# The new DLV Grounder: External Computations, Interoperability and Customizability

Francesco Calimeri[1,2], Davide Fuscà[1], Giovambattista Ianni[1], Giovanni Melissari[1], and Jessica Zangari[1]

[1] Department of Mathematics and Computer Science, University of Calabria, Italy, {calimeri,fusca,ianni,melissari,zangari}@mat.unical.it

[2] DLVSystem Srl, Piazza Vermicelli, Polo Tecnologico UniCal, I-87036 Rende, Italy, calimeri@dlvsystem.com

**Abstract.** $\mathcal{I}$-DLV is the new intelligent grounder of DLV; while relying on the solid theoretical foundations of its predecessor, it has been re-designed and re-engineered ex-novo, both in algorithms and data structures. It now features full support to ASP-Core-2 standard language, deductive database capabilities, increased flexibility, significantly improved performance, and an extensible design that eases the incorporation of optimization techniques, language updates and customizability. In this paper we present $\mathcal{I}$-DLV, focusing on most recent advancements that mainly aim at easing the integration with external systems: the handling of external computations with explicit calls to Python scripts via external atoms, and interoperability mechanisms for the connection with relational and graph databases via explicit directives for importing/exporting data; furthermore, we discuss how the fine-grained customizability means allow to control the the internal grounding process, possibly positively affecting performance.

**Keywords:** KRR, Answer Set Programming, Artificial Intelligence, Deductive Database Systems, DLV, Grounding, Instantiation

## 1 Introduction

$\mathcal{I}$-DLV, the new instantiator of DLV [13], has been recently introduced in [5]; it shows good performance and stability, proving to be competitive both as an ASP grounder and as a deductive database system.

In this paper, after a brief overview of the system, we present some of the main recent advancements and ongoing work; we focus on a set of mechanisms and tools that have been introduced in $\mathcal{I}$-DLV with the aim of easing the interoperability and integration with external systems. In particular, $\mathcal{I}$-DLV now supports calls to Python scripts via *external atoms*, and connection with relational and graph databases via explicit directives for importing/exporting data. The external atoms are inspired by the ones in dlvhex [8], although relational inputs are not permitted: this choice allowed us to greatly simplify the evaluation strategy and improve the overall performance. Nonetheless, one of the main goals of the $\mathcal{I}$-DLV project is to obtain a novel, flexible tool for experimenting with ASP and its applications; to this end, it has been designed in order to allow a fine-grained control over the whole computational process, both via command-line options and inline annotations.

## 2 $\mathcal{I}$-DLV Overview

In the following, we briefly introduce the $\mathcal{I}$-DLV system; for further details we refer the reader to the more thorough discussion in [5]. The computational core of $\mathcal{I}$-DLV relies on a set of theoretical results and techniques that have already been proven to be effective in the old DLV grounder [9]. In particular, it follows a bottom-up evaluation strategy based on a semi-naïve approach [14]; this latter has been extended with a number of optimization techniques that have been explicitly designed by contextualizing it in the setting of an ASP grounder. Such techniques include [5, 9]: *rule body back-jumping*, *magic-sets*, and, in a significantly enhanced version, *body-reordering* and *indexing* strategies.

The first two techniques have been borrowed from the DLV grounder, and properly adapted to the new system architecture; the latter two have been significantly enhanced and/or extended. In particular, the body reordering technique now relies on linear interpolations, computing new statistics for variables involved in comparison predicates for covering cases that previously were not properly addressed; indexing strategies follow again a create-on-demand policy, yet featuring a more general approach: both single- and multiple-argument indices are allowed, along with a heuristics which can be used at will for selecting the best configuration.

Furthermore, a number of additional techniques and features have been designed and newly introduced in $\mathcal{I}$-DLV, that were not originally present in DLV, such as the *aligning substitutions* mechanism, the *two-fold management* of isolated variables, the *anticipated evaluation of strong constraints* [5]. As a remark, the experience over the DLV grounder proved that having a monolithic set of optimizations, most of which were activated or deactivated at the same time, does not pay in general. For such reason, even though $\mathcal{I}$-DLV comes with a general-purpose default configuration, it has been conceived in order to provide the user with a fine-grained control over the whole computational process; $\mathcal{I}$-DLV allows to enable, disable, and customize every single strategy, hence resulting in a flexible tool for experimenting with ASP and its applications.

Notably, $\mathcal{I}$-DLV introduces a new mechanism for further fine-tuning customizations, via annotations within the ASP code. One can enrich the input programs in a Java-like fashion: directives, that are embedded in comments, can express explicit steering instructions to the internal grounding process, both at global and rule level, as discussed below. It is worth mentioning also that $\mathcal{I}$-DLV fully supports the ASP-Core-2 [3] language standard.

## 3 Customizability

One of the main goals of the $\mathcal{I}$-DLV project is to obtain a novel, flexible tool for experimenting with ASP and its applications; to this end, it has been designed in order to allow a fine-grained control over the whole computational process. Indeed, all stages and tasks can be controlled, by activating/deactivating techniques and customizing them by setting proper options via command line. A comprehensive list of such customization possibilities, along with further details, is available in [5] and via the online documentation [7].

Besides the command line, system customization and tuning is further eased by a new special feature of $\mathcal{I}$-DLV: *annotations* of ASP code. Annotations and meta-data have been applied in different programming paradigms and languages; Java annotations, for instance, have no direct effect on the code they annotate: a typical usage consists in analysing them at runtime in order to change the code behaviour.

$\mathcal{I}$-DLV annotations allow to give explicit directions on the internal grounding process, at a more fine-grained level with respect to the command-line options: they "annotate" the ASP code in a Java-like fashion, while embedded in comments: hence, the resulting programs can still be given as input to other ASP systems that do not support them, without any modification. In particular, our annotations can have two different scopes: at the global level, meaning that they are applied to the whole program, or at the rule level, and hence annotations act just on the rule they precede. Syntactically, all annotations start with the prefix "%@" and end with a dot ("."). Among the annotations currently supported, we mention here those that are meant for customizing two of the major aspects of the grounding process: *body ordering* and *indexing*.

Body ordering heavily affects the efficiency of the rule grounding process: before grounding a rule, the order of literals in the body is analyzed and possibly changed in a way inspired by the database setting. Several ordering strategies have been defined and implemented in $\mathcal{I}$-DLV, based on different heuristics; they perform differently, each one featuring some advantages case by case. By means of annotations, a specific body ordering strategy can be explicitly requested for any rule, simply preceding it with the line:

```
%@rule_ordering(@value=Ordering_Type).
```

where `Ordering_Type` is a number representing an ordering strategy [5]. In addition, it is possible to specify a particular partial order among atoms, no matter the employed ordering strategy, by means of `before` and `after` directives. For instance, in the next example, $\mathcal{I}$-DLV is forced to always put literals $a(X,Y)$ and $X = \#count\{Z : c(Z)\}$ before literal $f(X,Y)$, whatever the order chosen:

```
%@rule_partial_order(
   @before={a(X,Y), X=#count{Z:c(Z)}},
    @after={f(X,Y)}).
```

The use of indices for the retrieval of matching instances from the predicate extensions is another effective optimization technique employable while grounding a rule. $\mathcal{I}$-DLV's indexing schema is very general: any predicate argument can be indexed, allowing both single- and multiple-argument indices, on the basis of a heuristics that defines a proper default configuration. $\mathcal{I}$-DLV allows the user to control the indexing strategy in order to handle situations where the default behaviour is not satisfactory. In particular, the indexing module can set per each predicate in the program a single- or multiple-index, on the desired arguments. Annotations allow to provide directives on a per-atom basis; the next annotation, for instance, requests that, in the subsequent rule, atom $a(X, Y, Z)$ is indexed, if possible, with a double-index on the first and third arguments:

```
%@rule_atom_indexed(@atom=a(X,Y,Z),
    @arguments={0,2}).
```

Multiple preferences can be expressed via different annotations; in case of conflicts, priority is given to the first appearing in the program. In addition, preferences can also be specified at a global scope, by replacing the `rule` directive with the `global` one. Such kind of annotations are applied on the rules, if possible. While a `rule` annotation must precede the intended rule, `global` annotations can appear at any line in the input program. Both `global` and `rule` annotations can be expressed in the same program; in case of overlap on a particular rule/setting, priority is given to the more specific `rule` ones.

Intuitively, the way annotations change the grounding mechanisms can noticeably affect performance on the program at hand; we discuss this also on the basis of some experimental results in Section 6.

## 4 External Computations

$\mathcal{I}$-DLV supports the call to external sources of computations within ASP programs by means of *external atoms* in the rule bodies. An external atom of the form $\&p(t_0, \ldots, t_n; u_0, \ldots, u_n)$, where $n, m \geq 0$, is an instance of an *external predicate*. The name of the external predicate starts with a & symbol, $t_0, \ldots, t_n$ are intended as *input terms*, and are separated from the *output terms* $u_0, \ldots, u_m$ by a semicolon (";"); note that an input or output term can be either a *constant* or a *variable*. 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 its input terms are safe accordingly to the ASP-Core2 definition of safety [4].

Intuitively, output terms are computed on the basis of the input ones, according to a semantics which is externally defined; currently, $\mathcal{I}$-DLV supports such definition via Python scripts. In particular, for each external predicate $\&p$ featuring $n/m$ input/output terms, the user must define a Python function whose name is $p$, and featuring $n/m$ input/output parameters. The function has to be compliant with Python[3] version 3. Note that each instance of an external predicate must appear with the same number of input and output terms throughout the program. As an example, let us consider the following rule, that makes use of an external predicate with two input and one output terms:

$$compute\_sum(X, Y, Z) \coloneq number(X), number(Y), X <= Y, \&sum(X, Y; Z).$$

A program containing this rule must come along with the proper definition of *sum* within a Python function, as, for instance, the one reported next.

```
def sum(X,Y):
    return X+Y
```

It is worth noting that the external atoms are completely evaluated by $\mathcal{I}$-DLV as true or false; hence, they never appear in the produced instantiation. Each value returned by the Python function defining an external predicate can be of one of these types: *numeric*, *boolean* or *string*[4]. These values are internally mapped to ground values accordingly to the following default policy: an integer returned value is mapped to a corresponding *numeric constant*; all other values are tentatively associated to a *symbolic constant*, if the form is compatible to the ASP-Core-2 syntax, and associated to a *string constant* in case of failure. However, $\mathcal{I}$-DLV allows the user to customize the mapping policy of a particular external predicate by means of a global *annotation* of the form

`%@global_external_predicate_conversion(@predicate=&p,@type=@T`$_1$`,...,@T`$_N$`).`

that specifies the sequence of conversion types for an external predicate $\&p$ featuring $n$ output terms. For instance, if in the program containing the previous rule the following annotation is added:

```
%@global_external_predicate_conversion(@predicate=&sum,@type=@Q_CONST).
```

where `Q_CONST` stands for quoted string, then for each external call, the output variable $Z$ is bounded to the value returned by the Python function interpreted as a quoted string. Further details about options and conversion types are available at [7]. External atoms can be both functional and relational, i.e., they can return a single tuple or a set of tuples, as output. In the example, $\&sum$ 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 *sequence*[5] containing $m$ values. If $m = 1$, the output can be either a 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

[3] https://docs.python.org/3
[4] https://docs.python.org/3/library/stdtypes.html
[5] https://docs.Python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

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.

## 5 Interoperability

As emerged from the previous sections, $\mathcal{I}$-DLV is intended to ease the interoperability of ASP with external sources of knowledge. To further comply with this purpose, its input language, still supporting ASP-Core-2, has also been enriched with explicit directives for connecting with relational and graph databases. In particular, $\mathcal{I}$-DLV inherits from DLV directives for *importing/exporting* data from/to relational DBs.

For details about syntax and usage, we refer to the online manual [7]; slight differences with respect to DLV are due to the fact that $\mathcal{I}$-DLV complies with the ASP-Core-2 syntax, hence supporting, for instances, predicate names with multiple arities.

As for graph databases, data can be imported via SPARQL queries, thanks to new directives that were not featured by DLV. *Local* DBs in RDF files and *remote* SPARQL *EndPoints* can both be queried by directives of the form:

```
#import_local_sparql("rdf_file","query",predname,predarity,[,typeConv])
#import_remote_sparql("endpnt_url","query",predname,predarity,[,typeConv])
```

where `query` is a SPARQL statement defining data to be imported and `typeConv` is optional and specifies the conversion for mapping RDF data types to ASP-Core-2 terms. For the local import, `rdf_file` can be either a local or remote URL pointing to an RDF file: in this latter case, the file is downloaded and treated as a local RDF file; in any case, the graph is built in memory. On the other hand, for the remote import, the `endpnt_url` refers to a remote endpoint and building the graph is up to the remote server; this second option might be convenient in case of large datasets. Further details are available at [7].

Concerning implementation, for a fast prototyping, we started with a solution based on external atoms. For instance, for each remote SPARQL import directive, an auxiliary rule of the following form is added to the input program: $predname(X_0, ..., X_N) :\!-\, \&sparqlEndPoint(\text{“}endpnt\_url\text{”}, \text{“}query\text{”}; X_0, ..., X_N)$. The head atom has `predname` as name, and contains a number of variables that corresponds to `predarity`. The body contains an external atom $\&sparqlEndPoint$ in charge of performing the remote query. Intuitively, when grounding the rule, the extension of the specified predicate will be filled in with information extracted by the query.

Even though this approach relying on external atoms perfectly reaches the original goal, there are some reasons in favour of a “native” support for such features. First of all, it is easy to imagine that native support should enjoy much better performance, as we discuss in Section 6.3; furthermore, in many scenarios (as it is often the case in the deductive database settings) the use of external atoms is not crucial, whilst accessing standard knowledge sources is vital: in such

| Problem | # inst. | I-DLV | | I-DLV+Annotations | |
|---|---|---|---|---|---|
| | | #solved | time | #solved | time |
| 3rd Comp. - Grammar Based | 10 | 10 | 76,01 | 10 | 22,12 |
| 6th Comp. - Complex Optimization | 20 | 20 | 70,09 | 20 | 28,51 |
| 6th Comp. - Labyrinth | 20 | 20 | 2,03 | 20 | 0,93 |
| 6th Comp. - Nomistery | 20 | 20 | 4,15 | 20 | 2,36 |

**Table 1.** Customizability: experimental results.

cases, taking care of the burden of the external Python runtime machinery does not look useful. Hence, the idea is to incorporate into the system the directives that are most likely to be used "per se", and let external atoms address cases that need extended functionalities.

## 6 Experimental evaluation

In the following we present the results of some experimental activities that have been carried out in order to assess performance of the discussed $\mathcal{I}$-DLV features.

In particular, experiments on customizability and external computations have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 processors and 128 GiB of main memory, running Linux Ubuntu 14.04.5 kernel v. 4.4.0-45-generic. Binaries were generated with the GNU C++ compiler v. 4.9. As for memory and time limits, we allotted 15 GiB and 600 seconds for each system, per each single run; while for the interoperability mechanisms, experiments have been performed on machine equipped with an Intel Core i7-4770 processor and 16GiB of main memory, running Linux Ubuntu 14.04.5 kernel v.3.13.0-107-generic. Binaries were generated with the GNU C++ compiler v. 4.9. As for memory and time limits, we allotted 15 GiB 600 seconds for each system, per each single run.

### 6.1 Playing with Customizations

Table 1 shows the impact of using ad-hoc $\mathcal{I}$-DLV configurations, rather than the default one, over a set of benchmarks taken from the Third ASP Competition [6], the Sixth ASP Competition [11]: for each benchmark the custom configuration has been defined either via command-line options or via inline annotations. It is easy to see that significant improvements can be obtained by playing with grounding options. In order to give an idea of why this happens, we illustrate a couple of interesting cases, namely Labyrinth and Visit-All, where performance is significantly affected by the possibility to choose different strategies for the body orderings from rule to rule. Let us consider the rule:

$$\begin{aligned}&reach(X,Y,T) :- reach(XX,YY,T),\\ &\quad dneighbor(D,XX,YY,X,Y),\\ &\quad conn(XX,YY,D,T),\ conn(X,Y,E,T),\end{aligned}$$

$inverse(D, E),\ step(T).$

which is taken from the encoding of *Labyrinth*. In this case, by annotating the rule with:

```
%@rule_partial_order(
   @before={inverse(D,E)},
   @after={reach(XX,YY,T),
   dneighbor(D,XX,YY,X,Y),
   conn(XX,YY,D,T), conn(X,Y,E,T),
   step(T).}).
```

that corresponds to ask $\mathcal{I}$-DLV to select as first literals $inverse(D, E)$ in its ordering strategy no matter how the other literals are positioned, average grounding time over all instances is reduced up to 90%. Similarly, by annotating the following rule in the *Visit-all* encoding:

`%@rule_ordering(@value=5).`
$atother(N, T)\ {:-}\ connected(C, N),\ C! = N,$
$\quad step(T),\ atrobot(O, T),\ O! = N.$

that requires the use of criterion $ord_5$ instead of the default one on this specific rule, the average grounding time computed over all instances reduces by 15%; the same improvement is obtained with the ordering criterion $ord_6$. Having a closer look at what changes, we can observe that while the default algorithm orders the rule body as already reported, the mentioned criteria orders the body as follows:

$atother(N, T)\ {:-}\ atrobot(O, T),\ step(T),$
$\quad connected(C, N),\ C! = N,\ O! = N.$

Let us give some insights about the reasons behind the performance improvements. In the case of Labyrinth instances, intuitively, since the extension of the predicate $inverse$ is very small, it is better to add it as soon as possible, possibly at first. As for Visit-all, both criteria $ord_5$ and $ord_6$ try to prefer literals binding output variables (i.e., variables appearing in head or in body literals with unsolved predicates) in order to facilitate the backjumping mechanism. In that rule, the output variables are $N$, $T$ and $O$, and thus the literal $atrobot(O, T)$ is inserted as soon as possible (in fact, it is the one containing the largest number of output variables).

It is worth noting that in all reported cases, acting at a global scope, as one could do via command-line options, does not bring the same improvements: indeed, experiments showed that the gain due to the change of strategy over the reported rules is overshadowed by corresponding losses over the rest of the program; the flexible customization means featured by $\mathcal{I}$-DLV, that allow to

| Problem | # inst. | DLVHEX | | GRINGO | | I-DLV | |
|---|---|---|---|---|---|---|---|
| | | #solved | time | #solved | time | #solved | time |
| Attachment | 10 | 0 | TO | 10 | 149,55 | 10 | 45,50 |
| Growth | 10 | 0 | TO/MO | 9 | 164,21 | 10 | 67,20 |
| Move | 10 | 0 | TO/MO | 9 | 163,89 | 10 | 68,08 |
| Contact | 10 | 6 | 93,05 | 10 | 11,21 | 10 | 4,94 |
| Disconnect | 10 | 8 | 127,72 | 10 | 10,85 | 10 | 4,86 |
| Discrete | 10 | 8 | 127,44 | 10 | 10,85 | 10 | 4,96 |
| Equal | 10 | 8 | 101,07 | 10 | 10,95 | 10 | 4,93 |
| Externally Connect | 10 | 8 | 100,43 | 10 | 10,79 | 10 | 4,68 |
| Nontangential Proper Part | 10 | 7 | 107,14 | 10 | 10,89 | 10 | 4,88 |
| Overlap | 10 | 6 | 92,80 | 10 | 11,11 | 10 | 4,94 |
| Part Of | 10 | 8 | 126,56 | 10 | 11,00 | 10 | 4,93 |
| Partially Overlap | 10 | 8 | 126,37 | 10 | 11,00 | 10 | 4,83 |
| Proper Part | 10 | 6 | 93,34 | 10 | 11,21 | 10 | 4,99 |
| Tangential Proper Part | 10 | 7 | 106,54 | 10 | 10,90 | 10 | 4,72 |
| String Concatenation | 5 | 0 | TO/MO | 5 | 64,73 | 5 | 52,09 |
| Prime Numbers | 10 | 1 | 93,34 | 10 | 21,94 | 10 | 13,26 |
| Reachability | 10 | 0 | TO/MO | 10 | 36,71 | 10 | 37,18 |
| **Solved Instances** | | **81/165** | | **163/165** | | **165/165** | |
| **Total Running Time** | | **59341** | | **8362** | | **3109** | |

**Table 2.** External computations: experimental results (TO/MO stands for Time/Memory Out).

configure and fine-tune the grounder as needed, even at a rule level, are exactly aimed at better dealing with such scenarios.

## 6.2 External Computations Benchmarks

We compared $\mathcal{I}$-DLV with other already available systems that support similar mechanisms for dealing with external sources of computations via Python: the ASP grounder *gringo* [10] and the *dlvhex* [8] system; in particular, we considered the latest available releases at the time of writing, respectively, *clingo* 5.2.0 executed with the `--mode=gringo` and *dlvhex* 2.5.0 executed with the default provided ASP solver that combines *gringo* 4.5.4 and *clasp* 3.1.4.

It is worth stating that our aim was not to asses the ASP computation capabilities/performance of the systems; rather, we wanted to assess their efficiency at integrating external computations: hence, we first properly adapted a set of already-proposed problems, and then we enriched them with further domains testing different aspects.

The first set of benchmarks is focused on the spatial representation and reasoning domain; these problems originally appeared in [15]. In this setting, two scenarios have been taken into account:

– The first scenario requires the determination of relations among randomly-generated circular objects in a 2-$D$ space. For each pair of circles one is interested in knowing which of the following relations hold: having some

contacts, being disconnected, being externally connected, overlapping or partially overlapping, one being part of the other, one being proper part of the other, one being tangential proper part of the other, one being non-tangential proper part of the other. For each possible relation, an ASP encoding that makes use of an external Python script checks if it holds. The encodings have been paired with instances of increasing sizes containing random generated circles, from 10 to 190.
– In a second scenario we re-adapted the encodings of *Growth*, *Move* and *Attachment* problems introduced in [15], that solves some geometrical problems over triples of circular objects in a 2D space. Again, instances of increasing size have been given to the tested systems: in this case, we generated triples of circles, from 7 to 70.

Notably, ASP-Core-2 only supports integer numbers; hence, the encodings have been re-adapted in order to result independent from the way data are expressed. Each object is associated with an identifier, and information about coordinates and dimensions are stored in a csv file; thus, from the ASP side, objects are managed via their ids, and computations involving real numbers are handled externally via the same Python scripts that, in turn, are invoked by the external mechanisms typical of each tested system. Hence, the encodings reported in [15] have been carefully translated into ASP-Core-2 encodings. In this respect, it is worth noticing that in [15] two versions for problem Attachment are reported: in our setting, due to the described translation, they coincide.

Since the benchmarks introduced above involve non-disjunctive stratified encodings and only numeric (integer) constants as ground terms, we considered three further domains: the reachability problem, where edges are retrieved via Python scripts; concatenation of two randomly-generated strings with arbitrary lengths varying from 1000 to 3000 chars; generation of first $k$ prime numbers, with $k$ ranging from 0 to 100000.

Results, reported in Table 2, show satisfactory performance for $\mathcal{I}$-DLV, both in comparison with *gringo*, which solves approximately the same number of instances but spending larger times, and with *dlvhex*, which, yet offering a more complete support for external source of knowledge, suffers from its architecture that makes use of an ASP solver as a black box.

## 6.3 Interoperability Benchmarks

We wanted to analyze the effective gain on performance obtainable with a native support of SQL/SPARQL local import directives against the same directives implemented via Python scripts (see Section 5). In particular, we compared two different importing approaches: (1) a version exploiting explicit directives natively implemented in `C++`, (2) a latter version where the import mechanism is performed externally.

The benchmarks are divided into two categories:

– Importing data from a Relational Database, by means of SQL statements. To this end, a DB containing a randomly generated table has been created: such

| Problem | I-DLV-C++ | I-DLV-Python |
|---|---|---|
| sql-100K | 0,55 | 1,63 |
| sql-200K | 0,98 | 3,06 |
| sql-300K | 1,49 | 4,58 |
| sql-400K | 2,07 | 6,19 |
| sql-500K | 2,47 | 7,61 |
| sql-600K | 2,99 | 9,20 |
| sql-700K | 3,51 | 10,64 |
| sql-800K | 4,17 | 11,94 |
| sql-900K | 4,69 | 13,24 |
| sql-1M | 5,02 | 15,19 |
| sparql-lubm -1 | 7,86 | 13,92 |
| sparql-lubm-2 | 16,62 | 29,49 |
| sparql-lubm-3 | 25,03 | 44,39 |
| sparql-lubm-4 | 31,74 | 56,86 |
| sparql-lubm-5 | 38,08 | 66,85 |

**Table 3.** Interoperability: experimental results.

table contains 1000000 tuples and features three columns, one of integer type and two of alphanumeric type. Several encodings have then been tested, each one importing a different number of tuples from the aforementioned table, ranging from 100000 to 1000000. In both cases, each SQL column is mapped, respectively, to a numeric term, a symbolic constant and a string constant (we refer the reader to ASP-Core-2 term types [4]).

– SPARQL imports from a local RDF file. In particular, we generated some OWL ontologies via Data Generator(UBA) [12]. Such ontologies are referred to a University context: each university has a number of departments ranging from 15 to 22. The generated encoding selects graduate and undergraduate students, and each encoding imports the students from a different number of universities, ranging from 1 to 5.

Results, reported in Table 3, show that the native approach clearly outperforms the other by 66% when dealing with SQL directives, and by 43% when dealing with SPARQL local import directives. Intuitively, an internal management of import/export mechanism can be performed directly interfacing `C++` and SQL/SPARQL, while with external atoms Python acts as a mediator causing an overhead which is not always negligible as our tests evidenced.

## 7 Conclusion and Ongoing Work

$\mathcal{I}$-DLV is a project actively under development; besides improvements of the features presented above, both in functionalities and performance, further enhancements are planned, and relate to language extensions, customizability means, performance, and a tight integration with the ASP solver *wasp* [2] in the new full-fledged ASP system DLV2 [1]. More in detail, more native directives for interoperating with external data will be added; the $\mathcal{I}$-DLV language will be extended with constructs for explicitly managing complex terms such as prolog-like lists; the set of annotations for a fine-grained control of the grounding process will be enlarged; a new set of annotations for integrating and tuning *wasp* will be added; in addition, we are studying a proper way of manipulating the produced ground program in order to better fit with the computational mechanisms carried out by the solver.

## References

1. Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP System DLV2. In *14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017), Espoo, Finland*, volume To appear, 2017.
2. Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, pages 40–54, 2015.
3. Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. Asp-core-2: Input language format, 2012.
4. Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: 4th ASP Competition Official Input Language Format, 2013. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf.
5. Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.
6. Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The Third Open Answer Set Programming Competition. *TPLP*, 14(1):117–135, 2014.
7. Francesco Calimeri, Simona Perri, Davide Fuscà, and Jessica Zangari. $\mathcal{I}$-DLV homepage, since 2016. https://github.com/DeMaCS-UNICAL/I-DLV/wiki.
8. Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *TPLP*, 16(4):418–464, 2016.
9. Wolfgang Faber, Nicola Leone, and Simona Perri. The Intelligent Grounder of DLV. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, pages 247–264. Springer, 2012.
10. Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, pages 345–351, 2011.
11. Martin Gebser, Marco Maratea, and Francesco Ricca. The Design of the Sixth Answer Set Programming Competition. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *LNCS*, pages 531–544. Springer, 2015.
12. Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.
13. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, July 2006.

14. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
15. Przemyslaw Andrzej Walega, Carl P. L. Schultz, and Mehul Bhatt. Non-Monotonic Spatial Reasoning with Answer Set Programming Modulo Theories. *CoRR*, abs/1606.07860, 2016.