*TLP* **13** (4–5): Online Supplement, July 2013. © 2013 [VÍTOR SANTOS COSTA and DAVID VAZ]
URL: http://dx.doi.org/10.1017/S1471068413000501

# *BigYAP: Exo-compilation meets UDI*

VÍTOR SANTOS COSTA and DAVID VAZ
*CRACS - DCC/FCUP, University of Porto, Portugal*

*submitted 10 April 2013; revised 23 June 2013; accepted 5 July 2013*

## Abstract

The widespread availability of large data-sets poses both an opportunity and a challenge to logic programming. A first approach is to couple a relational database with logic programming, say, a Prolog system with MySQL. While this approach does pay off in cases where the data cannot reside in main memory, it is known to introduce substantial overheads. Ideally, we would like the Prolog system to deal with large data-sets in an efficient way both in terms of memory and of processing time. Just In Time Indexing (JITI) was mainly motivated by this challenge, and can work quite well in many application.

Exo-compilation, designed to deal with large tables, is a next step that achieves very interesting results, reducing the memory footprint over two thirds. We show that combining exo-compilation with Just In Time Indexing can have significant advantages both in terms of memory usage and in terms of execution time.

An alternative path that is relevant for many applications is User-Defined Indexing (UDI). This allows the use of specialized indexing for specific applications, say the spatial indexing crucial to any spatial system. The UDI sees indexing as pluggable modules, and can naturally be combined with Exo-compilation. We do so by using UDI with exo-data, and incorporating ideas from the UDI into high-performance indexers for specific tasks.

## 1 Introduction

The last few years have seen the convergence of two trends. On the one hand, there has been a steady improvement in hardware, both in terms of processing and in terms of storage. On the other hand, more and more data is stored in computer databases. Most of this data, be it web data, medical records data, spatial data, or text data is essentially read-only. Thus, efficient access to the data is often achieved by loading copies of the data in main memory, either of a single computational unit or in a distributed network. and processing it in-line.

Logic Programming provides a high-level description of data that naturally fits relational databases (but that can also be a target for other types of databases). It further provides a powerful reasoning mechanism to query the data. We claim that these in-memory big databases provide an excellent opportunity for logic programming, and namely, for Prolog systems. Moreover, these opportunities offer a number of interesting implementation challenges.

The first challenge is *memory management*. Prolog sees data as a program. More precisely, most Prolog systems use an emulator and data is stored as a set of abstract-machine instructions. Executing data improves running-time, but has a cost in memory usage. Moreover, often Prolog systems store each fact (or row in a table), as a separate data-structure with its own headers and footers, further increasing the memory overhead. Clause management overheads can be addressed by packing clauses together either at program load-time, as done in XSB (Swift and Warren 2012), or at run-time, as done in YAP (Santos Costa *et al.* 2012). Abstract-machine instruction overhead can be reduced by coalescing different instructions into a single instruction, and by specializing the merged instruction for a fixed set of arguments. In that way, a fact with three arguments can be compiled as a single WAM instruction, reducing the representation overhead to a single opcode (Santos Costa 2007). Ultimately, one can go further and separate data from code, the so-called *exo-compilation* (Demoen *et al.* 2007). This idea was originally used in Mercury (Conway *et al.* 1995), but it has never been widely implemented in Prolog systems. Indeed, to the best of our knowledge, only hProlog (Demoen and Nguyen 2000) actually implemented exo-code. On other systems, such as YAP, coalescing instructions seemed to work well enough at reducing memory overhead.

The second challenge stems from the need to access the database efficiently. Full scans of a table with millions of rows take a long time. Good indexing is necessary but raises several difficulties. Prolog execution often is about searching for terms that match a query term, a process that can be implemented efficiently using hash-tables. One first question is which arguments or combinations of arguments should have associated tables, or indices? Indexing all possible combinations is very expensive, $2^{arity}$. Traditionally, Prolog systems index the first argument only, as suggested for the PLM (Warren 1977) and original WAM (Warren 1983), but this is not effective for databases. One solution is to support multi-argument indexing, but according to a user-given or compile-time order (Demoen *et al.* 1989; Van Roy 1994). A more ambitious solution is to index a combination of arguments only when they are actually needed: this idea is called *just-in-time-indexing* (JITI) and it is used in the YAP and XXX Prolog systems (Santos Costa *et al.* 2007; Santos Costa *et al.* 2012), and more recently in SWI-Prolog (Wielemaker *et al.* 2012).

The JITI allows Prolog systems to process effectively databases with millions of facts, but at a cost of extra memory usage. More precisely, we pay two costs in memory: the actual storage required for the extra hash-tables, and the fragmentation costs incurred when creating and later releasing the temporary data-structures necessary to store the indices. In our experience, the first cost is often higher than the memory needed to actually store the data, and the second cost is not negligible for large databases.

A second limitation of Prolog indexing is that it it hard to efficiently answer even simple aggregated queries over a table. As an example, imagine that we have a table of patient prescriptions and we wanted to know when patient `Michael` started taking `warfarin`. The query could be written as:

```
?- min(Date, hasdrug(michael, Date, warfarin) ).
```

A Prolog program needs either to run a `setof/3` and extract the head of the list, or it can implement the query as:

```
?- hasdrug(michael, Date, warfarin),
   \+ (hasdrug(michael, _D, warfarin), _D < Date).
```

A more elegant solution to this specific query is mode-directed tabling (Guo and Gupta 2008), but this technique unfortunately requires creating a new intermediate table, which may be quite expensive space-wise, and does not immediately address more complex queries such as:

```
?- hasdrug(michael, Date0, warfarin),
   diagnoses(michael, DateF, bleeding),
   Date0 < DateF.
```

One alternative idea in this case is user-defined indexing (UDI) (Vaz *et al.* 2009). The principle is to allow the user to include low-level code specialized for say, indexing in the presence of numeric range operators. The operations of interest are then plugged-in as extensions to the Prolog engine.

Next, we argue that in order to apply Prolog to data-sets with hundreds of millions of elements and still have a reasonable memory footprint, we should reconsider exo-compilation and user-defined indexing. The goal should not be so much to be efficient, but to have a compact representation while avoiding fragmentation and large temporary data-structures. Moreover, as many of the queries of interest need more than Herbrand semantics, we need to be able to fit user-defined indexing with this approach.

We next present a novel, exo-compilation based representation scheme for facts and for indices, that allows integration with user-defined indexing. We show substantial savings space-wise, and good performance time-wise. Most important, we have been able to process effectively very large datasets with up to 300 million facts in mid-range machines (up to 24GB).

## 2 Exo-Code

Exo-code separates data from byte-code by simply storing data as an array (Demoen *et al.* 2007). For simplicity sake but also because we are interested in databases, we start by assuming the data is in first normal form (Codd 1970), that is, we only include Prolog atoms and numbers (integers to start with). Given this, there are two decisions:

- Should we store tagged or un-tagged objects?
- Should we use a vertical representation, i.e., separate facts, a horizontal representation, i.e., separate arguments, an hybrid representation or a single block of data?

These decisions are not as crucial as they would be in the WAM: one important advantage of exo-compilation is that we encapsulate the data. Thus, we can feel free to experiment with different representations. Our system currently implements a single block of tagged terms, where each entry corresponds to an argument of a fact. We thus implement the following relation:

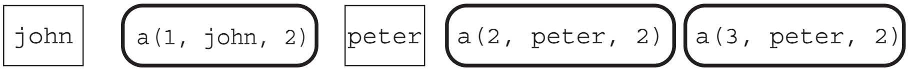


Fig. 1. Space fragmentation when compressing clauses. When YAP compiles a file with three facts, the parser first adds the constant `john` to the symbol table, then YAP compiles `a(1,john,2)`, then the parser adds `peter`, and then YAP compiles the second and third clauses. When YAP tries to release space by packing the three clauses, the first clause will become a hole.

$$d(A_i^1, A_i^2, \ldots, A_i^{arity}) \Leftrightarrow (P[i-1][0], P[i-1][1], \ldots, P[i-1][arity-1])$$

where $P$ points to the base of the array and we follow `C` notation for indices. The actual implementation reuses the WAM abstract machine register $S$ as a pointer to the *current fact*, that is, when entering exo-code clause $i+1$ we always execute first $S \leftarrow P + i * arity$, thus ensuring that $S$ points to the beginning of fact $i+1$.

### 2.1 Instructions

Accessing the data is performed by a variation on the WAM's `get_atom` instruction. We remind the reader that in the WAM the instruction

`get_atom` $X_i$, `C`

will unify the contents of argument register $X_i$ and the constant `C`. Remember that in our exo-code, the code is a sequence of terms and the WAM abstract machine register $S$ initially points to the first argument of the clause. Thus, the corresponding exo-code instruction is

`get_exo_atom` $i$

that unifies the contents of argument register $X_i$ and the constant at position $S[i-1]$.

### 2.2 Database loading

An important consideration in the design of the exo-code is loading. Currently, we use a two step process:

1. run through the database and verify the database size (number of clauses) and the type of each argument.
2. Compute size and allocate space for predicates.
3. run through the database and copy each fact to the allocated space sequentially.

This model avoids fragmentation, as atoms are allocated in the symbol table by the first step, thus before we allocate any space for the predicate. In contrast, by default YAP reads clauses one by one, mixing allocation of atoms in the symbol table with allocation of compiled code for individual clauses. As both are allocated from the same pool, it is impossible to recover all space when consolidating the clauses together, as described in Figure 1.

## 3 Indexing exo-code

Indexing is key to the implementation of exo-code. In fact, our design is such that, arguably, all code is indexing code. We start by describing the general flow of the execution when we enter an exo-code procedure.

Every exo-procedure starts with the instruction `exo`, that:

1. builds up a map with the instantiation state on every argument;
2. does a linear scan on a collection of indices:
   (a) If one index matches the map, it executes the index.
   (b) If no index matches, it generates a new one in JITI style.

The next step of exo-execution corresponds to executing an index, and consists of the following steps:

1. Construct a key using the map of instantiated arguments.
2. Look-up the key in a hash-table, and:
   (a) if not there, execute the `fail` instruction;
   (b) If it matches several clauses, make `S` point to the first clause, and jump to the non-determinate code entry;
   (c) If it matches a single clause, make `S` point to the first code, and jump to the determinate code entry.

The global structure of the exo-code is shown in Figure 2. Each index block includes a map of instantiated arguments, a hash-table of keys and matching clauses, and a block of emulator code with two entry points: the determinate and the non-determinate entry. Next, we discuss the hash-table and the block of emulator instructions in more detail.

### *3.1 The hash table*

The hash-table was designed to be compact and to not require allocation of temporary memory to be built. Speed is relevant but it was not considered the major goal. We implemented it as two arrays, both containing indices of clauses:

- the *keys* array points to a clause and is used for matching;
- the *chains* array contains pointers to a chain of clauses that match the clause.

In some more detail, if $i(G)$ is an index block that has the same instantiation map as a query $G$, and assuming there are no collisions, we have that:

- $h \leftarrow hash(G)$, where $h$ is the value of the $i(G)$ hash function for either a term $G$ or a clause $C$;
- $k \leftarrow i(G).keys[h]$ refers to a clause $C$ such that $hash(C) = h$;
- $i(G).chains[k]$ refers to the second clause $C'$ such that $hash(C') = h$;
- $i(G).chains[i(G).chains[k]]$ refers to the third clause $C''$ such that $hash(C'') = h$;

A chain terminates when we find the $OUT\_OF\_DOMAIN$ value. Notice that if indexing on the goal makes the goal always determinate, a all values in $i(G).chains$ are $OUT\_OF\_DOMAIN$, and thus the *chains* data-structure can be discarded.

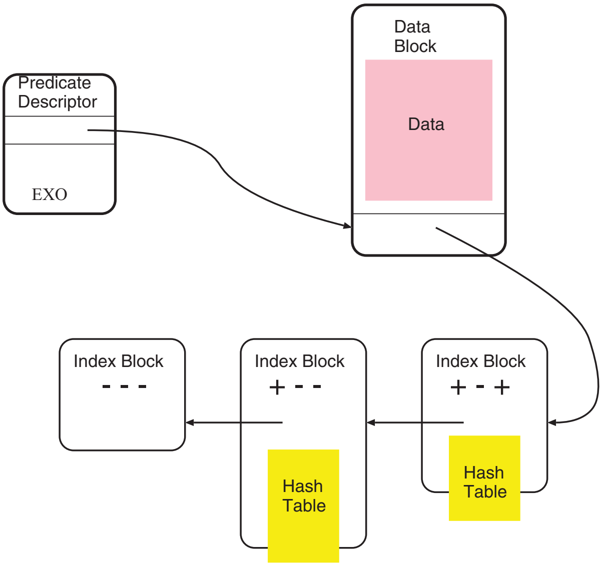


Fig. 2. (Colour online) Exo-code: the predicate descriptor includes the first opcode to be executed. in this case `exo`. It also points to the data block that includes the code, stored as an array. The header of the code block points to a linked list of indexing blocks, one per each mode of the predicate's argument. Each indexing block may (or may not) include a hash table and always includes byte-code.

### 3.2 The hash function

Experience showed that the algorithm can be quite sensitive to the quality of the hash function, especially for very large tables. We settled on using the FNV non-cryptographic hash function (Henke *et al.* 2007).

```
$hash \leftarrow \texttt{FNV\_basis};$
for $octet\_of\_data$ do
    $hash \leftarrow hash * \texttt{FNV\_prime};$
    $hash \leftarrow hash \oplus octet\_of\_data;$
end for
return hash
```

Notice that we need to walk all the octets in the representation of all bound terms (the key). Table 1 justifies our decision by comparing this function against other widely used hashing functions. We present the times necessary to construct hashes on three tables from an Observational Medical Outcomes Partnership (OMOP) simulated medical record database (Murray *et al.* 2012). The times are dominated by the lookup operation and by collision handling, so they give a good estimate on how effective the algorithms are.

Note that these hash functions are much more expensive to compute than the module hash used in the WAM, but they are useful at reducing collisions on larger tables.

Hash table size defaults to three times the size of the table. If the number of collisions is very low (possibly because there are few different keys), we contract the

Table 1. *Hash algorithm performance*

| Dataset | Entries | RS | DJB2 | FVN-1A | Murmur3 |
|---|---|---|---|---|---|
| conditions | 175,496,758 | 16,824ms | 17,055ms | 16,635ms | 19,823ms |
| hasdrug | 118,541,933 | 12,770ms | 12,387ms | 11,744ms | 14,238ms |
| dates | 266,297 | 18ms | 25ms | 18ms | 23ms |

size of the table and rebuild. If the number of a sequence of collisions exceeds a threshold, by default 32, we stop, expand the table size and rebuild from scratch.

### *3.3 The index code*

Every index block has associated a code sequence. For a predicate of arity 4, assuming that arguments 1 and 4 were bound, the code would be:

```
L_non_det_enter:
       try_me_exo
       retry_me_exo
L_det_enter:
       get_exo_atom 2
       get_exo_atom 3
       proceed
```

The two labels correspond to the non-determinate and determinate entry points, respectively. Determinate execution enters the code on the first `get_exo_atom`, unifies, and proceeds.

*Choice-point instructions* The `try_me_exo` instruction creates a choice-point whose alternative points to the `retry_me_exo` instruction. It works very much as the WAM's `try_me` instruction, except that if we currently match clause $C$:

1. it pushes the value of $i(G).chains[k]$, where $k$ is the index of the current clause, to the new choice-point.
2. after executing, control jumps to `L_det_enter`, that is, it skips the `retry_me_exo` instruction.

The `retry_me_exo` instruction combines the WAM's `retry_me` and `trust_me` instructions:

1. it pulls $n \leftarrow i(G).chains[k]$ from the stack, and it updates `S` to point at the clause at offset $n$;
2. it computes $v \leftarrow i(G).chains[n]$;
3. if $v = OUT\_OF\_DOMAIN$ the chain terminates, so execute a `trust_me`.
4. otherwise, execute a `retry_me`, but do not update the choice-point's alternative.

### 3.4 The default block

The indexing code always has a default indexing block, that corresponds to the case where no arguments are instantiated. The default block has an empty hash-table, and the code for our example would be:

```
L_non_det_enter:
       try_exo      S0
       retry_exo    Sf
       get_exo_atom 1
       get_exo_atom 2
       get_exo_atom 3
       get_exo_atom 4
       proceed
```

The instructions are similar to the ones discussed above. Two differences:

1. try_exo sets $S$ to $S_0$, which points to the first clause;
2. retry_exo executes trust when $S == S_f$.

*Optimizations* A number of optimizations are possible. We briefly mention three:

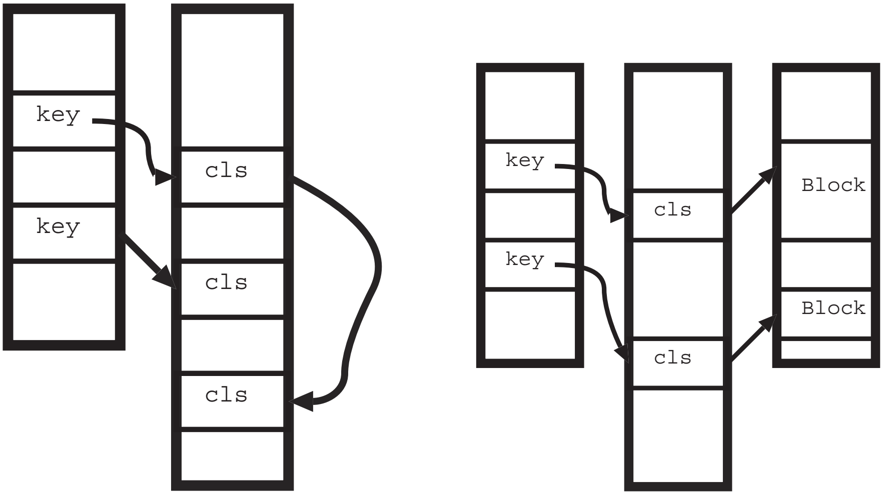
- If an index block is determinate, a more instantiated block cannot be a better indexer; thus, if such blocks exist they can be discarded.
- We use 32-bits offsets to represent addresses. Offsets complicate and slow-down code, but they halve memory usage.
- As proposed in the original exo-code paper, if the value of an argument does not matter, we can simply skip code for that argument (Demoen *et al.* 2007). We may also want to declare arguments to be output, and never build indices on them.

## 4 The UDI and exo-code

The UDI allows users to integrate their own indexing engines within YAP. This is important when we want to obtain indexing that depends on a certain interpretation of terms, say, if we want to process number ranges in a special way. The UDI can benefit from exo-coding, as exo-coding isolates the data from the Prolog engine. Next, we describe how the UDI integrates with exo-compilation.

To illustrate our approach, we refer back to the prescription query in page 800. Consider we want to extend exo-code to support range searches over the second argument, given that the first and third arguments are instantiated. We need to address three questions:

- *Where?* How do we know that we want to give a special semantics to the second argument? The UDI solution is to have declarations that specify the type for this argument, and to use attributed variables to constrain execution (Vaz *et al.* 2009).



(a) Default hash table for exo-indexing

(b) An UDI hash-table for exo-indexing, using integer ordering: chains of integers are moved into blocks

Fig. 3. Reshaping an exo-hash table by using the UDI.

- *When?* In contrast with the initial UDI, we now implement UDI indexing at run-time. This complicates the design of the UDI somewhat, but allows for integration between the UDI and the JITI.
- *How?* We call the UDI code after the exo-index is constructed, and we assume full access to the data-structures (that are mostly independent of the WAM's).

The idea is presented in Figure 3. The left-side shows the exo-hash table as described before: the key array indirectly indexes a second array, the chains array, that points to a sequence of entries. The sequence is represented as a linked-list, and follows Prolog ordering. If we want to find the largest and the smallest integer it will takes time $O(n)$, where $n$ is the chain size, bounded by the number of clauses.

In contrast with the previous approach for UDI compilation, in exo compilation space and time are paramount, so we assume the UDI code can directly access the exo hash table data-structure. In this case, we assume that the data is read-only so there is no point in building an incremental and updatable data-structure, such as a B-Tree. Instead, in order to achieve logarithmic search, we organize clauses as sorted blocks, one per different value of the key. The approach is shown in Figure 3(b). The keys table did not change, but the chains table was made to point to a new table, that contains blocks with sequences of clauses. Each block contains:

- a header giving the size of the block;
- plus a sequence of pointers to clauses, ordered by the numeric argument.

This very straightforward approach guarantees constant-time access to the minimal and maximal elements, and logarithmic time for other elements. The space overhead is $O(C)$, where $C$ is the number of clauses.

The interface occurs at three-points:

*Compilation*: the exo-compiler calls the UDI code after constructing a new hash-table (the interface may also allow calling before). In the example, the UDI code collects the entries matching a key, sorts them, and stores them in the new sequence table. Last, the chains table is made to point to the sequence table.

*Procedure Entry*: the `exo` instruction detects that the matching indexing block is UDI, so it calls the UDI code after performing look-up in the hash table. The UDI code follows the chain pointer, executes based on a constraint on the numeric argument. Examples of constraints include `min`, implemented by take the first element; `max`, implemented by reading the number of elements in the chain and take the last; and `gt X` implemented by searching for `X` and set-up a choice-point starting at clause $A_i >$ `X` and ending at last element

Notice that the ordering of returned clauses does not follow traditional Prolog order. The search for `X` can be binary or linear, depending on the chain-size. Last, if `X` is unconstrained one should enumerate all clauses.

*Instruction retry*: in this case the update of the choice point and the detection of the last instruction are performed by the UDI code. This requires storing the limits in the choice-point. and calling the UDI code before the main body of `retry` or `trust` starts.

The proposed UDI is motivated by medical domains, where often we want to find the first time a patient has been prescribed a certain drug, and whether he took the drug after having reported some pre-specified condition. Other approaches would be needed for spatial data, for example. Last, notice that the approach requires sorting each bucket of clauses. A step-wise algorithm, like a B-Tree, may be a better approach if we have few keys and large buckets.

## 5 Evaluation

We first evaluate exo-code on a gene expression and on a web mining application. These experiments were performed on a Intel Xeon E5620 Linux Red Hat v6.3 machine, running in 64 bit mode, with 24GB memory. Exo-code is using 64-bit addresses throughout for comparison fairness. As our first dataset, we use the human gene ontology (Ashburner *et al.* 2000). The application takes advantage of the YAP-R interface (Angelopoulos *et al.* 2013) to consult differentially expressed genes from gene expression data, and then uses the gene ontology database to detect the main functions of those genes. YAP loads the gene ontology as three relations: a set of concepts, a graph connecting those concepts, and a table instantiating the concepts in the context of the human genome. In the second dataset we have a tokenized extract of an Italian ecology blog. Again we use `R`, but this time to construct a cloud of words with the most popular stem words in the ecology blog. Most work is Prolog counting how many times the different words occur in the blog. We show two versions: the first 5 million tokens, and the full blog with 38M tokens.

Table 2 shows the loading time using exo and default compilation. The results show a speedup in loading time (exo-compilation does not actually need to compile), and a compression factor between two and 1.5. Notice that the byte-code already

Table 2. *Exo-code and Emulated code loading times in msec and Sizes in KB*

| | | Exo | | Compact Byte-code | |
|---|---|---|---|---|---|
| File | Facts | Time | Size | Time | Size |
| Annotations | 425,093 | 1,978 | 8,386 | 5,254 | 16,772 |
| Terms | 39,222 | 300 | 2,145 | 560 | 3,677 |
| Graph | 992,790 | 5,062 | 46,537 | 12,366 | 69,805 |
| Tokens5M | 5,000,000 | 29,719 | 195,312 | 39,133 | 312,500 |
| Tokens | 38,331,459 | 199,088 | 1,301,989 | 253,962 | 2,083,183 |

Table 3. *Exo-code and JITI byte-code index size in KB*

| File | Exo<br>Index Size (KB) | Byte-code<br>Index Size (KB) |
|---|---|---|
| Annotations | 8,386 | 17,286 |
| Terms | 4,598 | 4,014 |
| Graph | 46,537 | 161,034 |
| Tokens5M | 351,563 | 588,551 |
| Tokens | 2,343,582 | 3,831,397 |

uses coalesced instructions and compacted clauses (Santos Costa 2007). We do not consider the fragmentation overhead at this point.

Table 3 compares indexing size. Notice that exo-indexing actually takes more space in the `Terms` predicate. This is because this table needs indexing on a single argument. The WAM in this case builds a very compact table and YAP does not need to generate choice-point manipulation instructions. Exo-indexing does best when one has to do complex accesses and one has to backtrack through the database. This occurs in the `Graph` table, where exo-code is four times more compact.

Table 4 compares query run times and memory usage by YAP, as measured by the Operating System. On the one hand, exo-code has a higher execution overhead, and thus should be slower. On the other hand, the hash function used in the exo-code does a much better job on very large tables than the standard WAM radix hash. The results confirm this: the “annotator” benchmark is slower when running with exo-code, but on the larger data-sets exo-code is actually faster.

Table 4. *Query Cputime in sec. and memory usage in number of used pages, as reported by the Operating System*

| | Exo | | Compact Byte-code | |
|---|---|---|---|---|
| | Run Time | Max Pages | Run Time | Max Pages |
| Annotator | 52s | 1,965,808 | 35s | 2,889,056 |
| Tokens5M | 9s | 1,941,904 | 17s | 7,193,392 |
| Tokens | 43s | 11,763,312 | 63s | 43,056,080 |

Table 5. *Data Mining in the OMOP database: we compare byte-code, exo and exo plus UDI. Load Time is the time to load the database, Run Time is cputime taken to run the query, Total Time is wall-time reported for full execution, and Memory is total memory usage in MBytes as reported by the Operating System*

| | Exo | Exo + UDI | Byte-code |
|---|---|---|---|
| Load Time (sec) | 690 | 692 | 2,200 |
| Run Time (sec) | 6,769 | 5,108 | 5,469 |
| Total Time (sec) | 7,459 | 5,800 | 7,669 |
| Memory (MB) | 21,690 | 23,388 | 60,632 |

We also show the difference in number of resident pages, as it gives a good indication of total memory usage (there was no swapping). Note that exo-compilation does better than what we get just when considering the actual code, as the YAP JITI uses very dynamic data structures that cause substantial fragmentation.

As a final result, we evaluate performance on a machine-learning task. The goal is to mine health-record data for adverse drug events (Page *et al.* 2012). We use Inductive Logic Programming (ILP) on simulated electronic health records data (Murray *et al.* 2012). We compare run-times on a i7-3930K with 64GB of ram, running Ubuntu in 64-bits mode. Altogether, the data has approximately 300 million tuples: 175M tuples in a table of prescriptions, 118M tuples on a table of diagnosis, and 10M tuples on a table of persons. Execution time is dominated by 1100 queries that are performed on 8M examples. We use the UDI to sort by the dates at which patients were diagnosed or had prescriptions. The main queries search for the first time a patient developed a certain symptom given that he had been exposed to a certain drug. Results are shown in Table 5.

As expected, exo-code load times and memory usage do very well in this last test: memory usage drops to a third, making it possible to use large datasets in mid-range machines. The run-time performance is actually faster using YAP byte-code than with exo-code. We believe this is because the default JITI constructs a hash-table per instantiated argument in each usage mode. Instead, exo-compilation builds a single hash table. Even so, performance is close.

The run-time improves with UDI indexing, as a substantial amount of time is spent in range queries, such as finding the first time a drug or condition was reported. Although we most often do not have that many different reports per patient and condition or drug, UDI indexing largely improves the run-time with only a small overhead in memory, in this case 10 %. The use of UDI actually outweighs the exo-code overhead when compared with traditional compilation. Last, we again notice that these results were obtained using 64-bit addresses in the hash-tables. Using 32-bit offsets reduces total memory usage in this application to a maximum of 15 and 16GB.

## 6 Conclusions and future work

We introduce an implementation of exo-compilation for Prolog. In contrast with the initial proposal (Demoen *et al.* 2007), the fundamental goal is not so much to obtain a compact representation of the database but to obtain a compact representation of the indices, as they dominate overall performance, while striving at supporting scalable access to data in Prolog. This requires compilation algorithms to be ideally $O(n)$.

The second contribution of the paper is the integration of the exo and UDI work. This is motivated by practical applications of Prolog: we need efficient search in ordered values. On the other hand, we do not want to encumber generic Prolog with the machinery required for all types of different indexing. The UDI offers an excellent solution towards this problem and naturally fits with the exo-compilation ideas.

Our results so far have been excellent. The overhead of the exo instructions and the hash function seems to be offset by the gains in quality from using a better hash key, even for relatively small data-sets. Most importantly, we have been able to process very large data-sets, with hundreds of millions of facts, in mid-sized machines (24GB of memory with current hardware). This opens the door to exciting new results in using Prolog.

The current implementation has some important limitations. First, we do not support floating point numbers and ground complex terms. This can be easily addressed by computing how much space we need in the first loading step, but complicates hashing. Second, compilation is noticeably slow for large data-sets. Third, as described in Demoen *et al.* (2007), one important advantage of exo-compilation is the ability to optimize for different types of queries, say, one can generate code specialized for first-solution queries. We would like to incorporate these ideas in our system. It is interesting to reuse our results on hash tables in the context of the standard WAM. Last, using these large datasets raises a number of interesting questions: how to best run the queries? Where to insert constraints and where to reorder goals? Should we consider combining with bottom-up execution? And how will reuse and tabling work in these very large sets (Zhou and Have 2012)? We believe these are important research questions and worth of future investigation.

## Acknowledgments

We would like to acknowledge the referees for their very valuable comments. This work is partially financed by the ERDF European Regional Development Fund through the COMPETE Program (operational program for competitiveness) and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects LEAP PTDC/EIA-CCO/112158/2009 and ADE PTDC/EIA-EIA/121686/2010.

## References

Angelopoulos, N., Santos Costa, V., ao Azevedo, J., Wielemaker, J., Camacho, R. and Wessels, L. 2013. Integrative functional statistics in logic programming. In *Proc. of Practical Aspects of Declarative Languages*, LNCS, vol. 7752, Rome, Italy. Accepted.

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., Harris, M. A., Hill, D. P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J. C., Richardson, J. E., Ringwald, M., Rubin, G. M. and Sherlock, G. 2000. Gene ontology: Tool for the unification of biology. The Gene Ontology Consortium. *Nature genetics 25,* 1 (May), 25–29.

Codd, E. F. 1970. A relational model for large shared data banks. *Communications of the ACM 13,* 6, 377–387.

Conway, T., Henderson, F. and Somogyi, Z. 1995. Code Generation for Mercury. In *Proceedings of the International Symposium on Logic Programming*, J. Lloyd, Ed. MIT Press, Cambridge, 242–256.

Demoen, B., Mariën, A. and Callebaut, A. 1989. Indexing in Prolog. In *Proceedings of the North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek, Eds. Cleveland, Ohio, USA, 1001–1012.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *LNAI 1861, Proceedings Computational Logic - CL 2000*. Springer-Verlag, 1240–1254.

Demoen, B., Nguyen, P.-L., Santos Costa, V. and Somogyi, Z. 2007. Dealing with large predicates: Exo-compilation in the WAM and in Mercury. In *Proceedings of the Seventh Colloquium on the Implementation of Constraint and Logic Programming (CICLOPS 2007)*, S. Abreu and V. Santos Costa, Eds. Porto, Portugal, 117–131.

Guo, H.-F. and Gupta, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience 38,* 1, 75–94.

Henke, C., Schmoll, C. and Zseby, T. 2007. *Empirical Evaluation of Hash Functions for Multipoint Measurements* . Tech. rep., Fraunhofer Institute FOKUS. 11.

Murray, R. E., Ryan, P. B. and Reisinger, S. J. 2012. Design and validation of a data simulation model for longitudinal healthcare data. In *Proc. of the ANIA Annual Symposium*, Vol. 1176–1185, Washington DC, USA.

Page, D., Santos Costa, V., Natarajan, S., Peissig, P., Barnard, A. and Caldwell, M. 2012. Identifying adverse drug events from multi-relational healthcare data. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence, AAAI-12*, J. Hoffmann and B. Selman, Eds.

Santos Costa, V. 2007. Prolog performance on larger datasets. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14–15, 2007.*, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 4354, Springer, 185–199.

Santos Costa, V., Damas, L. and Rocha, R. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12,* Special Issue 1-2, 5–34.

Santos Costa, V., Sagonas, K. and Lopes, R. 2007. Demand-driven indexing of Prolog clauses. In *Proceedings of the 23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 305–409.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *TPLP 12,* 1-2, 157–187.

Van Roy, P. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19/20.*

Vaz, D., Santos Costa, V. and Ferreira, M. 2009. User defined indexing. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14–17, 2009. Proceedings*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649, Springer, 372–386.

Warren, D. H. D. 1977. *Implementing Prolog - Compiling Predicate Logic Programs*. Tech. Rep. 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

Warren, D. H. D. 1983. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI International.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *TPLP 12,* 1-2, 67–96.

Zhou, N.-F. and Have, C. T. 2012. Efficient tabling of structured data with enhanced hash-consing. *TPLP 12,* 4-5, 547–563.