User Defined Indexing

David Vaz[1], Vítor Santos Costa[2], and Michel Ferreira[3]

[1] LIACC - DCC/FCUP, University of Porto, Portugal
[2] CRACS - DCC/FCUP, University of Porto, Portugal
[3] Instituto de Telecomunicações - DCC/FCUP, University of Porto, Portugal

Abstract. Logic programming provides an ideal framework for tackling complex data, such as the multi-dimensional vector-based data used to represent spatial databases. Unfortunately, the usefulness of logic programming systems if often hampered by the fact that most of these systems have to rely on a single unification-based mechanism as the only way to search in the database. While unification can usually take effective advantage of hash-based indexing, it is often the case that queries over more complex and structured data, such as the vectorial terms stored in spatial databases, cannot.
We propose a new extension to Prolog indexing: User Defined Indexing (UDI). In this mechanism, the programmer may add extra information to Prolog indices so that only interesting fragments of the database will be selected. UDI provides a general extension of indexing, and can be used for both instantiated and constrained variables. As a test case, we demonstrate how UDI can be combined with a constraint system to provide an elegant and efficient mechanism to generate and execute range queries and spatial queries. Experimental evaluation shows that this mechanism can achieve orders of magnitude speedups on non-trivial datasets.

1 Introduction

Logic programming provides an ideal framework for tackling complex data, using a single and universal representation of pieces of such data as *logic terms*. A logic term can represent things such as an unbound variable, a constant or an integer, or more complex and structured entities such an interval over reals or a vectorial polygon. The universality of this representation is a key feature for the declarative flavor of logic programs. In particular, it is the basis for the generic handling of data through the single mechanism of *unification*. While this unification and the term-based representation of the world are fundamental flagships of the logic programming paradigm, they can also entangle the usefulness and effectiveness of logic programming for data-intensive applications. Performing search through unification can be terribly ineffective because of the match-based process associated with it. Indexing tries to overcome this inefficiency, and has been coupled to the earliest Prolog implementations [**?**], in order to narrow the number of clauses to try. Indexing is however based on the representation of data, which is universally term-based, and is very much designed in Prolog systems around the

lexical or syntactic form of such terms, rather than its *semantic*. A conspicuous example of that is the position-based indexing proposed in the WAM [**?**]. The divergence from semantic is contrary to the Prolog's focus on the *what*, instead of the *how*, but is rooted on the fact that indexing is an implementation issue, rather than a programmer's concern.

Lexical indexing is usually well performed through hashing techniques, which cope naturally with the match-based mechanism of unification. Efficient execution of Prolog queries requires the programmer to be aware of this close relationship between hash-based indexing and unification. The following two queries:

```
?- p(A), q(B), A=B.
?- p(A), q(A).
```

are semantically equivalent, but have very different performances, as the indexing over goal `q/1` is only effective when unification is pulled to the argument of the call. This is an important difference between Prolog and relational databases querying, where the execution of the later is preceded by an optimizer that is able to look globally to the query and define an execution plan that maximizes efficiency. Systems incorporating global analysis in the compilation of Prolog, such as Ciao [**?**], are also able to perform some goal reordering that would use the constraint over variable `B` on `q/1` goal [**?**]. However, hash-based indexing of Prolog predicates is not able to take advantage of constraints over arguments that are not based on unification. The following query:

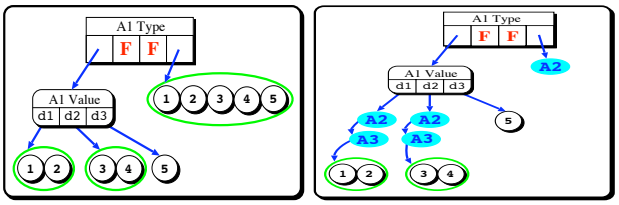
```
?- p(A), q(B), A>B.
```

has no possible rewriting in Prolog that would make it efficient, even if we could pull the constraint to the call of goal `q/1`. This is due to the fact that the lexical, hash-based, indexing of predicate `q/1` is not able to take advantage of the *semantic* constraint of order, either defined in a term-based representation domain of numbers or strings, between variables `A` and `B`. In the same way, if the argument of predicates `p/1` and `q/1` are terms representing a vectorial polygon, then a query such as:

```
?- p(A), q(B), overlaps(A,B).
```

is also unable to improve efficiency based on a hash-based index over the argument of predicate `q/1`. Efficient indexing over spatial terms is particularly important, not only because of the usual mammoth size of such predicates, but also because of the computationally expensive execution of spatial operators.

In this paper we propose and implement a *semantic-oriented* indexing of Prolog predicates, where the programmer is able to define the indexing mechanism based on *what* the terms in the arguments of a predicate are meant to represent. This User Defined Indexing (UDI) allows users to provide an indexing function that selects a subset of the clauses of a predicate, given a set of constrained variables or bound Prolog terms. This function implements the type of indexing the user deems appropriate for the predicate, from specialized hash-based functions to multi-dimensional indexing suited for spatial terms. We propose a constraint



(a) (b)

Fig. 1. WAM and JITI Code for `has_property/3`

based syntax over the logic variables that can affect the efficiency of indexing, retaining the declarative style of Prolog querying.

The remainder of this paper is organized as follows: Section 2 presents the current state-of-the-art of Prolog indexing; Section 3 explains the indexing mechanisms used to efficiently access data structured in ranges and multi-dimensional objects; Section 4 presents our proposal of User Defined Indexing and addresses the engine modifications to implement it in Yap; Section 5 gives some examples of user defined indexers and Section 6 performs a complete evaluation over very large datasets; Section 7 concludes the paper.

2 Indexing Prolog Programs

Indexing is a key feature in Prolog implementations and has been supported since Warren's DEC-10 Prolog system [**?**]. Both DEC-10 Prolog and Warren's WAM [**?**] implement indexing on the first argument, and this has become standard in Prolog systems. Figure 1(a) shows the WAM code for a small database shown next:

```
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).
```

Figure 1(a) shows WAM indexing code as a tree, with *switch* nodes, that implement clause selection, and *clause chain* nodes, that either jump to clauses or support backtracking through a set of clauses. There are two different switch nodes in the WAM. The first, *switch_on_type*, selects according whether the first argument, A_1, is unbound, constant, pair or structure. The second type, say *switch_on_constant* selects clauses that match a *value*. Given a large enough number of different values, the WAM will implement this operation as a lookup in the hash table. Looking up an hash table takes constant time in average, hence indexing can in the best case improve query execution from linear in the number of clauses to constant-time, with only a small overhead.

A natural step from the WAM is to index on multiple arguments. Systems such as Prolog by BIM [**?**] and SWI-Prolog [**?**] do so in an user-specified fashion. This approach requires prior knowledge on which modes of usage are going to be used in the program, though. Just In Time Indexing (JITI) [**?**] addresses this problem by generating indexing code only when needed. For example:

```
?- has_property(d1, _, _).
```

would result on the JITI code shown in Figure 1(b): notice that the code is very similar to the original WAM code, but it includes "wait nodes", shown as filled ovals. Imagine next the queries:

```
?- has_property(d1, salmonella, _).
?- has_property(_, salmonella_, _).
```

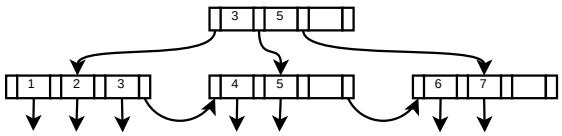
The JITI has the ability to expand the tree in Figure 1(b) with hashes on the first *two* arguments first, and then later building an alternative index that hashes on the *second* argument. This is implemented by generating new trees rooted at the *wait* nodes. The JITI seems to address well the cases where one wants to lookup a value in a database. This is an important application of Prolog, but not the only one.

3 Indexing Ranges and Spatial Data in RDMs

Indices based on exact matching of values are useful when searching for the value that matches some constraints, and they are usually implemented with hash tables. On the other hand, quite often users are interested in different styles of queries. One typical example is finding all values that are larger than some X; another are "ranges queries", that is, finding all values that are between two predefined boundaries. Such queries can be naturally written as logic programs, but are difficult to implement effectively with hash tables. More recently, there has been wide interest in storing and manipulating geographical data. These problems have motivated a large body of research in the Relational Database Management Systems (RDMs) community, which has proposed a number of indexing structures, such as B+-Trees [**?**] and R-Trees [**?**]. We briefly review these data structures next.

A B+-Tree is a self-balanced tree based on a B-Tree: it is often used in RDMs because it allows for logarithmic time selections, insertions and deletions. In B+-Trees data is stored in leaf nodes and only keys are stored in inner nodes (index nodes). Leaves in B+-Trees are linked to one another in a linked list. The main advantage of B+-Trees versus Hash Tables is that data is kept in order, making range queries (inequalities) possible and efficient. Figure 2(a) shows an example of a B+-Tree. The inner node separates the tree into three ranges $X < 3$, $3 < X \leq 5$ and $X > 5$. Given a key, search executes by going down from root of the tree and taking the branch covering the key, as shown in Figure 2(b).

B+-Trees are useful when addressing ordered values, but are not sufficient to index complex multi-dimensional data, such as spatial data. In this case an



(a) Example

```
search (node, key)
  if (node is leaf)
    find key in node
  else
    find branch in node
    search (branch, key)
```

(b) Search

Fig. 2. B+Tree.

important operation is to compute whether two geographical objects *intersect*. Quite often, an object's Minimum Bounding Rectangle (MBR) or Bounding Box, is used towards this goal. Namely, MBRs are used to implement the *R-Tree*, a major datastructure used in databases such as PostGIS [**?**] to quickly find all objects in a given area, e.g., "find all lakes in Switzerland".

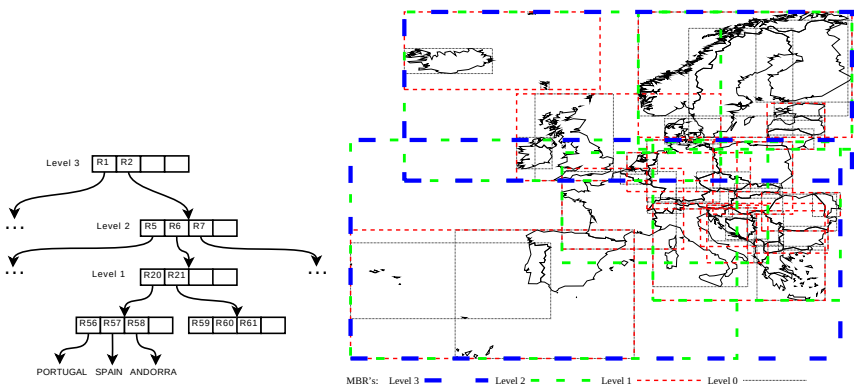
R-Trees are inspired on B+-Trees. The key idea is that R-Trees use MBRs to index data. Each leaf nodes stores an object, and is keyed by the object's MBR. Inner nodes are keyed by an MBR that is the union of all MBRs below. Notice, that in contrast to B+-Trees, keys cannot be sorted as there is no order. On the other hand, searches in R-Tree are similar to searches in B+-Tree, except that *several MBRs in the same node may overlap with the searched MBR*. As a result we can have several valid branches at each node, and it is not possible to guarantee good worst-case performance. Indeed, in the worst case scenario, a query MBR can contain the whole dataset; in this case the complete indexing structure will need to be searched. Nevertheless, on most datasets the tree will maintain a shape that allows the search algorithm to quickly discard irrelevant regions.

Figure 3 shows an example R-Tree designed to store the boundaries of European countries. Figure 3(a) details part of the index structure, and Figure 3(b) graphically depicts the actual boundaries and MBRs that define the R-Tree. Notice that although European countries do not overlap, their MBRs do. The tree has three levels. The root node (Level 3) contains two MBRs, R_1 and R_2, shown as the wider lines. Notice that there is some overlap, as we cannot find a disjoint balanced union of MBRs that covers the whole of Europe. The overlap is even more evident on Level 2, Also observe that whereas Iceland, Greece and Portugal belong to a single box *for each level*, the central Alps region in Europe is covered by a large number of overlapping MBRs *at all levels*.

4 User Defined Indexing

In this work we are motivated by our desire to query complex databases declaratively and *efficiently*. Say, in Prolog in order to find all cities with more than 5 million people, we would state the query:

```
?- city(X,Pop), Pop > 5000000.
```



(a) R-Tree Structure (b) MBR Containment

Fig. 3. R-Tree of European Countries.

Execution of this query visits every city, returning only the ones with population over 5 million. This is arguably the most inefficient execution one can follow, especially if only a few cities have population above 5 million. In order to constrain the search we need to know that `Pop` must be over 5 million people first. This is not possible in Prolog, but it is possible in the framework of constraints:

```
?- Pop #> 5000000, city(X,Pop).
```

Notice that stating this constraint is not sufficient to improve performance: we must *use it to narrow search* over `city/2`. In this work, we propose to do so through indexing. This requires addressing two challenges:

1. We must be able to index on this constraint.
2. Because constraints provide a powerful and flexible language, it is not possible beforehand to implement an abstract machine that will address all possible constraints: we need a generic framework for indexing on unknown terms.

We address the latter problem first through our UDI mechanism, that allows programmers to define a function that selects a subset of clauses, given a class of attributed variables or Prolog terms. Next, we discuss the UDI in more detail.

4.1 Principles

Given a program P and a procedure Q defined as a set of clauses $\{c_1, \ldots, c_n\}$, Q's indexing code I^P is a function defined as follows: given a goal $G\pi$ and a matching procedure Q, where π is a set of constraints, then $I^P : (Q, G\pi) \rightarrow Q'$ selects a set of clauses $Q' \subset Q$ such that if reducing $G\pi$ against $c \in Q$ succeeds then $c \in Q'$.

Clearly, the most trivial indexing function is the identity function: $Q' = Q$. In general, as I^P incurs an overhead, one has to make sure that the benefits

of computing I^P outweigh this overhead. One way to do so is to restrict how indexers are constructed. Typically, Prolog systems restrict I^p as follows:

1. I^P is known at compile-time; that is, the function I^P must be explicit before querying the program.
2. $I^P(Q, G\pi)$ is local, that is it depends on Q only and not on $P - Q$.
3. Indexing uses Herbrand constraints, that is, $I^P(Q, G\pi) = I^P(Q, G\sigma)$, where $\sigma \subset \pi$ are the Herbrand constraints in π.

Work on indexing has proceed by relaxing these constraints. YAP's JITI, for example, relaxes the first constraint: essentially the JITI implements an interesting subset of an "ideal" indexing function by only coding the cases shown to be useful. The second constraint is relaxed in systems such as Ciao [**?**] that can look at the whole program P to understand the possible queries and improve the quality of indexing code.

To the best of our knowledge, there is little work on indexing non-Herbrand constraints in the context of logic programming. We are interested in doing so through an user defined indexer, U^P. Our first observation is that we would not expect U^P to be the only indexer in the system: in general, it must be able to work with a default, system indexer. A simple approach would be to choose one indexing per procedure or query. But, ideally, we would want to have different indexers working together *over the same query*.

We can now state the properties of the user indexer U_i^P: **(i)** it must be correct; **(ii)** it must not perform arbitrarily worse than the default indexer; and **(iii)** it must work together with other indexers.

4.2 Pragmatics

User indexer are constructed and used as a three step process:

1. The programmer declares a predicate Q that will benefit from UDI, at this point the engine will initialize the respective UDI through `udi_init(Q)`;
2. The engine consults a new clause C for an user indexed procedure, the engine will call `udi_extend(C)`;
3. A call to an user-indexed goal, the engine will call `udi_exec(G)`.

Next, we use the `pop/2` example to show our implementation. We shall assume there are two U^P: one for B+-Trees and one for R-Trees.

Declarations We use *declarations* to inform which U^P will be used by a procedure Q. Each declaration specifies which program-dependent interpretations we will give to the arguments of a procedure. A declaration is as follows:

```
:- udi pop(-,btree(int)).
```

First, the Prolog engine tags the procedure as user indexed. Next, for both U^Ps the engine calls `udi_init(pop(-,btree(int)))`. In the example, the btree indexer will **(i)** initialize a new, empty, B+-Tree of integers for `pop/2`; **(ii)** declare

that the key to the tree will be the second argument; and **(iii)** store a pointer to the new tree in a record. This B+-Tree record will be returned to the engine as an opaque *handle*. The engine stores the handle in a table towards fast lookup of all UDI indexers for a procedure.

The rtree indexer will also be called. It will simply consult the declaration, and return `NULL`.

Asserting Every time a clause for a user indexed predicate has been asserted, and *after* it has been compiled, the Prolog engine searches for UDI records. For each record, it calls `udi_extend()` with a pointer to the term describing the clause, a pointer to the compiled code, and the *handle*. In our example, the btree code would **(i)** recover the tree from the handle; **(ii)** fetch the key from the second argument, given the clause's source; **(iii)** insert a new record new key in the B+-Tree; **(iv)** associate the new record with the clause code. The UDI code then returns control to the engine.

The engine is not informed of what the indexing code does but it does assume the clause code will be respected.

Execution Currently, we assume that each predicate has at most one user indexer. UDI code is supported by the following YAP instruction:

```
yamop *new = Yap_udi_search(P->u.lp.p);
if (!new) P = PREG->u.lp.l;
else P = new;
JMPNext();
```

The function `Yap_udi_search` receives a pointer to the procedure descriptor and fetches the handle matching the procedure table. It then calls `call_udi` using the handle as argument. If `call_udi` returns a `NULL` pointer YAP will fall back the default indexing code. Otherwise, YAP will execute the code returned by the UDI, which can be:

- a pointer to code, usually clause code;
- a pointer to a set of clauses;
- `NULL`, as explained above;
- `FAIL`: execution just fails.

From the engine point of view, the non-trivial case is when the engine must process a set of clauses. It must construct instructions that can enumerate every clause. In our case, we decided that the constructed object should be discardable on backtracking (or we will risk filling up memory). The current YAP implementation relies on "blobs", or opaque terms, to implement this functionality. Essentially, YAP creates an "opaque term" which just contains WAM code of the form `try-retry-trust`. These objects can be easily stack shifted, but choicepoints may point to instructions within the blob, making garbage collection difficult.

5 User Defined Indexers

Next, we propose two constraint systems that rely on the UDI for efficient execution. We will use the following methodology:

- Data will be represented as a standard Prolog database.
- Constraints will be used to represent our queries. Thus we shall follow Datalog with constraints style [**?**]. Other styles such as HiLog would be possible [**?**], but we chose Datalog because it has a natural application to our indexing algorithms.
- We shall use UDI code to associate semantics to special procedures.

As explained above a typical query will be as follow:

```
?- Pop #> 5000000, city(X,Pop).
```

set queries can be written in Prolog style:

```
?- setof(Pop, (Pop #> 5000000, city(X,Pop)), Cities).
```

Notice that In this work we are interested in reasonably simple queries *executed on large databases*: indexing, and not constraint propagation, will be fundamental.

In the above example, the first step is to implement the constraint `#>`. YAP supports Demoen's implementation of attributed variables [**?**]. In this case, the constraint could be set by the following (simplified) code:

```
A '#>' B :-
        attributes:put_att_term(A,range(_,gt(B))),
        attributes:put_att_term(B,range(_,lt(A))).
```

The calls to the `put_att_term` built-in associate variable A and B with the constraint. Following Demoen, the constraint is represented as a compound term. The functor represents the constraint module, or package: in this case, B+-Trees are used to to verify satisfiability of `range` constraints. The first argument chains constraints for different modules on the same variable. The $N-1$ remaining arguments correspond to the constraints for the same module: in this case, and towards readability, we represent them explicitly.

Notice that the code is simplified: the definition should be symmetric and cumulative and it should handle the cases where either A or B are instantiated.

The second step of execution is `city(X,Pop)`. The UDI code for `udi_exec` is then called and access the arguments of the predicate through the C-interface. The indexer executes as follows:

- Fetch the second argument A_2
- Verify whether A_2 is an attributed variable: if not, return `NULL`.
- Verify whether A_2 contains a term with main functor `range`: if not, return NULL.
- Translate the constraint(s) into a query on B+-trees.

– If the query returns no matching clauses, return `FAIL`
– If the query returns a matching clauses C , return C
– If the query returns several matching clauses, call the C-interface to construct a "blob" that will allow backtracking through the code.

Next we will discuss how to use UDI in our two examples: ranges and vectorial data. They both use trees as indexing structures, and therefore most of their definitions are similar. In both cases, `udi_init` and `udi_extend` are very similar: `udi_init` stores which arguments are indexed and initializes the tree; `udi_extend` will insert the indexed arguments in the trees, saving the clause pointer to use as return value in searches. Each UDI example works in a specific domain so the tree structure and set of constraints will be shown next.

5.1 Ranges

We propose two UDIs for range data: one for integers and one for floating point numbers. We will discuss them together, given the obvious similarity. We support seven constraints, two unary and five binary constraints:

```
max A    min A    A #> B    A #>= B    A #< B    A #=< B    A #= B
```

In our simplified implementation each constraint term has 6 arguments. One represents a constraint `max`, `min`, and the other four the maximum and minimum limits, and whether we can match that limit. For example, a range query of the form:

```
?- Pop #> 100, Pop #< 1000, city(X,Pop).
```

will result in setting the following range constraint on `Pop`:

```
range(_,false,100,false,1000,false)
```

The UDI code searches for this `range` structure and translates it into a range query returning all values in the database such that their second argument is between 100 and 1000 (if any). If the second argument was set to `max` it would return the maximum value in this range: other queries such as average or mode can easily be implemented.

5.2 Vectorial Terms

The original motivation to this work was our interest in using vectorial terms or spatial terms as defined in previous work [**?**]. These are simple geometry types based on 2D points. Notice that the simplicity of the primitives does not mean that the terms themselves are simple. For example, the European countries boundaries in Figure 3(b) are represented in Prolog as multipolygons with several hundred points each (and this is a low resolution sample).

Here we use R-Trees as the indexing structure, following the ideas in Section 3. In this work, we will use `overlaps` binary constraint `&&`, the key operator on the Postgis spatial RMS [**?**]: `A && B` constraint is satisfied if `A`'s bounding box overlaps `B`'s bounding box. A query is shown next:

```
?- country(spain,P1), P2 && P1, country(Country,P2).
```

Here as `P1` is instantiated by the time `P2 && P1` is reached `P2` will be attributed by `overlap(_,P1)`, thus second call of `country` will search the tree succeeding only with Countries that have overlapping boundaries MBR with *spain*.

Notice that the `&&` only approximates overlapping the actual intersection must be performed after. For example:

```
?- country(spain,P1), P2 && P1, country(C,P2),
   intersection(P1,P2,P3).
```

The query searches for countries that intersect with Spain. The overlapping constraint prunes the results to Portugal, France and Andorra, but only the latter will eventually succeed. Notice that the same result would be achieved without the use of UDI, but with a high penalization in time:

```
?- country(spain,P1), country(C,P2), overlap(P1,P2),
   intersection(P1,P2,P3).
```

6 Experiments

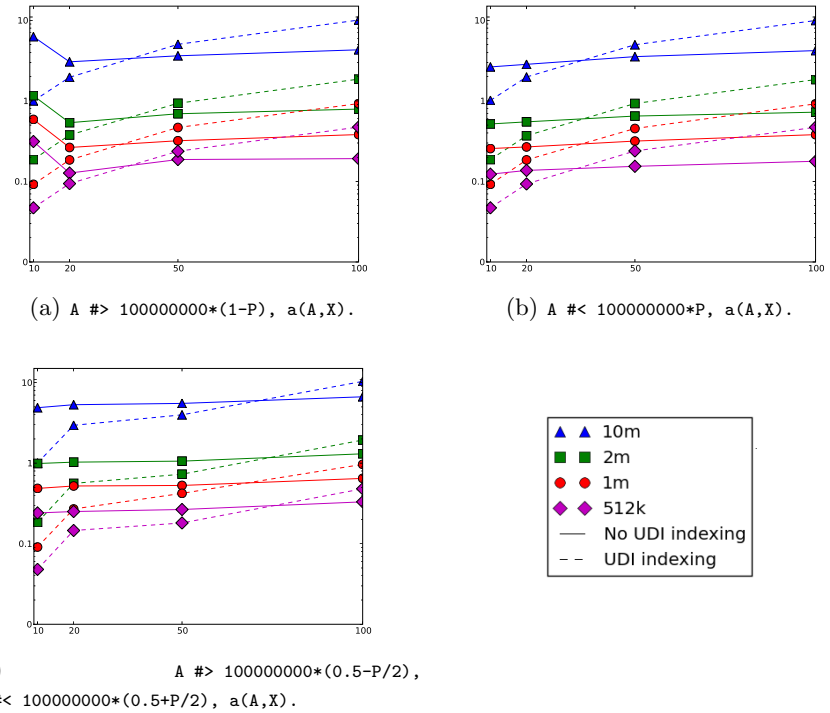
In this section we discuss the performance of our two UDI indexers. We compare against the default JITI indexing in YAP.

6.1 Experimental Setup

We performed our experiments on a Core 2 Duo P9500 @ 2.53GHz machine with 4GB of memory running Linux 2.6.27 in 64 bit mode.

Our goal in evaluating range queries was to compare Prolog and UDI as we vary selectivity and database size. As a first step, we created four datasets, with sizes between 512K and 10M tuples. Each dataset was filled in with a uniform distribution of random integers in the interval $[1, 100000000]$. Next, we experimented with simple queries, as shown in Figure 4. The first two queries select all values above or below a certain threshold, the third query selects a range in the database. We control how many tuples should be selected: values are 10%, 20%, 50% and 100% of all tuples. Figure 4 shows the execution time in all cases. Notice that we only show the constraint queries, the Prolog queries must be written with the tests after the database call.

Regarding the evaluation of vectorial data, we are interested in performance on common spatial queries such as: "Find road intersections", "Find railway crosses", "Find road bridges over streams"; in all cases the queries are about overlapping objects. Our methodology was as follows: **(i)** we select two sources of geographical objects S_1 and S_2; **(ii)** for every object O in S_1, we query how many objects in S_2 O overlaps. Figures 5(a) and 5(b) displays the actual Prolog code used in both cases. Notice that we only compare if the object's MBRs overlap in these tests. We use datasets of Germany and California geographical



(a) `A #> 100000000*(1-P), a(A,X).`

(b) `A #< 100000000*P, a(A,X).`

(c) `A #> 100000000*(0.5-P/2), A #< 100000000*(0.5+P/2), a(A,X).`

Fig. 4. B+-Tree UDI testing. Times in seconds (Y axis) of UDI versus no UDI, varying the result percentage over dataset size (X axis). Times are given in log scale.

data in these experiments, obtained at http://www.rtreeportal.org. Figure 5(c) shows the results; in Germany we overlap roads with utilities and level lines, and in California we compare roads and streams. Notice that the California roads dataset is over 2.1 million objects whereas Germany largest dataset only have 36k objects: we include the size of each dataset in brackets.

6.2 Results and Discussion

Figure 4 shows that, as expected, in all cases performance of the UDI code varies linearly with the size of the output and with database size. Prolog does not benefit from tuple selection: as a result, performance tends to be independent of output size, although it still varies linearly with dataset size.

If the output size is close to the database size, there is no benefit in using the UDI. In this case, pure Prolog is faster than the UDI code, as it can use static data structures; the UDI has been designed to construct answer lists dynamically. The UDI starts to performs better as the output size decreases, and is up to 10 times faster for 10% output size. Notice that 10% of, say, 10MB is still quite

```
:- [data1].
:- [data2].

overlap([(X1,Y1),(X2,Y2)],[(X3,Y3),(X4,Y4)]):-
       X1 =< X4, X3 =< X2, Y1 =< Y4, Y3 =< Y4.

:- time((data1(A,B),data2(C,D),
         overlap(B,D),fail)).
```

(a) Native Indexing.

```
:- udi data1(-,rtree).
:- [data1].
:- udi data2(-,rtree).
:- [data2].

:- time((data1(A,B), D && B,
         data2(C,D),fail)).
```

(b) UDI Indexing.

data1	data2	native (a)	UDI (b)	ratio
Germany				
road (30k)	road (30k)	334.165s	0.170s	1965x
utility (17k)	road (30k)	219.259s	0.067s	3272x
road (30k)	utility (17k)	194.000s	0.090s	2155x
rrline (36k)	road (30k)	402.150s	0.106s	3793x
road (30k)	rrline (36k)	416.943s	0.095s	4388x
California				
streams (96k)	roads (2.1m)	81665.457s	13.543s	6030x
roads (2.1m)	roads (2.1m)	n.a.	80.989s	

(c) Results

Fig. 5. R-Tree UDI testing.

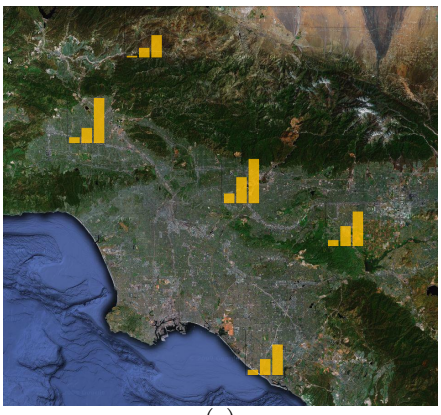
large, so the UDI is doing very well although it is constructing very large data structures, and results will be even better for queries that have very small output sizes.

Figure 5(c) shows even better results for R-Trees. Notice that we just compute overlap, we do not execute spatial operators. In this case, Prolog uses an $O(n \times m)$ algorithm versus the RTrees average $O(n \times (log(m)))$, easily justifying two orders of magnitude speedups. The performance of the R-Tree UDI results is of same order of the magnitude as the results obtained by Postgis, an extension to the well known RDMs PostgreSQL, and the main Open Source RDMs solution in Geographical Information Systems.

6.3 Example Application

Our work in the UDI has been motivated by previous work [**?**] in the geographical viewer `simplegraphics`, where as the user zooms-in we need to display fewer objects or might display the objects in view in greater detail. In general, the ability to quickly prune spatial objects is fundamental for performance in these applications. Next, we show an example of how a straightforward logic program can be at the heart of a geographical querying system.

Los Angeles is subject to earthquakes on a daily basis, due to its location in the Pacific Ring of Fire. We have gathered the location of the last five major earthquakes, shown in Figure 6. By using a roads database and different ranges of action we can estimate how many roads could be affected by a new earthquake. The predicate `in_danger` specifies the problem and we use a simple graphical interface based on the site http://maps.google.com to depict the results, as shown



(a)

```
:- udi roads(-,rtree).
:- [roads].
:- [earthquakes].

in_danger(ID,Count) :-
        earthquakes(ID,Epicenter),
        e_area(Epicenter,D),
        findall(ID2,
             (R && D, roads(R,ID2)),
                L),
        length(L,Count).
```

Earthquake	20km	40km	60km
1933	41.038	123.956	236.430
1971	12.799	75.223	173.749
1987	75.797	202.980	343.914
1994	45.816	117.734	206.294
2008	46.604	154.739	268.228

(b)

Fig. 6. Los Angeles five major earthquakes.

in Figure 6. The UDI execution mechanism takes less than a second to run these queries, hence allowing us to quickly experiment with different epicenters and with different ranges. Moreover, this small program can be easily adapted as other sources of information, such as population counts, become available.

7 Conclusions

The use of Prolog as a general-purpose language, solving a wide variety of different problems, is clearly not limited by the expressiveness of logic terms. Declarative and intuitive representations of entities such as range intervals over reals or vectorial representations of spatial objects are easily expressed as logic terms. Such conceptual efficiency is naturally expected in a language built around the motto of the *"what"*. The *"how"*'s efficiency, however, is almost completely left, in Prolog, to sophisticated compilation techniques, where indexing fits. We argued, in this paper, that the current state-of-the-art of this Prolog indexing, disconnected from *what* a term represents, can entangle the general use of the language with data-intensive problems from novel domains, such as vectorial spatial databases. Our results provide unequivocal evidence of the advantages of UDI, showing real-world queries that take hours to execute based on hash-based indexing, and are completed within tenths of a second when suitable indexing is used.

Our proposal of UDI, coupled to a declarative constraining of logic variables, is able to allow the user to redefine *how* indexing is to be done for a particular predicate, based on *what* the arguments of the predicate represent, from scalar values to multi-dimensional objects. It is clear that UDI provides the user some explicit control over the procedural execution of Prolog code, which is very much justifiable when such explicit control is able to improve the efficiency of Prolog

programs by several orders of magnitude, allowing an efficient handling of data in novel areas of application.

Acknowledgments

This work has been partially supported by projects JEDI (PTDC/EIA/66924/2006) and STAMPA (PTDC/EIA/67738/2006) and funds granted to LIACC, CRACS and IT through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. David Vaz is funded by FCT PhD grant SFRH/BD/29648/2006.

References

1. Warren, D.H.D.: Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh (1977)
2. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (1983)
3. Hermenegildo, M., Bueno, F., Puebla, G.: The CIAO multi-dialect compiler and system: An experimentation workbench for future (C) LP systems. In Parallelism and Implementation of Logic and Constraint Logic Programming (1999)
4. Puebla, G., Stuckey, P.: Optimization of logic programs with dynamic scheduling. In: Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming, MIT Press (1997)
5. Demoen, B., Mariën, A., Callebaut, A.: Indexing prolog clauses. In: NACLP. (1989) 1001–1012
6. Wielemaker, J.: SWI-Prolog 5.5: Reference Manual. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands. (2008)
7. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-driven indexing of prolog clauses. In Dahl, V., Niemelä, I., eds.: Proceedings of the 23rd International Conference on Logic Programming. Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 305–409
8. Comer, D.: The ubiquitous b-tree. ACM Comput. Surv. **11**(2) (1979) 121–137
9. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In Yormark, B., ed.: SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984, ACM Press (1984) 47–57
10. The Postgis Development Team: Postgis adds support for geographic objects to the postgresql object-relational database. Available from `http://postgis.refractions.net/`.
11. Revesz, P.: Introduction to constraint databases. Springer-Verlag New York, Inc., New York, NY, USA (2002)
12. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. J. Log. Program. **15**(3) (1993) 187–230
13. Demoen, B.: Dynamic attributes, their hprolog implementation, and a first evaluation. Technical report, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2002)
14. Vaz, D., Ferreira, M., Lopes, R.: Spatial-yap: A logic-based geographic information system. In Dahl, V., Niemelä, I., eds.: ICLP. Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 195–208