Efficient and Scalable Induction of Logic Programs using a Deductive Database System

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

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

Abstract. A consequence of ILP systems being implemented in Prolog or using Prolog libraries is that, usually, these systems use a Prolog internal database to store and manipulate data. However, in real-world problems, the original data is rarely in Prolog format. In fact, the data is often kept in Relational Database Management Systems (RDBMS) and then converted to a format acceptable by the ILP system. Therefore, a more interesting approach is to link the ILP system to the RDBMS and manipulate the data without converting it. This scheme has the advantage of being more scalable since the whole data does not need to be loaded into memory by the ILP system. In this paper we study several approaches of coupling ILP systems with RDBMS systems and evaluate their impact on performance. We propose to use a Deductive Database (DDB) system to transparently translate the hypotheses to relational algebra expressions. The empirical evaluation performed shows that the execution time of ILP algorithms can be effectively reduced using a DDB and that the size of the problems can be increased due to a non-memory storage of the data.

Keywords: Implementation, Performance, Deductive Databases.

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 biotech data that, as a result of automation in biochemistry, doubles its size every three to six months [1, 2]. Most of this data is structured and stored in relational databases and, in more complex applications, 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 store 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 [3] or to tables of prohibitive size [4].

On the other hand, Inductive Logic Programing (ILP) systems are able to learn patterns from relational data. However, ILP systems usually store and manipulate data in Prolog databases as a result of being implemented in Prolog [5–7]

or using Prolog libraries [8]. The approach often followed by ILP practitioners is to convert the data in the relational database to a format acceptable by the ILP system. A consequence of learning from Prolog databases is that the data is loaded into main memory, thus limiting ILP ability to process larger data-sets. Although ILP systems load the data into main memory, they are known as being computationally expensive. To find a model, ILP systems repeatedly examine sets of candidate clauses, which in turn involves evaluating each clause on all data to determine its *quality*. On complex or sizable applications, evaluating individual clauses may take considerable time, and thus, to compute a model, an ILP system can take several hours or even days. Efficiency and scalability are thus two of the major challenges that current ILP systems must overcome.

In this work we show how an ILP system can be transparently coupled with a Relational Database Management System (RDBMS) by using a Deductive Database (DDB) system, and how this coupled environment provides an excellent framework for the efficient and scalable induction of logic programs. In particular, we will use April [9] as the ILP system and MYDDAS [10] as the DDB system. By using a DDB system, the ILP system is able to process larger databases, since the memory issues disappear, and can transparently exploit advanced features of relational databases, such as powerful indexing schemes, query optimization, efficient aggregation and joining algorithms. In particular, we describe *mode based indexing*, an optimization that many ILP systems may easily perform.

The idea of coupling ILP with relational databases is not new [11–14], but very little has been reported about the impact on performance of learning from a relational database. In fact, there is a general idea that ILP systems become slower when coupled to a RDBMS. To clarify this, we investigate the effectiveness of several high-level strategies of coupling an ILP system with a DDB. We wish to evaluate the potential performance gains that result from learning from a relational database as opposed to the more traditional approach of learning from Prolog databases. In the experiments we used four artificially generated problems [15] that allowed us to perform the evaluations while considering different data-set sizes and hypotheses complexity (number of joins in a hypothesis).

The remainder of the paper is organized as follows. First, we revise the background concepts of relational algebra operations in Prolog and introduce the problem of coverage computation in ILP. Then, we describe our approaches to couple ILP with DDB and discuss some implementation details. Next, we present the results of an empirical evaluation on the performance of the proposed approaches. We end by discussing related work and by outlining some conclusions.

2 Preliminaries

In this section we revise relevant concepts of relational algebra and the encoding of its operations in Prolog syntax. We also introduce the problem of coverage computation in the context of ILP systems.

2.1 Prolog and Relational Algebra

If we abstract the notion of order on the clauses of a Prolog predicate and restrict these clauses to ground facts with atomic arguments, then this predicate is equivalent to a database relation. Database relations are queried by RDBMS using relational algebra. In [16], Codd defined five primitive operations of relational algebra: *selection*, *projection*, *cartesian product*, *set union* and *set difference*. We can define Prolog predicates which are equivalent to these relational algebra operations. Assuming that $\mathcal{Q}$ and $\mathcal{R}$ are database relations with an arbitrary number of attributes, and that q and r are their associated Prolog predicates, Table 1 defines a new relation $\mathcal{P}$ and a new Prolog predicate p, based on the five primitive relational algebra operations and their equivalent encoding in Prolog syntax.

Selection	$\mathcal{P} \leftarrow \sigma_{\$i=val}(\mathcal{Q})$ $p(X_1,...,X_{i-1},val,X_{i+1},...,X_n) :- q(X_1,...,X_{i-1},val,X_{i+1},...,X_n).$
Projection	$\mathcal{P} \leftarrow \pi_{\$i}(\mathcal{Q})$ $p(X_i) :- q(X_1,...,X_i,...,X_n).$
Cartesian Product	$\mathcal{P} \leftarrow \mathcal{Q} \times \mathcal{R}$ $p(X_1,...,X_n,Y_1,...,Y_m) :- q(X_1,...,X_n), r(Y_1,...,Y_m).$
Set Union	$\mathcal{P} \leftarrow \mathcal{Q} \cup \mathcal{R}$ $p(X_1,...,X_n) :- q(X_1,...,X_n).$ $p(X_1,...,X_n) :- r(X_1,...,X_n).$
Set Difference	$\mathcal{P} \leftarrow \mathcal{Q} - \mathcal{R}$ $p(X_1,...,X_n) :- q(X_1,...,X_n), not\ r(X_1,...,X_n).$

Table 1. Relational algebra operations in Prolog

An important difference between Prolog and relational algebra is that the Prolog's inference engine operates *tuple-at-a-time*, while the database manager operates *set-at-a-time*. To get the Prolog system to compute the equivalent of the relational algebra operations of Table 1, we need to use the $findall/3$ built-in: $findall(p(X_1,...,X_n), p(X_1,...,X_n), L)$, which will force backtracking to occur on goal $p(X_1,...,X_n)$, the second argument, collecting all solutions as $p(X_1,...,X_n)$ terms, the first argument, in list L.

Codd's relational algebra has been extended to include higher-order operations, such as aggregate functions that compute values over sets of attributes. Virtually every database system supports the following aggregate functions over relations: $sum()$, $avg()$, $count()$, $min()$ and $max()$, which compute the sum, the average, the number, the minimum and the maximum of given attributes. In relational algebra, aggregation operations are represented by ${}_{group}\mathcal{F}_{fun}(\mathcal{Q})$, where $\mathcal{F}$ is the aggregation operator, $group$ is an optional list of attributes of relation $\mathcal{Q}$ to be grouped and fun is the list of aggregation functions. For example, a relational algebra expression returning a relation with a single tuple representing the number of values for the ith attribute of a relation $\mathcal{Q}$ would be: $\mathcal{P} \leftarrow \mathcal{F}_{count\ \$i}(\mathcal{Q})$.

Because of its tuple-at-a-time nature, Prolog is particularly inefficient for higher-order computations. Coupled DDB systems thus try to transfer these computations to the database manager. In the context of DDB the logic syntax

to encode the aggregation operations of relational algebra is as follows:

$$p(X_i, ..., X_j, Y_1, ..., Y_m) :- Y_1 \; is \; X_i \; \hat{} ... \hat{} X_j \; \hat{} fun_1(X_k, q(X_1, ..., X_k, ..., X_n)),$$
$$..., $$
$$Y_m \; is \; X_i \; \hat{} ... \hat{} X_j \; \hat{} fun_m(X_l, q(X_1, ..., X_l, ..., X_n)).$$

where the $X_i, ..., X_j$ are the grouping attributes and the $Y_1, ..., Y_m$ are the aggregate values associated to the $fun_1, ..., fun_m$ aggregation functions. The above example of $\mathcal{P} \leftarrow \mathcal{F}_{count \; \$i}(\mathcal{Q})$ would be written in Prolog as:

$$p(Y_1) :- Y_1 \; is \; count(X_i, q(X_1, ..., X_i, ..., X_n)).$$

Common database queries typically combine several primitive relational algebra operations. For instance, a *natural join* such as $\mathcal{P} \leftarrow \mathcal{Q} \bowtie_{\$i=\$j} \mathcal{R}$ is implemented by a composition of cartesian product, selection and projection operations: $\mathcal{P} \leftarrow \pi_{X_1,...,X_n,Y_1,...,Y_{j-1},Y_{j+1},...,Y_m}(\sigma_{\$i=\$(n+j)}(\mathcal{Q} \times \mathcal{R}))$.

An equivalent composition results in the following Prolog clause to implement the same natural join:

$$p(X_1, ..., X_i, ..., X_n, Y_1, ..., Y_{j-1}, Y_{j+1}, ..., Y_m) :- q(X_1, ..., X_i, ..., X_n),$$
$$r(Y_1, ..., Y_{j-1}, X_i, Y_{j+1}, ..., Y_m).$$

Every composition of relational algebra can be expressed in Prolog, while the reverse is not true. The subset of Prolog, extended with the $findall/3$ predicate, equivalent to relational algebra is referred as Datalog [17]. Prolog predicates which involve either direct or indirect recursion cannot be expressed in relational algebra. Relational tuples also cannot represent Prolog facts containing unbound or compound arguments.

2.2 Coverage Computation in ILP

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 [18]. In general, the background knowledge and the set of examples can be arbitrary logic programs.

To derive a theory with the desired properties, many ILP systems follow some kind of *generate-and-test* approach to traverse the *hypotheses space* [8]. 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 each of these hypotheses 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 hypotheses evaluated, or the number of positive examples covered, or time. A simplified algorithm for the coverage computation of a clause is presented next in Fig. 1.

```
compute_coverage(Clause,ScorePos,ScoreNeg) :-
   assert(Clause,Ref),
   reset_counter(pos,0), reset_counter(neg,0),
   (
      select_positive_example(Goal), once(Goal),
      incr_counter(pos), fail
   ;
      true
   ),
   (
      select_negative_example(Goal), once(Goal),
      incr_counter(neg), fail
   ;
      true
   ),
   counter(pos,ScorePos), counter(neg,ScoreNeg),
   erase(Ref).
```

Fig. 1. Coverage computation

The *compute_coverage*/3 predicate starts by asserting the clause being evaluated[1] and by resetting a counter *pos*. Next, the *select_positive_example*/1 predicate binds variable *Goal* to the first positive example, which is then called using the *once*/1 primitive. The *once*/1 primitive is used to avoid backtracking on alternative ways to derive the current 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.

3 Coupling Approaches

In this section we describe several approaches to divide the coverage computation work between the logic system and the relational database system. We will describe the coupling approaches starting with the base coverage computation, and then incrementally transferring computational work from the logic system to the database system.

3.1 Selection Approach

On a typical coupled DDB system, the tuples defined extensionally in database relations are transparently mapped to Prolog predicates by using a directive such as:

$$:- db\_import(rel\_name, pred\_name, conn).$$

This directive is meant to associate a predicate *pred_name* with a database relation *rel_name* that is accessible through a connection with the database system named *conn*. What this directive does is implementing the communication layer between the Prolog engine and the database system, which involves the translation of queries written in Prolog syntax to their equivalent relational

[1] Here we consider the general case where the clauses being evaluated can be recursive.

algebra expressions, as explained in subsection 2.1. Typical interfaces with relational database systems do not include support for relational algebra expressions in their textual form, requiring their further translation to SQL, the *lingua franca* of database systems.

Based on the above directive and assuming that *rel_name* is a two field relation, the query goal $pred\_name(val, A)$ will be translated to the following relational algebra expression: $\sigma_{\$1=val}(rel\_name)$, which is in turn translated to the SQL expression:

$$SELECT\ val,\ A.attr2\ FROM\ rel\_name\ A\ WHERE\ A.attr1 = val;$$

where $attr1$ and $atrr2$ are the attributes names of relation *rel_name*. This expression is then sent to the database system and the obtained result set is navigated *tuple-at-a-time* using backtracking. Note that the database system executes the *selection operation*, returning only the tuples that unify with the logic goal, thus freeing the logic system from the unification operation. This selection approach requires just the declaration of the background knowledge and the positive and negative examples predicates through *db_import*/3 directives. Coverage computation is done exactly as in Fig. 1.

3.2 Join Approach

A fundamental improvement to the selection approach is to transfer the computation of the join of the several database goals in the body of a clause to the database system. Prolog efficiency is compromised by the strict execution mechanism of SLD-resolution, while the query optimiser of database systems is able to use goal-reordering and extended indexing schemes to improve the efficiency of join computation.

In order to transfer the join computation to the database system, the interface of the DDB system must group together conjunctions of extensional goals and Prolog built-ins that can be expressed in relational algebra. This can be done automatically during compilation using a simple program analysis, or can be done explicitly by the user. Currently, MYDDAS follows the later approach, through a *db_view*/3 directive:

$$:- db\_view(view(A_i, ..., A_j), (db\_goal_1(A_1, ..., A_n), ..., db\_goal_m(A_k, ..., A_l)), conn).$$

where the first argument specifies the attributes to be fetched from the database, the second argument specifies the selection restrictions and join conditions, and the third argument identifies the connection with the database system.

The *compute_coverage*/3 predicate still works as before, but instead of asserting the given clause, it now creates a view for the goals in the body of the clause and then replaces the clause's body with the created view. For example, considering the clause '$h(A) :- p1(A, B), p2(B)$.', where $p1/2$ and $p2/1$ represent the database relations $r1$ and $r2$, the *compute_coverage*/3 predicate now creates the view:

$$db\_view(view(A), (p1(A, B), p2(B)), conn)$$

and asserts the clause '$h(A) :- view(A).$'. The relational algebra expression generated for the view when evaluating a given example, $e1$ for instance, is:

$$\pi_{\$1}((\sigma_{\$1=e1}(r1)) \bowtie_{\$2=\$1} r2)$$

3.3 Reduced-Join Approach

Some very important issues in the coverage algorithm of Fig. 1 arise for the *once*/1 primitive: (i) the coupling interface must support deallocation of queries result sets when the *once*/1 primitive prunes the search space [19]; (ii) instead of unnecessarily computing all the alternative solutions, 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 primitive to the database view. The asserted clause should include an *once*/1 predicate on the view definition and the DDB interface should be able to translate it to a relational algebra expression that can be efficiently executed by the database system. We introduce an extension to the relational algebra selection operation, $\sigma_{(conditions,rows)}(R)$, where the *rows* argument defines a limit to the number of tuples that the selection operation should return. In particular, if this selection operation is composed with a join operation, the query optimizer can prune the join computation as soon as the required number of tuples is reached. With this approach, the *compute_coverage*/3 predicate can be used as before and we can drop the *once*/1 call from its code. For our previous example, the view is now:

$$db\_view(view(A), once(p1(A, B), p2(B)), conn)$$

and the relational algebra operation generated when evaluating example $e1$ is:

$$\pi_{\$1}((\sigma_{(\$1=e1,1)}(r1)) \bowtie_{\$2=\$1} r2)$$

Based on this relational algebra expression, the MYDDAS interface is able to send the following SQL query to the database system:

$$SELECT\ A.attr1\ FROM\ r1\ A,\ r2\ B$$
$$WHERE\ A.attr1 = e1\ AND\ A.attr2 = B.attr1\ LIMIT\ 1;$$

3.4 Aggregation 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 clause. The *compute_coverage*/3 predicate uses extra-logical global variables to perform this counting operation, as it would be too inefficient otherwise.

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 clause to the program code, include the positive or negative examples relation as a goal co-joined with the goals in the body of the current clause, and include a *count*/2 predicate on the view definition for the attributes holding the positive or negative examples. Again, the join should only test for the existence of one tuple in the body goals for each of the examples, using the *once*/1 primitive on the view definition. For our example, the view would be:

$$db\_view(view(C), C\ is\ count(A, (h(A), once(p1(A, B), p2(B)))), conn)$$

The composition of these relational operations results in the following relational algebra expression:

$$\mathcal{F}_{count\ \$1}(r0 \bowtie_{\$1=\$1} (\sigma_{(\epsilon,1)}(r1) \bowtie_{\$2=\$1} r2))$$

where $r0$ is the database relation associated with $h/1$ and ϵ represents the empty condition. We have extended the MYDDAS interface in order to generate an efficient translation to SQL for such expressions. The above view generates the following SQL expression:

$$SELECT\ COUNT(A.attr1)\ FROM\ r0\ A$$
$$WHERE\ EXISTS\ (SELECT\ *\ FROM\ r1\ B,\ r2\ C$$
$$WHERE\ A.attr1 = B.attr1\ AND\ B.attr2 = C.attr1\ LIMIT\ 1);$$

Although the '$LIMIT$ 1' keyword 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. On the other hand, the '$LIMIT$ 1' suffix has no impact on performance when using an Oracle RDBMS, as we shall see. This observation shows that MySQL query optimizer is failing somewhere on its task.

The four coupling approaches, *Selection*, *Join*, *Reduced-Join* and *Aggregation*, are gradually transferring computation from the logic system to the database system. In a future approach we plan to further transfer computation work to the database system, implementing a many-at-once optimization, as illustrated by the query packs technique [20]. Not only do some database systems perform caching of queries, but we can also extend the Prolog-to-relational-algebra translation in order to be able to send *packs* of logic queries to the database system and have their coverage computed by the database system, at once, using relational grouping operators optimized for redundancy elimination.

4 Implementation

The coupling approaches described above were implemented in the April ILP system [9] coupled with the DDB system MYDDAS. Both April and MYDDAS systems run on top of the Yap Prolog engine.

Being able to abstract the Prolog to SQL translation, task performed by MYDDAS, we concentrated in implementing the various coupling approaches,

with different distributions of work between the logic system and the database system, and considered some optimizations such as mode-based indexing, presented in the next subsection. The integration of both systems required minor changes to April's code and, in particular, to its clause evaluation component.

The impact for the ILP practitioner of using a DDB as opposed to using the Prolog database is kept to a minimum. The user first indicates, through a configuration option, which coupling approach wants to use, and then only needs to provide information regarding the database where the data resides (name, user, password, and host) and, if using the aggregation approach, the names of the tables of the positive and, if available, negative examples. When the examples are stored in tables, the ILP system automatically creates new tables for each class of examples with extra attributes that are used to keep temporary information generated during execution.

4.1 Mode-Based Indexing

ILP systems often use some kind of input/output mode declarations to supply information concerning the arguments of each predicate that may appear in the hypotheses [21, 22]. These declarations specify if an argument of a predicate is intended to be a constant, an input or an output argument. Although the mode declarations are usually provided by the user, they can be also automatically extracted from the background knowledge [23].

There are two major advantages in the use of mode declarations. First, the ILP system can guarantee termination by ensuring that the hypotheses it generates are accordingly to the mode. Second, ILP systems can use the mode information to automatically create indexes in the database in order to optimize query execution. We proceed as follows. For each mode declaration (that affects some table) we create two indexes. The first index is created on the attributes indicated as constants or input arguments. The rationale is that all hypotheses (queries) generated will be mode conform and, thus, the join and projection operations in the queries will always be performed over the constant and/or the input attributes. The second index is created on all the attributes of the corresponding table (predicate). In the following section, we show that this automatic index creation, when used with the aggregation approach, reduces the execution speed significantly.

5 Performance Evaluation

We have performed a set of experiments in order to evaluate our work. The goals of the experiments were two-fold:

- Empirically compare the four coupling approaches.
- Assess if our proposal of coupling ILP with a DDB can improve the efficiency and scalability of ILP systems.

5.1 Materials and Methodology

We have used four artificially generated problems [15]. Table 2 characterizes the problems in terms of number of examples, number of relations in the background knowledge, and number of tuples. All experiments were performed using 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 indexing in run-time on all arguments.

Problem	# Examples	# Relations	# Tuples
$p.m8.l27$	200	8	321,576
$p.m11.l15$	200	11	440,000
$p.m15.l29$	200	15	603,000
$p.m21.l18$	200	21	844,200

Table 2. Problems characterization

A set of 688 clauses was generated for each artificial problem. The clauses were randomly generated and equally distributed by length, ranging from 1 to the number of relations available in the data-set. We used the April ILP system to randomly generate the sets of clauses.

Since the weight of coverage computation on the total execution time of an ILP system varies accordingly to the system or algorithm used, we have chosen to implement the approaches for coverage computation through simple Prolog programs[2]. This allows us to perform a comparison independent of the ILP system and to correctly measure the time spent in coverage computation. Using April's execution time as the measure does not allow us to do a precise performance evaluation since the gains would vary, depending on the search algorithm used and on the optimizations that April can perform during run-time.

5.2 Comparing the Coupling Approaches

Table 3 shows the best execution time of 5 runs, in seconds, for each problem. For the basic ILP approach, the clauses were evaluated using Yap with indexing on the first argument (IFA) and using Yap with indexing on all arguments (IAA). For the coupling approaches, the clauses were evaluated using mode-based indexing (MBI) as described in subsection 4.1. For comparison purposes, we also show the execution time for the *Aggregation* approach without mode-based indexing. Without mode-based indexing the relational tables still include indexing, but only based on the primary indexes associated to the primary keys.

A first observation should be made regarding the impact in the execution time when using full indexing in Prolog (*Basic ILP + IAA*) as opposed to use indexing solely in the first argument (*Basic ILP + IFA*). We will use the times taken by the *Basic ILP + IAA* as the base times, although full indexing is available in only a few Prolog engines.

[2] Available from `http://www.ncc.up.pt/MYDDAS/ilpddb.html`

Approach	Problem			
	p.m8.l27	**p.m11.l15**	**p.m15.l29**	**p.m21.l18**
Basic ILP + IFA	149	409	>1 day	>1 day
Basic ILP + IAA	15	50	33,972	>1 day
Selection + MBI	35,583	>1 day	>1 day	>1 day
Join + MBI	n.a	n.a	n.a	n.a
Reduced-Join + MBI	99	628	2,975	33,229
Aggregation	>1 day	>1 day	>1 day	>1 day
Aggregation + MBI	5	14	251	734

Table 3. Performance for the different approaches (execution time in seconds)

The core time of communication between the ILP system and the database system dilutes as we increase the computation work of the database system. For problems involving relations with thousands of tuples the *Selection + MBI* approach is unrealistic. This approach does not transfer any computation work to the database system, other than selecting tuples from individual relations. Furthermore, 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 *Join + MBI* approach, as expected, we could not obtain the execution times for any problem, 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 ran out of memory when trying to compute a join of several relations, each with thousands of tuples.

For the *Reduced-Join + MBI* 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 + IAA* approach is explained by the number of queries that are sent to the database system, one for every positive and negative example. This means that a total of 200 queries (the number of positive and negative examples) are sent to the database system for each of the 688 clauses. As the size of the joins grows larger, as with $p.m15.l29$, the time spent in communication of the queries becomes irrelevant compared to the time taken for computing the joins. This and the huge amount of backtracking performed by the *Basic ILP + IAA* approach for the two largest artificial problems, as the *Basic ILP + IAA* approach runs on a *tuple-at-a-time* form against the *set-at-a-time* database approaches, explains the speedup obtained with this approach.

On the *Aggregation* approach only two queries per clause are sent to the database system, one to compute positive coverage and one to compute negative coverage. Since all the coverage computation work is transferred to the database system, the core time of sending and storing the result set for the two queries is insignificant. However, the results are disappointing due to the lack of useful indexes on the tables.

The results obtained with the *Aggregation + MBI* approach are very good. The performance gains over the *Basic ILP + IAA* approach are clear: a 2.8 speedup for $p.m8.l27$, and a 3.4 speedup for $p.m11.l15$, and a 135 speedup for

$p.m15.l29$. These results show a clear tendency for higher speedups as the size of the problems grow.

In conclusion, the *Aggregation + MBI* approach clearly outperforms the other approaches. It significantly reduces the execution time and may allow ILP systems to handle larger problems, thanks to the non-memory storage of data-sets, thus contributing to improving the scalability of ILP systems.

5.3 Impact of Query Transformations

The results presented in the previous section showed that *Aggregation + MBI* approach has the best results, even when compared to the times obtained to the *Basic ILP + IAA* approach. Although Prolog's indexing on all arguments is an improvement to indexing on the first argument (often provided by Prolog engines), that is not the only technique that may be used to improve query execution in Prolog.

Several techniques have been proposed to this effect, namely perform transformations in the query so that it can be executed more efficiently [24, 20], compute an approximate evaluation [25, 26] as opposed to an exact evaluation, store and reused the computations [27, 28], or by exploiting parallelism [29]. Determining which technique or combinations of techniques produces the best results is out of scope of this paper. Instead, we selected one ILP technique similar to the query optimization performed by RDBMS and that has been shown to yield good results - the query transformations (QT) proposed in [24]. The results obtained with query transformations are presented in Table 4 and compared with two other approaches and with the times taken in an Oracle RDBMS.

Approach	**Problem**			
	p.m8.l27	**p.m11.l15**	**p.m15.l29**	**p.m21.l18**
Basic ILP + IAA	15	50	33,972	>1 day
Basic ILP + IAA + QT	1	4	16	39
Aggregation + MBI (MySQL)	5	14	251	734
Aggregation + MBI (Oracle)	6	9	101	164

Table 4. Comparing with query transformations (execution time in seconds)

It is obvious that the results presented are dependent of the RDBMS used. In our experiments we used MySQL which is known to be a fast database. However, our experiments suggest that the optimizer is not efficient. The results obtained for the *Aggregation + MBI* approach in the Oracle RDBMS show that it outperforms the MySQL RDBMS by almost a factor of 5 for the largest problem. This speedup also shows a clear trend to increase as the size of the data-sets grows. The Oracle optimizer also proves to be more intelligent when computing the existential sub-query on the *Aggregation* approach. When using the Oracle system, the '$LIMIT$ 1' keyword is actually redundant in terms of performance.

The impact of query transformations is impressive when compared to the basic ILP approach. Compared to our *Aggregation + MBI* approach in Oracle it is still 4 times faster for $p.m21.l18$. However, it requires the full data-set to

be loaded to memory, which might not be possible for larger problems. Another interesting possibility we are exploring is the translation to relational algebra of the Prolog goals *after* applying the queries transformations optimization. Our preliminary results showed that this can in fact improve the performance on the database side. At this time, the MYDDAS interface translating Prolog to relational algebra expressions does not support the usage of the cut (*!*) predicate. The query transformations uses this extra-logical predicate to optimize queries. In Prolog, the cut operator is used to prune the search tree, but on the relational algebra the semantics of the `!/0` predicate gives the notion of an existential sub-query. Preliminaries results of translating the cut to existential sub-queries for the goals generated by the queries transformation optimization indicate that it may have a positive impact on performance.

6 Related Work

Several previous implementations have already coupled ILP systems with relational databases, some mapping logical predicates into database relations, others translating logical clauses into SQL statements [11–14], and others using both [30]. The level of transparency (for the user) in these implementations is quite variable, ranging from no transparency (the user manually defines the views for each literal that may appear in a clause) [30] to completely transparent [14].

The idea of coupling ILP with DDB is also not new - the ILP system Warmr has been coupled with a DDB system to mine association rules [31]. The difference to our work is twofold. First, Warmr loads the data into main memory from a relational database, while in our proposal the data remains in the database. Secondly, we have performed an empirical performance study of several coupling approaches and proposed to exploit mode based indexing.

7 Concluding Remarks

In this work we have studied several approaches to couple ILP systems with RDBMS by using a DDB system. The strategy of using a DDB system brings to ILP systems the technology of relational database systems, which are very efficient in dealing with large amounts of data. 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 system.

The results of evaluating the several approaches to couple ILP with RDBMS show that the *Aggregation + MBI* approach: i) outperforms the other coupling approaches and significantly reduces the execution time when compared to the use of a Prolog engine (even with indexing on all arguments) and ii) may allow ILP systems to handle larger problems, thanks to the non-memory storage of data-sets, thus contributing to improving the scalability of ILP systems.

The results also indicate that further research should be done in order to make learning from RDBMS competitive, in terms of execution time, with a fast Prolog engine (using indexing on all predicate's arguments and performing query transformations). For instance, a possible line of research could be the adaptation of techniques already developed in the ILP context (see e.g., [20, 24]) to be used while learning from RDBMS. As further work we also plan to be able to implement the transformations from Prolog to SQL described in this paper as a compilation step, based on program analysis, which takes into account factors such as the size of data, database indexing information and complexity of queries. This information should guide an automatic translation of parts of a Prolog program to database accesses, using SQL as a compiler target language and a database system as an abstract machine.

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 and Programa POSC. Tiago Soares is funded by FCT PhD grant SFRH/BD/23906/2005.

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. Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J., Wheeler, D.: GenBank. Nucleic Acids Research **33** (2005) 235–242
3. Wrobel, S.: Inductive Logic Programming for Knowledge Discovery in Databases. In: Relational Data Mining. Springer-Verlag (2001) 74–101
4. Raedt, L.D.: Attribute Value Learning versus Inductive Logic Programming: The Missing Links. In: Inductive Logic Programming. Number 1446 in LNAI, Springer-Verlag (1998) 1–8
5. Raedt, L.D., Laer, W.V.: Inductive Constraint Logic. In: International Conference on Algorithmic Learning Theory, Springer-Verlag (1995) 80–94
6. Raedt, L.D., Dehaspe, L.: Clausal Discovery. Machine Learning **26** (1997) 99–146
7. Srinivasan, A.: The Aleph Manual. (2003) Available from `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph`.
8. Muggleton, S., Firth, J.: Relational Rule Induction with CProgol4.4: A Tutorial Introduction. In: Relational Data Mining. Springer-Verlag (2001) 160–188
9. Fonseca, N.A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System. In: European Conference on Logics in Artificial Intelligence. Number 4160 in LNAI, Springer-Verlag (2006) 481–484
10. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)
11. Shen, W.M., Leng, B.: Metapattern Generation for Integrated Data Mining. In: Knowledge Discovery and Data Mining. (1996) 152–157
12. Brockhausen, P., Morik, K.: Direct Access of an ILP Algorithm to a Database Management System. In: MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing. (1996) 95–100

13. Morik, K.: Knowledge Discovery in Databases - an Inductive Logic Programming Approach. In: Foundations of Computer Science: Potential - Theory - Cognition, Springer-Verlag (1997) 429–436
14. Bockhorst, J., Ong, I.M.: FOIL-D: Efficiently Scaling FOIL for Multi-Relational Data Mining of Large Datasets. In: International Conference on Inductive Logic Programming. LNAI, Springer-Verlag (2004) 63–79
15. 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
16. Codd, E.F.: A relational model for large shared data banks. Communications of the ACM **13** (1970) 377–387
17. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1989)
18. Muggleton, S., Raedt, L.D.: Inductive Logic Programming: Theory and Methods. Journal of Logic Programming **19/20** (1994) 629–679
19. 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
20. 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 Machine Learning Research **16** (2002) 135–166
21. Muggleton, S.: Inverse Entailment and Progol. New Generation Computing, Special Issue on Inductive Logic Programming **13** (1995) 245–286
22. Blockeel, H., Raedt, L.D.: Top-Down Induction of First-Order Logical Decision Trees. Artificial Intelligence **101** (1998) 285–297
23. McCreath, E., Sharma, A.: Extraction of meta-knowledge to restrict the hypothesis space for ILP systems. In: Australian Joint Conference on Artificial Intelligence, World Scientific (1995) 75–82
24. Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., Laer, W.V.: Query Transformations for Improving the Efficiency of ILP Systems. Journal of Machine Learning Research **4** (2002) 465–491
25. Srinivasan, A.: A study of two sampling methods for analysing large datasets with ILP. Data Mining and Knowledge Discovery **3** (1999) 95–123
26. DiMaio, F., Shavlik, J.W.: Learning an Approximation to Inductive Logic Programming Clause Evaluation. In: International Conference on Inductive Logic Programming. LNAI, Springer-Verlag (2004) 80–97
27. Berardi, M., Varlaro, A., Malerba, D.: On the Effect of Caching in Recursive Theory Learning. In: International Conference on Inductive Logic Programming. LNAI, Springer-Verlag (2004) 44–62
28. Rocha, R., Fonseca, N.A., 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
29. Fonseca, N.A., Silva, F., Camacho, R.: Strategies to Parallelize ILP Systems. In: International Conference on Inductive Logic Programming. Number 3625 in LNAI, Springer-Verlag (2005) 136–153
30. Weber, I.: Discovery of First-Order Regularities in a Relational Database Using Offline Candidate Determination. In: International Workshop on Inductive Logic Programming, Springer-Verlag (1997) 288–295
31. Dehaspe, L., Toironen, H.: Discovery of Relational Association Rules. In: Relational Data Mining. Springer-Verlag (2000) 189–208