Storage and Retrieval of First Order Logic Terms in a Database

Peter Gurský

Department of Computer Science, Faculty of Science
P.J.Šafárik University Košice
Jesenná 9, 040 01, Košice
gursky@vk.science.upjs.sk

Abstract. In this paper we present a storage method for sets of first order logic terms in a relational database using function symbols based indexing method of Discrimination trees. This is an alternative method to a published one, based on attribute indexing. This storage enables effective implementation of several retrieval operations: unification, generalization, instantation and variation of a given query term in the language of first order predicate calculus. In our solution each term has unique occurrence in the database. This is very useful when we need to store a large set of terms that have identical many subterms.

Key words: first order logic terms, relational database storage and retrieval, first order logic term indexing

1 Introduction

A term in the alphabet of first order logic theory is defined inductively as follows: A variable or a constant is a term and if f is an n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term. A constant can be sometimes consider as the 0-ary function symbol. In this paper the notion term has always this meaning. Please do no confuse it with other usage of the expression 'term' in information retrieval, digital libraries or other parts of computer science. Terms constitute the basic representational unit of information in several disciplines of computer science such as automated deduction, term rewriting, symbolic computing, logic and functional programming and inductive logic programming. Computation is done by operations such as, e.g., unification, generalization, instantation or variation. Often these operations are performed on large collections of terms. For instance, in logic programming, deductive databases, and theorem-proving for model elimination we need to select all candidate clause-heads in the program that unify with a given goal. In the absence of techniques for speeding up the retrieval of candidate terms, the time spent in identifying candidates may be overshadow the time spent in performing other useful computation. See [1].

In almost all programs that work with terms or sets of terms, there is a question how to store and retrieve[1] them effective. Majority of them take, at first, all the program and transform text representations of the terms, occurred in the program, to their internal structures in the main memory. If this program contains a large amount of terms, then this initialization can be expensive. On the other side, it is possible, that we do not have enough main memory. This leads to the problem of storing terms in persistent form on the disk. One solution is to use the relational database as the standard application to store a large amount of data that are related together. It is also the standard device for sharing data between the systems. Implementation described in this paper is based on theoretical results of [1] and [2] and provides an alternative solution for all of mentioned retrieval operations. We can store the sets of terms and quickly find the terms, that fulfills given requirement with the use of indexing technique called Discrimination trees. (See [1])

The paper is organized as follows: section 2 describes the problem of first order logic term indexing and explains some useful expressions. Section 3 provides deeper look on Discrimination trees indexing technique. Section 4 shows how to represent the structure of a term with the use of directed acyclic graph. In section 5 previously described structures are implemented in a relational database. Finally, section 6 mentions conclusions.

2 First order logic term indexing

The problem of first order logic term indexing by [1] can be formulated abstractly as follows. Given a set $\mathcal{L}$ (called the *set of indexed terms* or the *indexed set*), a binary relation R over terms (called the *retrieval condition*) and a term t (called the *query term*), identify the subset $\mathcal{M}$ of $\mathcal{L}$ consisting of all of the terms l such that $R(l,t)$ fulfills ($\mathcal{M} = \{l : R(l,t)\}$).

In some applications it is enough to search for a superset $\mathcal{M}'$ of $\mathcal{M}$ ($\mathcal{M}' \supseteq \mathcal{M}$), i.e., to also retrieve terms l for which $R(l,t)$ does not hold, but we would naturally like to minimize the number of such terms in order to increase the effectiveness of indexing. In information retrieval terminology we aim for a complete query answering with possibly lower precision which can be improved by additional search. When this happen, we say that indexing performs *imperfect filtering* in terminology of [1]. Retrieved terms, in this case, we will call *candidate terms*.

A *substitution* σ in a first order logic is a finite set of the form $\{v_1 \rightarrow t_1, ..., v_n \rightarrow t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, ..., v_n$ are distinct.

Let $\sigma = \{v_1 \rightarrow t_1, ..., v_n \rightarrow t_n\}$ be a substitution and s be a term. Then $s\sigma$ is the term obtained from s by simultaneously replacing each occurrence of the variable v_i in s by the term t_i. We emphasize that it is important that

[1] In this paper the notion *retrieve* means, that we search for terms, that fulfill some kind of condition

replacements are done simultaneously. For example let $s = p(x, y, f(a))$ and $\sigma = \{x \to b, y \to x\}$, then $s\sigma = p(b, x, f(a))$.

In the context of term indexing, it is usually the case that the relation R of interest is such that $R(s, t)$ fulfills if there exists substitution σ and β such that $s\sigma = t\beta$, and furthermore, these substitutions satisfy certain additional constraints. In addition in order to identifying the terms that fulfills the retrieval condition, we sometimes need to compute the substitutions σ and β as well.

Under retrieval condition we will understand unification, generalization, instantation or variation. Given a query term t and indexed set $\mathcal{L}$, the retrieval operation is concerned with the identification of subset $\mathcal{M}$ of those terms in $\mathcal{L}$ that have specified relation R to t. The retrieval relation R identifies those terms $l \in \mathcal{L}$ that need to be selected. We will be interest in these retrieval conditions:

unif(l,t) $\Leftrightarrow \exists$ substitution σ: $l\sigma = t\sigma$;
inst(l,t) $\Leftrightarrow \exists$ substitution σ: $l = t\sigma$;
gen(l,t) $\Leftrightarrow \exists$ substitution σ: $l\sigma = t$;
var(l,t) $\Leftrightarrow \exists$ substitution σ: ($l\sigma = t$ and σ is a renaming substitution).

To understand these retrieval conditions we can make following example:

Let $l = f(x, g(a))$, $t = f(g(y), x)$, $s = f(g(b), g(a))$ and $u = f(g(x), y)$ where x and y are variables and a and b are constants.

Then unif(l,t) holds with substitution $\sigma = \{x \to g(a), y \to a\}$,
inst(s,l) holds with substitution $\sigma = \{x \to g(b)\}$,
gen(t,s) holds with substitution $\sigma = \{y \to b, x \to g(a)\}$
and var(t,u) holds with substitution $\sigma = \{x \to y, y \to x\}$.

On the other side e.g. unif(l,u), gen(s,l), inst(t,s) and var(t,s) do not hold, because there are no substitutions to fulfill the corresponding equality.

Thus, the retrieval condition is based on identification of a substitution between the query term and indexed terms, with various constraints placed on the substitution. The question of whether the retrieval condition holds between the query term and an indexed term is determined by the function symbols in both these terms. Thus, in every positions where both the query term and candidate term contain a function symbol, these symbols must be identical, because substitution does not change function symbols, it changes only variables. So we can make use of the function symbols in the indexed terms in determining the candidate terms. Most known term indexing techniques are based on this observation, and we refer to such techniques broadly as *function symbols based indexing*, or simply as *symbol-based indexing*. Representative of this techniques is also an indexing technique called Discrimination trees described in [1,2], that we will use below.

An alternative to symbol-based indexing is *attribute-based indexing*. In attribute-based indexing, we map som features of a term t into a simple-valued (say, integer-valued) attribute a_t. This solution is based on the assumption that a relation involving simple-valued attributes is much easier to compute than performing term matching or unification. However, according to [1] it has several disadvantages. Firstly, the precision of attribute-based indexing is typically low.

Second, if the index set is large, the coarse filter may still be inefficient as it may involve checking the retrieval relation from each term in a set. For details see [1,5].

In [5] authors use the attribute-based indexing for storing the terms in a relational database. Our solution is an alternative method with a use of more powerful filtering of symbol-based indexing also stored in a relational database.

3 Discrimination trees

Now, if we know that function symbols of the language of predicate calculus of first order logic in every position in the query term and the candidate term must be identical, we need some method to compare them. One solution is to construct a string of symbols from the query term, and identify a candidate term if this string matches the string constructed from the indexed terms. We can make these strings by writing out the symbols occurring in a term in some sequence. However, such an approach may lose some of the information captured by the term structure.

In discrimination trees we generate a single string (called the *path string* or the *p-string*) from each of indexed terms. These p-strings are obtained via a preorder traversal (left-to-right, depth-first direction) of the terms. To construct index structure we mount these p-strings in the index trie. The trie structure is described in [1]. We can see that there is a unique correspondence between the string obtained by preorder traversal (i.e. by writing out the symbols occurring on this traversal) and the text representation of a term. If we say that we will go through the terms only by preorder traversal, we can generate p-string very fast. If we have the only traversal, we don't need any added information about positions captured to function symbols, as it is in many other indexing techniques described in [1], e.g. Path indexing.

Except function symbols, and constants (that can be seen as function symbol with null arity), there are variables in terms too. This method speeds up finding of candidate terms by replacing variables by symbol *. This leads, of course, to imperfect filtering (lowers precision but preserves completeness), because it is impossible to search for substitutions or check out, if retrieval condition holds between the query and the indexed terms. We have to compute resultant substitutions, that is very expensive, after retrieval of candidate terms.

We can illustrate discrimination tree indexing using the example set of indexed terms and the p-strings generated from these terms.

{1}	f(g(a,*),c)	f.g.a.*.c
{2}	f(g(*,b),*)	f.g.*.b.*
{3}	f(g(a,b),a)	f.g.a.b.a
{4}	f(g(*,c),b)	f.g.*.c.b
{5}	f(*,*)	f.*.*

The retrieval of generalization of query term $f(g(a,c),b)$ from the indexed trie obtained from these p-strings is shown in Figure 1. To understand the process

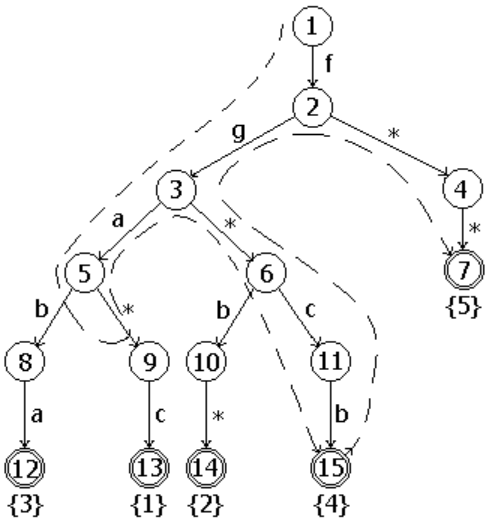

Fig. 1. Index and retrieval of generalizations of the query term *f(g(a,c),b)*

of indexing, note that the string corresponding to the query term is $f.g.a.c.b$. We compare the symbols in this p-string successively with the symbols on the edges in the path from the root to state 5. At this point, we cannot take the left path, as the symbol b on this edge conflicts with the symbol c in the query term. However, the symbol $*$ on the edge leading to state 9 is entirely plausible, since taking this edge corresponds finding a generalization (namely, a variable) of the subterm c of the query term. However, we cannot proceed further from state 9 because of constant c, so we have to backtrack to state 3. At this point, we can follow down the $*$ branch all the way down the final state 15, identifying candidate term 4. If we are interested in all generalizations, we have to backtrack further to state 2, and then finally follow to state 7, identifying candidate term 5.

In order to perform retrieval of unifiable terms and instances, we must efficiently deal with situations where the query term has a variable at a point where the indexed terms contain a function symbol. In such a case, we need a mechanism to efficiently skip the corresponding subterms in the indexed terms. It is also not trivial, witch part of p-string generated from query term, we need to skip, if we find $*$ on some edge of index.

To perform traversal of a term t we will need two operations on term positions explained in [1]: $next_t$ and $after_t$, witch can be informally explained as follows. Represent the term t as a tree and imagine a term traversal in the left-to-right, depth-first direction (i.e. preorder traversal). Suppose that s is a descendant of t and its (unique) position in the tree is p. Then $next_t(p)$ is the position of subterm

of t visited immediately after s, and $after_t(p)$ is the position of subterm visited immediately after traversal of all subterms of s.

Figure 2 illustrates the behavior of next and after on the positions in the term $f(g(a,b),c)$, when we mark the position of the symbol t by Λ, the position of the symbol g by 1, a by 1.1, b by 1.2 and the position of the symbol c by 2. We also need a special object ε, that is representative of the "end position" in the term.

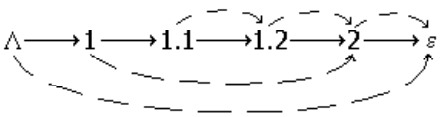

Fig. 2. $next_t$ and $after_t$ on the positions in term t = f(g(a,b),c). Solid straight lines represents $next_t$ and dashed lines represents $after_t$

When we have these two functions, it is easy to perform traversal through the query term during the retrieval operation. Thus, if we make traversal in the index trie and find the function symbol on the edge, we have to compare it with relevant function symbol in the query term, and if they match, we call the function $next_t$ to determine the next comparing position. If we find symbol * on the edge, we can call the function $after_t$ for the position of next comparing. Thus, we said that we can substitute the symbol * with all the subterm on this position, and next comparing will be with his next sibling in the query term tree or next symbol in preorder traversal after all this subterm.

Function $after_t$ in the case of query term we cannot use, of course, when the retrieval condition is instance, but we need mechanism similar to this function in the index trie. This requirement stands out also when we need to find unifiable terms. For this purpose we will use the structure named *Jump lists* from [1]. It can be seen that there must be an analogy with the function $after_t$.

We can make following example. We can add new term t = f(g(b,c),*) with its p-string f.g.b.c.* into the index trie in figure 1. Now imagine its traversal functions $next_t$ and $after_t$. Those structure is identical to that in Figure 2. It can be seen, that there is reciprocal corresponding between the function $next_t$ and the respective branch in the indexed trie. We can see added branch on Figure 3. In every state on this branch (except the last, that represents the end position ε) we will add a link to the states corresponding those positions in term t that determine the function $after_t$. In this case, we will add in the edge 1 the link to state 18, similar in state 2 to state 17, in state 3 to state 16, in state 16 to state 17 and in state 17 to state 18.

In Figure 3 we can see retrieval of terms unifiable with $f(g(b,*),a)$. Dashed lines are Jump lists only for those nodes where jump links go to a state different from the immediate child of a node, and except links from root to the leafes.

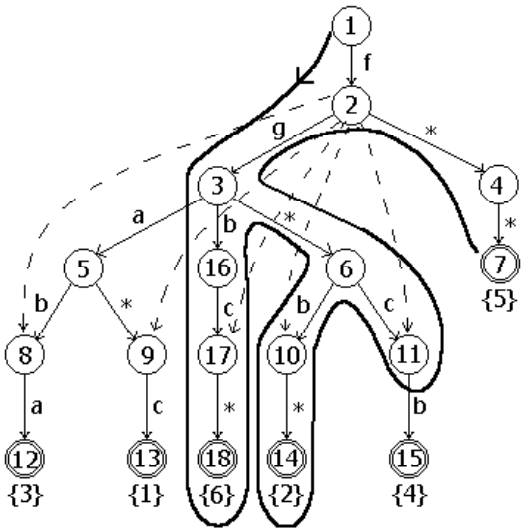

Fig. 3. Retrieval of terms unifiable with $f(g(b,*),a)$

All the structure of discrimination trees including index trie with jump lists we implement in a relational database in section 5.

4 Storing of terms

When we add a new first order logic term to the database, the standard input is the text-representation of this term and the name or the id of the set, where this term have to be a member. Storing only the pure text-representation of terms is not suitable. It is known that, term can be represented as a tree or *directed acyclic graph* (*DAG*) [1,2]. The root node of the tree representation of a term t contains the root symbol of t and pointers to nodes that correspond to immediate subterms of t. As compared to a tree representation, a DAG representation presents an opportunity to share subterms. We are trying to ensure that exists only a single copy of a term, regardless of number of contexts in which it occurs. This solution is called *perfect sharing* or *aggressive sharing* [2]. Such sharing can contribute to as much as an exponential decrease in the size of the term. The example in Figure 4 shows DAG representation of term $f(a,g(1,h(b),a),h(4,h(b)))$ with aggressive sharing.

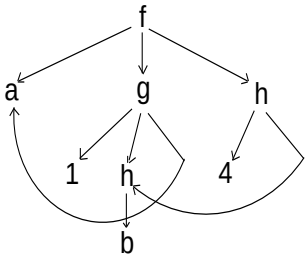

Fig. 4. DAG representation of $f(a, g(1, h(b), a), h(4, h(b)))$ with aggresive sharing

The benefits of sharing are significant in practice. It is very useful when we want, for example, to unificate two terms. There is known some unification algorithms like the *unification on term dags* or *an almost-linear algorithm* described in [2], that need on the input the structure contains two DAGs of the terms with shared subterms that we want to unificate. These algorithms are more powerful than any other unification algorithms based on any different structures. Those main advantage is that each substitution is computed only once, because there is always only one instance of each variable and after computing of any particular substitution this can be implement on the each of positions immediately at once and the substituted variable will never occur again.

When we do not share subterms, unification algorithm can be very inefficient. In the worst case, its running time can be an exponential function of the length of the input, for example when we want to unificate terms $s = p(x_1, ..., x_n)$ and $t = p(f(x_0, x_0), ..., f(x_{n-1}, x_{n-1}))$. For details see [2,5].

5 Implementation in relational database

In chapter 2 we have followed the informal description of the algorithms for retrieving of candidate terms and in previous chapter we have said how we can represent first order theory terms according to [1], [2] and [3]. In this chapter we present our method of storing terms in a relational database. The decomposition of data to the relational schema is shown in Table 1. For better notion of relationships between the tables, there is the database diagram on Figure 5.

For better understand of how to store and retrieve terms with use of this decomposition, we will show this on following example.

Imagine that we want to insert term t=$f(a, g(1, h(b), a), h(4, h(b)))$ where a and b are variables, whose DAG representation is shown in Figure 4. We also need to know the name or id of the set, of which this term have to be a member. For simplicity suppose that its name is $Set1$ with id 1 already stored in table SET, with root_state 1. The root_state is a number, that tell us, which state in index trie belong to this set is a root one. It is better to store for each set its

SET(id, name, root_state)
INSERTED(id, id_term, id_set)
TERM(id, id_symbol)
ATTRIBUTE(id_father, id_son, position)
SYMBOL(id, name, arity)
STATE(id, id_symbol, next)
JUMP(state, jump)

Table 1. Relational schema

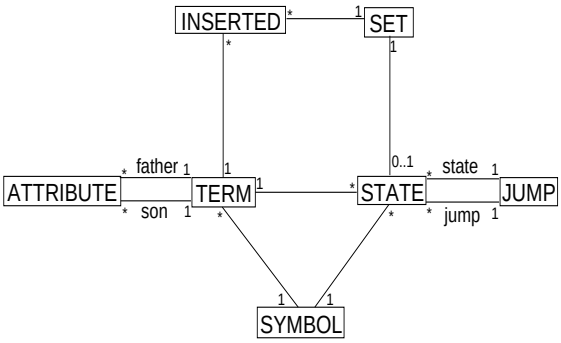

Fig. 5. Database diagram

own index trie. This solution is not suitable only if we want to retrieve always from all stored terms. In this case is the information about sets needless and, of course, we do not need table SET. In other cases our implementation allow to have a smaller index tries and faster retrieval from the individual sets of indexed terms. At first, we will point a look on storing of DAG representation of terms. An example can be seen in Table 2. For better understand of relations we do not start to generate id-s in table TERM from 1 but 11.

The symbols occurred in terms with their arities are stored in table SYMBOL. The column *arity* in the table SYMBOL is captured with the name, because it can really speed up retrieval from the index. Arity is also useful, when we can read all the structure of any term for the database. In this case we know, if we have to seek the table ATTRIBUTE for the subterms. Note that the arity -1 denotes a variable and the arity 0 denotes a constant.

The structure of the term is covered in table ATTRIBUTE. For example, as we can see in Table 2, the term with id 11 (whose text representation is symbol f) has as children the terms with *id*-s 12, 13 and 17 respectively. The order of these subterms is determined by the column *position*.

A expert fluent in databases would say that information in table TERM is redundant. It became useful when there is an identical function symbol with the same arity but with different subterms. In this case, if we have not the table

SYMBOL		
id	**name**	**arity**
1	f	3
2	a	-1
3	g	3
4	1	0
5	h	1
6	b	-1
7	h	2
8	4	0

TERM	
id	**id_symbol**
11	1
12	2
13	3
14	4
15	5
16	6
17	7
18	8

ATTRIBUTE		
id_father	**id_son**	**position**
11	12	1.
11	13	2.
11	17	3.
13	14	1.
13	15	2.
13	12	3.
15	16	1.
17	18	1.
17	15	2.

Table 2. Example of representation of inserted term $f(a, g(1, h(b), a), h(4, h(b)))$

TERM, we should have to add a new symbol as a row to the table SYMBOL that would have the same name and arity as already stored symbol, just with different *id*. For example if we want to add the term h(a), we can use fifth symbol, e.g. we add the couple (19,5) to the table TERM and triple (19,12,1) to the table ATTRIBUTE.

We can see, that we store only one instance of each subterm. We suppose that aggressive sharing is not concern only on individual terms, and there is only one instance of each subterm or term in all the database. Thus, if we add another term, that has any subterm identical to some subterm, that was stored before, we simply refer to the stored one. This happen also when this term is stored in another set.

There is one another table INSERTED, that was not mentioned. It stores the information, which term was original inserted (more precise it stores its root *id*) and to which set.

Now, when we have stored the DAG representation, we need to know, how we represent the index trie of Discrimination trees. As we have described in section 3, at first we make the p-string and add it as a new branch into the index trie. Structure of the branch from our example can be seen in Figure 6.

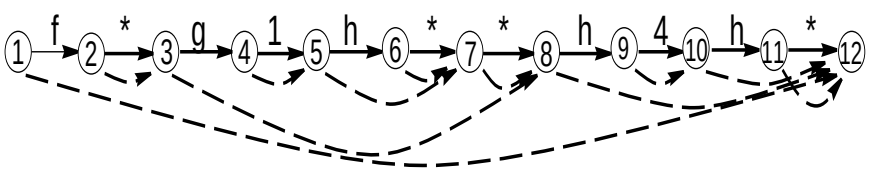

Fig. 6. Index representation of term $f(a, g(1, h(b), a), h(4, h(b)))$. Solid lines represents *next* and dashed *jump*

We can store this structure to the tables NEXT and JUMP as we can see in Table 3.

SYMBOL		
id	**name**	**arity**
1	f	3
2	a	-1
3	g	3
4	1	0
5	h	1
6	b	-1
7	h	2
8	4	0

STATE		
id	**id_symbol**	**next**
1	1	2
2	NULL	3
3	3	4
4	4	5
5	5	6
6	NULL	7
7	NULL	8
8	7	9
9	8	10
10	5	11
11	NULL	12
12	1	0

JUMP	
id_state	**id_jump**
1	12
2	3
3	8
4	5
5	7
6	7
7	8
8	12
9	10
10	12
11	12

Table 3. Representation of indexing structure for term $f(a, g(1, h(b), a), h(4, h(b)))$

The structure of these tables is, almost all, obvious from the Figure 6, when we say, that we are replacing a symbol * by NULL value for simpler differentiate between function symbols and variables with preserving of integer value of column *id_symbol*. The only difference between Figure 6 and Table 2 is on the last raw of table STATE. On this place we are making a trick, and saying, that when in the column *next* is value 0, than we are on leaf of index trie, and in the column *id_symbol* is the id of one of the terms, that are attached to this final state and stored under this id in table TERM or table INSERTED (vote is on the man, who want to implement it).

Now, when we are familiar in database structure, we can demonstrate, how to retrieve terms, that fulfill a retrieval condition. Let us have the text or DAG representation of a query term, retrieval condition and the name of set, that we want to seek for retrieval. At first, we need a list of symbols with arities equal to arities of the query term and its subterms. Then we must have a look to the table SYMBOL to get *id*-s of these symbols. If we do not find relevant rows for function symbols (with arity greater or equal to 0), than if the retrieval condition is instance or variation, we can say, that there is no relevant candidates in database. In other cases we can assign to such a symbols or variables unused *id*-s, e.g. negative numbers. Now we can seek indexed trie of given set of terms with root symbol registered in table SET. Traversal on this trie was described in section 3 with the difference that we do not match symbols but *id*-s and symbol * was substituted with NULL value. The function $next_t$ we easy can simulate with the use of table STATE and the function $after_t$ with the use of table JUMP.

On this part of enumeration we have two possibilities. We can wait for all the set of candidate terms or use the aspect of Discrimination trees, that we can receive candidates in sequence one by one. This allows to do next enumerations with the use of threading.

Further, it remains us to compute substitution for each couple of the query and the candidate term. We have to select all the structure of a candidate term from the database. One advantage is, that we obtain the structure, that answers the DAG representation with aggressive sharing. So we can use very fast algorithms as it was written in section 4. Finally, we have to delete those candidates, for which the substitution cannot be computed.

This solution of implementation of sets of terms in database provides a structures for storing and retrieval. This system is suitable primarily for applications, where retrieval performance is important and efficiency of maintenance operations is not a concern. If we want to insert or delete a branch from index trie we need to take care of shared states and jump lists. Similar, when we want to insert or delete a term from set (or more complicated from several sets) there is need to see if any subterm is a part of some other term or as a term is a member of a different set. In spite of that, there is a lot of programs, in which the sets of terms are almost static and the primary requirement is retrieval time.

6 Conclusions

In this paper we have presented a storage method for sets of first order logic terms in a relational database using Discrimination trees. Our solution is an alternative to a [5], based on attribute indexing. This storage enables effective implementation of retrieval operations unification, generalization, instantation and variation of a given query term. In our solution each term has unique occurrence in the database and can be easy converted to the DAG representation of terms. This provides very fast verifying of candidates returned from the index.

Acknowledgement. I would like to express my thanks to my master thesis supervisor RNDr. Peter Eliaš PhD.

Supported by project VEGA 1/0385/03.

References

1. R.Sekar, I.V.Ramakrishnan, Andrei Voronkov *Term indexing* in Alan Robinson, Andrei Voronkov *Handbook of automated reasoning*. Elsevier Science Publishers B.V. 2001
2. Franf Baader, Wayne Snyder *Unification theory* in Alan Robinson, Andrei Voronkov *Handbook of automated reasoning*. Elsevier Science Publishers B.V. 2001
3. Peter Gurský *Implementation of formal structures in database systems*. Master thesis under supervision of Peter Eliaš (in Slovak). Košice 2003.

4. J.W.Lloyd *Foundations of logic programming*. Springer-Verlag New York Berlin Heidelberg 1987. ISBN 3-540-18199-7, 0-387-18199-7
5. Paul Singleton, O.Pearl Brereton *Storage and retrieval of first-order terms using a relational database*, 1993. ISSN 1353-7776