing the facts (1) from Section 3.1.

Figure 5 represents the generated hypergraph after having encoded the knowledge base fact.

3.2 Storage Algorithms

In order to store large knowledge bases and to prevent the issues previously described, we have implemented the generic storage algorithm below.

Algorithm 1: Input Manager store method

Input: S a stream of atoms, f an IFact,*bSize* an integer
Output: a boolean value

```
1  begin
2      buffer ⟵ an empty array of size bSize;
3      counter ⟵ 0;
4      foreach Atom a in S do
5          if counter = bSize then
6              f.addAtoms(buffer,null);
7              counter ⟵ 0;
8          buffer[counter] = a;
9          counter++;
10     f.addAtoms(buffer,counter); return true;
11 end
```

Algorithm 1 illustrates the manner the Input Manager handles the stream of atoms received as input, as well as the manner it creates a buffer and sends atoms for the storage system in groups and not one-by-one nor all-at-once.

As one may notice, the algorithm calls the *addAtoms* method of the fact f also given as input. The genericity of ALASKA platform certifies that any storage system connected to the platform must implement this method (among others). A fact is differently encoded according to the storage system at target according to the transformations explained in Sections 3.1.

3.3 Querying Algorithms

In order to measure the querying efficiency of each of the different storage paradigms (relational databases or graphs) we have implemented a generic backtrack algorithm that makes use of the fact retrieval primitives of each system. Please note that numerous improvements are possible for this algorithms (cf. [9] or [10]) but this work is out of the scope of this paper. We are currently investigating the use of a constraint satisfaction solver (Choco) for the optimisation of the backtrack algorithm as such.

4 Evaluation

In this section we describe our strategy for evaluating ALASKA and its capability of comparing different storage systems. The results of this evaluation are presented once the data sets and respective queries for the tests are chosen.

Algorithm 2: Backtrack algorithm

```
Input: K a knowledge base
Output: a boolean value
1  begin
2      if mode = graph then
3          g ⟵ empty graph;
4      else
5          while Atom a in A do
6              n ⟵ empty array of nodes;
7              foreach Term t in a.terms do
8                  if exists node with label t then
9                      id ⟵ node.id;
10                 else
11                     create new node with id newId;
12                     id ⟵ newId;
13                 n.push(id);
14             create hyperedge with label a.predicate between all nodes in n;
15         return true;
16 end
```

As already stated in Section 1, our initial choice was to perform our tests using parts of data already available with ABES. Unfortunately such information were not available for this paper due to confidentiality reasons. By browsing the literature in the SPARQL benchmarking we found different datasets that were still pertinent to our problem. We thus used the knowledge base generator supplied by the SP2B project [16]. The generator enables the creation of knowledge bases with a certain parametrised quantity of triples maintaining a similar structure to the original DBLP knowledge base. Please refer to the paper [16] for a discussion on the relevance of this dataset. Using those generated knowledge bases then requires an initial translation from RDF into first order logic expressions (done offline, according to [3]). Please note that the initial RDF translation step has not been taken into account when reporting on storage times.

4.1 Queries

Another very interesting feature of the SP2B project is the fact that it also provides a set of 10 SPARQL queries that covers the whole specifications of the SPARQL language. As we decided to use their generated knowledge within our storage tests, we have also planned to use their set of queries for our querying tests. Using such set of queries however was not possible due to the fact that half of their queries use some SPARQL keywords (OPTIONAL, FILTER) which makes that the query cannot be directly translated into a Datalog query, which is the query language we

have chosen for our generic backtrack algorithm. We thus created our own set of queries for our knowledge bases structurally similar to the ones proposed in SP2B:

1. $type(X, Article)$
2. $creator(X, PaulErdoes) \wedge creator(X, Y)$
3. $type(X, Article) \wedge journal(X, 1940) \wedge creator(X, Y)$
4. $type(X, Article) \wedge creator(X, PaulErdoes)$

Size of the stored knowledge bases					
System	5M	20M	40M	75M	100M
DEX	55 Mb	214.2 Mb	421.7 Mb	785.1	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 Mb	11.6 Gb	15.5 Mb
Jena TDB	1.1 Gb	3.9 Gb	7.5 Gb	-	-
RDF File	533.2 Mb	2.1 Gb	4.2 Gb	7.8 Gb	10.4 Gb

Fig. 6 Table comparing knowledge bases sizes in different systems.

4.2 Results

For our tests, RDF files of 1, 5, 20, 40, 75 and 100 million triples were generated then stored on disk with storage systems selected from the available systems in ALASKA. On such tests, we not only measure the time elapsed during the storing process, but also the final size of the stored knowledge base. As time results are important in order to see which is the most efficient storage system, disk usage results are equally important in the context of OCQA.

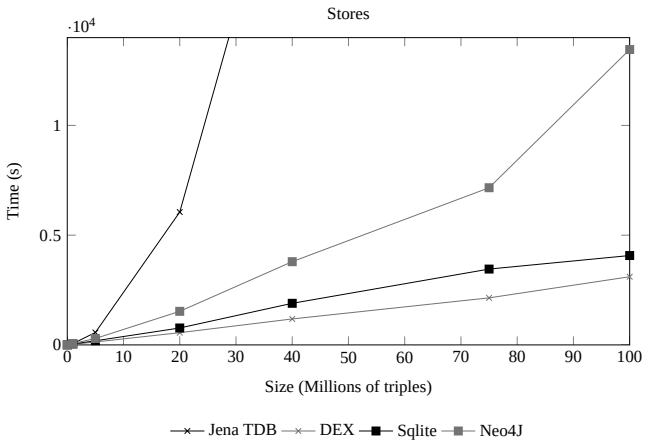

Fig. 7 Storage performance using ALASKA for large knowledge bases

Figure 7 shows the time results of the tests performed on the storage systems selected due to their intermediate performance on smaller data sets. On such tests,

we observe that DEX is the most efficient system from all the systems selected. Sqlite is second, followed by Neo4J. Jena TDB results are far from convincing. Not only its storing time is very high, but also the testing program could not go beyond the 40M triples knowledge base. We believe that such inefficiency may come from the parsing issues linked to the Jena framework. Our aim is not to optimize a storage system in particular but to be able to give a fast and good overview of available storage solutions.

Finally, Figure 8 shows the number of answers for each query in different sizes knowledge bases, while Figure 9 show the amount of time needed to perform those queries using the generic backtrack algorithm over the two best storage systems SQL Lite and DEX.

...				
Size	Q1	Q2	Q3	Q4
1 K	129 answers	104 answers	30 answers	50 answers
20 K	1685 answers	458 answers	30 answers	100 answers
40 K	3195 answers	567 answers	30 answers	100 answers
80 K	6084 answers	678 answers	30 answers	100 answers
100 K	7441 answers	721 answers	30 answers	100 answers
200 K	14016 answers	838 answers	30 answers	100 answers
400 K	25887 answers	967 answers	30 answers	100 answers

Fig. 8 Number of answers for each query

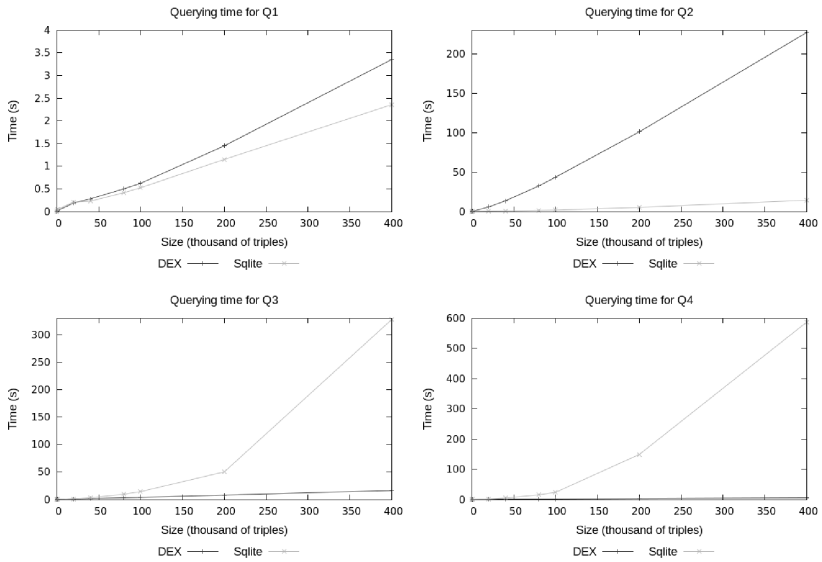

Fig. 9 Results of querying tests after having performed the 4 selected queries on the tested systems

4.3 Implementation Aspects

Storing large knowledge bases using a straight-forward 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 implemented an RDF parser which does not store the facts in main memory, but feeds them one at a time to the ALASKA input file.

The use of **transactions** also became necessary in order to store large knowledge bases properly.

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.

In order to address both transaction and garbage collection issues, an atom buffer was created. 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 Discussion

A novel abstract platform (ALASKA) was introduced in order to provide an unified logic-based architecture for ontology-based data access. In its current state ALASKA can be considered as the bottom layer of a generic KR architecture. Today, ALASKA is the only software tool that provides for a logical abstraction of ontological languages and the possibility to encode factual knowledge (DB, ABox) in one's storage system of choice. The transparent encoding in different data structures made thus possible the comparison of storage and querying capabilities of relational databases and graph databases.

While the initial aim of devising and describing a platform able to compare systems for OCQA has been archived, we are currently taking this work further in two main directions.

First we are optimising the backtrack algorithm and comparing the optimised version with the native SQL / SPAQRL engines. We are currently using a constraints satisfaction solver in order to benefit from the different backtrack optimisations in the literature. Second, we will extend our various tests over different size knowledge bases and different expressivity queries in order to be able to give "best storage system recipes" to knowledge engineers. Obtaining such comparative results is a long and tedious process (due to different implementation adaptations ALASKA needs to take in account), but the final decision support system it could generate could be of great interest for application aspects of OCQA community.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Baader, S. Brandt, and C. Lutz. Pushing the el envelope. In *Proc. of IJCAI 2005*, 2005.
3. J.-F. Baget, M. Croitoru, A. Gutierrez, M. Leclère, and M.-L. Mugnier. Translations between rdf(s) and conceptual graphs. In *ICCS*, pages 28–41, 2010.
4. 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*, pages 712–717. IJCAI / AAAI, 2011.
5. C. Basca and A. Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *ISWC Posters&Demos*, 2010.
6. 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.
7. 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.
8. D. Calvanese, G. De 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.
9. M. Chein and M.-L. Mugnier. *Graph-based Knowledge Representation and Reasoning—Computational Foundations of Conceptual Graphs*. Advanced Information and Knowledge Processing. Springer, 2009.
10. 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.
11. M. Croitoru, L. Guizol, and M. Leclère. On link validity in bibliographic knowledge bases. In *Proc. of IPMU*, 2012.
12. B. Haslhofer, E. M. Roochi, B. Schandl, and S. Zander. Europeana RDF store report. Technical report, University of Vienna, Vienna, Mar. 2011.
13. P. Hayes, editor. *RDF Semantics*. W3C Recommendation. W3C, 2004. http://www.w3.org/TR/rdf-mt/.
14. A. Hertel, J. Broekstra, and H. Stuckenschmidt. Rdf storage and retrieval systems. *Handbook on Ontologies*, pages 489–508, 2009.
15. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS 2002*, 2002.
16. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.
17. M. Sensoy, G. de Mel, W. W. Vasconcelos, and T. J. Norman. Ontological logic programming. In *WIMS*, page 44, 2011.