First Steps towards Reasoning on Big Data with DLV

Nicola Leone[1], Simona Perri[1], Francesco Ricca[1], Pierfrancesco Veltri[1,2], and Jessica Zangari[1]

[1] Department of Mathematics and Computer Science, University of Calabria, Italy
{lastname}@mat.unical.it

[2] DLVSystem s.r.l.
P.zza Vermicelli, Polo Tecnologico, Rende, Italy
veltri@dlvsystem.com

Abstract. Answer Set Programming (ASP), that extends Datalog with powerful knowledge modeling constructs, is suitable for modeling both database-oriented applications and more complex combinatorial optimization tasks arising in decision-making. However, ASP systems were not conceived having the challenges of Big Data in mind; thus they are not applicable tout court in this new setting. This paper moves the first steps towards enabling the specification of reasoning tasks on Big Data with ASP. In particular we present our ongoing work in the direction of extending the well-known DLV system to interact in a plausible way with Big Data repositories.

Keywords: ASP, Datalog, Big Data

1 Introduction

Modern information systems are facing an unprecedented phenomenon: the massive production and storage of enormous amounts of data (Big Data). The need for processing Big Data repositories has recently implied a technological switch including the development of new hardware and software architectures [10, 8] to handle the exponentially growing quantity of data coming from network connected devices, large enterprise information systems, etc. Nonetheless, one of the most relevant challenges in this setting is to get real value from Big Data by modeling, understanding, and reasoning on the information present in large data stores [13], thus enabling the development of powerful decision-making systems and high-revenue applications.

SEBD 2018, June 24-27, 2018, Castellaneta Marina, Italy.

The area of computing dedicated to representing and solving complex tasks using information is called knowledge representation and reasoning (KRR) [9]. A main goal of KRR is to introduce formalisms that simplify the task of designing and building complex applications. Among the mainstream approaches to KRR, logic-based formalisms are well-known for their expressiveness and powerful knowledge-modeling capabilities. In particular, Answer Set Programming (ASP) [3], that extends Datalog with powerful knowledge modeling constructs, is suitable for modeling both database-oriented applications and more complex combinatorial optimization tasks arising in decision-making [3]. However, ASP systems were not conceived having the challenges of Big Data in mind; thus they are not applicable tout court in this new setting.

Recently less expressive logic languages, like Datalog [16], or logic-programs with a different semantics [14], were implemented on top of Big Data platforms. However, no ASP system for Big Data is available at the moment (cfr. a recent survey [12]) and connecting ASP with mainstream technologies for Big Data is still an open problem.

This paper moves the first steps towards enabling the specification of reasoning tasks on Big Data with ASP. In particular we present our ongoing work in the direction of extending the well-known DLV system [11] (actually, its most recent version [1]) to interact in a plausible way with Big Data repositories. In our proposal DLV is connected to SQL-based data warehousing tools such as Hive via a translation component, that allows to demand the computation of space-demanding tasks/queries to well-assessed Big Data software, and leave to the standard ASP engine the burden of evaluating more computationally complex decision-making tasks over a selection of the original data.

2 Answer Set Programming

In this Section, we briefly recall syntax of the ASP language. For ASP semantics we refer the reader to [7].

A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \dots t_n$ are terms and f is a function symbol of arity n, then $f(t_1, \dots, t_n)$ is a *functional term*. If $t_1, \dots, t_k$ are terms and p is a *predicate symbol* of arity k, then $p(t_1, \dots, t_k)$ is an *atom*. A *literal* l is of the form a or *not* a, where a is an atom; in the former case l is *positive*, otherwise *negative*. A *rule* r is of the form $\alpha_1 | \cdots | \alpha_k :- \beta_1, \dots, \beta_n, not\ \beta_{n+1}, \dots, not\ \beta_m.$ where $m \geq 0$, $k \geq 0$; $\alpha_1, \dots, \alpha_k$ and $\beta_1, \dots, \beta_m$ are atoms. We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ (the *head* of r) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of r), where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ (the *positive body*) and $B^-(r) = \{not\ \beta_{n+1}, \dots, not\ \beta_m\}$ (the *negative body*). If $H(r) = \emptyset$ then r is a *(strong) constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then r is a *fact*. A rule r is safe if each variable of r has an occurrence in $B^+(r)$. An ASP program is a finite set P of safe rules. A program (a rule, a literal) is said to be *ground* if it contains no variables. A predicate is defined

We remark that this definition of safety is specific for rules featuring only classical literals. For a complete definition we refer the reader to [5].

by a rule if the predicate occurs in the head of the rule. A predicate defined only by facts is an *EDB* predicate, the remaining predicates are *IDB* predicates. The set of all facts in P is denoted by $Facts(P)$; the set of instances of all *EDB* predicates in P is denoted by $EDB(P)$.

3 The DLV2 System

DLV2 [1] is the new version of the Answer Set Programming (ASP) system DLV. The system has been completely re-engineered: it now combines $\mathcal{I}$-DLV [4], a fully-compliant ASP-Core-2 grounder, with the well-assessed solver *wasp* [2]. Besides performance improvements w.r.t. to its predecessor, DLV2 is enriched with novel features such as support to annotations and directives that customize heuristics of the system and extend its solving capabilities. Moreover, the system offers some means to ease the interoperability with external sources of knowledge. In particular, by means of some $\mathcal{I}$-DLV features, it supports connections with relational and graph databases via explicit *directives* for importing/exporting data and calls to Python functions via *external atoms*. An accurate description of the system and the underlying techniques is out of the scope of this paper; we refer the interested reader to [4, 1]. Rather, in the following, we describe in more detail external atoms, since our proposal for integrating Big Data computation and ASP explicitly relies on them.

External Computation in DLV2 As anticipated above, the new DLV2 system, via its grounder $\mathcal{I}$-DLV, supports a special form for atoms, namely *external atoms*, geared towards external computations, whose extension is specified by means of external defined Python functions.

Formally, an external atom is of the form $\&p(t_0, \ldots, t_n; u_0, \ldots, u_m)$, where $n + m \geq 0$, $\&p$ is an *external predicate*, $t_0, \ldots, t_n$ are intended as *input terms*, and are separated from the *output terms* $u_0, \ldots, u_m$ by a semicolon (";"). In the following, we denote an external atom by $\&p(In; Out)$, where In and Out represent input and output terms, respectively.

An external literal is either *not* e or e, where e is an external atom, and the symbol *not* represents default negation. An external literal is safe if all input terms are safe, according to the safety definition in ASP-Core-2 standard [5]. External literals can appear only in the rule bodies, and each instance of an external predicate must appear with the same number of input and output terms throughout the whole program.

Given an external atom $\&p(In; Out)$ and a substitution σ, a ground instance of such external atom is obtained by applying σ to variables appearing in In and Out, obtaining $\sigma(\&p(In; Out)) = \&p(In_g; Out_g)$. The truth value of a ground external atom is given by the value $f_{\&p}(In_g, Out_g)$ of a decidable $n + m$-ary two-valued oracle function, where n and m are the lengths of In_g and Out_g, respectively. A negative ground external literal *not* e is *true*/*false* if e is *false*/*true*.

Intuitively, output terms are computed on the basis of the input ones, according to a semantics which is provided externally (i.e., from the outside of the logic

program) by means of the definition of oracle functions written in Python. Basically, for each external predicate $\&p$ featuring n/m input/output terms, the user must define a Python function whose name is p, and having n/m input/output parameters. The function has to be compliant with Python version 3.

Example 1. As a simple example, let us consider the following rule, that makes use of an external predicate with two input and one output terms:

append(X,Y,Z) :– string(X), string(Y), &append_strings(X,Y;Z).

A program containing this rule must come along with the proper definition of *append_strings* within a Python function, as, for instance, the following:

```
def append_strings(X,Y): return str(X)+str(Y)
```

According to the given definition, as they are completely evaluated by $\mathcal{I}$-DLV as true or false, external predicates do not appear in the produced instantiation.

External atoms can be both functional and relational, i.e., they can return a single tuple or a set of tuples, as output. In Example 1, *&append_strings* is *functional*: the associated Python function returns a single value for each combination of the input values. In general, a functional external atom with $m > 0$ output terms must return a Python *sequence* containing m values. If $m = 1$, the output can be either as sequence containing a single value, or just a value, as in the example; if $m = 0$, the associated Python function must be boolean. A relational external atom with $m > 0$ is defined by a Python function that returns a sequence of m-sequences, where each inner sequence is composed by m values.

Example 2. The following rule uses a relational external atom:

prime_factor(X,Z) :– number(X), &compute_prime_factors(X;Z).

Intuitively, given a number X, the rule computes prime factors of X, demanding this task to a relational external atom, that receives as input the number X, and returns as output its factors. The semantics has to be provided via a Python function called *compute_prime_factors* returning a sequence of numbers each one representing a different factor of X.

4 Big DLV: ASP + Big Data

In this section we describe the main idea underlying our approach for integrating ASP computation, and in particular the DLV2 system, with Big Data sources. We first introduce a motivating example that will be used through the rest of the paper for illustrating the idea underlying our approach. Then, we provide some details about the usage of external atoms for our purposes.

4.1 KRR with Big Data

The following example showing a possible use case can be useful to grasp the intuition behind our approach.

`https://docs.python.org/3`

Let us consider the following scenario: Giovanni wants to go out for dinner and wants to invite a given number of people, possibly inviting friends of friends, in order to spend some funny and pleasant time. Invited people should be selected from a social network, choosing among the Giovanni's friendship net. In such social network subscribed people can add friends and mark some of these as close friends; moreover, the social network performs some stats on the degree of dislike among two people in the network. The following ASP program can be used to explore the friendship relations of such social network and then to provide Giovanni with suitable suggestions on people that could be invited.

$$
\begin{array}{ll}
 & nfriends(10). \quad averageAge(25). \\
r_1 & possible\_friend(Y) \text{:-} close\_friend(giovanni, Y). \\
r_2 & possible\_friend(X) \text{:-} possible\_friend(Y), close\_friend(Y, X). \\
r_3 & suggested\_friend(Y, A) \text{:-} possible\_friend(Y), person(Y, A), A > 18. \\
r_4 & invite(X) \mid \text{-}invite(X) \text{:-} suggested\_friend(X). \\
r_5 & \text{:-} \#count\{X : invite(X)\}! = N, nfriends(N). \\
r_6 & \text{:-} \#sum\{A, X : suggested\_friend(X, A, \_), invite(X)\} < AVG * N, \\
 & nfriends(N), averageAge(AVG). \\
r_7 & :\sim invite(X), suggested\_friend(X, \_), dislike(giovanni, X, D). \ [D@1, X]
\end{array}
$$

The former two rules compute the transitive closure of the friendship relation of Giovanni, restricting the computation to close friends only; rule r_3 suggests a person Y if he is a possible friend and is older than 18; rule r_4 guesses if a suggested friend should be invited or not; rule r_5 ensures that the number of invited friends is exactly the desired one; r_6 imposes that the average age of the invited friends is not smaller than a given value; eventually, rule r_7 prefers solutions in which the total degree of dislike among Giovanni and invited people is minimized.

Intuitively, computing the transitive closure of the friendship relation in a social network could be very expensive when performed on a huge database. Thus, traditional main memory ASP systems cannot handle it; indeed, also the simple import of the friend relation is not feasible in practice. So, the idea is to delegate this heavy part of the computation to an external Big Data source. The remaining part that takes as input a heavily reduced part of the friendship, instead, can be conveniently expressed in a purely declarative way by means of ASP rules as reported above. We remark that from a computational perspective, this latter part encodes a NP-hard task that cannot be expressed by means of canonical RDB query languages such as SQL, Datalog, etc.

4.2 Interfacing DLV2 with Hive via External Atoms

In order to delegate the computation of data-intensive rules to Big Data tailored software, one can make use in the ASP program of a new external atom. Roughly, the external atom connects to a DB, may export some input facts to the DB, converts some given rules to queries, executes them on the DB and returns as output the extension of a specific predicate. In order to enable the external

evaluation on a DB *db*, the ASP program should contain a rule r having in the body an external atom called &*bigasp*, that receives as input:

db	the name of the ODBC DSN (Data Source Name);
user	the name of the user who connects to the DB;
password	the password of that user;
rules	a string containing some Datalog rules that define the operations that will be performed on the DB;
output	a string representing the name of the relation to be imported in the ASP program;
input	[optional] a string containing a set of ASP facts that have to be exported to the DB.

During the evaluation, DLV2 invokes the external atom, imports the results of the evaluation and fills in the extension of the predicate *output* that has to appear as head in the rule r. Note that, this approach could be adopted to retrieve data from diverse Big Data sources.

Let us consider again our running example. We can delegate the computation of the transitive closure of the friendship relation to an external Big Data platform by replacing the first three rules in the program above with the following one:

$$\begin{aligned} &suggested\_friend(Y,A) :- \&bigasp(\text{“}my\_db\text{”}, \text{“}my\_user\text{”}, \text{“}my\_pass\text{”}, \text{“} \\ &\quad possible\_friend(Y) :- close\_friend(giovanni, Y). \\ &\quad possible\_friend(X) :- possible\_friend(Y), close\_friend(Y, X). \\ &\quad suggested\_friend(Y,A) :- possible\_friend(Y), person(Y,A), A > 18.\text{”}, \\ &\quad \text{“}suggested\_friend\text{”}; Y, A). \end{aligned}$$

Apart from connection parameters and the name of the output relation, the external atom takes as input the extracted rules; it invokes the machinery for enabling their evaluation on the external *my_db* and returns the results as a sequence of tuples representing the suggested friends with their age. Such tuples populate the extension of the *suggested_friend* relation so that the traditional ASP evaluation can continue.

5 System Architecture

We now describe an architecture where we indicate the software components that can be used to realize our idea. The general architecture of a framework for Big Data Analytics with ASP is composed by 3 macro-components as illustrated in Figure 1. In particular, the data layer of the system could rely on Hive, a data warehouse software project built on top of Hadoop for providing data summarization, query and analysis.

The choice of this tool is due to the fact that Hive is the mainstream for OLAP analysis over Big Data. Hive brings three main advantages: (i) it exploits

`http://hive.apache.org/`

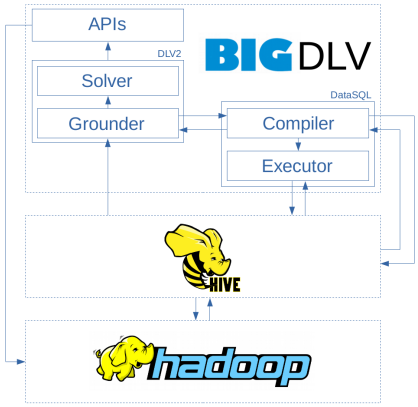

Fig. 1. General architecture of the framework for ASP reasoning over Big Data

the computational power of Hadoop clusters, (ii) it implements a standard relational model, that is equivalent to the data model supported by ASP, and (iii) it supports a querying language called H-QL, which is very close to the standard SQL. On top of Hive we find *BigDLV*, the core of our framework for Big Data Analytics. BigDLV should allow for reasoning in ASP over data stored and pre-processed within Hive. In particular, BigDLV should be able to extract extensions of input predicates from the underlying Big Data repository and, moreover, delegate parts of the computation to Hive. Figure 1 shows the internal architecture of BigDLV and focuses possible interactions with the underlying layers. This component encapsulates the DLV2 system which allows to define external atoms as discussed in Section 3: when an input program embeds a Datalog rule to be evaluated against the Hive repository, DLV2, by means of an external atom, invokes a Python component, called *DataSQL*, managing the interaction between DLV2 and Hive. DataSQL is in turn composed by two sub-modules, the *Compiler* and the *Executor*. The former compiles the rules received from DLV2 into an SQL query, while the latter runs the compiled query over Hive and retrieves the result. Results from Hive are first filtered out by the Executor and then returned back to DLV2. An end-user could ask the framework to store the final output of the whole evaluation process into the cluster. Note that, the Compiler module interacts directly with Hive in order to get information about table schemata matching the rule predicates. The implementation of the Compiler can follow the lines of [15] and thus supporting only stratified Datalog programs. Communication between DataSQL and Hive can be performed via *PyHive*, providing a collection of python DB-API interfaces for Hive.

`https://github.com/dropbox/PyHive`
`https://www.python.org/dev/peps/pep-0249`

6 Ongoing and Future Work

In this paper we described our idea for interfacing DLV2 with Big Data. Currently, we are implementing all the components mentioned in Section 4 and 5. Moreover we are developing a case of study that combines a Datalog sub-program modeling a heavy data-mining task (demanded to a Big Data platforms), with another component that models a computationally hard decision-making task (demanded to the ASP system). The case of study is part of the research project S^2BDW we are currently involved in and that focuses on a prognostic problem for foreseeing possible breaks on trains by analyzing data coming from sensors. Currently, we are focusing on applications in which the computation demanded to the external sources involves Big Data but returns an output which can be profitably handled in main memory. As a future work, we plan to better analyze the theoretical boundaries and limitations of our approach, in order to determine which kind of reasoning can be outsourced (e.g., going along the lines of [6]).

Acknowledgments. Work partially supported by MISE under project "Smarter Solutions in the Big Data World (S^2BDW)" (n. F/050389/01-02-03/X32).

References

1. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: LPNMR. Lecture Notes in Computer Science, vol. 10377, pp. 215–221. Springer (2017)
2. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 40–54. Springer (2015)
3. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
4. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I-DLV: the new intelligent grounder of DLV. Intelligenza Artificiale **11**(1), 5–20 (2017). https://doi.org/10.3233/IA-170104, http://dx.doi.org/10.3233/IA-170104
5. Calimeri, F., Ianni, G., Krennwallner, T., Ricca, F.: The answer set programming competition. AI Magazine **33**(4), 114 (2012)
6. Fan, W., Geerts, F., Libkin, L.: On scale independence for querying big data. In: PODS. pp. 51–62. ACM (2014)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. vol. 88, pp. 1070–1080 (1988)
8. Ghemawat, S., Gobioff, H., Leung, S.: The google file system. In: SOSP. pp. 29–43. ACM (2003)
9. van Harmelen, F., Lifschitz, V., Porter, B.W. (eds.): Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3. Elsevier (2008)
10. Harrison, G.: Next Generation Databases NoSQLand Big Data. Apress 2015 (2015)
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3), 499–562 (2006)
12. Lierler, Y., Maratea, M., Ricca, F.: Systems, engineering environments, and competitions. AI Magazine **37**(3), 45–52 (2016)

13. Sivarajah, U., Kamal, M.M., Irani, Z., Weerakkody, V.: Critical analysis of big data challenges and analytical methods. Journal of Business Research **70**, 263 – 286 (2017). https://doi.org/https://doi.org/10.1016/j.jbusres.2016.08.001
14. Tachmazidis, I., Antoniou, G., Faber, W.: Efficient computation of the well-founded semantics over big data. TPLP **14**(4-5), 445–459 (2014)
15. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. Theory and Practice of Logic Programming **8**(2), 129–165 (2008)
16. Yang, M., Shkapsky, A., Zaniolo, C.: Scaling up the performance of more powerful datalog systems on multicore machines. VLDB J. **26**(2), 229–248 (2017)