

# Executing Query Flocks for ILP

*Bart Demoen  
Gerda Janssens  
Henk Vandecasteele*

*Report CW280, September 1999*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Executing Query Flocks for ILP

*Bart Demoen*  
*Gerda Janssens*  
*Henk Vandecasteele*

*Report CW280, September 1999*

Department of Computer Science, K.U.Leuven

## **Abstract**

The aim of data mining is to derive rules that describe characteristics of sets of data. Inductive Logic Programming does this by repeatedly constructing sets of related queries and by determining the success or failure of all these queries for a given data set in order to select the best fitting query. In the ILP systems of the Leuven ML team, each query is actually a conjunction of calls to Prolog predicates appearing in the data set or appearing in the background knowledge. The number of related queries can be huge and queries with common prefixes occur frequently. This paper investigates the *flock* construct that makes this overlap between queries explicit and offers an opportunity to reduce the execution time for the set of queries. Three implementations of the flock construct are discussed: a meta-interpreter, a source-to-source transformation and an implementation in a WAM-setting. Preliminary experiments with the latter are very promising, e.g., in the case of a benchmark from an existing application the flock construct realises a speedup by an order of magnitude. We also discuss briefly ways to exploit the fact that a prefix succeeds.

**Keywords :** Inductive Logic Programming, Machine Learning, WAM.

**CR Subject Classification :** I.2.6, D.1.6

# Executing Query Flocks for ILP

Bart Demoen Gerda Janssens Henk Vandecasteele

Department of Computer Science  
Katholieke Universiteit Leuven  
B-3001 Heverlee, Belgium  
{bmd,gerda,henkv}@cs.kuleuven.ac.be

## Abstract

The aim of data mining is to derive rules that describe characteristics of sets of data. Inductive Logic Programming does this by repeatedly constructing sets of related queries and by determining the success or failure of all these queries for a given data set in order to select the best fitting query. In the ILP systems of the Leuven ML team, each query is actually a conjunction of calls to Prolog predicates appearing in the data set or appearing in the background knowledge. The number of related queries can be huge and queries with common prefixes occur frequently. This paper investigates the *flock* construct that makes this overlap between queries explicit and offers an opportunity to reduce the execution time for the set of queries. Three implementations of the flock construct are discussed: a meta-interpreter, a source-to-source transformation and an implementation in a WAM-setting. Preliminary experiments with the latter are very promising, e.g., in the case of a benchmark from an existing application the flock construct realises a speedup by an order of magnitude. We also discuss briefly ways to exploit the fact that a prefix succeeds.

## 1 Introduction

The Leuven ML team works since many years on the efficient implementation of learning systems in Prolog [2, 3]. Such systems strain Prolog because huge amounts of data is involved, and rapid context switch between examples is crucial. Lots of improvements can be made in that area. Another efficiency bottle neck is the fact that often similar queries must be evaluated over the same data set. In its most simple form, the similarity consists in the fact that a given query  $Q$  which succeeded for a data set  $D$ , is extended to a potentially large set of queries  $\{Q, E_i | i = 1 \dots\}$  which must be evaluated over the same data set  $D$ . The  $Q$  part is named the common prefix of the set  $\{Q, E_i\}$ . One can exploit the following knowledge about such a set of queries:

- the fact that they have a common prefix
- the fact that the prefix succeeds on  $D$
- the existence and particular form of dependencies between the prefix and an extension
- the fact that in the inductive learning setting, one is interested in the success/failure of a query but not in the bindings, neither in the number of times it succeeds

Each of these can be used for optimising a certain aspect of the execution of one of the  $Q, E_i$  or of the whole set. This will be shown in the next sections.

As soon as a mechanism is found to exploit the common prefix in a set of extensions, this will lead immediately to the consideration - in the IL context - of large and possibly deeply nested sets of extended queries. Such a mechanism can be easily defined conceptually, and implemented by meta interpretation (see section 3.3), but an implementation by a meta interpreter is too slow for the purpose of IL. It is therefore worthwhile to investigate whether an efficient implementation can be achieved in the context of a WAM implementation. This will be shown in section 3.5 by the design of an extension to the instruction set of WAM (and a corresponding extension of the compiler): the implementation is relatively easy and can be performed by modifying any existing Prolog implementation. In particular we have implemented the design in ToC [6]<sup>1</sup>, XSB Prolog and also a new Prolog-like system called *ilProlog* which is the dedicated engine in an inductive learning environment under construction. We present measurements that show the usefulness of the approach in the context of ToC and *ilProlog* (section 3.6).

In the sequel we will rely on understanding of Prolog code and also on some understanding of the WAM.

The data set  $D$  over which a query must be evaluated, is never explicitly mentioned: in fact as far as this paper is concerned, it is always the same data set.

## 2 Optimisations based on the success of the prefix

To focus the ideas, consider a query  $Q$  that is refined with the extension  $E$ . Evaluation now requires the execution of the query:

?- once((Q,E)), write(success).<sup>2</sup>

Since it is known that the goal once(Q) is true, one can exploit this in at least two ways: by splitting off an independent part of  $Q$  and by remembering the state of the success of  $Q$ . These are described in the next two sections.

### 2.1 Removing an independent part of the prefix.

Suppose the query  $Q$  is extended with  $E$ . It is known that  $Q$  succeeds. The new query  $(Q,E)$  succeeds if and only if the query  $(Q',E)$  succeeds where  $Q'$  is the component of  $Q$  that is *connected* to  $E$ . The *connected* relation between goals in a query, is the transitive closure of the relation *directly connected*. Two goals  $G_1, G_2$  are directly connected if they share a source variable. Examples clarify how this notion is used:

```
Q = f(X,Y), g(Y,Z)
E = t(Z)
then
(Q',E) = f(X,Y), g(Y,Z), t(Z)
```

```
Q = f(X,A), g(Y,Z)
E = t(Z)
then
(Q',E) = g(Y,Z), t(Z)
```

Finding  $Q'$  from  $Q$  and  $E$  is implemented in the Tilde system ([2]) by a predicate named “smartcall”. Although it might seem silly that a query is extended with a goal that is not connected to the whole query, it proves actually very useful and improves speed. Further details are beyond the scope of this paper.

If language bias<sup>3</sup> prevents extensions that are not connected to a query, then smartcall gains nothing.

<sup>1</sup>ToC is WAM based Prolog system that is developed at the K.U.Leuven and compiles to C.

<sup>2</sup>once/1 can be thought of as defined by once(G) :- call(G), !.

<sup>3</sup>Language bias determines which kind of extensions are allowed and explored.

## 2.2 Memo-ing the success state of the prefix

From the implementation point of view, the success state of  $Q$  is the collection of WAM-stacks (heap, local stack, choice point stack and trail) at the moment of the success, i.e. the point in execution just before the cut in the definition of  $\text{once}/1$ , for the query  $\text{once}(Q)$ . How does this help the execution of the extended query? Imagine that  $Q$  consists of just the goal  $f(l, X)$  and the extension  $E$  of just the goal  $g(X)$  and let the data set be the set of facts  $f(a,1)$ .  $f(b,2)$ .  $f(c,3)$ . ...  $f(l, 12)$ . ...  $f(z,26)$ . Also let  $g(X)$  succeed only if  $X > 13$ . For the sake of the example, also assume that the Prolog system that executes these goals has no first-argument indexing. Now, executing the query  $f(l, X), g(X)$  without using the prior success of  $f(l, X)$ , will start scanning the sequence of facts for  $f/2$ , until it finds the first succeeding one, binding  $X$  to 12. If the state of the success were remembered and installed before executing the extended query, search in the  $f/2$  facts would resume from the point  $f(l,12)$ . It is clear that this can lead to arbitrary savings.

Saving the stacks at some point in execution, is quite easy.

There are however a few drawbacks to this method: usually the stacks will be small because queries often contain only calls to ground facts. But in the presence of background knowledge, the state may be arbitrarily large. One can try to memo enough of the state to *re-execute* without backtracking the prefix; [4] describes in detail how this can be done. The drawback of that method is that the data set (the definition of  $f/2$  in the example above) as well as the background knowledge must be transformed and that every query evaluation is penalized. Still, this seems feasible.

However, the main drawback of memo-ing in some form the success of the prefix, is that one must remember the success of the same prefix *for many different data sets*. This means that memo-ing success of prefixes, must be accompanied by a good heuristic that decides when to forget memo-ed success in favour of a re-computation. This definitely complicates the implementation a lot.

## 2.3 Remembering all the successes of the prefix

It is clear that one could also try to memo *all* the success bindings of the prefix: this is again a situation in which there is trade-off between the necessity to compute all answers, the cost of storing them (for a possibly large number of data sets) versus re-computation: for simplicity of implementation re-computation is preferred, but only empirical data can show the usefulness of memo-ing techniques in IL.

## 3 Evaluating a set of queries with a common prefix

While the previous two methods used the fact that the prefix succeeded before, the method in this section makes only use of the fact that the prefix is common to a set of extended queries. For the sake of simplicity, assume we have the query  $Q = f(X)$  and the extensions  $E_1 = g(X)$  and  $E_2 = h(X)$ . We have thus two extended queries:  $f(X), g(X)$  and  $f(X), h(X)$ . We can execute these in isolation, but that would mean that in the worst case, the goal  $f(X)$  is run twice to completion. This can be avoided by combining the two queries using the Prolog disjunction  $;/2$  as follows:  $f(X), (g(X); h(X))$  This will make sure that the common prefix is executed at most once. However, in the current form, we cannot exploit easily the fact that we are interested in only one success of each of the original queries and in the worst case,  $g(X)$  will be tried with values that  $f(X)$  produces, even after the first disjunctive branch has succeeded for the first time, because  $h(X)$  hasn't succeeded yet. One cannot simply put a cut after the goal  $g(X)$ , because that would stop  $f(X)$  from producing values for  $g/1$ . Neither is some form of soft cut that Prolog systems all provide internally, sufficient.

Instead of transforming to an ordinary Prolog disjunction, we will transform to the new construct  $f(X)$ ,

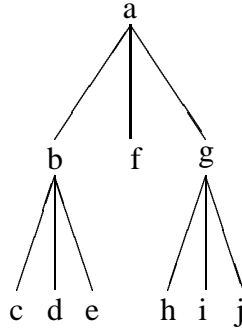


Figure 1: A graphical representation of the example flock (tree)

$(g(X) \text{ or } h(X))$  and we will define the semantics of the new *or* operator later.<sup>4</sup> In principle, there can be more than two or-branches and moreover, they can be nested. A compound query like

$a, (b, (c \text{ or } d \text{ or } e) \text{ or } f \text{ or } g, (h \text{ or } i \text{ or } j))$

results from considering in one go the set of queries:

$a, b, c$   
 $a, b, d$   
 $a, b, e$   
 $a, f$   
 $a, g, h$   
 $a, g, i$   
 $a, g, j$

The form  $a, (b, (c \text{ or } d \text{ or } e) \text{ or } f \text{ or } g, (h \text{ or } i \text{ or } j))$ , is called a *flock of queries*. Note that it does not contain a real Prolog disjunction and that or-s can be nested but do never occur in conjunction with each other: or-branches never join. Each branch corresponds to one extension. The atoms in the example could in general be arbitrary conjunctions of non-ground terms. A graphical representation of the example at hand is a tree shown in Figure 1.

### 3.1 The “semantics” in words

The procedural meaning of the flock  $a, (b, (c \text{ or } d \text{ or } e) \text{ or } f \text{ or } g, (h \text{ or } i \text{ or } j))$  is that as soon as a or-branch has succeeded, it is removed from the query, and that this change survives backtracking. Otherwise it behaves like a disjunctive query. The correctness of this behaviour w.r.t. the expectations the IL system has of a set of queries, follows from the fact that one is only interested in one success and not in bindings.

We have implemented this construct in several ways: by a meta-interpreter, a program transformation (both using assert/retract) and a low level addition to WAM. It is important that optimal complexity is achieved as in practice the breadth and depth of the flocks can become large. This means that true destructive update is needed. Still, the meta-interpreter does not obey this requirement.

Before we present the different implementations, we point out the worst and best cases for the flock construct in the next section.

---

<sup>4</sup>the name *or* was chosen because of its similarity with disjunction

### 3.2 Worst and best cases for flock queries

In the worst case a query flock performs within a constant factor from the original set of queries if activated separately. The following example shows when this occurs:

```
% the data set:
a(1).          b(1).          c(1).

% the flock query
?- a(X), (b(X) or c(X)).

% the set of queries
?- a(X), b(X).
?- a(X), c(X).
```

In the above example, the flock execution suffers from the set up of the mechanism, but since this cost is linear in the number of or-branches, it is just a constant factor (which is actually low). The other cost that the flock execution incurs is at every success of an or-branch: again, there is a related cost whenever a non-flock query succeeds.

We first compare the or-construct with the ordinary disjunction: arbitrary performance gain can be obtained, as the following example shows.

```
% the data set
a(1).          b(1).
a(2).          b(2) :- arbitrary_computation.

c(2).

% the flock query
?- a(X), (b(X) or c(X)).

% the disjunction
?- a(X), (b(X) ; c(X)).
```

The goal *arbitrary\_computation* will not be executed when the flock query is evaluated, while in the query with the ordinary disjunction, it is.

The example in section 2.2 already showed the possible gain of the query flock over the set of individual queries: however, this gain is limited by a factor equal to the number of or-branches.

### 3.3 A meta-interpreter

The runtime behaviour for a flock of queries can be described with the a meta-interpreter. Before using the flock of queries it is first annotated with some information. This information is composed from three types of numbers:

- **Node numbers** All nodes in the tree are numbered, depth first, from left to right. Leafs not included.
- **Leaf numbers** Each leaf is numbered, from left to right. If the original queries would be numbered sequentially, then the leaf numbers correspond with these.
- **Branch number** For each node with N branches, all branches from the node are numbered from 1 up to N sequentially.

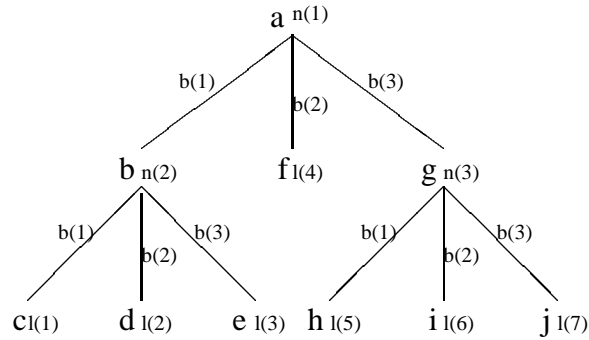


Figure 2: Node numbers  $n(i)$ , leaf numbers  $l(i)$  and branch numbers  $b(i)$  in our example

These numbers for Example 1 is shown in Example 2.

Then each node (not the leafs) is annotated with: the node number of its father, its own node number, its branch number with respect to its father node and the number of branches starting in the node. The number of the father node of the top node is assumed to be zero. At the end of each query (the leafs) a term `leaf/3` is added. Its arguments are: the node number of its father, its branch number corresponding to this father node and its own leaf number.

During the execution of the meta-interpreter, `solved/2` facts are asserted. Each `solved(Node, Branch)` fact denotes that a **branch** from a **node** has succeeded. Such facts are asserted when reaching a leaf and when all branches of a node have succeeded. The meta-interpreter only executes branches for which no `solved/2` fact has been asserted.

Note that the time-complexity of this meta-interpreter is not yet as desired. Execution of a node will always be dependent on the number of original branches, instead of the number of remaining (as yet unsuccessful) branches.

```
run_flock(Flock) :-
    preprocess(Flock, NewFlock, 0, 1, 1, 1, _, _),
    % The code for preprocessing is given in appendix
    retractall(solved(_, _)),
    meta(NewFlock),
    solved(0, _), !.

meta((A,B)):- !,
    A,
    meta(B).
meta(or(Flock, PrevNode, Node, Branch, Length)):- !, % 'or' corresponds to a node
    handleflock(Flock, Node, 1),
    all_solved(Node, 0, Length),
    assert(solved(PrevNode, Branch)).
meta(leaf(PrevNode, Branch, Leaf)):- !, % corresponds to the end of a leaf
    write(succeed(Leaf)), nl,
    assert(solved(PrevNode, Branch)).

handleflock([], _, _).
handleflock([A|_], Node, Branch):-
    \+(solved(Node, Branch)),
    once(meta(A)), fail.
handleflock(_[_]As, Node, Branch):-
```

```

Branch1 is Branch + 1,
handleflock(As, Node, Branch1).

```

```

all_solved(Node, Branch, Length):-
  (Branch = Length -> true;
   Branch1 is Branch + 1,
   solved(Node, Branch1),
   all_solved(Node, Branch1, Length)
  ).

```

For Example 1 the input for the call to meta/1 is:

```

(a, or([ (b, or([ (c, leaf(2,1,1)), (d, leaf(2,2,2)), (e, leaf(2,3,3)) ], 1,2,1,3)),
        (f, leaf(1,2,4)),
        (g, or([ (h, leaf(3,1,5)), (i, leaf(3,2,6)), (j, leaf(3,3,7)) ], 1,3,3,3)) ],
  0,1,1,3))

```

### 3.4 A program transformation

We present a program transformation that describes exactly the semantics of the flock of queries. The nice thing about the transformation is that it only uses builtin predicates that are either standard, or at least present in some form in every implementation we know of. We present the transformation immediately for a nested flock; assume the query is:

```

a, (b, (c or d) or e, (f or g))

```

Then the transformed query is:

```

mark(B), a, bcdefg(_,Kill), (Kill = yes -> cut(B) ; true)

```

to be evaluated against the program in Figure 3.

Note that this code works under reasonable assumptions on the implemented database update view. In particular, this code works well for the logical and immediate update view.

Some explanation about the meaning of Kill and FKill in the program; take for example the code for the first clause of bcdefg/2: if the goal cd(⌊,Kill) has unified Kill with *yes*, it means that both (all) clauses of cd/2 have succeeded (and have been killed), so the backtracking over the conjunction (b, cd(⌊,Kill)) must be stopped (this is done by the cut(Kill)) and that the first clause of bcdefg/2 can be retracted. Which in consequence makes it worthwhile testing whether there are any clauses for bcdefg/2 left: if not the FKill in the first clause of bcdefg/2 is unified with *yes* - propagating the effect of retracting clauses - otherwise to *no*, meaning that there are still alternatives of bcdefg/2 with non succeeded alternatives.

In the query, *Kill = yes* means that all alternatives have succeeded, so backtracking can be stopped.

Note that for the leaves of the flock (in the above example these are c,d,f and g) the code can be specialised, in particular the code for cd/2 can be specialized to:

```

cd(1,FKill) :- once(c), r(cd(1,⌊),FKill).
cd(2,FKill) :- once(d), r(cd(2,⌊),FKill).

```

The predicates mark/1 and cut/1 exist in most implementations, if only to implement a meta-interpreter that handles !/0 correctly. In SICStus Prolog for example, they are named 'CHOICE IDIOM'/1 and 'CUT IDIOM'/1 (SICStus 3 #5: Tue Feb 23 23:14:45 MET 1999) and awkward to use.<sup>5</sup> In XSB they are named

<sup>5</sup>they only work in compiled code, meaning they cannot occur in dynamic code

:- dynamic bcdefg/2, cd/2, fg/2, c/2, d/2, f/2, g/2.

```

bcdefg(1,FKill) :-      mark(B),
                        b,
                        cd(_,Kill),
                        s(Kill,B,bcdefg(1,_),FKill).
bcdefg(2,FKill) :-      mark(B),
                        e,
                        fg(_,Kill),
                        s(Kill,B,bcdefg(2,_),FKill).

cd(1,FKill) :-          mark(B),
                        c(1,Kill),
                        s(Kill,B,cd(1,_),FKill).
cd(2,FKill) :-          mark(B),
                        d(1,Kill),
                        s(Kill,B,cd(2,_),FKill).

fg(1,FKill) :-          mark(B),
                        f(1,Kill),
                        s(Kill,B,fg(1,_),FKill).
fg(2,FKill) :-          mark(B),
                        g(1,Kill),
                        s(Kill,B,fg(2,_),FKill).

c(1,FKill) :-           mark(B),
                        c,
                        Kill = yes, % because no alternative must be done
                        s(Kill,B,c(1,_),FKill).
d(1,FKill) :-           mark(B),
                        d,
                        Kill = yes, % because no alternative must be done
                        s(Kill,B,d(1,_),FKill).
f(1,FKill) :-           mark(B),
                        f,
                        Kill = yes, % because no alternative must be done
                        s(Kill,B,f(1,_),FKill).
g(1,FKill) :-           mark(B),
                        g,
                        Kill = yes, % because no alternative must be done
                        s(Kill,B,g(1,_),FKill).

r(Goal,Kill) :-         write(Goal-succeeded),nl, fail.
r(Goal,Kill) :-         retractall(Goal), fail.
r(Goal,Kill) :-         functor(Goal,Name,Arity),
                        functor(Head,Name,Arity),
                        clause(Head,_),
                        !,
                        Kill = no. % There is still a clause for Name/Arity.
r(_,yes). % all clauses have already succeeded.

s(Kill,B,Goal,FKill) :- (Kill = yes ->
                        cut(B),
                        r(Goal,FKill)
                        ;
                        FKill = no
                        ).

```

Figure 3: Transformed program

'\_\$\_savecp'/1 and '\_\$\_cutto'/1 and not meant for the programmer. In ProLog-by-BIM, they are named mark/1 and cut/1 and figure in the user manual.

It is straightforward to deal with arguments to goals in the original query and how one deals with. Also the generalisation to an or-disjunction of more than two goals is not difficult.

Although reasonably portable, it is clear that the above implementation has some drawbacks: there is some unneeded inefficiency in the Prolog builtins used, and also, it relies on having the flock of queries being transformed to dynamic code. These inefficiencies can be avoided by the careful introduction of a set of new primitives or adding some features to the underlying WAM engine. We have explored both approaches. We report on WAM enhancement only in the next section.

The approach in the current section has only been tested "by hand" and the transformation was not automated. Consequently, we will not present any measurements about it.

### 3.5 An implementation in a WAM based engine

As an introduction, we briefly discuss how a non-nested flock is treated.

Let us remind the reader that a body like a, (b ; c ; d), ... results in WAM code like:

```
        call a
        ortry @2
        call b
        jump @4
@2:     orretry @3
        call c
        jump @4
@3:     ortrust
        call d
        jump @4
@4:     ...
```

Consider in analogy the flock a, (b or c or d) which we propose to translate to very similar code:

```
        call a
        flock_try 17
@1:     call b
        flock_success
@2:     call c
        flock_success
@3:     call d
        flock_success
```

The 17 in the above code refers to a table which contains the addresses @1, @2 and @3 in this order. It is convenient to put also a sentinel - the nil address will do.

We extend the WAM machine with one new register: the current flock choice point pointer *FB*. It is maintained on creation and deletion of flock choice points by the instructions *flock\_try* and *flock\_retry* and only saved in those choice points.

The instruction *flock\_try* creates a flock choice point and picks up from a table with number 17 the first address and assigns it to the instruction pointer *P*. The ALT field of the choice point is put to a new instruction, *flock\_retry*, which is described later. As arguments, the choice point gets 17 (the table number) and 2 - the next alternative in the table to be tried. Also *FB* is saved and set to the just created choice point.

Flock\_retry looks at the second argument of the choice point; suppose it is *i*; it sets P to the *i*-th address in the table and increments the second argument. If the address is nil, the choice point is removed and execution fails.

Flock\_success is a new instruction that cuts the choice points up to FB and removes from the table (that FB points to) the just executed flock alternative: how we propose to do this exactly is detailed later. When the table becomes empty, the computation halts. This last behavior is inadequate when the flock is nested.

So, consider as before the nested flock

```
a, (b, (c or d) or e, (f or g))
```

There are three or-constructs in this flock, and we'll number their corresponding tables with t1, t2 and t3. The generated code we propose is:

```
      call a
      flock_try t1
@1:   call b
      flock_try t2
@2:   call c
      flock_success
@3:   call d
      flock_success
@4:   call e
      flock_try t3
@5:   call f
      flock_success
@6:   call g
      flock_success
```

A flock table contains the following items:

```
the number of flock alternatives: NFA
the addresses of the alternatives - as an array[1..NFA]
```

So, t1 looks like:

```
2
@1
@4
```

The code for flock\_try t then becomes:

```
set up an ordinary choice point
B[FB] := FB;
B[TABLE] := t;
B[CEA] := 1; % CEA: currently executing alternative
make ALT point to flock_retry
P := t.alt_array[1];
FB := B;
```

The code for flock\_retry is:

```

% the choice point being backtracked to has a reference to t
B[CEA]++;
P := t.alt_array[B[CEA]];
if (P == nil)
{
    FB := B[FB];
    B := B[B];
    fail;
}

```

Flock\_success is implemented as:

```

B := FB; % i.e. cut up to the nearest flock choice point
% now remove the alternative from t
t := B[TABLE];
t.alt_array[B[CEA]] := t.alt_array[t.NFA];
t.alt_array[t.NFA] := nil;
B[CEA]--;
t.NFA--;
while (t.NFA == 0)
{
    % cut up to previous FB
    % remove alternative from table
}
do_whatever_has_to_be_done_at_success_of_query then fail;

```

There remains to explain how the flock tables are created at runtime: we have found this most convenient by generating new instructions **flock\_init** and **flock\_fill** in the beginning of the clause that represents the flock query: the details about these instructions can be easily reconstructed.

We have implemented this schema in ToC, XSB Prolog and finally in ilProlog. The next section presents some preliminary measurements in ToC and ilProlog.

### 3.6 Evaluation

Basically two experiments were done, the first experiment uses a small artificial example<sup>6</sup>. The benchmark consists of 8 flocks. In the experiment all 8 flocks were run after each other, 1000 times. The benchmark was run:

- without using flocks,
- using the meta-interpreter defined above (only on ToC <sup>7</sup>) and
- using the WAM-based approach.

The results are collected in Table 1. All computations were performed on a Pentium II 233 MHz with Linux. The time unit is the millisecond. For the meta-interpreter it turns out that the overhead of the meta-level destroys any possible advantage of using the flock-mechanism. For WAM-flocks we notice a significant speed-up of more than 40 percent in both the ToC and the ilProlog approach.

We tested the mechanism on a realistic benchmark as well. This benchmark set contains three sets of queries (qs1, qs2 and qs3 in Table 2): the second and third set are realistic, as they are taken from an existing

---

<sup>6</sup>part of the example can be found in appendix

<sup>7</sup>ilProlog has currently no assert-like predicates

	No flocks	Meta-interpreter	WAM flocks
ToC	360	2000	210
ilProlog	430	-	230

Table 1: Timings in milliseconds on a small artificial example

application. This example was run on the ilProlog system. Each set of queries was run 100 times, each time on several examples in the database. The WAM-flocks show a speed-up of respectively 2.5, almost 8 and 42!

	Number of queries	No flocks	WAM flock	speed-up
qs1	51	400	150	2.67
qs2	681	7800	980	7.95
qs3	1106	12000	280	42.85

Table 2: Results on realistic queries in ilProlog

## 4 Future work

Since the figures above exclude the meta interpreter approach, and show that the compiled approach buys in practice a significant performance gain over the individual execution, we will continue the implementation within the WAM setting. With the tools at our disposal now, it takes too long to compile large flocks of queries. Indeed, even the SICStus compiler takes more than 10 seconds to compile the largest flock: this must be reduced significantly. Since the form of flocks is quite regular, a dedicated flock compiler will be build and is expected to perform acceptably. A second technical problem will eventually arise: as flocks grow, they might contain too many variables for the underlying logic engine. Typically, WAM implementations restrict the number of temporary and permanent variables to 255. Just lifting this restriction might be the wrong thing to do, as this possibly affects aversely the speed of the overall system. The alternative is to allow flocks to be spread over several clauses. We will look into how to define the semantics and implementation of multiple-clause flocks. Finally, sometimes the success of one branch of a flock determines the success or the failure of another branch: such dependencies can perhaps be transformed away at the source level, but some lower level support could be mandatory from the performance point of view. Such support will be investigated as well. Eventually, this work will be integrated in a data mining tool.

## 5 Concluding remarks

Inductive learning systems are becoming more and more sophisticated. Some of them have for historical reasons been implemented in Prolog or in Prolog-like languages. The speed of the underlying Prolog engine was for a long time the main determinant of the speed of the IL system. Now that larger data sets are tackled and better understanding of the incremental process of refinement trees has emerged, plain Prolog can no longer meet the requirements of an IL system: dedicated support is needed, e.g. in the context switching between examples, dealing with large external data bases and as is the subject of this paper, query

processing. This motivated us to write a dedicated Prolog-like engine *ilProlog* which incorporates already the compilation of a flock of queries and will in the future also give support for the other needs of Tilde [2] and WARMR [3].

## Acknowledgement

We thank the Leuven ML team of Leuven, in particular Hendrik Blockeel, Luc De Haspe and Wim Van Laer, for the specification of the problem and the test examples.

## References

- [1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] Blockeel, H. and De Raedt, L., *Top-down Induction of First Order Logical Decision Trees* Journal of Artificial Intelligence, Vol. 101, 1-2, pp. 285-297
- [3] L. Dehaspe and H. Toivonen. *Discovery of frequent Datalog patterns* Data Mining and Knowledge Discovery 3(1): 7-36, 1999.
- [4] A. Mariën, B. Demoen. Findall without findall/3 Proceedings of ICLP'93 Budapest
- [5] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
- [6] H. Vandecasteele. Compiling Prolog to ANSI C. In Y. Deville, editor, *Proceedings of the Eight Benelux Workshop on Logic Programming*, 1996.

## Appendix

### Preparing the query for the meta-interpreter

Note that the following preprocessor assumes that the flock of the form  $a, (b, (c \text{ or } d \text{ or } e) \text{ or } f \text{ or } g, (h \text{ or } i \text{ or } j))$  was already transformed to the form  $(a, \text{or}([(b, \text{or}([c, d, e])), f, (g, \text{or}([h, i, j]))]))$

```
preprocess((A, B), (A, NewB), PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr1, LeafNr1):-!,
    preprocess(B, NewB, PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr1, LeafNr1).
preprocess(or(Queryys), or(NQueryys, PrevNode, NodeNr0, BranchNr, Length),
    PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr1, LeafNr1):-!,
    NodeNr2 is NodeNr0 + 1,
    preprocessbranches(Queryys, NQueryys, NodeNr0, NodeNr2, LeafNr0, 1, NodeNr1, LeafNr1, Length).
preprocess(A, (A, leaf(PrevNode, BranchNr, LeafNr0)), PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr0, LeafNr1):-
    LeafNr1 is LeafNr0 + 1.

preprocessbranches([], [],-, NodeNr, LeafNr, BranchNr, NodeNr, LeafNr, BranchNr).
preprocessbranches([Query|Queryys], [NewQuery|NewQueryys],
    PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr1, LeafNr1, Length):-
    preprocess(Query, NewQuery, PrevNode, NodeNr0, LeafNr0, BranchNr, NodeNr2, LeafNr2),
    BranchNr1 is BranchNr + 1,
    preprocessbranches(Queryys, NewQueryys, PrevNode, NodeNr2, LeafNr2, BranchNr1, NodeNr1, LeafNr1, Length).
```

## A small example of query flocks

```
% example where the queries act on
sendback.
worn(wheel).
worn(control_unit).

% background knowledge
replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).

% Four from the eight flocks
q1:- worn(_186), ( true, success(1);
                true, success(2)
                ).

q2:- worn(_188), ( worn(_227), not(_227=_188), success(1);
                  replaceable(_188), true, success(2);
                  not_replaceable(_188), success(3)
                  ).

q3:- worn(_217), ( not_replaceable(_217),
                  ( worn(_292), not(_292=_217), success(1);
                    replaceable(_217), success(2)
                  );
                  replaceable(_217), worn(_238), not(_238=_217), success(3);
                  worn(_212), not(_212=_217), worn(_199),
                  not(_199=_217), not(_199=_212), success(4)
                  ).

q4:- worn(_206), ( worn(_325), not(_325=_206), worn(_312),
                  not(_312=_206), not(_312=_325),
                  ( worn(_419), not(_419=_206), not(_419=_325),
                    not(_419=_312), success(1);
                    replaceable(_312), true, success(2);
                    not_replaceable(_312), success(3)
                  );
                  replaceable(_206), worn(_232), not(_232=_206),
                  ( replaceable(_232), true, success(4);
                    not_replaceable(_232), success(5)
                  );
                  not_replaceable(_206), worn(_196), not(_196=_206),
                  not_replaceable(_196), success(6)
                  ).
```