Demand-Driven Indexing of Prolog Clauses[⋆]

Vítor Santos Costa[1], Konstantinos Sagonas[2], and Ricardo Lopes

[1] LIACC- DCC/FCUP, University of Porto, Portugal
[2] National Technical University of Athens, Greece

Abstract. As logic programming applications grow in size, Prolog systems need to efficiently access larger and larger data sets and the need for any- and multi-argument indexing becomes more and more profound. Static generation of multi-argument indexing is one alternative, but applications often rely on features that are inherently dynamic which makes static techniques inapplicable or inaccurate. Another alternative is to employ dynamic schemes for flexible demand-driven indexing of Prolog clauses. We propose such schemes and discuss issues that need to be addressed for their efficient implementation in the context of WAM-based Prolog systems. We have implemented demand-driven indexing in two different Prolog systems and have been able to obtain non-negligible performance speedups: from a few percent up to orders of magnitude. Given these results, we see very little reason for Prolog systems not to incorporate some form of dynamic indexing based on actual demand. In fact, we see demand-driven indexing as only the first step towards effective runtime optimization of Prolog programs.

1 Introduction

The WAM [1] has mostly been a blessing but occasionally also a curse for Prolog systems. Its ingenious design has allowed implementors to get byte code compilers with decent performance — it is not a fluke that most Prolog systems are still based on the WAM. On the other hand, *because* the WAM gives good performance in many cases, implementors have not incorporated in their systems many features that drastically depart from WAM's basic characteristics. For example, first argument indexing is sufficient for many Prolog applications. However, it is clearly sub-optimal for applications accessing large data sets; for a long time now, the database community has recognized that good indexing is the basis for fast query processing.

As logic programming applications grow in size, Prolog systems need to efficiently access larger and larger data sets and the need for any- and multi-argument indexing becomes more and more profound. Static generation of multi-argument indexing is one alternative. The problem is that this alternative is often unattractive because it may drastically increase the size of the generated byte code and do so unnecessarily. Static analysis can partly address this concern, but in applications that rely on features which are inherently dynamic (e.g., generating hypotheses for inductive logic programming data sets during runtime) static analysis is inapplicable or grossly inaccurate. Another alternative, which has not been investigated so far, is to do flexible indexing on demand during program execution.

[⋆] Dedicated to the memory of our friend, colleague and co-author Ricardo Lopes. We miss you!

This is precisely what we advocate with this paper. More specifically, we present a small extension to the WAM that allows for flexible indexing of Prolog clauses during runtime based on actual demand. For static predicates, the scheme we propose is partly guided by the compiler; for dynamic code, besides being demand-driven by queries, the method needs to cater for code updates during runtime. Where our schemes radically depart from current practice is that they generate new byte code during runtime, in effect doing a form of just-in-time compilation. In our experience these schemes pay off. We have implemented demand-driven indexing in two different Prolog systems (YAP and XXX) and have obtained non-trivial speedups, ranging from a few percent to orders of magnitude, across a wide range of applications. Given these results, we see very little reason for Prolog systems not to incorporate some form of indexing based on actual demand from queries. In fact, we see demand-driven indexing as only the first step towards effective runtime optimization of Prolog programs.

Organization. After commenting on the state of the art and related work concerning indexing in Prolog systems (Sect. 2) we briefly review indexing in the WAM (Sect. 3). We then present demand-driven indexing schemes for static (Sect. 4) and dynamic (Sect. 5) predicates, their implementation in two Prolog systems (Sect. 6) and the performance benefits they bring (Sect. 7). The paper ends with some concluding remarks.

2 State of the Art and Related Work

Many Prolog systems still only support indexing on the main functor symbol of the first argument. Some others, such as YAP version 4, can look inside some compound terms [2]. SICStus Prolog supports *shallow backtracking* [3]; choice points are fully populated only when it is certain that execution will enter the clause body. While shallow backtracking avoids some of the performance problems of unnecessary choice point creation, it does not offer the full benefits that indexing can provide. Other systems such as BIM-Prolog [4], SWI-Prolog [5] and XSB [6] allow for user-controlled multi-argument indexing. Notably, ilProlog [7] uses compile-time heuristics and generates code for multi-argument indexing automatically. In all these systems, this support comes with various implementation restrictions. For example, in SWI-Prolog at most four arguments can be indexed; in XSB the compiler does not offer multi-argument indexing and the predicates need to be asserted instead; we know of no system where multi-argument indexing looks inside compound terms. More importantly, requiring users to specify arguments to index on is neither user-friendly nor guarantees good performance results.

Recognizing the need for better indexing, researchers have proposed more flexible indexing mechanisms for Prolog. For example, Hickey and Mudambi proposed *switching trees* [8], which rely on the presence of mode information. Similar proposals were put forward by Van Roy, Demoen and Willems who investigated indexing on several arguments in the form of a *selection tree* [9] and by Zhou et al. who implemented a *matching tree* oriented abstract machine for Prolog [10]. For static predicates, the XSB compiler offers support for *unification factoring* [11]; for asserted code, XSB can represent databases of facts using *tries* [12] which provide left-to-right multi-argument indexing. However, in XSB none of these mechanisms is used automatically; instead the user has to specify appropriate directives.

Long ago, Kliger and Shapiro argued that such tree-based indexing schemes are not cost effective for the compilation of Prolog programs [13]. Some of their arguments make sense for certain applications, but, as we shall show, in general they underestimate the benefits of indexing on EDB predicates. Nevertheless, it is true that unless the modes of predicates are known we run the risk of doing indexing on output arguments, whose only effect is an unnecessary increase in compilation times and, more importantly, in code size. In a programming language such as Mercury [14] where modes are known the compiler can of course avoid this risk; indeed in Mercury modes (and types) are used to guide the compiler generate good indexing tables. However, the situation is different for a language like Prolog. Getting accurate information about the set of all possible modes of predicates requires a global static analyzer in the compiler — and most Prolog systems do not come with one. More importantly, it requires a lot of discipline from the programmer (e.g., that applications use the module system religiously and never bypass it). As a result, most Prolog systems currently do not provide the type of indexing that applications require. Even in systems such as Ciao [15], which do come with a built-in static analyzer and more or less force such a discipline on the programmer, mode information is not used for multi-argument indexing.

The situation is actually worse for certain types of Prolog applications. For example, consider applications in the area of inductive logic programming. These applications on the one hand have high demands for effective indexing since they need to efficiently access big datasets and on the other they are unfit for static analysis since queries are often ad hoc and generated only during runtime as new hypotheses are formed or refined. Our thesis is that the abstract machine should be able to adapt automatically to the runtime requirements of such or, even better, of all applications by employing increasingly aggressive forms of dynamic compilation. As a concrete example of what this means in practice, in this paper we will attack the problem of satisfying the indexing needs of applications during runtime. Naturally, we will base our technique on the existing support for indexing that the WAM provides, but we will extend this support with the technique of demand-driven indexing that we describe in the next sections.

3 Indexing in the WAM

To make the paper relatively self-contained we review the indexing instructions of the WAM and their use. In the WAM, the first level of dispatching involves a test on the type of the argument. The switch_on_term instruction checks the tag of the dereferenced value in the first argument register and implements a four-way branch where one branch is for the dereferenced register being an unbound variable, one for being atomic, one for (non-empty) list, and one for structure. In any case, control goes to a bucket of clauses. In the buckets for constants and structures the second level of dispatching involves the value of the register. The switch_on_constant and switch_on_structure instructions implement this dispatching: typically with a fail instruction when the bucket is empty, with a jump instruction for only one clause, with a sequential scan when the number of clauses is small, and with a hash table lookup when the number of clauses

```
has_property(d1,salmonella,p).
has_property(d1,salmonella_n,p).
has_property(d2,salmonella,p).
has_property(d2,cytogen_ca,n).
has_property(d3,cytogen_ca,p).
```

(a) Some Prolog clauses

```
switch_on_constant r1 5 T1
try L1
retry L2
retry L3
retry L4
trust L5
```

T_1:

Hash Table Info	
d1	try L_1 trust L_2
d2	try L_3 trust L_4
d3	jump L_5

(b) WAM indexing

```
L1: get_constant r1 d1
    get_constant r2 salmonella
    get_constant r3 p
    proceed
L2: get_constant r1 d1
    get_constant r2 salmonella_n
    get_constant r3 p
    proceed
L3: get_constant r1 d2
    get_constant r2 salmonella
    get_constant r3 p
    proceed
L4: get_constant r1 d2
    get_constant r2 cytogen_ca
    get_constant r3 n
    proceed
L5: get_constant r1 d3
    get_constant r2 cytogen_ca
    get_constant r3 p
    proceed
```

(c) Code for the clauses

```
switch_on_constant r1 5 T1
dindex_on_constant r2 5 3
dindex_on_constant r3 5 3
try L1
retry L2
retry L3
retry L4
trust L5
```

T_1:

Hash Table Info	
d1	try L_1 trust L_2
d2	try L_3 trust L_4
d3	jump L_5

(d) Any arg indexing

Fig. 1. Part of the Carcinogenesis dataset and WAM code that a byte code compiler generates

exceeds a threshold. For this reason the switch_on_constant and switch_on_structure instructions take as arguments the hash table T and the number of clauses N the table contains. In each bucket of this hash table and also in the bucket for the variable case of switch_on_term the code sequentially backtracks through the clauses using a try-retry-trust chain of instructions. The try instruction sets up a choice point, the retry instructions (if any) update certain fields of this choice point, and the trust instruction removes it.

The WAM has additional indexing instructions (try_me_else and friends) that allow indexing to be interspersed with the code of clauses. We will not consider them here. This is not a problem since the above scheme handles all programs. Also, we will feel free to do some minor modifications and optimizations when this simplifies things.

Let's see an example. Consider the Prolog code shown in Fig. 1(a), a fragment of the machine learning dataset *Carcinogenesis*. These clauses get compiled to the WAM code shown in Fig. 1(c). The first argument indexing code that a Prolog compiler generates is shown in Fig. 1(b). This code is typically placed before the code for the clauses and the switch_on_constant is the entry point of the predicate. Note that compared with vanilla WAM this instruction has an extra argument: the register on the value of which we index (r_1). This extra argument will allow us to go beyond first argument indexing. Another departure from the WAM is that if this argument register contains an unbound variable instead of a constant then execution will continue with the next instruction; in effect we have merged part of the functionality of switch_on_term into the switch_on_constant instruction. This small change in the behavior of switch_on_constant will allow us to get demand-driven indexing. Let's see how.

4 Demand-Driven Indexing of Static Predicates

For static predicates the compiler has complete information about all clauses and shapes of their head arguments. It is both desirable and possible to take advantage of this information at compile time and so we treat the case of static predicates separately. We will do so with schemes of increasing effectiveness and implementation complexity.

4.1 A simple WAM extension for any argument indexing

Let us initially consider the case where the predicates to index consist only of Datalog facts. This is commonly the case for all extensional database predicates where indexing is most effective and called for.

Refer to the example in Fig. 1. The indexing code of Fig. 1(b) incurs a small cost for a call where the first argument is a variable (namely, executing the switch_on_constant instruction) but the instruction pays off for calls where the first argument is bound. On the other hand, for calls where the first argument is a free variable and some other argument is bound, a choice point will be created, the try-retry-trust chain will be used, and execution will go through the code of all clauses. This is clearly inefficient, more so for larger data sets. We can do much better with the relatively simple scheme shown in Fig. 1(d). Immediately after the switch_on_constant instruction, we can statically generate dindex_on_constant (demand indexing) instructions, one for each remaining argument. Recall that the entry point of the predicate is the switch_on_constant instruction. The dindex_on_constant r_i N A instruction works as follows:

- if the argument r_i is a free variable, execution continues with the next instruction;
- otherwise, demand-driven indexing kicks in as follows. The abstract machine scans the WAM code of the clauses and creates an index table for the values of the corresponding argument. It can do so because the instruction takes as arguments the number of clauses N to index and the arity A of the predicate. (In our example, the numbers 5 and 3.) For Datalog facts, this information is sufficient. Because the WAM byte code for the clauses has a very regular structure, the index table can be created very quickly. Upon its creation, the dindex_on_constant instruction gets transformed to a switch_on_constant. Again this is straightforward because of the two instructions have similar layouts in memory. Execution of the abstract machine then continues with the switch_on_constant instruction.

Figure 2 shows the index table T_2 which is created for our example and how the indexing code looks after the execution of a call with mode `(out,in,?)`. Note that the dindex_on_constant instruction for argument register r_2 has been appropriately patched. The call that triggered demand-driven indexing and subsequent calls of the same mode will use table T_2. The index for the second argument has been created.

The main advantage of this scheme is its simplicity. The compiled code (Fig. 1(d)) is not significantly bigger than the code which a WAM-based compiler would generate (Fig. 1(b)) and, if demand-driven indexing turns out unnecessary during runtime (e.g. execution encounters only open calls or with only the first argument bound), the extra overhead is minimal: the execution of some dindex_on_constant instructions for the open call only. In short, this is a simple scheme that allows for indexing on *any single* argument. At least for big sets of Datalog facts, we see little reason not to use it.

```
switch_on_constant r1 5 T1
switch_on_constant r2 5 T2
dindex_on_constant r3 5 3
try L1
retry L2
retry L3
retry L4
trust L5
```

T_1:

Hash Table Info	
d1	try L_1 trust L_2
d2	try L_3 trust L_4
d3	jump L_5

T_2:

Hash Table Info	
salmonella	try L_1 trust L_3
salmonella_n	jump L_2
cytrogen_ca	try L_4 trust L_5

Fig. 2. WAM code after demand-driven indexing for argument 2; T_2 is generated dynamically

Optimizations. Because we are dealing with static code, there are opportunities for some easy optimizations. Suppose we statically determine that there will never be any calls with `in` mode for some arguments or that these arguments are not discriminating enough.[3] Then we can avoid generating dindex_on_constant instructions for them. Also, suppose we know that some arguments are most likely than others to be used in the `in` mode. Then we can simply place the dindex_on_constant instructions for them before the instructions for other arguments. This is possible since all indexing instructions take the argument register number as an argument; their order does not matter.

4.2 From any argument indexing to multi-argument indexing

The scheme of the previous section gives us only single argument indexing. However, all the infrastructure we need is already in place. We can use it to obtain any fixed-order multi-argument demand-driven indexing in a straightforward way.

Note that the compiler knows exactly the set of clauses that need to be tried for each query with a specific symbol in the first argument. For multi-argument demand-driven indexing, instead of generating for each hash bucket only try-retry-trust instructions, the compiler can prepend appropriate demand indexing instructions. We illustrate this on our running example. The table T_1 contains four dindex_on_constant instructions: two for each of the remaining two arguments of hash buckets with more than one alternative. For hash buckets with none or only one alternative (e.g., for `d3`'s bucket) there is obviously no need to resort to demand-driven indexing for the remaining arguments. Figure 3 shows the state of the hash tables after the execution of queries `has_property(C,salmonella,T)`, which creates T_2, and `has_property(d2,P,n)` which creates the T_3 table and transforms the dindex_on_constant instruction for `d2` and register r_3 to the appropriate switch_on_constant instruction.

Implementation issues. In the dindex_on_constant instructions of Fig. 3 notice the integer 2 which denotes the number of clauses that the instruction will index. Using this number an index table of appropriate size will be created, such as T_3. To fill this table we need information about the clauses to index and the symbols to hash on. The clauses can be obtained by scanning the labels of the try-retry-trust instructions following dindex_on_constant; the symbols by looking at appropriate byte code offsets (based on the argument register number) from these labels. In our running example, the symbols can be obtained by looking at the second argument of the get_constant instruction whose argument register is r_2. In the loaded bytecode, assuming the argument

[3] In our example, suppose the third argument of `has_property/3` was the atom `p` throughout.

```
switch_on_constant r1 5 T1
switch_on_constant r2 5 T2
dindex_on_constant r3 5 3
try L1
retry L2
retry L3
retry L4
trust L5
```

T_1:

Hash Table Info	
d1	dindex_on_constant r_2 2 3 dindex_on_constant r_3 2 3 try L_1 trust L_2
d2	dindex_on_constant r_2 2 3 switch_on_constant r_3 2 T_3 try L_3 trust L_4
d3	jump L_5

T_2:

Hash Table Info	
salmonella	dindex_on_constant r_3 2 3 try L_1 trust L_3
salmonella_n	jump L_2
cytrogen_ca	dindex_on_constant r_3 2 3 try L_4 trust L_5

T_3:

Hash Table Info	
p	jump L_3
n	jump L_4

Fig. 3. demand-driven indexing for all arguments; T_1 is static; T_2 and T_3 are created dynamically

register is represented in one byte, these symbols are found $sizeof(\text{get\_constant}) + sizeof(opcode) + 1$ bytes away from the clause label; see Fig. 1(c). Thus, multi-argument demand-driven indexing is easy to get and the creation of index tables can be extremely fast when indexing Datalog facts.

4.3 Beyond Datalog and other implementation issues

Indexing on demand clauses with function symbols is not significantly more difficult. The scheme we have described is applicable but requires the following extensions:

1. Besides dindex_on_constant we also need dindex_on_term and dindex_on_structure instructions. These are the demand-driven indexing counterparts of the WAM's switch_on_term and switch_on_structure.
2. Because the byte code for the clause heads does not necessarily have a regular structure, the abstract machine needs to be able to "walk" the byte code instructions and recover the symbols on which indexing will be based. Writing such a code walking procedure is not hard.
3. Indexing on a position that contains unconstrained variables for some clauses is tricky. The WAM needs to group clauses in this case and without special treatment creates two choice points for this argument (one for the variables and one per each group of clauses). However, this issue and how to deal with it is well-known by now. Possible solutions to it are described in a paper by Carlsson [16] and can be readily adapted to demand-driven indexing. Alternatively, in a simple implementation, we can skip demand-driven indexing for positions with variables in some clauses.

Before describing demand-driven indexing more formally, we remark on the following design decisions whose rationale may not be immediately obvious:

- By default, only table T_1 is generated at compile time (as in the WAM) and the additional index tables $T_2, T_3, \ldots$ are generated dynamically. This is because we do not want to increase compiled code size unnecessarily (i.e., when there is no demand for these indices).
- On the other hand, we generate dindex_on_* instructions at compile time for the head arguments.[4] This does not noticeably increase the generated byte code but it

[4] The dindex_on_* instructions for T_1 can be generated either by the compiler or the loader.

greatly simplifies code loading. Notice that a nice property of the scheme we have described is that the loaded byte code can be patched *without* the need to move any instructions.

– Finally, one may wonder why the dindex_on_* instructions create the dynamic index tables with an additional code walking pass instead of piggy-backing on the pass which examines all clauses via the main try-retry-trust chain. Main reasons are: 1) in many cases the code walking can be selective and guided by offsets and 2) by first creating the index table and then using it we speed up the execution of the queries and often avoid unnecessary choice point creations.

Note that all these decisions are orthogonal to the main idea and are under compiler control. For example, if analysis determines that some argument sequences will never demand indexing we can simply avoid generation of dindex_on_* instructions for them. Similarly, if some argument sequences will definitely demand indexing we can speed up execution by generating the appropriate tables at compile time instead of dynamically.

4.4 Demand-driven index construction and its properties

The idea behind demand-driven indexing can be captured in a single sentence: *we can generate every index we need during program execution when this index is demanded*. Subsequent uses of these indices can speed up execution considerably more than the time it takes to construct them (more on this below) so this runtime action makes sense.

Let p/k be a predicate with n clauses. At a high level, its indices form a tree whose root is the entry point of the predicate. For simplicity, assume that the root node of the tree and the interior nodes corresponding to the index table for the first argument have been constructed at compile time. Leaves of this tree are the nodes containing the code for the clauses of the predicate and each clause is identified by a unique label $L_i, 1 \leq i \leq n$. Execution always starts at the first instruction of the root node and follows Algorithm 1. The algorithm might look complicated but is actually quite simple. Each non-leaf node contains a sequence of byte code instructions with groups of the form $\langle I_1, \ldots, I_m, T_1, \ldots, T_l \rangle, 0 \leq m \leq k, 1 \leq l \leq n$ where each of the I instructions, if any, is either a switch_on_* or a dindex_on_* instruction and each of the T instructions either forms a sequence of try-retry-trust instructions (if $l > 1$) or is a jump instruction (if $l = 1$). Step 2.2 dynamically constructs an index table $\mathcal{T}$ whose buckets are the newly created interior nodes in the tree. Each bucket associated with a single clause contains a jump to the label of that clause. Each bucket associated with many clauses starts with the I instructions which are yet to be visited and continues with a try-retry-trust chain pointing to the clauses. When the index construction is done, the instruction mutates to a switch_on_* WAM instruction.

Complexity properties. Index construction during runtime does not change the complexity of query execution. First, note that each demanded index table will be constructed at most once. Also, a dindex_on_* instruction will be encountered only in cases where execution would examine all clauses in the try-retry-trust chain.[5] The construction visits these clauses *once* and then creates the index table in time linear in the number of clauses as one pass over the list of $\langle c, L \rangle$ pairs suffices. After index construction,

[5] This statement is possibly not valid in the presence of Prolog cuts.

Algorithm 1 Actions of the abstract machine with demand-driven indexing

```
1. if the current instruction $I$ is a switch_on_*, try, retry, trust or jump, act as in the WAM;
2. if the current instruction $I$ is a dindex_on_* with arguments $r, l$, and $k$ ($r$ is a register) then
   2.1 if register $r$ contains a variable, the action is a goto the next instruction in the node;
   2.2 if register $r$ contains a value $v$, the action is to dynamically construct the index:
      2.2.1 collect the subsequent instructions in a list $\mathcal{I}$ until the next instruction is a try;
      2.2.2 for each label $L$ in the try-retry-trust chain inspect the code of the clause with
            label $L$ to find the symbol $c$ associated with register $r$ in the clause; (This step
            creates a list of $\langle c, L\rangle$ pairs.)
      2.2.3 create an index table $\mathcal{T}$ out of these pairs as follows:
            • if $I$ is a dindex_on_constant or a dindex_on_structure then create an index
              table for the symbols in the list of pairs; each entry of the table is identified by
              a symbol $c$ and contains:
              ∗ the instruction jump $L_c$ if $L_c$ is the only label associated with $c$;
              ∗ the sequence of instructions obtained by appending to $\mathcal{I}$ a try-retry-trust
                chain for the sequence of labels $L'_1, \ldots, L'_l$ that are associated with $c$
            • if $I$ is a dindex_on_term then
              ∗ partition the sequence of labels $\mathcal{L}$ in the list of pairs into sequences of labels
                $\mathcal{L}_c, \mathcal{L}_l$ and $\mathcal{L}_s$ for constants, lists and structures, respectively;
              ∗ for each of the four sequences $\mathcal{L}, \mathcal{L}_c, \mathcal{L}_l, \mathcal{L}_s$ of labels create code:
                · the instruction fail if the sequence is empty;
                · the instruction jump $L$ if $L$ is the only label in the sequence;
                · the sequence of instructions obtained by appending to $\mathcal{I}$ a try-retry-trust
                  chain for the current sequence of labels;
      2.2.4 transform the dindex_on_* $r, l, k$ instruction to a switch_on_* $r, l, \mathcal{T}$ instruction;
      2.2.5 continue execution with this instruction.
```

execution will visit a subset of these clauses as the index table will be consulted. Thus, in cases where demand-driven indexing is not effective, execution of a query will at most double due to dynamic index construction. In fact, this worst case is pessimistic and unlikely in practice. On the other hand, demand-driven indexing can change the complexity of query evaluation from $O(n)$ to $O(1)$ where n is the number of clauses.

4.5 More implementation choices

The observant reader has no doubt noticed that Algorithm 1 provides multi-argument indexing but only for the main functor symbol. For clauses with compound terms that require indexing in their sub-terms we can either employ a program transformation such as *unification factoring* [11] at compile time or modify the algorithm to consider index positions inside compound terms. This is relatively easy to do but requires support from the register allocator (passing the sub-terms of compound terms in appropriate registers) and/or a new set of instructions. Due to space limitations we omit further details.

Algorithm 1 relies on a procedure that inspects the code of a clause and collects the symbols associated with some particular index position (step 2.2.2). If we are satisfied with looking only at clause heads, this procedure needs to understand only the structure of get and unify instructions. Thus, it is easy to write. At the cost of increased implementation complexity, this step can of course take into account other information that may exist in the body of the clause (e.g., type tests such as `var(X)`, `atom(X)`, aliasing constraints such as `X = Y`, numeric constraints such as `X > 0`, etc.).

A reasonable concern for demand-driven indexing is increased memory consumption. In our experience, this does not seem to be a problem in practice since most applications do not have demand for indexing on many argument combinations. In applications where it does become a problem or when running in an environment with limited memory, we can easily put a bound on the size of index tables, either globally or for each predicate separately. For example, the `dindex_on_*` instructions can either become inactive when this limit is reached, or better yet we can recover the space of some tables. To do so, we can employ any standard recycling algorithm (e.g., LRU) and reclaim the memory of index tables that are no longer in use. This is easy to do by reverting the corresponding `switch_on_*` instructions back to `dindex_on_*` instructions. If the indices are demanded again at a time when memory is available, they can simply be regenerated.

5 Demand-Driven Indexing of Dynamic Predicates

We have so far lived in the comfortable world of static predicates, where the set of clauses to index is fixed and the compiler can take advantage of this knowledge. Dynamic code introduces several complications:

– We need mechanisms to update multiple indices when new clauses are asserted or retracted. In particular, we need the ability to expand and possibly shrink multiple code chunks after code updates.
– We do not know a priori which are the best index positions and cannot determine whether indexing on some arguments is avoidable.
– Supporting the logical update (LU) semantics of ISO Prolog becomes harder.

We briefly discuss possible ways of addressing these issues. However, note that Prolog systems typically provide indexing for dynamic predicates and thus already deal in some way or another with these issues; demand-driven indexing makes the problems more involved but not fundamentally different than with only first argument indexing.

The first complication suggests that we should allocate memory for dynamic indices in separate chunks, so that these can be expanded and deallocated independently. Indeed, this is what we do. Regarding the second complication, in the absence of any other information, the only alternative is to generate indices for all arguments. As optimizations, we can avoid indexing predicates with only one clause and exclude arguments where some clause has a variable.

Under LU semantics, calls to dynamic predicates execute in a “snapshot” of the corresponding predicate. Each call sees the clauses that existed at the time when the call was made, even if some of the clauses were later retracted or new clauses were asserted. If several calls are alive in the stack, several snapshots will be alive at the same time. The standard solution to this problem is to use time stamps to tell which clauses are *live* for which calls. This solution complicates freeing index tables because: (1) an index table holds references to clauses, and (2) the table may be in use (i.e., may be accessible from the execution stacks). An index table thus is killed in several steps:

1. Detach the index table from the indexing tree.
2. Recursively *kill* every child of the current table; if a table is killed so are its children.
3. Wait until the table is not in use, that is, it is not pointed to from anywhere.
4. Walk the table and release any references it may hold.
5. Physically recover space.

6 Implementation in XXX and in YAP

The implementation of demand-driven indexing in XXX follows a variant of the scheme presented in Sect. 4. The compiler uses heuristics to determine the best argument to index on (i.e., this argument is not necessarily the first) and employs switch_on_* instructions for this task. It also statically generates dindex_on_constant instructions for other arguments that are good candidates for demand-driven indexing. Currently, an argument is considered a good candidate if it has only constants or only structure symbols in all clauses. Thus, XXX uses only dindex_on_constant and dindex_on_structure instructions, never a dindex_on_term. Also, XXX does not perform demand-driven indexing inside structure symbols. For dynamic predicates, demand-driven indexing is employed only if they consist of Datalog facts; if a clause which is not a Datalog fact is asserted, all dynamically created index tables for the predicate are simply removed and the dindex_on_constant instruction becomes a noop. All this is done automatically, but the user can disable demand-driven indexing in compiled code using an option.

YAP implements demand-driven indexing since version 5. The current implementation supports static code, dynamic code, and the internal database. It differs from the algorithm presented in Sect. 4 in that *all indexing code is generated on demand*. Thus, YAP cannot assume that a dindex_on_* instruction is followed by a try-retry-trust chain. Instead, by default YAP has to search the whole predicate for clauses that match the current position in the indexing code. Doing so for every index expansion was found to be very inefficient for larger relations: in such cases YAP will maintain a list of matching clauses at each dindex_on_* node. Indexing dynamic predicates in YAP follows very much the same algorithm as static indexing: the key idea is that most nodes in the index tree must be allocated separately so that they can grow or shrink independently. YAP can index arguments where some clauses have unconstrained variables, but only for static predicates, as in dynamic code this would complicate support for LU semantics.

YAP uses the term JITI (Just-In-Time Indexing) to refer to demand-driven indexing. In the next section we will take the liberty to use this term as a convenient abbreviation.

7 Performance Evaluation

We evaluate JITI on a set of benchmarks and applications. Throughout, we compare performance of JITI with first argument indexing. For the benchmarks of Sect. 7.1 and 7.2 which involve both systems, we used a 2.4 GHz P4-based laptop with 512 MB of memory. For the benchmarks of Sect. 7.3 which involve YAP 5.1.2 only, we used a 8-node cluster, where each node is a dual-core AMD 2600+ machine with 2GB of memory.

Table 1. Performance of some benchmarks with 1st vs. demand-driven indexing (times in msecs)

(a) When JITI is ineffective

	YAP		XXX	
Benchmark	1st	JITI	1st	JITI
tc_l_io (8000)	13	14	4	4
tc_r_io (2000)	1445	1469	614	615
tc_d_io (400)	3208	3260	2338	2300
tc_l_oo (2000)	3935	3987	2026	2105
tc_r_oo (2000)	2841	2952	1502	1512
tc_d_oo (400)	3735	3805	4976	4978
compress	3614	3595	2875	2848

(b) When JITI is effective

	YAP			XXX		
	1st	JITI	**ratio**	1st	JITI	**ratio**
sg_cyl	2,864	24	119×	2,390	28	85×
muta	30,057	16,782	1.79×	26,314	21,574	1.22×
pta	5,131	188	27×	4,442	279	16×
tea	1,478,813	54,616	27×	—	—	—

7.1 Performance of demand-driven indexing when ineffective

In some programs, demand-driven indexing does not trigger[6] or might trigger but have no effect other than an overhead due to runtime index construction. We therefore wanted to measure this overhead. As both systems support tabling, we decided to use tabling benchmarks because they are small and easy to understand, and because they are a bad case for JITI in the following sense: tabling avoids generating repetitive queries and the benchmarks operate over extensional database (EDB) predicates of size approximately equal to the size of the program. We used **compress**, a tabled program that solves a puzzle from an ICLP Prolog programming competition. The other benchmarks are different variants of tabled left, right and doubly recursive transitive closure over an EDB predicate forming a chain of size shown in Table 1(a) in parentheses. For each variant of transitive closure, we issue two queries: one with mode `(in,out)` and one with mode `(out,out)`. For YAP, indices on the first argument and try-retry-trust chains are built on all benchmarks under demand-driven indexing. For XXX, demand-driven indexing triggers on no benchmark but the dindex_on_constant instructions are executed for the three **tc_?_oo** benchmarks. As can be seen in Table 1(a), demand-driven indexing, even when ineffective, incurs a runtime overhead that is at the level of noise and goes mostly unnoticed. We also note that our aim here is *not* to compare the two systems, so the **YAP** and **XXX** columns should be read separately.

7.2 Performance of demand-driven indexing when effective

On the other hand, when demand-driven indexing is effective, it can significantly improve runtime performance. We use the following programs and applications:

sg_cyl The same generation DB benchmark on a $24 \times 24 \times 2$ cylinder. We issue the open query.
muta A computationally intensive application where most predicates are defined intentionally.
pta A tabled logic program implementing Andersen's points-to analysis. A medium-sized imperative program is encoded as a set of facts (about 16,000) and properties of interest are encoded using rules. Program properties are then determined by the closure of these rules.
tea Another implementation of Andersen's points-to analysis. The analyzed program, the `javac` benchmark, is encoded in a file of 411,696 facts (62,759,581 bytes in total). Its compilation exceeds the limits of the XXX compiler (w/o JITI). So we run this benchmark only in YAP.

[6] In XXX only; even 1st argument indexing is generated on demand when JITI is used in YAP.

As can be seen in Table 1(b), demand-driven indexing significantly improves the performance of these applications. In **muta**, which spends most of its time in recursive predicates, the speed up is only 79% in YAP and 22% in XXX. The remaining benchmarks execute several times (from 16 up to 119) faster. It is important to realize that *these speedups are obtained automatically*, i.e., without any programmer intervention or by using any compiler directives, in all these applications.

7.3 Performance of demand-driven indexing on ILP applications

The need for demand-driven indexing was originally noticed in inductive logic programming applications. These applications tend to issue ad hoc queries during execution and thus their indexing requirements cannot be determined at compile time. On the other hand, they operate on lots of data, so memory consumption is a reasonable concern. We evaluate JITI's time and space performance on some learning tasks using the Aleph system [17] and the datasets of Fig. 4 which issue simple queries in an extensional database. Several of these datasets are standard in the ILP literature.

Time performance. We compare times for 10 runs of the saturation/refinement cycle of the ILP system; see Table 2(a). The **Mesh** and **Pyrimidines** applications are the only ones that do not benefit much from indexing in the database; they do benefit through from indexing in the dynamic representation of the search space, as their running times improve somewhat with demand-driven indexing.

The **BreastCancer** and **GeneExpression** applications use unstructured data. The speedup here is mostly from multiple argument indexing. **BreastCancer** is particularly interesting. It consists of 40 binary relations with 65k elements each, where the first argument is the key. We know that most calls have the first argument bound, hence indexing was not expected to matter much. Instead, the results show demand-driven indexing to improve running time by more than an order of magnitude. This suggests that even a small percentage of badly indexed calls can end up dominating runtime.

IE-Protein_Extraction and **Thermolysin** are example applications that manipulate structured data. **IE-Protein_Extraction** is the largest dataset we consider, and indexing is absolutely critical. The speedup is not just impressive; it is simply not possible to run the application in reasonable time with only first argument indexing. **Thermolysin** is smaller and performs some computation per query, but even so, demand-driven indexing improves its performance by an order of magnitude. The remaining benchmarks improve from one to more than two orders of magnitude.

Space performance. Table 2(b) shows memory usage when using demand-driven indexing. The table presents data obtained at a point near the end of execution; memory usage should be at the maximum. These applications use a mixture of static and dynamic predicates and we show their memory usage separately. On static predicates, memory usage varies widely, from only 10% to the worst case, **Carcinogenesis**, where the index tables take more space than the original program. Hash tables dominate usage in **IE-Protein_Extraction** and **Susi**, whereas try-retry-trust chains dominate in **BreastCancer**. In most other cases no single component dominates memory usage. Memory usage for dynamic predicates is shown in the last two columns; this data is mostly used to store the search space. Observe that there is a much lower overhead in this case. A

Table 2. Time and space performance of JITI on Inductive Logic Programming datasets

(a) Time (in seconds) | (b) Memory usage (in KB)

	Time			Static code		Dynamic code	
Benchmark	1st	JITI	**ratio**	Clauses	Index	Clauses	Index
BreastCancer	1,450	88	16×	60,940	46,887	630	14
Carcinogenesis	17,705	192	92×	1,801	2,678	13,512	942
Choline	14,766	1,397	11×	666	174	3,172	174
GeneExpression	193,283	7,483	26×	46,726	22,629	116,463	9,015
IE-Protein_Extraction	1,677,146	2,909	577×	146,033	129,333	53,423	1,531
Mesh	4	3	1.3×	802	161	2,149	109
Pyrimidines	487,545	253,235	1.9×	774	218	25,840	12,291
Susi	105,091	307	342×	5,007	2,509	4,497	759
Thermolysin	50,279	5,213	10×	2,317	929	116,129	7,064

GeneExpression learns rules for yeast gene activity given a database of genes, their interactions, and micro-array gene expression data;

BreastCancer processes real-life patient reports towards predicting whether an abnormality may be malignant;

IE-Protein_Extraction processes information extraction from paper abstracts to search proteins;

Susi learns from shopping patterns;

Mesh learns rules for finite-methods mesh design;

Carcinogenesis, Choline, Pyrimidines try to predict chemical properties of compounds and store them as tables, given their chemical composition and major properties;

Thermolysin also manipulates chemical compounds but learns from the 3D-structure of a molecule's conformations.

Fig. 4. Description of the ILP datasets used in the performance comparison of Table 2

more detailed analysis shows that most space is occupied by the hash tables and by internal nodes of the tree, and that relatively little space is occupied by try-retry-trust chains, suggesting that demand-driven indexing is behaving well in practice.

8 Concluding Remarks

Motivated by the needs of applications in the areas of inductive logic programming, program analysis, deductive databases, etc. to access large datasets efficiently, we have described a novel but also simple idea: *indexing Prolog clauses on demand during program execution*. Given the impressive speedups this idea can provide for many LP applications, we are a bit surprised similar techniques have not been explored before. In general, Prolog systems have been reluctant to perform code optimizations during runtime and our feeling is that LP implementation has been left a bit behind. We hold that this should change. Indeed, we see demand-driven indexing as only a first, very successful, step towards effective runtime optimization of logic programs.

As presented, demand-driven indexing is a hybrid technique: index generation occurs during runtime but is partly guided by the compiler, because we want to combine it with compile-time WAM-style indexing. More flexible schemes are of course possible. For example, index generation can be fully dynamic (as in YAP), combined with user declarations, or driven by static analysis to be even more selective or go beyond fixed-order indexing. Last, observe that demand-driven indexing fully respects Prolog semantics. Better performance can be achieved in the context of one solution computations, or in the context of tabling where order of clauses and solutions does not matter and repeated solutions are discarded.

Acknowledgments This work is dedicated to the memory of our friend and colleague Ricardo Lopes. We miss you! V{ıtor Santos Costa was partially supported by CNPq and would like to acknowledge support received while visiting at UW-Madison and the support of the YAP user community. 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.

Acknowledgments Vítor Santos Costa was partially supported by CNPq and would like to acknowledge support received while visiting at UW-Madison and the support of the YAP user community. 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.

References

1. Warren, D.H.D.: An abstract Prolog instruction set. Tech. Note 309, SRI International (1983)
2. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. (2002)
3. Carlsson, M.: On the efficiency of optimising shallow backtracking in compiled Prolog. In Levi, G., Martelli, M., eds.: Proceedings of the Sixth ICLP, MIT Press (June 1989) 3–15
4. Demoen, B., Mariën, A., Callebaut, A.: Indexing in Prolog. In Lusk, E.L., Overbeek, R.A., eds.: Proceedings of NACLP, MIT Press (1989) 1001–1012
5. Wielemaker, J.: SWI-Prolog 5.1: Reference Manual. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands. (1997–2003)
6. Sagonas, K.F., Swift, T., Warren, D.S., Freire, J., Rao, P.: The XSB Programmer's Manual. State University of New York at Stony Brook. (1997)
7. Tronçon, R., Janssens, G., Demoen, B., Vandecasteele, H.: Fast frequent quering with lazy control flow compilation. Theory and Practice of Logic Programming (2007) To appear.
8. Hickey, T., Mudambi, S.: Global compilation of Prolog. JLP **7**(3) (November 1989) 193–230
9. Van Roy, P., Demoen, B., Willems, Y.D.: Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In: TAPSOFT'87, Springer (1987) 111–125
10. Zhou, N.F., Takagi, T., Kazuo, U.: A matching tree oriented abstract machine for Prolog. In Warren, D.H.D., Szeredi, P., eds.: ICLP90, MIT Press (1990) 158–173
11. Dawson, S., Ramakrishnan, C.R., Ramakrishnan, I.V., Sagonas, K., Skiena, S., Swift, T., Warren, D.S.: Unification factoring for the efficient execution of logic programs. In: Conference Record of POPL'95, ACM Press (January 1995) 247–258
12. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient access mechanisms for tabled logic programs. Journal of Logic Programming **38**(1) (January 1999) 31–54

13. Kliger, S., Shapiro, E.: A decision tree compilation algorithm for FCP(|,:,?). In: Proceedings of the Fifth ICSLP, MIT Press (August 1988) 1315–1336
14. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. JLP **26**(1–3) (December 1996) 17–64
15. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Science of Computer Programming **58**(1–2) (2005) 115–140
16. Carlsson, M.: Freeze, indexing, and other implementation issues in the WAM. In Lassez, J.L., ed.: Proceedings of the Fourth ICLP, MIT Press (May 1987) 40–58
17. Srinivasan, A.: The Aleph Manual. (2001)