Experiencing ASP with real world applications

G. Terracina[1], E. De Francesco[1], C. Panetta[1], N. Leone[1]

Dipartimento di Matematica, Università della Calabria,
I-87036 Rende (CS), Italy
terracina,defrancesco,panetta,leone@mat.unical.it

Abstract

Disjunctive logic programming under answer set semantics (DLP, ASP) is a powerful formalism for knowledge representation and reasoning. The language of DLP is very expressive, and allows for modelling complex combinatorial problems. However, despite the high expressiveness of this language, the success of DLP systems is still dimmed when the applications of interest become data intensive (current DLP systems work only in main memory) or they involve some inherently procedural sub-tasks or the handling of complex data structures. The main goal of this paper is precisely to improve efficiency and usability of DLP systems in these contexts, and verify these improvements by a benchmarking activity on real-world applications. We present a DLP system which: *(i)* carries out as much as possible of the reasoning tasks in mass memory without degrading performances, thus allowing to deal with data-intensive applications; *(ii)* extends the expressiveness of DLP with external function calls, yet improving efficiency (at least for procedural sub-tasks) and knowledge-modelling power; *(iii)* extends the expressiveness of DLP for supporting also the management of recursive data structures (lists). We test the system on four main areas: data-integration, combinatorial problems, data transformation, and string similarity computation. The experimental results are very encouraging: the proposed system can handle significantly larger amounts of data than competitor systems, and it is also faster in response time.

1 Introduction

Disjunctive logic programming under answer set semantics (DLP, ASP) is a powerful rule-based formalism for knowledge representation and reasoning. The recent development of efficient DLP systems, like DLV [9], Cmodels [7], Gnt2 [8], and ClaspD [5], has renewed the interest for DLP in modern application areas.

However, current DLP systems present three main drawbacks in real world scenarios: they are not capable of handling data intensive applications (they work in main memory only), they are not well suited for modelling inherently procedural problems, and they can not reason about recursive data structures and infinite domains, such as XML/HTML documents, time, etc.

Proceedings of the 15th International RCRA workshop (RCRA 2008):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Udine, Italy, 12–13 December 2008

Recently, we presented a database-oriented variant of DLV, namely DLV^{DB} [15], representing a first step towards overcoming these drawbacks. In fact, [15] carries out all of its tasks in mass memory, thus enabling data intensive applications, but only for limited forms of reasoning (only disjunction free, stratified programs are allowed).

The goal of our current work is to enhance DLV^{DB} features to improve its efficiency and usability in the contexts outlined above, for an effective exploitation of DLP in real world scenarios. The proposed enhancements include: *(i)* full support to disjunctive datalog with unstratified negation, and aggregate functions; *(ii)* extension of DLP with external function calls, particularly suited for solving inherently procedural sub-tasks but also for improving knowledge-modelling power; *(iii)* extension of DLP to support list terms; *(iv)* an evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory, thus enabling complex reasonings in data intensive applications without degrading performances.

In order to make the above enhancements possible, various challenges had to be faced:

1. Data intensive applications usually must access, and modify, data stored in autonomous enterprise databases and these should be accessed also by other applications.

2. Evaluating the stable models of an ASP program directly in mass-memory data-structures, could be highly inefficient.

3. Using the main memory to accommodate both the input data (hereafter, EDB) and the inferred data is usually impossible for data intensive applications due to the limited amount of available main memory.

4. The introduction of functions and list terms makes the evaluation of programs more complex.

Note that, from points 2. and 3. it comes out that some amount of data *must* be loaded in main memory, but this should be as small as possible.

In order to face challenge 1. DLV^{DB} is interfaced with external databases via ODBC. ODBC allows a very straightforward way to access and manipulate data over, possibly distributed, databases. Note that challenge 1. makes the adoption of deductive systems integrating proprietary DBMSs not effective.

As far as challenge 2. is concerned we adopt a mixed strategy, which is outlined in Section 3; intuitively, the evaluation can be divided in two distinct phases: the grounding and the model generation. Grounding is completely performed in the database, whereas the model generation is carried out in main memory; this allows also to address challenge 3. In fact, in several cases, only a small portion of the ground program is actually needed

for the model generation phase, since most of the inferred data is "stable" and belongs to every stable model (and is already derived during the grounding phase). Finally, as for challenge 4. we exploit database stored functions to implement external function calls; these are also the basis for supporting list terms, which are handled with suitable manipulation functions.

It is worth pointing out that the resulting ASP system is very powerful: it can encode any computable function in a rich and fully declarative language, allowing to solve very complex combinatorial problems.

We have dedicated special attention also to efficiency; in fact, while language extensions and mass memory evaluations usually tend to degrade systems efficiency, our implementation presents comparable, and in several cases even better, performances than competitor main memory systems, yet allowing the handling of the highest amounts of data. The proposed system has been in fact compared with state-of-the-art ASP systems. Test results, reported in the paper, show that DLV^{DB} is well suited for data intensive applications both for time and space requirements.

2 System Language

The language supported by the proposed system is disjunctive datalog, extended with functions, aggregates, and list terms. Syntax and semantics of this language are sketched next; details can be found in [2, 3]. Rules accepted by the system have the form:

$$\alpha_1 \vee \cdots \vee \alpha_k \text{ :- } \beta_1, \ldots, \beta_n, \text{not } \beta_{n+1}, \ldots, \text{not } \beta_m. \quad (1)$$

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$, are ordinary atoms, and $\beta_1, \ldots, \beta_m$ are (ordinary, external, or aggregate) atoms. External and aggregate atom predicate names are conventionally preceded by "#". Arguments of atoms are terms that can be either constants, variables, or lists. Rules with $k = 0$ and $m > 0$ are called *constraints*, whereas rules such that $k = 1$ and $m = 0$ are called *facts*.

An example of external atom could be #concat(X,Y,Z), which takes two strings X and Y as input and returns a string Z corresponding to the concatenation of X and Y. Examples of aggregate functions are `#count` (number of terms) and `#sum` (sum of rational numbers).

A list term can be defined using the following two forms:

- $[t_1, \ldots, t_n]$ where t_1, ..., t_n are terms;
- $[h|t]$ where h (the head of the list) is a term, and t (the tail of the list) is a list term.

Moreover, two special external atoms are reserved for lists manipulation, namely #head(L,H), which receives a list L and returns its head H, and

#tail(L,T), which returns the tali T of L.[1]

Functions introduced in the program by external atoms are expected to be defined as scalar stored functions in the database coupled with DLV^{DB}; in fact, as it will be clear in the following, DLV^{DB} performs most of its evaluations directly on a working database specified by the user. Moreover, programs must be value-invention restricted (cfr. [2]), i.e. new values possibly introduced by external atoms must not propagate through recursion; this avoids the generation of infinite-sized answer sets.

Stored functions in databases can return only one scalar value; as a consequence, DLV^{DB} adopts the convention that the last variable of the external atom corresponds to the result returned by the function call, whereas all the other variables are the input for the stored function.

3 Evaluation Strategies

The proposed evaluation strategy puts its basis on the sharp distinction existing between the grounding of the input datalog program and the generation of its stable models. Then, two distinct approaches can be adopted depending on whether the input program is non disjunctive and stratified (in this case everything can be evaluated on the DBMS) or not. The former case has already been addressed in [15]; here we concentrate our attention on the latter one and on the new features of the system.

Evaluation of disjunctive programs with unstratified negation The evaluation strategy we adopt carries out the grounding completely in the database, by the execution of suitable SQL queries. This phase generates two kinds of data: ground atoms (facts) valid in every stable model (and thus not requiring further elaboration in the model generation phase) and ground rules, summarizing possible values for a predicate and the conditions under which these can be inferred.

Facts compose the so called *solved* part of the program, whereas ground rules form the *residual program*, not completely solved by the grounding. As previously pointed out, one of the main challenges in our work is to load the smallest amount of information as possible in main memory; consequently, the residual program generated by the system should be as small as possible.

Model generation is then executed in main memory with the technique described in [9].

[1] Actually, our current implementation imposes some restrictions on the generic definition of list terms. In particular, in $[t_1, \ldots, t_n]$ only (possibly nested lists of) constants are allowed, whereas in $[h|t]$, h can be either a constant or a variable and t can be only a variable. Note that these restrictions, coupled with the availability of #head and #tail do not limit language expressiveness.

Definition 1. Let p be a predicate of a program $\mathcal{P}$, p is said to be *unsolved* if: *(i)* it is in the head of a disjunctive rule; or *(ii)* it is the head of at least one rule involved in unstratified negation; or *(iii)* the body of a rule having p as head contains at least one unsolved predicate. p is said to be *solved* otherwise.

In our evaluation strategy, a solved predicate is associated with facts only in the ground program and, thus, with *certainly-true* values, i.e. instantiations of the predicate that make the resulting atom true in every stable model. On the contrary, an unsolved predicate p may be defined by both facts (certainly-true values) and ground rules; the latter identify *possibly-true* values for p, i.e. instantiations of p that make the resulting atom true in some stable models.

Given an unsolved predicate p we indicate the set of its certainly-true values as p^s and the set of its possibly-true values as p^u.

As previously pointed out, rules having an unsolved predicate may generate ground rules in the instantiation. Since we are interested in generating the smallest residual program as possible, ground rules are "epurated" of certainly-true values.

Definition 2. A *simplified ground rule* (g-rule in the following) of a program $\mathcal{P}$ is a ground rule not involving any certainly-true values of $\mathcal{P}$.

It is now possible to illustrate the evaluation strategy implemented in our system. Given a program $\mathcal{P}$, the evaluation is carried out in five steps:

> **Step 1.** Preprocess $\mathcal{P}$ for the database-oriented instantiation; this produces a rewriting $\mathcal{P}'$ of $\mathcal{P}$;
>
> **Step 2.** Translate each rule of $\mathcal{P}'$ into a corresponding SQL statement;
>
> **Step 3.** Compose and execute the query plan of statements generated in Step 2 on the DBMS;
>
> **Step 4.** Generate the residual program and load it in the Model Generator of DLV;
>
> **Step 5.** Execute the residual program in main memory and show the results.

Step 1. The objective of Step 1 is to "prepare" rules of $\mathcal{P}$ to be translated in SQL almost straightforwardly, in order to generate a residual programs as small as possible. In more detail, for each rule r in $\mathcal{P}$ three kinds of rule are generated in $\mathcal{P}'$:

A. If the head of r has one atom only, a rule (hereafter denoted as A-rule) is created for deriving only certainly-true values of r's head; note that if r is disjunctive no certainly-true values can be derived from it.

B. A set of rules (hereafter, B-rules) supporting the generation of the g-rules of r. The heads of these rules contain both the variables of unsolved predicates in the body of r and the variables in the head of r. Ground values obtained for these variables with B-rules are then used to instantiate r with possibly-true values only.

C. A set of rules (hereafter C-rules) for generating the set of possibly-true values of unsolved predicates as projections on B-rules obtained previously.

Step 2. Translation of the rules obtained in Step 1. into SQL is carried out with the technique already presented in [15] for non disjunctive and stratified programs.

Step 3. In order to compile the query plan, the dependency graph D associated with $\mathcal{P}$ is considered [10]. In particular, D allows the identification of a partially ordered set $\{\text{Comp}_i\}$ of program components where lower components must be evaluated first.

Then, given a component Comp and a rule r in Comp, if r is not recursive, then the corresponding portion of query plan is as follows[2]: *(1)* evaluate (if present) the A-rule associated with r; *(2)* evaluate each B-rule obtained from r; *(3)* for each predicate in the head of r evaluate the corresponding C-rule.

If r is recursive, the portion of query plan above must be included in a fix-point semi-naïve evaluation, as described in [15].

Step 4 and 5. The generation of the residual program requires the analysis of values derived by B-rules only. Then, for each rule r and each corresponding B-rule (say, r.B_i), r is instantiated with values of r.B_i; during this phase some checks are suitably carried out to remove certainly-true values from the corresponding g-rule instances. The residual program is then loaded in main memory for the generation of stable models. Note that each answer set found on this residual program shall be enriched with certainly-true values determined during the grounding.

More details on the implementation of Steps 1-5 are presented in [14].

Evaluation of programs with functions Recall that, by convention, given an external atom $\#f(X_1, \ldots, X_n, O)$ used in a rule r, only the last variable O can be considered as an output parameter, while all the other variables must be intended as input for f. This corresponds to the function call $f(X_1, \ldots, X_n) = O$ on the DLV^{DB} working database. Moreover, O can be: *(i)* bound to other variables in r's body, *(ii)* bound to a constant, *(iii)*

[2]Here, for simplicity of exposition, we refer to rules, indicating that the corresponding SQL statements must be evaluated on the database.

a variable of r's head. Then, in the SQL statement corresponding to r, a function call is introduced in the WHERE part to implement cases *(i)* and *(ii)* and in the SELECT part to implement case *(iii)*.

As an example, consider the rule: mergedNames(ID, N) :- person(ID, FN, LN), #concat(FN, LN, N). This rule belongs to case *(iii)* above and is translated into:

```
INSERT INTO mergedNames
(SELECT person.ID, concat(person.FN,person.LN) FROM person);
```

Evaluation of programs with list terms In our approach, list terms are handled by suitable function calls; in particular, programs containing list terms are automatically rewritten to contain only terms and function calls. Three basic operations can be singled out to handle lists: *(i)* initialization, *(ii)* packing of a term as head of a list, *(iii)* unpacking of a list in the head term and its tail.

Lists are internally handled as strings, starting (resp., ending) with a '[' (resp., ']') where terms are separated by a ','. Initialization is then implicitly implemented by the transformation of the list in a string (recall that we currently limit lists of the form $[t_1, \ldots, t_n]$ to contain only – possibly nested – lists of constants).

Packing of a list is carried out by a function `#pack` which receives a term H and a list T and returns the list L=[H|T].[3] E.g. the rule p([H|T]):- dom(H),list(T) is translated into p(L):-dom(H), list(T), #pack(H,T,L).

Handling the unpacking is a bit more tricky. In fact, the corresponding function should return two values (the head and the tail) but database stored functions can output one value only and can not have side effects on existing tables. Then, unpacking of a list must be carried out through two different calls to functions `#head` and `#tail` introduced in Section 2.

As an example, a rule of the form q(H):- dom(H), list(T), list([H|T]) is translated into q(H):-dom(H), list(T), list(L), #head(L,H), #tail(L,T).

The corresponding SQL statement will then be

```
INSERT INTO q (SELECT dom.H,
FROM dom, list l1, list l2 WHERE head(l1.L)=dom.H AND tail(l1.L)=l2.L).
```

Note that availability of #head and #tail functions allows also the manipulation of nested lists.

As a final remark, in order to simplify the evaluation process, we currently associate each occurrence of a list term in the head (resp., body) of a rule with a call to #pack (resp., #head and #tail). This may be not always the best choice in terms of efficiency, but provides a very easy way to compose multiple lists in the same rule.

[3] Here and in the following functions handling lists are supposed to be already loaded on the working database.

4 Experiments and Benchmarks

In order to asses the performance of the proposed system, we carried out several tests on four categories of real world applications: data-integration, combinatorial problems, data transformation, and string similarity computation. Next we report results obtained for each kind of test in a separate section. All tests have been carried out on a Pentium IV with 500Mb of RAM. The working database of DLV^{DB} was defined on Microsoft SQL Server 2005.

4.1 Testing on a real data integration setting

In this section we describe the tests we carried out in querying inconsistent and incomplete data. We exploited the data integration framework developed in the INFOMIX project (IST-2001-33570) [4] which integrates real data from a university context.

Compared systems We compared DLV^{DB} with state-of-the-art ASP systems, namely DLV [9], Gnt2 [8], ClaspD [5], Smodels [11], and Cmodels [7]. DLV^{DB} and DLV include an internal proprietary grounder, whereas the other systems require an external grounder; we tested both Lparse [13] and GrinGo [6] for this purpose; precisely, given a grounder x and a system y, we run $x|y$ so as to direct the output of x into y; the output of the systems have been directed to null in order to eliminate printing times from the computation of the overall execution times.

It is worth pointing out that all systems but DLV^{DB} and DLV do not explicitly support non-ground queries; in order to carry out our tests, we asked these systems to compute all answer sets. However, since tested queries are all non-ground (see below) answer sets must be all computed anyway. Note also that Smodels and GrinGo do not support disjunction; since the data integration framework required some disjunctive rules for handling data inconsistencies, we adopted a semantic preserving rewriting when using these systems to remove disjunctions[4].

Tested queries We tested four queries, ranging from simple selections to more complicated ones. Two of these queries have been also used for studying the scalability of tested systems:

- Q_1: select the student IDs and the course descriptions of the examinations they passed (this query involves possible inconsistencies in student IDs, exam records, and course descriptions).

[4] We used ClaspD also for non disjunctive programs with GrinGo. However, we checked that running times of Clasp are the same as those of ClaspD in these queries.

- Q_2: select the first and second names of the professors stored in the database (this query involves possible inclusion dependency violations in relationships involving professors, and possible inconsistencies in exam records).
- Q_3: select pairs of students having at least one common exam (this query involves possible inconsistencies in student IDs and exam records). We leveraged the complexity of this query by filtering out different subsets of exam records.
- Q_4: select pairs of students and course codes of passed examinations such that the professor's first name of the corresponding courses is the same (this query involves possible inconsistencies in student IDs, exam records, and course descriptions). We leveraged the complexity of this query by filtering out different subsets of exam records.

All tested queries are non-ground. Due to space constraints we can not show here their complete encodings. The interested reader can find them in the on-line Appendix [1].

Results and discussion Test results are shown in Figure 1. In the graphs, we used the notation x:y to denote the system y coupled with the grounder x; moreover, to simplify the notation, we used symbol L (resp. G) to denote Lparse (resp. GrinGo).

Results of queries Q_1 and Q_2 are shown in Figure 1(a). We can observe that the amount of data involved by these queries is still manageable by all tested systems in main memory. DLV^{DB} and DLV present comparable performances and they are at least 50% faster than other systems. In these queries, there is no substantial difference in using Lparse or GrinGo.

The scalability of query Q_3 is illustrated in Figure 1(b). Here (and in Figure 1(c)) the line of a system stops when it (or the associated grounder) has not been able to solve the query. Note that no system but DLV^{DB} has been capable of handling 100% of input data, due to lack of memory. Specifically, for this query, grounders were able to complete the computation, but systems not. As for obtained results, it is possible to observe that in this query, when coupled with GrinGo, systems behave generally better than with Lparse, at least for small inputs. Performances of DLV^{DB} are comparable to those of the other systems with Lparse for small inputs, but it behaves much better for bigger data sizes. Notably ClaspD with GrinGo presents the best performance for Q_3 until it is able to handle data in main memory.

Results for query Q_4 are shown in Figure 1(c). Here, Lparse has not been able to complete the grounding in reasonable time even for the smallest data set (we stopped it after 12 hours). Hence, only results with GrinGo are presented (which has been able to complete the grounding for plotted

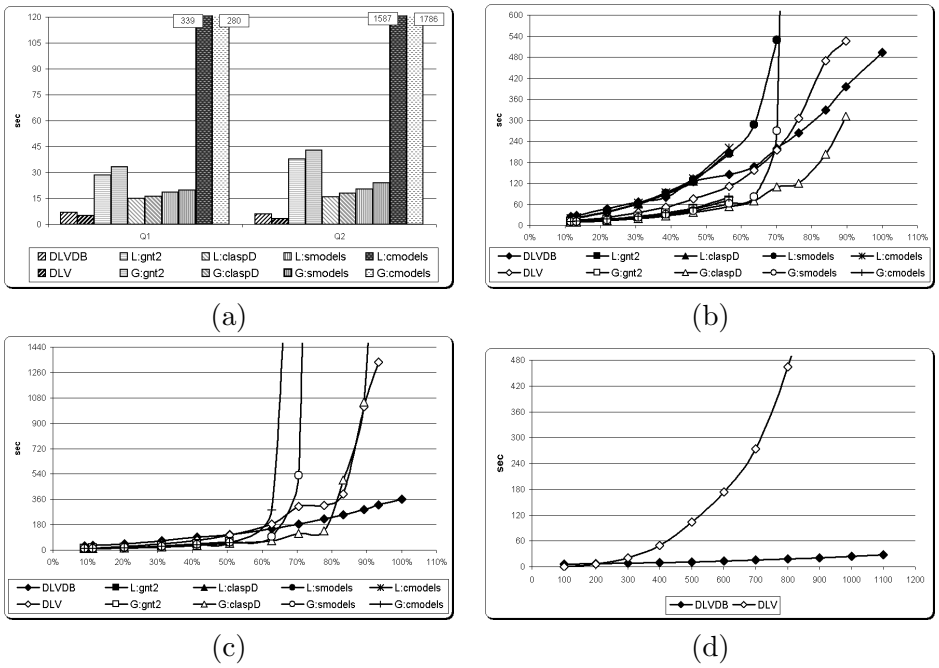


Figure 1: Results for : (a) Q_1, and Q_2; (b) Q_3; (c) Q_4; (d) FastFoods.

data). Here, again, DLV^{DB} allows handling bigger data sizes than the other systems which, at some point, are subject to memory overflow. Also, the performances of DLV^{DB} in small data sets are extremely competitive.

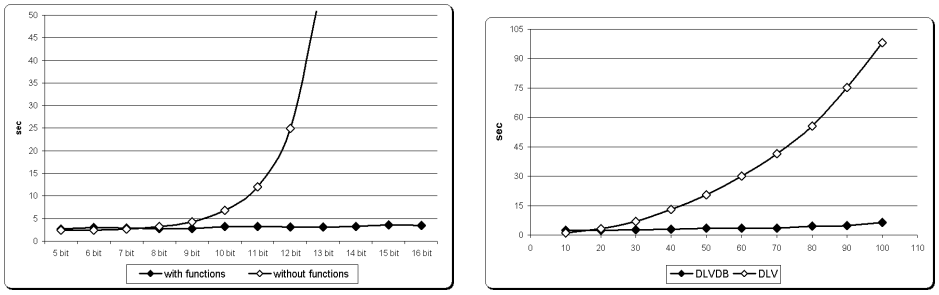
4.2 Testing on a combinatorial problem

In this test, we considered a combinatorial problem, we call it FastFoods, which checks whether a depot allocation has minimal supply costs among all depot allocations of the same cardinality. Inputs to the problem are a set of *restaurants* and a set of *depots*, each characterized by a Name and a Position (Km). The output is an alternative set of *depots*, if available. The complete encoding of this problem can be found in the on-line Appendix [1].

Note that we could test only DLV and DLV^{DB} on this problem. In fact, the encoding of FastFoods is heavily based on aggregate functions, especially assignment aggregates which are not supported by the other systems.

Results showing response times for increasing numbers of restaurants are illustrated in Figure 1(d) [5]. It clearly emerges that DLV^{DB} is much more effective than DLV in aggregating data for increasing input sizes; this can be justified by the fact that DLV^{DB} exploits DBMS aggregation functions during the grounding.

[5]We fixed the number of depots to 50.



(a) (b)

Figure 2: Results for (a) Int2Bin, (b) HammingDistances.

4.3 Testing on data transformation problems

We tested the capability to improve usability and efficiency of DLV^{DB} via functions for a typical real world problem, namely data transformation. Data transformation is particularly relevant, e.g. in data integration, to uniform data formats among different sources.

In particular, we considered the problem of transforming integer numbers in their binary representation. This task can be encoded both in pure datalog and in datalog with functions (see the on-line Appendix [1]). We then designed a test program, named Int2Bin, aiming simply at transforming integers stored in an input table to binaries. We defined two variants of Int2Bin, one with and one without function calls. In order to measure the scalability of DLV^{DB} in this test, we considered output binary numbers having 5 to 16 bits. Obtained results are shown in Figure 2(a).

The figure clearly shows the significant advantage of using functions in this context. In fact, the execution time of Int2Bin with functions is almost constant because it requires a fixed number of function calls (one for each mark to convert), independently of the number of bits. To the contrary, the standard datalog version must generate all the binary numbers in the admissible range; this explains the exponential growth of the response time.

4.4 Testing on string similarity computation

String similarity computation is an important task in several application areas. In particular, in Bioinformatics, it is essential for measuring several parameters between portions of DNA or proteins and to identify frequently repeated patterns. ASP (with some extensions) has already been exploited also in this context, see e.g. [12].

In this test, we considered the computation of the Hamming distance between pairs of strings, which is at the basis of several similarity measures. It is defined as the number of positions in which the corresponding symbols of two strings of the same length are different. This problem is inherently procedural and, even if a declarative solution for it is possible, this is quite

unnatural.

We then considered the following problem, referred as HammingDistances in the following: given a set of strings compute the Hamming distance between each string pair. Note that, in classical ASP, in order to properly compute the hamming distance, strings must be represented as a set of pairs (POS, CHAR); to the contrary, a function-based solution can directly handle the whole string.

We then designed two encodings for the problem, one using functions and one not; specifically, in the former case input strings are represented as `string(ID,S)`, whereas in the latter case, strings are expressed as `string(ID, CHAR, POS)`. Note that we did not count the time for converting the strings from one format to the other in our tests. In both cases, the output has the form `hd(ID1, ID2, H)`. The complete encodings can be found in the Appendix [1].

Results are shown in Figure 2(b) for increasing numbers of input strings. The gain provided by DLV^{DB} is similar to that we have observed in the previous test, thus confirming the advantage of using functions to solve procedural sub-tasks.

5 Conclusions

In this paper we shown how efficiency and usability of DLP systems can be improved for their exploitation in real world applications. Specifically, we presented some enhancements to the DLV^{DB} system devoted to improve both its expressiveness and its efficiency.

Our extensive experimental evaluation showed that proposed improvements make DLV^{DB} particularly suited for data intensive applications and that DLV^{DB} can exemplify the usage of DLP for those problems characterized by both declarative and procedural components.

As for future work we plan to further improve the program evaluation techniques, especially in answering user queries. In this context, we plan to exploit query unfolding techniques and query distribution approaches.

References

[1] http://www.mat.unical.it/terracina/rcra08/Appendix.pdf.

[2] F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50:333–361, 2007.

[3] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. of the 18th Int. Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, 2003.

[4] N. Leone et al. The infomix system for advanced integration of incomplete and inconsistent data. In *Proc. of 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, 2005. ACM Press.

[5] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *Clasp* : A conflict-driven answer set solver. In *Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Tempe, AZ, USA*, pages 260–265, 2007.

[6] M. Gebser, T. Schaub, and S. Thiele. GrinGo : A new grounder for answer set programming. In *Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Tempe, AZ, USA*, pages 266–271, 2007.

[7] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Jornal of Automated Reasoning*, 36(4):345–377, 2006.

[8] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. *TOCL*, 7(1):1–37, 2006.

[9] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, July 2006.

[10] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, 1997.

[11] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In *Proc. of the 8th Int. Workshop on Non-Monotonic Reasoning (NMR'2000)*, Colorado, USA, April 2000.

[12] L. Palopoli, S. Rombo, and G. Terracina. Flexible pattern discovery with (extended) disjunctive logic programming. In *Proc. of 15th International Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, pages 504–513, Saratoga Springs, New York, USA, 2005. Lecture Notes in Artificial Intelligence (3488), Springer-Verlag.

[13] T. Syrjänen. Lparse 1.0 user's manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[14] G. Terracina, E. De Francesco, C. Panetta, and N. Leone. Enhancing a DLP system for advanced database applications. In *Proc. of International Conference on Web Reasoning and Rule Systems (RR 2008)*, Karlsruhe, Germany, 2008. Lecture Notes in Computer Science, Springer.

[15] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165, 2008.