On Applying Deductive Databases to Inductive Logic Programming: a Performance Study

Tiago Soares, Michel Ferreira, Ricardo Rocha, and Nuno A. Fonseca

DCC-FC & LIACC
University of Porto, Portugal
{tiagosoares,michel,ricroc,nf}@ncc.up.pt

Abstract. Inductive Logic Programming (ILP) tries to derive an intensional representation of data (a *theory*) from its extensional one, which includes positive and negative examples, as well as facts from a *background knowledge*. This data is primarily available from relational databases, and has to be converted to Prolog facts in order to be used by most ILP systems. Furthermore, the operations involved in ILP execution are also very database oriented, including selections, joins and aggregations. We thus argue that the Prolog implementation of ILP systems can profit from a hybrid execution between a logic system and a relational database system, that can be obtained by using a coupled deductive database system. This hybrid execution is completely transparent for the Prolog programmer, with the deductive database system abstracting all the Prolog to relational algebra translation. In this paper we propose several approaches of coding ILP algorithms using deductive databases technology, with different distributions of work between the logic system and the database system. We perform an in-depth evaluation of the different approaches on a set of real-size problems. For large problems we are able to obtain speedups of more than a factor of 100. The size of the problems that can be solved is also significantly improved thanks to a non-memory storage of data-sets.

1 Introduction

The amount of data collected and stored in databases is growing considerably in almost all areas of human activity. A paramount example is the explosion of bio-tech data that, as a result of automation in biochemistry, doubles its size every three to six months [1]. Most of this data is structured and stored in relational databases and, in more complex applications, it can involve several relations, thus being spread over multiple tables. However, many important data mining techniques look for patterns in a single relation (or table) where each tuple (or row) is one object of interest. Great care and effort has to be made in order to squeeze as much relevant data as possible into a single table so that propositional data mining algorithms can be applied. Notwithstanding this

preparation step, propositionalizing data from multiple tables into a single one may lead to redundancy, loss of information [2] or to tables of prohibitive size [3].

On the other hand, Multi-Relational Data Mining (MRDM) systems are able to analyse data from multiple relations without propositionalizing data into a single table first. Most of the multi-relational data mining techniques have been developed within the area of Inductive Logic Programming (ILP) [4]. However, on complex or sizable applications, ILP systems suffer from significant limitations that reduce their applicability in many data mining tasks. First, ILP systems are computationally expensive - evaluating individual rules may take considerable time, and thus, to compute a model, an ILP system can take several hours or even days. Second, most ILP systems execute in main memory, therefore limiting their ability to process large databases. Efficiency and scalability are thus two of the major challenges that current ILP systems must overcome.

The main contribution of this paper is thus the proposal of applying Deductive Databases (DDB) to ILP, allowing the interface with a relational database system to become transparent to the ILP system. In particular, we will use the MYDDAS system [5], which couples YapTab [6] with MySQL [7], as the DDB system, and April [8], as the ILP system. Being able to abstract the Prolog to SQL translation, we concentrate in describing and evaluating several high-level coupling approaches, with different distributions of work between the logic system and the database system. These alternative coupling approaches correspond to different formulations in Prolog of relational operations, such as joins and aggregations, that are transparently implemented by the DDB system. We evaluate the different approaches on a set of real-size problems, showing that significant improvements in performance can be achieved by coupling ILP with DDB, and that the size of the problems solved can be significantly increased due to a non-memory storage of the data-sets.

The remainder of the paper is organised as follows. First, we discuss the main aspects of a typical coupling interface between a logic programming system and a relational database. Then, we introduce some background concepts about ILP and describe a particular ILP algorithm. Next, we show how to improve ILP algorithms efficiency by performing the coverage computation of rules with the database system. We then present some experimental results and end by outlining some conclusions.

2 Coupling a Logic System with a Relational Database

On a coupled DDB system, the predicates defined extensionally in database relations usually require a directive such as:

```
:- db_import(edge_db,edge,my_conn).
```

This directive associates a predicate `edge/2` with the relation `edge_db` that is accessible through a connection with the database system named `my_conn`. In MYDDAS, what this directive does is asserting a clause such as the one in Fig. 1.

```
edge(A,B) :-
   translate(proj_term(A,B),edge(A,B),SQLQuery),
   db_query(my_conn,SQLQuery,ResultSet),
   db_row(ResultSet,[A,B]).
```

Fig. 1. Asserted clause for an imported database predicate

Of the three predicates in Fig. 1, `translate/3`, `db_query/3` and `db_row/2`, the simplest one is `db_query/3`. This predicate uses the connection identifier, `my_conn`, to send an SQL query, `SQLQuery`, to the database system that executes the query and returns a pointer to the set of matching tuples, `ResultSet`.

The `db_row/2` predicate is more interesting. It usually navigates through the result set tuple-at-a-time using backtracking. It unifies the current tuple in the result set with the arguments of a list or some other structure. Several optimizations can be implemented for `db_row/2` [9]. The most obvious is replacing the unification by a simple binding operation for the unbound variables, since normally the SQL query already returns only the tuples that *unify* with the list arguments. Another interesting feature of `db_row/2` is how it handles pruning through a *cut* over the result set [10].

The most interesting predicate is `translate/3`, which translates a query written in logic to an SQL expression that is understood by database systems [11]. For example, the query goal `'?- edge(10,B).'` will generate the call `translate(proj_term(10,B),edge(10,B),SQLQuery)`, exiting with `SQLQuery` bound to `'SELECT 10, A.attr2 FROM edge_db A WHERE A.attr1=10'`.

The `translate/3` predicate can still be used to implement a more complex division of work between the logic system and the database system. Suppose we write the following query goal:

```
?- edge(A,B), edge(B,A).
```

A DDB system might decide that this query is more efficiently executed if the joining of the two `edge/2` goals is performed by the database system, instead of by the logic system. The system will then generate the `translate/3` call `translate(proj_term(A,B),(edge(A,B),edge(B,A)),SQLQuery)` to obtain the query `'SELECT A.attr1, A.attr2 FROM edge_db A, edge_db B WHERE B.attr1=A.attr2 AND B.attr2=A.attr1'`. In MYDDAS, we use the `db_view/3` primitive to accomplish this. For the example given above, we should use a directive like `db_view(direct_cycle(A,B),(edge(A,B),edge(B,A)),my_conn)`, which will assert a clause for `direct_cycle/2` in a similar way to what was done for the `db_import/3` directive in Fig. 1. This is known as *view-level access*, in opposition to the previous *relation-level access*.

The `translate/3` predicate also allows specifying database logic goals that include higher-order operations, such as aggregate functions that compute values over sets of attributes. Although higher-order operations are not part of the relational database model, virtually every database system supports aggregate functions over relations, such as `sum()`, `avg()`, `count()`, `min()` and `max()` which compute the sum, the average, the number, the minimum and the maximum of given attributes.

In `translate/3`, aggregate functions are represented as a binary subgoal in the database goal, mapping the predicate symbol of such subgoal to the aggregate function name in the database system. The first argument of the subgoal is mapped to the attribute over which the aggregate function is to be computed and the second argument specifies the relation goal. The projection term is specified to include the result of the aggregation. As an example, if we want `translate/3` to generate an SQL query to compute the number of tuples from predicate `edge/2` that depart from point `10` we would write:

```
?- translate(count(X),(X is count(B,edge(10,B))),SQLQuery).
```

and this would bind `SQLQuery` to `'SELECT COUNT(A.attr2) FROM edge_db A WHERE A.attr1=10'`.

3 Inductive Logic Programming

The normal problem that an ILP system must solve is to find a consistent and complete *theory*, from a set of examples and prior knowledge, the *background knowledge*, that explains all given positive examples, while being consistent with the given negative examples [4]. In general, the background knowledge and the set of examples can be arbitrary logic programs. We next describe ILP execution in more detail by using the classical *Michalski train problem* [12].

In the Michalski train problem the theory to be found should explain why trains are travelling eastbound. There are five examples of trains known to be travelling eastbound, which constitutes the set of positive examples, and five examples of trains known to be travelling westbound, which constitutes the set of negative examples. All our observations about these trains, such as size, number, position, contents of carriages, etc, constitutes our background knowledge. We present in Fig. 2 the set of positive and negative examples, together with part of the background knowledge (describing the train *east*1).

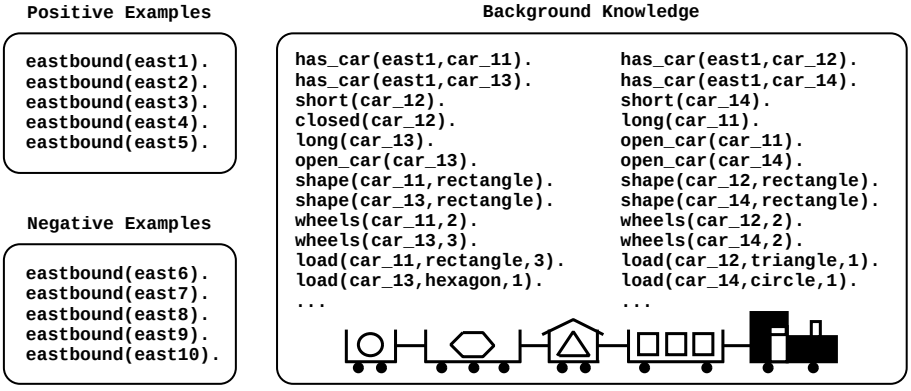

Fig. 2. Examples and background knowledge for the Michalski train problem

To derive a theory with the desired properties, many ILP systems follow some kind of *generate-and-test* approach to traverse the *hypotheses space* [13,

14]. A general ILP system spends most of its time evaluating hypotheses, either because the number of examples is large or because testing each example is computationally hard. For instance, a possible sequence of hypotheses (clauses) generated for the Michalski train problem would be:

```
eastbound(A) :- has_car(A,B).
eastbound(A) :- has_car(A,C).
eastbound(A) :- has_car(A,D).
eastbound(A) :- has_car(A,E).
eastbound(A) :- has_car(A,B), short(B).
eastbound(A) :- has_car(A,B), open_car(B).
eastbound(A) :- has_car(A,B), shape(B,rectangle).
eastbound(A) :- has_car(A,B), wheels(B,2).
eastbound(A) :- has_car(A,B), load(B,circle,1).
...
```

For each of these clauses the ILP algorithm computes its *coverage*, that is, the number of positive and negatives examples that can be deduced from it. If a clause covers all of the positive examples and none of the negative examples, then the ILP system stops. Otherwise, an alternative stop criteria should be used, such as, the number of clauses evaluated, the number of positive examples covered, or time. A simplified algorithm for the coverage computation of a clause is presented next in Fig. 3. In the evaluation section we name this approach the *Basic ILP Approach*.

```
compute_coverage(Clause,ScorePos,ScoreNeg) :-
   assert(Clause),
   reset_counter(pos,0),
   (
      positive_examples(X),
      process(Clause,X,GoalP),
      once(GoalP),
      incr_counter(pos),
      fail
   ;
      true
   ),
   counter(pos,ScorePos),
   reset_counter(neg,0),
   (
      negative_examples(Y),
      process(Clause,Y,GoalN),
      once(GoalN),
      incr_counter(neg),
      fail
   ;
      true
   ),
   counter(neg,ScoreNeg),
   retract(Clause).
```

Fig. 3. Coverage computation

Consider now that we call the `compute_coverage/3` predicate for clause `'eastbound(A) :- has_car(A,B), short(B).'`. Initially, it starts by asserting the clause to the program code, resetting a counter `pos`, and by calling the

predicate representing the positive examples. The `positive_examples/1` predicate binds variable `X` to the first positive example, say `east1`, and the `process/3` predicate creates the `eastbound(east1)` goal, which is called using the `once/1` primitive. The `once/1` primitive is used to avoid backtracking on alternative ways to derive the current goal. It is defined in Prolog as `'once(Goal) :- call(Goal), !.'`. If the positive example succeeds, counter `pos` is incremented and we force failure. Failure, whether forced or unforced, will backtrack to alternative positive examples, traversing all of them and counting those that succeed. The process is repeated for negative examples and finally the asserted clause is retracted.

4 Coupling ILP with a Deductive Database System

The time spent in the coverage computation of the rules generated by an ILP system represents the larger percentage of the total execution time of such systems. In this section we describe several approaches to divide the work between the logic system and the relational database system in order to maximize overall efficiency of the coverage computation. We will present this coverage computation starting with its original implementation, and then incrementally transferring computational work from the logic system to the database system. In what follows, we name each of the approaches in order to compare their performance in the evaluation section.

4.1 Relation-Level Approach

In coupled DDB systems the level of transparency allows the user to deal with relationally defined predicates exactly as if they were defined in the Prolog source code. Basically, predicates are transparently mapped to database tables or views. The extensionally defined predicates are mapped to tables, while the intensionally defined predicates are mapped to views. This mapping scheme provides a transparent solution for the designer of the ILP engine (if the system is implemented in a first order language like Prolog). However, it results in increased communication with the relational database system since many accesses are made to evaluate each single clause.

In particular, the `compute_coverage/3` predicate of Fig. 3 can be used when the background knowledge and the positive and negative examples are declared through the `db_import/3` directive. However, using only the `db_import/3` directives, the coverage computation uses *relation-level access* to retrieve the tuples from the database system. This means an uneven division of work between the logic system and the database system. We name this approach the *Relation-Level Approach*.

4.2 View-Level Approach

A fundamental improvement to the *Relation-Level Approach* is to transfer the joining effort of the background knowledge goals, in the body of the current clause, to the database system.

In MYDDAS, we use the `db_view/3` predicate to convert the relation-level accesses in *view-level accesses*, as explained in section 2. Following our previous example, instead of asserting the clause `'eastbound(A) :- has_car(A,B), short(B).'`, we now create the view using the directive `db_view(view(A,B), (has_car(A,B),short(B)),my_conn)` and assert the clause `'eastbound(A) :- view(A,B).'`. As the view just has to outlive the coverage computation of the current clause, an useful optimization is to use a predicate `run_view/3` which calls the view without asserting it: `'eastbound(A) :- run_view(view(A,B), (has_car(A,B),short(B)),my_conn).'`. The coverage computation algorithm of Fig. 3 works as before, with the joining computation performed now by the database system. We name this approach the *View-Level Approach*.

4.3 View-Level/Once Approach

Some very important issues in using the database system to compute the join of the goals in the body of the current clause and the algorithm of Fig. 3 arise for the `once/1` primitive. Not only the coupling interface must support deallocation of queries result sets through a `'!'` [10], but also the pruning of unnecessary computation work to derive alternative solutions is not being done by `once/1`, as intended. The database system has already computed all the alternative solutions when the `'!'` is executed. Reducing the scope of the join is thus fundamental and, for a given positive or negative example, the database system only needs to compute the first tuple of the join.

In order to reduce the scope of the join computed by the database system, we should push the `once/1` call to the database view. Therefore, the asserted clause includes a `once/1` predicate on the view definition which we can efficiently translate to SQL using the `'LIMIT 1'` SQL keyword. For our example, the asserted clause is now: `'eastbound(A) :- run_view(view(A,B),once(has_car(A,B), short(B)),my_conn)'`. For the first positive example `east1` the SQL expression generated for the view is: `'SELECT A.attr1, A.attr2 FROM has_car_db A, short_db B WHERE A.attr1='east1' AND B.attr1=A.attr2 LIMIT 1'`.

We can now drop the `once/1` primitive from the call on the code of Fig. 3. We name this approach the *View-Level/Once Approach*.

4.4 Aggregation/View Approach

A final transfer of computation work from the logic system to the database system can be done for the aggregation operation which counts the number of examples covered by a rule. The *Basic ILP Approach* uses extra-logical global variables to perform this counting operation, as it would be too inefficient without this feature.

To transfer the aggregation work to the database system we need to restrict the theories we are inducing to non-recursive theories, where the head of the clause can not appear as a goal in the body. With this restriction, we can drop the assertion of the current clause to the program code and use the `db_view/3` predicate with the aggregation operation `count/1` on an attribute of the relation holding the positive or negative examples. Also, the view now includes the positive or negative examples relation as a goal co-joined with the goals in the body of the current clause. Again, the join should only test for the existence of one tuple in the body goals for each of the examples. We introduce a predicate `exists/1`, similar to `once/1`, extending again the Prolog to SQL compiler, which will be translated to an SQL expression involving an existential sub-query. For our example clause, `'eastbound(A) :- has_car(A,B), short(B).'`, the view used to compute positive coverage would be the following:

```
db_view(count_examples(P),
        P is count(A,(eastbound(A),exists(has_car(A,B),short(B)))),
        my_conn).
```

which generates the SQL expression:

```
SELECT COUNT(A.attr1) FROM eastbound_db A
WHERE EXISTS (SELECT * FROM has_car_db B, short_db C
              WHERE B.attr1=A.attr1 AND B.attr2=C.attr1 LIMIT 1)
```

Although the `'LIMIT 1'` primitive may seem redundant for an existential sub-query, our experiments showed that MySQL performance is greatly improved if we include it on the sub-query. We name this approach the *Aggregation/View Approach*. Figure 4 presents a simplified algorithm for the coverage computation using this approach.

```
compute_coverage(':-'(Head,Body), ScorePos, ScoreNeg, Conn) :-
   process(pos, Head, HeadPos, AggrArgPos),
   run_view(count_positive_examples(ScorePos),
            (ScorePos is count(AggrArgPos,(HeadPos,exists(Body)))), Conn),
   process(neg, Head, HeadNeg, AggrArgNeg),
   run_view(count_negative_examples(ScoreNeg),
            (ScoreNeg is count(AggrArgNeg,(HeadNeg,exists(Body)))), Conn).
```

Fig. 4. Coverage computation with the database

Although our coverage computation predicate is very simple to implement in the context of a DDB, it brings with it a complex set of features which have been the subject of recent research in the implementation of ILP systems. The first of these features is efficient higher-order computation. Reasoning about a set of values is typically inefficient in Prolog, as we usually have to build the set of values and then traverse them again to compute the desired function. This can be overcome, as shown in Fig. 3, using non-logical extensions such as global variables. Database systems have efficient aggregation algorithms.

A second feature is powerful indexing. Typical Prolog systems indexing is restricted to the first argument of clauses. The inefficiency of this indexing scheme

for ILP algorithms, where efficient selections and joins are fundamental, motivated the development of the just-in-time, full arguments, indexing of the Yap Prolog system 5.0 [15]. Database systems allow the creation of a variety of index types over all the attributes of a relation.

A third feature is goal-reordering. The coverage computation goal is just to compute the number of positive and negative examples covered by a clause. The execution order of the goals in the body of a clause is irrelevant. Query optimization of database systems executes the computation of the join on the involved relations in the most efficient way, using transformations which are similar to goal-reordering in Prolog execution.

Another feature is parallelism. By using a parallel database system we can have the aggregation, selection and joining operations implemented using the parallel algorithms of parallel database systems.

5 Performance Evaluation

In order to evaluate and analyse the different approaches for coverage computation, we used the April ILP system [8] to obtain sets of hypotheses which are generated during the ILP algorithm search process. We then implemented the five different approaches for coverage computation through simple Prolog programs, as explained in the previous sections, and measured the time taken by each on the different sets of hypotheses. Implementing the different approaches in April and using April's execution time as the measure, did not allow us to do a precise performance evaluation. April can use different heuristics, optimizations and search strategies, and the time taken in the search process can mislead the real speed-up obtained in the different coverage computation approaches described in this paper.

We used MYDDAS 0.9, coupling Yap 5.1.0 with MySQL Server 4.1.5-gamma, on a AMD Athlon 64 Processor 2800+ with 512 Kbytes cache and 1 Gbyte of RAM. Yap performs a just-in-time, full arguments, indexing. We have used five ILP problems: the Michalski train problem [12] and four artificially generated problems [16]. Table 1 characterizes the problems in terms of number of examples, number of relations in the background knowledge, number of clauses generated, and number of tuples.

Problem	Examples	Relations	Clauses	Tuples
train	10	10	68	240
p.m8.l27	200	8	495	321,576
p.m11.l15	200	11	582	440,000
p.m15.l29	200	15	687	603,000
p.m21.l18	200	21	672	844,200

Table 1. Problems characterization

The clauses were randomly generated and equally distributed by length, ranging from 1 to the maximum number of relations. The clauses were then evaluated using each of the described approaches.

5.1 Coverage Performance

Table 2 shows the best execution time of five runs, in seconds, for coverage computation using our five approaches in each ILP problem.

Approach	Problem				
	train	**p.m8.l27**	**p.m11.l15**	**p.m15.l29**	**p.m21.l18**
Basic ILP	0.002	15.331	50.447	33,972.225	>1 day
Relation-Level	0.515	35,583.984	>1 day	>1 day	>1 day
View-Level	0.235	n.a	n.a	n.a	n.a
View-Level/Once	0.208	99.837	628.409	2,975.051	33,229.210
Aggregation/View	0.105	5.330	14.850	251.192	734.800

Table 2. Coverage performance for the different approaches

The **train** problem is a toy problem, useful to explain how an ILP algorithm works, but totally non-typical with respect to actual problems approached through ILP. The background knowledge together with the positive and negative examples totals less than 300 tuples (facts). This number clearly fits in memory and the communication overhead with a database represents most of the execution time, as the select or join operations involve very few tuples. We included this example as it is the only one where we could obtain execution times for all of the approaches. With regard to the database approaches, this example already shows gradual improvements as the computation work is transferred to the database engine, from 0.515 seconds using the *Relation-Level Approach* to 0.105 seconds using the *Aggregation/View Approach*. For this example, all the queries are executed almost instantly, and the time difference just translates the number of queries that are sent to the database system, which decreases from the *Relation-Level Approach* to the *Aggregation/View Approach*. Even sending a no-action query to the database system and obtaining the result set, takes a core execution time, which explains the difference to the *Basic ILP Approach* for this very small problem.

For the larger problems, the core time of communication between the logic system and the database system becomes diluted as we increase the computation work of the database system. For problems involving thousands of tuples in a number of relations, the *Relation-Level Approach* is unrealistic. This approach does not transfer any computation work to the database system, other than selecting tuples from individual relations. Also, the number of queries generated is a factor of the number of tuples in each relation, which explains execution times of days or weeks for problems larger than **p.m8.l27**.

For the *View-Level Approach*, as expected, we could not obtain the execution times for the large problems, due to insufficient memory to compute the joins involved. Note that this approach does not implement the `once/1` optimization,

therefore the entire join is computed instead of just the first tuple. MySQL runs out of memory when trying to compute a join of several relations, each with thousands of tuples.

For the *View-Level/Once Approach* the scope of the join is now reduced to compute just the first tuple. For problem **p.m11.l15** the slow-down factor compared to the *Basic ILP Approach* is explained by the number of queries that are sent to the database system, one for every positive and negative example. For the **p.m*** problems this means that for each of the 688 clauses a total of 200 queries (the number of positive and negative examples) are sent to the database system. As the size of the joins grows larger, as with problem **p.m15.l29**, the core time of these 200 queries becomes irrelevant compared to the time taken for computing the joins. This and the huge amount of backtracking performed by the *Basic ILP Approach* for the two largest problems, explains the speedup obtained with this approach.

On the *Aggregation/View Approach* only two queries are sent to the database system per clause, one to compute positive coverage and one to compute negative coverage. All the coverage computation work is transferred to the database system, and the core time of sending and storing the result set for just two queries is insignificant. The performance gains over the *Basic ILP Approach* are clear: a 2.8 speedup for problem **p.m8.l27**, a 3.4 speedup for problem **p.m11.l15** and a 135 speedup for the **p.m15.l29**. These results show a clear tendency for higher speedups as the size of the problems grow.

The results obtained with the *Aggregation/View Approach* are very significant. Not only we can now handle problems of size two orders of magnitude larger, thanks to the non-memory storage of data-sets, but we are also able to improve the coverage computation execution time by a very significant factor.

5.2 Coverage Analysis

The results presented in the previous section compare the different approaches for the coverage computation. In order to achieve a deeper insight on the behaviour of the DDB system usage, and therefore clarify some of the results obtained, we present in Table 3 data related to several activities of the coverage computation. These statistics were obtained by introducing a set of counters to measure the several activities. The columns in this table have the following meaning:

transl: the percentage of time spent on the `translate/3` predicate translating Prolog to SQL.

server: the percentage of time spent by the database server processing queries.

transf: the percentage of time spent in transferring result sets from the database to Prolog.

db_row: the percentage of time spent on the `db_row/2` predicate. It measures the time spent in unifying the results of the queries with the variables of the logic system.

prolog: the percentage of time spent on normal Prolog execution and not measured by the other activities.

queries: the total number of queries sent to the database server by the Prolog process.

rows: the total number of rows returned for the queries made. In parenthesis, it shows the amount of data transferred in KBytes.

Problem/Approach	Activities						
	transl	server	transf	db_row	prolog	queries	rows
train							
Relation-Level	9.4%	67.6%	2.0%	1.9%	19.1%	3402	5402(53)
View-Level/Once	9.6%	69.3%	1.3%	1.4%	18.4%	924	1362(7)
Aggregation/View	47.0%	41.4%	0.6%	0.5%	10.6%	154	154(0)
p.m08.l27							
View-Level/Once	4.6%	89.5%	0.3%	0.2%	5.5%	100378	236800(998)
Aggregation/View	1.5%	97.3%	0.1%	0.0%	1.1%	990	990(2)
p.m11.l15							
View-Level/Once	1.2%	97.2%	0.1%	0.1%	1.4%	117776	254000(1090)
Aggregation/View	0.8%	98.6%	0.0%	0.0%	0.6%	1164	1164(3)
p.m21.l18							
View-Level/Once	0.0%	99.9%	0.0%	0.0%	0.1%	100378	264055(1155)
Aggregation/View	0.0%	99.9%	0.0%	0.0%	0.0%	1344	1344(3)

Table 3. Activities analysis for the different approaches

For the **train** problem, the time spent on the database server is comparatively small to the other data-sets. The queries calculated are very small and easy to compute, so the other activities of the coverage computation gain relevance.

Considering only the problems that have a more interesting size, the data-set **p.m08.l27** presents the highest percentage of time spent on the **transl** activity. This can be explained by the fact that **p.m08.l27** is the smallest problem. As the size of the problems grow, the total execution time increases, therefore lowering the impact of the `translate/3` predicate on the final execution time. In fact, test results show that for each different type of approach, the core time spent on this predicate, is of the same order for any of the problems experimented. For the *Aggregation/View Approach* the times obtained were around 100 milliseconds, having a variance increase due to an enlargement of the logic clauses to be translated, as the data-sets grow in size. Notice also that in this approach only 2 queries are made to the database server per clause, one to count the positive examples that are covered, and another for the negative ones. On the *View-Level/Once Approach*, the number of queries sent to the server for each clause is augmented to 1 query per positive and negative example, which causes the time spent on the **transl** activity to increase.

Table 3 shows that the most significant part of the time is consumed in the **server** activity. Improving the efficiency of the database server in the execution of queries is thus fundamental to achieve good results. As ILP problems often use some kind of mode declarations to supply information concerning the arguments

of each predicate that may appear in the background knowledge, we use the mode information to automatically create indexes in the database in order to optimize query execution. Without indexing, the final execution time can increase more than 5000 times.

Table 3 also shows that for these approaches, only a small percentage of time is spent transferring the result sets from the database server to the Prolog system (**transf** activity). This is due to the small number of rows returned, and consequently, small amount of data transferred. This also shows that it is in fact the relational system that processes most of the work of the coverage computation. However, for the *View-Level/Once Approach*, the database returns more results than for the *Aggregation/View Approach*, which increases the time spent on unifying logic variables increases (**prolog** activity).

With respect to the **db_row** activity, we can see that for the *View-Level/Once Approach* and *Aggregation/View Approach* the time spent in this activity is not relevant. Results obtained for these approaches show that, on average, the time spent on the `db_row/2` predicate unifying the results of the queries with the logic variables, is around 2 and 400 milliseconds respectively for the *Aggregation/View Approach* and *View-Level/Once Approach*. Remember that the *Aggregation/View Approach* only returns two values for each query, while the *View-Level/Once Approach* produces far more results.

6 Conclusions and Future Work

In this work we have proposed to couple ILP systems with DDB systems. This strategy allows bringing to ILP systems the technology of relational database systems, which are very efficient in dealing with large amounts of data. Coupling ILP with DDB allows abstracting the Prolog to SQL translation from the ILP system. The ILP system just uses high-level Prolog predicates that implement relational operations that are more efficiently executed by the relational database system. We argue that this strategy is easier to implement and maintain than the approach that tries to incorporate database technology directly in the logic programming system. And, much more important, it allows a substantial increase of the size of the problems that can be solved using ILP since the data does not need to be loaded to memory by the ILP systems.

The performance results in execution speed for coverage computation are very significant and show a tendency to improve as the size of the problems grows. The size of the problems is exactly our most significant result, as the storage of data-sets in database relations allows an increase of more than 2 orders of magnitude in the size of the problems than can be approached by ILP systems.

In the future we plan to deal with recursive theories, through the YapTab tabling system [6], and to be able to send packs of clauses as a single query to the database system, using its grouping operators, to avoid redundant computation. We also plan to use the query packs technique [17], which is very similar to the tabling of prefixes technique [18], to perform a many-at-once optimization of SQL queries. Not only do some database systems perform caching of queries,

but it is also simple to implement similar techniques to query-packs on a DDB context.

A more ambitious future goal aims at a full integration of April and MYDDAS in a single programming environment where any program can be seen as a set of extensional data represented in a database, a set of intensional (and extensional) data represented by logic rules, and a set of undefined data that the ILP component of the system should be able to derive and compile to intensional data to be used by the program itself.

Acknowledgements

This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia (FCT) and Programa POSC. Tiago Soares is funded by FCT PhD grant SFRH/BD/23906/2005. Michel Ferreira was funded by FCT sabbatical grant SFRH/BSAB/518/2005, and thanks Manuel Hermenegildo and University of New Mexico for hosting his research.

References

1. Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., Bourne, P.E.: The protein data bank. Nucleic Acids Research (2000) 235–242
2. Wrobel, S.: Inductive Logic Programming for Knowledge Discovery in Databases. In: Relational Data Mining. Springer-Verlag (2001) 74–101
3. Raedt, L.D.: Attribute Value Learning versus Inductive Logic Programming: The Missing Links. In: Inductive Logic Programming. Volume 1446 of LNAI., Springer-Verlag (1998) 1–8
4. Muggleton, S., Raedt, L.D.: Inductive Logic Programming: Theory and Methods. Journal of Logic Programming **19/20** (1994) 629–679
5. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)
6. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
7. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
8. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)
9. Ferreira, M., Rocha, R., Silva, S.: Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In: Colloquium on Implementation of Constraint and LOgic Programming Systems. (2004) 71–82
10. Soares, T., Rocha, R., Ferreira, M.: Generic Cut Actions for External Prolog Predicates. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 16–30

11. Draxler, C.: Accessing Relational and Higher Databases Through Database Set Predicates. PhD thesis, Zurich University (1991)
12. Michalski, R.S., Larson, J.B.: Selection of Most Representative Training Examples and Incremental Generation of VL918 Hypotheses: The Underlying Mehtodology and the Description of Programs ESEL and AQ11. Technical Report 867, Department of Computer Science, University of Illinois at Urbana-Champaign (1978)
13. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A Midterm Report. In: European Conference on Machine Learning. Volume 667., Springer-Verlag (1993) 3–20
14. Muggleton, S., Firth, J.: Relational Rule Induction with CProgol4.4: A Tutorial Introduction. In: Relational Data Mining. Springer-Verlag (2001) 160–188
15. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. (2006) Available from `http://www.ncc.up.pt/~vsc/Yap`.
16. Botta, M., Giordana, A., Saitta, L., Sebag, M.: Relational Learning as Search in a Critical Region. Journal of Machine Learning Research **4** (2003) 431–463
17. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16** (2002) 135–166
18. Rocha, R., Fonseca, N., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning. Number 3720 in LNAI, Springer-Verlag (2005) 707–714