TR-401

Term Indexing for Retrieval by Unification

by
H. Yokota, H. Kitakami and
A. Hattori(Fujitsu)

June, 1988

Term Indexing for Retrieval by Unification

Haruo Yokota[†]

Hajime Kitakami[††]

Akira Hattori[†††]

Artificial Intelligence Laboratory

FUJITSU LABORATORIES LTD.

Kawasaki, Japan

ABSTRACT

This paper presents a method for indexing terms in a knowledge-base retrieval-by-unification (RBU) system. The term is a well-defined structure capable of handling variables to represent knowledge. RBU operations are an extension of relational database operations using unification and backtracking to retrieve terms from term relations. The term indexing we propose uses hashing and trie structures to reduce the number of comparisons between elements of a search condition and of an object term relation. Unification on a trie structure is suited to backtracking bindings of variables. The search and updating speed of an RBU prototype is measured to evaluate the indexing method. This method is effective in fast term retrieval for a large number of similar and varied form terms. The overhead for maintaining indexes in updating is low.

e-mail address:

† hyoko%motoko.stars.flab.fujitsu.junet@uunet.UU.NET

†† kami%motoko.stars.flab.fujitsu.junet@uunet.UU.NET

††† hattori%ayumi.stars.flab.fujitsu.junet@uunet.UU.NET

1. Introduction

Retrieval-by-unification (RBU) operations have been proposed [Yokota 86] for a knowledge base system of the Fifth Generation Computer Systems (FGCS) project in Japan. RBU operations are an extension of relational database operations for manipulating knowledge. A knowledge element is represented by a term, a well-defined structure capable of handling variables. A knowledge base consists of sets of terms called term relations. The RBU system searches the term relations for desired terms, those unifiable with a search condition.

The relational database presents a large amount of data and the knowledge stored in the RBU is easy to use. The SLD resolution can be implemented with a combination of RBU operations [Yokota 86]. The SLD resolution is a resolution algorithm for Horn logic, which is a subset of first-order logic [Lloyd 84]. The execution mechanism of Prolog systems is based on the SLD resolution. A parallel production system using RBU operations has been proposed for solving artificial intelligence problems [Yokota 88]. In these knowledge-based systems, Horn clauses and production rules are stored in term relations as knowledge bases. This chunk of knowledge is important in handling a large amount of knowledge.

Term retrieval speed influences the performance of knowledge-based systems. Different approaches have been proposed for fast term retrieval. One is dedicated hardware, e.g., parallel unification engines with a multiport page memory [Yokota 86, Morita 86, Itoh 87]. The engines perform unification on term streams in a pipeline fashion. The multiport page memory is shared by the engines. Superimposed code words for terms and another type of engine for manipulating the words has been also proposed [Wada 87].

A more economical approach is indexing using software without dedicated hardware. Prolog systems use hashing for indexing Prolog clauses. A hash vector for a join operation with unification has been proposed [Ohmori 87]. Hashing effectively retrieves a term from many terms in varied forms, but hash collisions due to similar form terms make the effectiveness of indexing low. Many similar form terms are stored in term relations for knowledge-based systems, and simple hashing is not enough to implement the RBU system. Backtracking in the index for variable binding and index-maintenance overhead in updating must also be considered in implementation.

We propose sophisticated indexing that uses hashing and trie structures. This is suited to retrieving terms from term relations containing a large number of terms in both similar and varied forms using backtracking, and to frequent updating of term relations. We implemented an RBU prototype using this indexing, and measured its search and updating speed to evaluate the indexing method. Section 2 defines the term relation and RBU operation. Section 3 presents indexing implementation. Section 4 reports the results of the RBU prototype evaluation.

2. Term relation and RBU operation

Our definition of "term" is the same as that of first-order logic [Chang and Lee 73]. We use "set" to define a term relation.

Definition Let F_n be a finite set of n-arity function symbols. Let V be an enumerably infinite set of variables where

$$\forall n, F_n \cap V = \emptyset .$$

Definition T is a term set when

i) If $t \in F_0$ or $t \in V$ then $t \in T$.

ii) If $t_1, \ldots, t_n \in T, f \in F_n$ $(n \geq 1)$ then $f(t_1, \ldots, t_n) \in T$.

An element of term set t is a term.

Definition Let $T_1, T_2, \ldots, T_m$ be term sets. TR_m is an m-attribute term relation when

$$T_1 \times T_2 \times \ldots \times T_m \supset TR_m \quad (m \geq 1).$$

tt_m is a term tuple when

$$tt_m = (t_1, t_2, \ldots, t_m) \in TR_m$$

$tt[i]$ is the i th item of term tuple tt.

Figure 1 shows a two-attribute term relation having six term tuples. In the examples in this paper, symbols beginning with capital letters are variables. When the variable X in the term $p(X,g(Y))$ at the first attribute of the first term tuple in Figure 1 is bound to a term $f(a,b)$, the term $r(X,Y)$ at the second attribute of the term tuple becomes $r(f(a,b),Y)$ because the variable scope is a term tuple. The variable binding $\theta = \{f(a,b)/X\}$ is called a substitution.

$p(X,g(Y))$	$r(X,Y)$
$q(f(a,X),g(X))$	$r(f(a,X),X)$
$p(X,g(b))$	$r(h(a,b),f(a))$
$q(f(X,Y),g(c))$	$s(X,g(Y,c))$
$p(f(a,b),h(X))$	$s(a,g(b,c))$
$p(f(a,X),h(X))$	$s(a,X)$

← Term tuple

First attribute ↑ Second attribute ↑

Figure 1. Example of term relation

Definition A substitution θ is called a unifier for terms t_1 and t_2, if and only if $t_1\theta = t_2\theta$.

Definition A unifier σ for terms t_1 and t_2 is a most general unifier, if and only if, for each unifier θ for these terms, there is a substitution λ such that $\theta = \sigma \bullet \lambda$.

Operations on term relations are based on relational algebra [Ullman 82]. There are several retrieval operations in relational algebra, including union, projection, join, and selection. We have implemented these operations, extended by unification in the RBU prototype, and have also implemented updating operations and defining operations for term relations. Because our theme is indexing, we focus on a simple selection operation with unification which we call "unification-restriction."

Definition Unification-restriction is an operation generating a new term relation TR'_m from TR_m and a search condition term t for the i th attribute:

$$tt'_m \in TR'_m \leftrightarrow \exists tt_m \in TR_m, \exists\sigma, tt_m[i]\sigma = t\sigma, tt'_m[k] = tt_m[k]\sigma \ (1 \le k \le m).$$

Here, σ is the most general unifier between $tt_m[i]$ and t.

Given the search condition term for the first attribute of the term relation in Figure 1 as $p(f(A,c),B)$, unification-restriction derives the first, third, and sixth term tuples of that term relation and generates a new term relation (Figure 2). The σs on the right of the table indicate the most general unifiers between the search condition term and each term tuple.

$p(f(A,c),g(Y))$	$r(f(A,c),Y)$	$\sigma = \{f(A,c)/X, g(Y)/B\}$
$p(f(A,c),g(b))$	$r(h(a,b),f(a))$	$\sigma = \{f(A,c)/X, g(b)/B\}$
$p(f(a,c),h(c))$	$s(a,c)$	$\sigma = \{c/X, a/A, h(c)/B\}$

Figure 2. Result of unification-restriction

3. Implementation of indexing

3.1. Representation of terms

Some representation of terms is required for storing and searching. A term can be viewed as a tree structure and unification as a type of pattern matching on the tree. A variable is bound to a subtree in the corresponding location. Figure 3 shows the tree structure of the term *p(f(a,b),h(X))*. When the term is unified with term *p(Y,Z)*, *Y* is bound to the subtree whose root is *f* and *Z* to the subtree whose root is *h*.

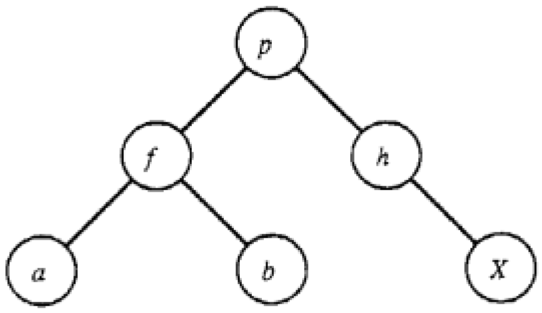

Figure 3. Tree structure of term *p(f(a,b),h(X))*

Level-order and family-order sequential representation are two ways to represent a tree structure [Knuth 73a]. The level-order sequential representation (LOSR) of the tree in Figure 3 is [*p-2*, *f-2*, *h-1*, *a-0*, *b-0*, *X*] and the family-order sequential representation (FOSR) is [*p-2*, *f-2*, *a-0*, *b-0*, *h-1*, *X*]. Each element denotes a function symbol and its arity (the number of branches of the node) in this representation to preserve the original tree structure.

Prolog systems use the FOSR. Since each node's symbol is followed recursively by the symbols for all child nodes, the FOSR is suited to traversing complicated variable bindings caused by repeated unification. A difference in term structure is found after examining fewer nodes of the

LOSR than of the FOSR. In the example, information that the top of the tree is constructed from the three nodes *p*, *f*, and *h* is derived by checking only the first three elements of the LOSR (independent of the size of the subtree-rooted *f*); five elements must be checked in the FOSR. The more nodes the subtree *f* has, the more elements must be checked in the FOSR. Because unification is repeated only a few times in the RBU system and most terms in a term relation have a similar top, we use the LOSR in the RBU prototype.

The length of the representation of a term is variable, so we use a cell structure to store the LOSR of a term. A cell corresponds to each element of the LOSR, and has four fields:

Atom table entry or null	Arity or variable number	Alternate cell	Next cell

The first and second fields are used to indicate element contents. For a function symbol, the first field contains a pointer for the corresponding entry of the atom table, and the second contains the arity of the function symbol. Function symbols are stored in an atom table because the symbol length is also variable and the same symbol may appear many times in term relations. For a variable, the first field contains a "null" pointer and the second contains the variable number. Symbols for variables are not important because they are renamed in unification. Variables are numbered sequentially in a term tuple because of the variable's scope, and the identical variable has the same variable number. The third field of the cell is used for constructing a trie structure, described later. The fourth field contains a pointer to the next cell and contains a "null" for tail elements.

Figure 4 shows the cell structure of the terms in the first attribute of the term relation in Figure 1. For readability, a function symbol is directly illustrated using a pair consisting of the symbol and arity (i.e., pointers for the atom table are omitted.) A letter V and its subscript indicate a variable and its variable number. The alternate-cell field is omitted and a slanted line in the next-cell field indicates a "null" pointer.

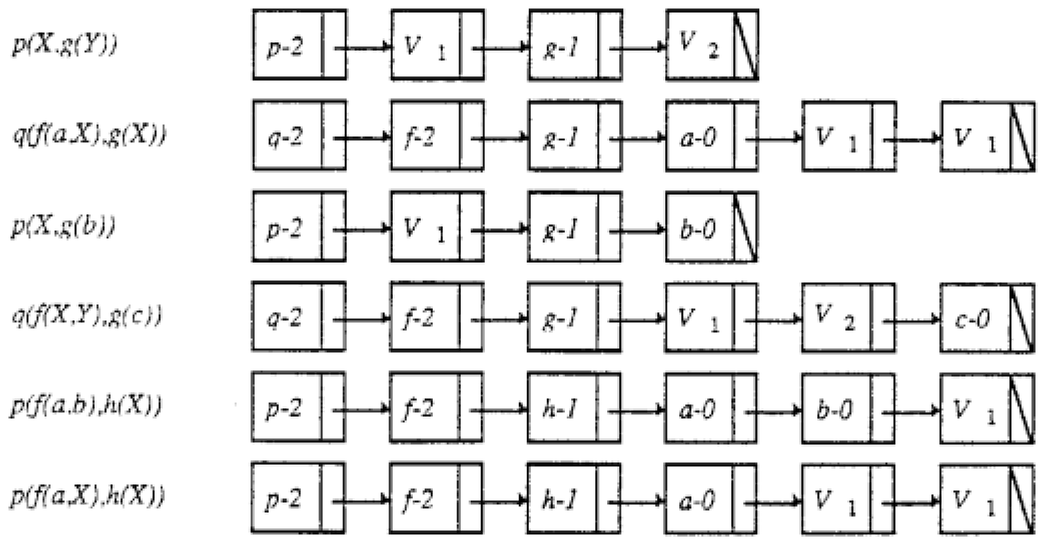

Figure 4. Cell structure for LOSR of terms

We use the cell for both storing terms and constructing indexes for key attributes of term relations. We call the cell structure in Figure 4 the flat cell list of a term.

3.2 Hashing and the trie structure

Hashing is a popular indexing technique for fast searching [Knuth 73b]. Terms with similar forms cause hash collisions and reduce the effectiveness of the indexing, however. A large number of similar form terms is stored in term relations for knowledge-based systems [Yokota 88]. Because the RBU operation requires backtracking of variable bindings, alternatives being bound to a variable are located close together in an index. We therefore combined the trie structure with hashing for our indexing scheme.

A trie structure is a type of tree for sharing identical elements [Knuth 73b]. It is usually used to store a sequence of numbers or letters. We apply a trie structure to the LOSR of terms having the same function symbol as their first element. The alternate-cell field of each cell is used for branching. The list of terms in Figure 4 is translated into two tries rooted at *p-2* and *q-2* (Figure 5). For example, the first three elements of the LOSR for *p(X,g(Y))* and *p(X,g(b))* are identical. These three elements are shared and the fourth element has a alternate cell. Terms resembling each other are located close to each other in the trie.

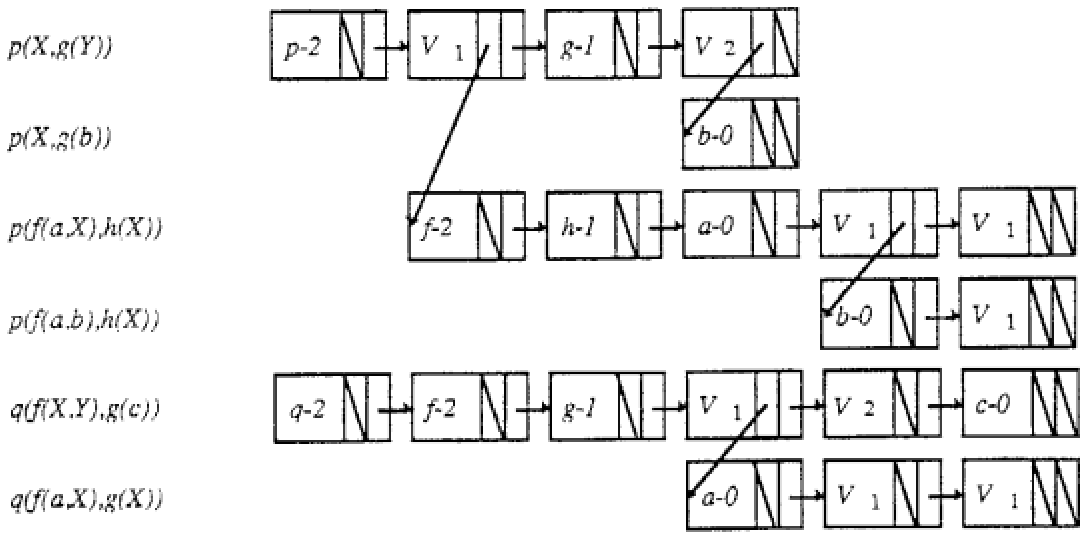

Figure 5. Trie structures for terms

The cost of the unification-restriction operation is proportional to the count of comparisons between elements of the condition and object terms. A trie reduces the number of comparisons when unification is performed on it.

Consider a searching of the set of terms in Figures 4 and 5 for terms unifiable with the condition term *p(f(a,b),h(c))*. Cells are traversed as follows to search for all such terms:

Without trie structures (Figure 4):

```
<start>
        p-2, V1(=f-2), g-1 <fail>,
        q-2 <fail>,
        p-2, V1(=f-2), g-1 <fail>,
        q-2 <fail>,
        p-2, f-2, h-1, a-0, b-0, V1(=c-0) <success>,
        p-2, f-2, h-1, a-0, V1(=b-0), V1 <fail>.
<end>
```

With trie structures (Figure 5):

```
<start>
        p-2, V1, g-1 <fail>,
                 f-2, h-1, a-0, V1(=b-0), V1 <fail>,
                                b-0, V1(=c-0) <success>,
        q-2 <fail>.
<end>
```

The first cell of the condition term is compared only twice with the first cell of the trie structures, but all six of the first cells of the flat list of terms must be compared. The size of this difference increases with the number of terms having the same first element. The same applies to the second and subsequent cells. The total number of cell comparisons for searching without the trie structure is $N \times M$ in the worst case when all elements except the last are same for all terms, where N is the number of term tuples and M is the average length of the flat list of terms. The trie structure reduces it to $N + M$ in that case.

Note that the amount of backtracking is also decreased with the trie structure. A sign at the end of each line of the above trace, <fail> or <success>, indicates invocation of a backtrack. The number of backtracks goes from six to four, making the total number of cell comparisons with the trie structure smaller than $N + M$, almost M, when the trie is balanced. In this example, the total number of cell comparisons is 11 using the trie structure, and 20 without.

We use hashing before the trie structure to store both similar and varied form term. A hash table is prepared for each key attribute in a term relation. The first element of the LOSR of terms in the key attribute is used as a hash key. A bit sequence of the atom table entry for the first element is folded by a bit width depending on the hash table size, and exclusive-ORed as a hash function to decide its location in the hash table. Even entries for similar function symbols are distributed well by this hash function.

The hash table contains pointers to the corresponding trie structures (Figure 6). The next-cell field of each leaf cell in the trie structure contains a pointer for a term tuple. The term tuple is a list of pointers for all attributes being flat cell lists of a term.

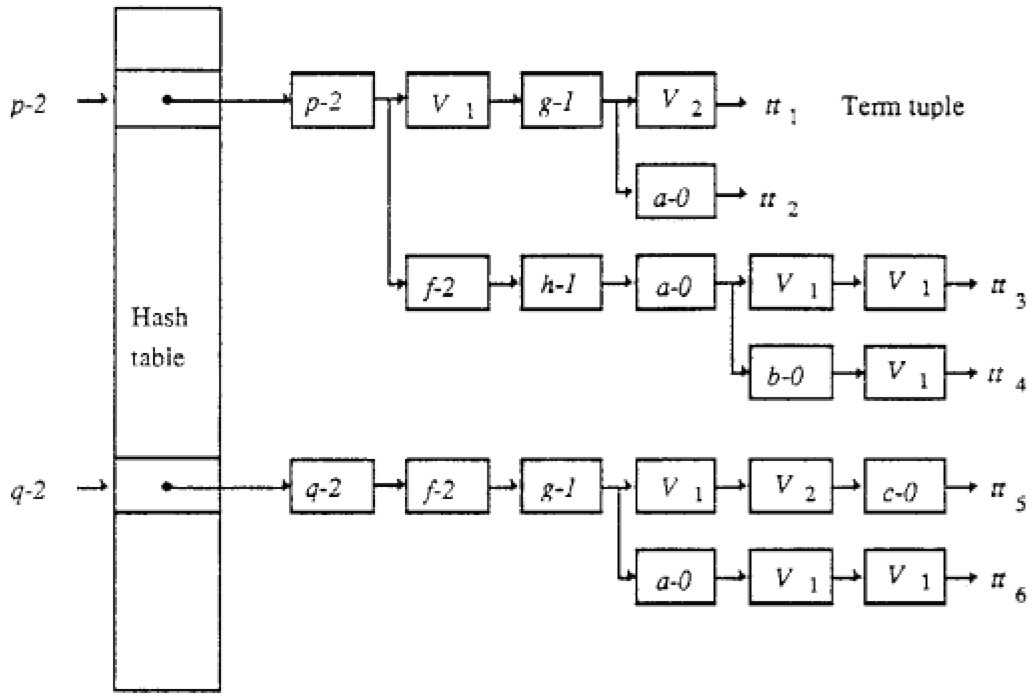

Figure 6. Combination of hashing and trie structures

The trie structure is treated as a mechanism for resolving hash collisions efficiently. Hashing is invoked only once for each search condition, and backtracking for the search can be performed in the same trie structure because the condition term is not unifiable with terms whose LOSRs have different first elements. Hashing is suited to narrowing a search space and the trie structure for traversing the space efficiently. The combination of hashing and trie structures is effective for a large number of both varied and similar form terms.

3.3. Unification on LOSR

The unification algorithm used in Prolog systems is based on the FOSR of terms. The RBU system requires a special mechanism to unify terms represented in the LOSR.

Two terms, *p(X,g(Y))* and *p(f(a,b),g(c))* , are unifiable because *X* can be bound to *f(a,b)*, and *Y* to *c* . The LOSRs are [*p*-2, V_1, *g*-*1*, V_2] and [*p*-2, *f*-2, *g-1, a-0, b-0, c-0*]. The first variable V_1 (= X) is bound to both *f-2* and [*a-0, b-0*] which are not located in a sequence. The second variable V_2(= Y) is bound to *c-0* which is in a different location than V_2 in the LOSR. A mechanism for expanding variables is needed to make a virtual LOSR [*p-2*, V_1, *g-1*, V'_1, V''_1, V_2]. We use first-in-first-out (FIFO) lists [Knuth 73a] as a working space to expand variables. Nonexpansion and expanded variable flags are prepared for the unification and pushed into the FIFOs corresponding to the arity number. The followings is a unification algorithm on LOSRs using these flags and FIFOs:

```
Unification algorithm on LOSR of terms:
Step1:  Let $L_A$ and $L_B$ be the LOSR of terms.
        Push a nonexpansion flag $n$ into each FIFO $W_A$ and $W_B$ as the initial
        state.
Step2:  If ( both $W_A$ and $W_B$ are empty ) then
          Unification succeeds
        Else
          Get flags $A$ and $B$ from the top of $W_A$ and $W_B$.
Step3:  Case1: (both $A$ and $B$ are $n$ )
          Get elements $E_A$ and $E_B$ from the top of $L_A$ and $L_B$.
          If (both $E_A$ and $E_B$ are function symbols $F_A$-$K_A$ $F_B$-$K_B$) then
            If ($F_A = F_B$ and $K_A = K_B$ ) then
              Push flag $n$ into $W_A$ and $W_B$ repeated $K$ times
              Go to Step2;
            Else
              Unification fails
          Else if ($E_{A/B}$ is $F$-$K$ and $E_{B/A}$ is a variable $V_i$) then
            Bind $V_i$ to $F$-$K$
            Push $V_i$ as an expanded variable flag into $W_{A/B}$
            repeated $K$ times.
        Push flag $n$ into $W_{B/A}$ repeated $K$ times
        Go to Step2.
    Else if (both $E_A$ and $E_B$ are variables) then
        Bind these variables each other
        Go to Step2.
Case2: ($A/B$ is $n$ and $B/A$ is $V_i$)
    Get an element $E_{A/B}$ from the top of $L_{A/B}$.
    If ($E_{A/B}$ is $F$-$K$) then
        Bind the variable $V_i$ to $F$-$K$
        Push $V_i$ into $W_{B/A}$ repeated $K$ times
        Push $n$ into $W_{A/B}$ repeated $K$ times
        Go to Step2.
    Else if ($E_{A/B}$ is a variable $V_j$ bound to $F$-$K$) then
        Bind the variable $V_i$ to $V_j$
        Push $V_j$ and $V_i$ into $W_{A/B}$ and $W_{B/A}$ repeated $K$ times
        Go to Step2.
    Else if ($E_{A/B}$ is a unbound variable $V_j$) then
        Bind the variable $V_i$ to $V_j$
        Go to Step2.
Case3: ($A$ is $V_i$ and $B$ is $V_j$)
        Bind each variables
        Go to Step2.
```

A variable bound to a function symbol F-K is expanded in the FIFOs as K expanded variable flags. Each element that must be bound to the variable is bound late corresponding to the flags. Figure 7 shows unification between [p-2, V_1, g-1, V_2] and [p-2, f-2, g-1, a-0, b-0, c-0] .

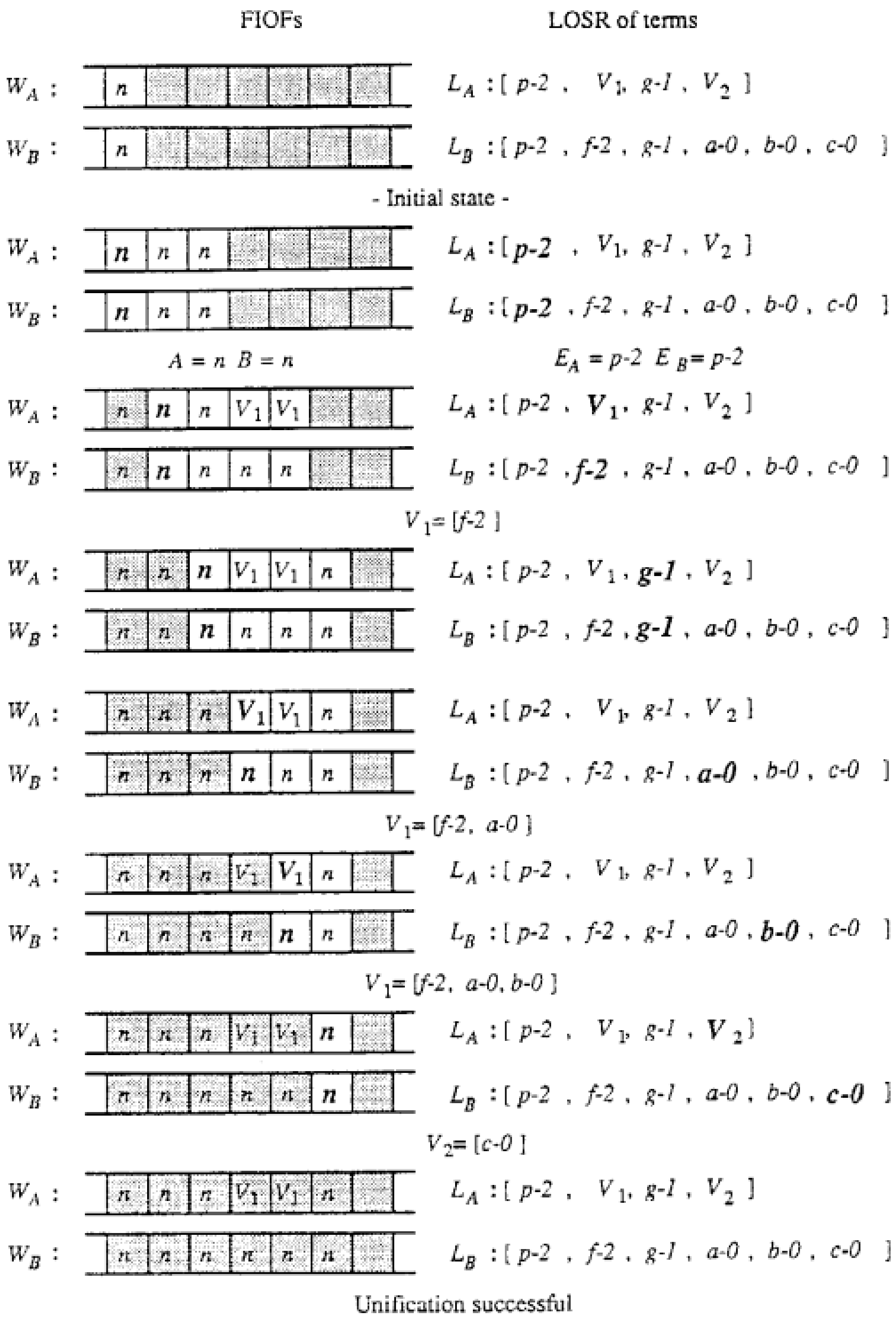

Figure 7. Unification process on LOSRs

3.4. Trie structure traversal

The unification algorithm is applied to trie-structure traversal for retrieving term tuples. Each node of the trie structure is treated as an element of L_A in the algorithm. L_B is the LOSR of a search condition term. Backtracking is required because the unification for branches of the trie structure starts from the middle of L_A and L_B.

Information for backtracking must be pushed into a stack when the unification algorithm examines an element having alternatives, i.e., a branch. The address of the E_B cell and of the alternate cell for E_A, the head and tail of FIFOs W_A and W_B, and the level of variable bindings at a backtracking point must be known. Each element bound to variables has its level increased by one at each backtracking point. The environment of variable bindings is recovered by unbinding elements whose level is higher than the level of the backtracking point. The unmeshed areas of FIFOs in Figure 7 are working spaces indicated by the head and tail. Because FIFO contents remain after the tail passes the element, the pointers to the head and tail of FIFOs are enough to revert the FIFOs to the previous state.

When backtracking occurs, the information at the top of the stack is popped and the FIFOs and variable bindings revert to the backtrack point. The next comparison starts from the cell address stored as the E_B cell and alternate cell for E_A.

When the unification succeeds at some leaf of the trie structure, the pointer for the corresponding term tuple and the variable bindings are retained as a retrieval result. This is another benefit of this method, deriving the variable bindings by indexing.

4. Performance evaluation

Using an RBU prototype implemented in C, we compared the search and updating speeds with those of the Quintus Prolog interpreter to evaluate the indexing method. Although the functions of these two systems are different, the speed of the Prolog system can be regarded as a general indicator of performance.

We prepared four types of term relations for evaluation. Examples of the index for each type are given in Figure 8. Each type has the following characteristics:

Type *A*: All terms have an identical function symbol in each element except the tail. When indexing is applied, the trie connected to only one hash entry for the term relation is a highly skewed tree whose branches are only its leaves.

Type *B*: Terms have one of 16 function symbols as their first element, and all other elements but the tail have an identical function symbol. When indexing is applied, all 16 tries connected to each hash entry are almost the same size and shaped the same as the trie in type *A*.

Type *C*: All terms have an identical function symbol as their first element, and the other elements have one of eight function symbols recursively. When indexing is applied, one hash entry contains a balanced trie. Each node of this trie has eight branches.

Type *D*: Terms have one of 16 function symbols as their first element, and the other elements have one of eight function symbols recursively. When indexing is applied, all 16 tries connected to each hash entry are almost the same size and shaped the same as the trie in Type *C*.

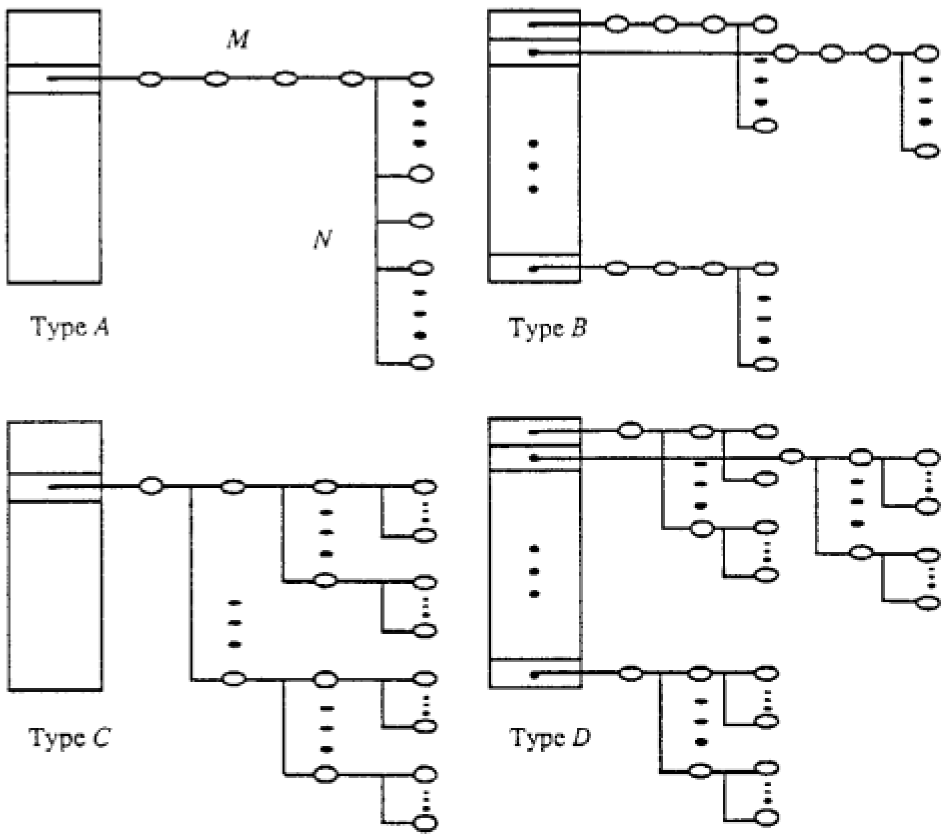

Figure 8. Index appearance for four type relations

We used five relations having 50, 250, 500, 750, and 1000 term tuples for each type. The search speeds of the Prolog interpreter and unification-restriction with and without indexing are compared for each relation. Figure 9, 10, 11, and 12 show the results for types A, B, C, and D.

Figure 9 shows the effect of using only a trie structure in the worst case. Hashing does not work for this type of relation. Because the tuple count (N) of backtracking occurs at the leaves of the trie structure, the search time increases with the tuple count, both with and without indexing. Differences in search speeds with and without indexing cause the number of comparison between the elements.

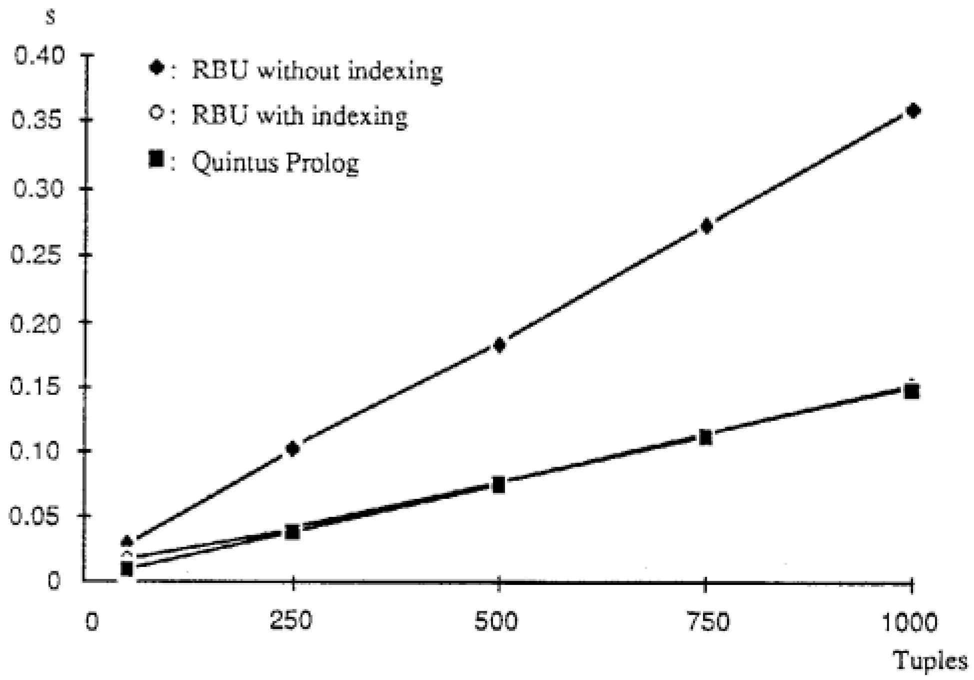

Figure 9. Search speeds for Type *A*

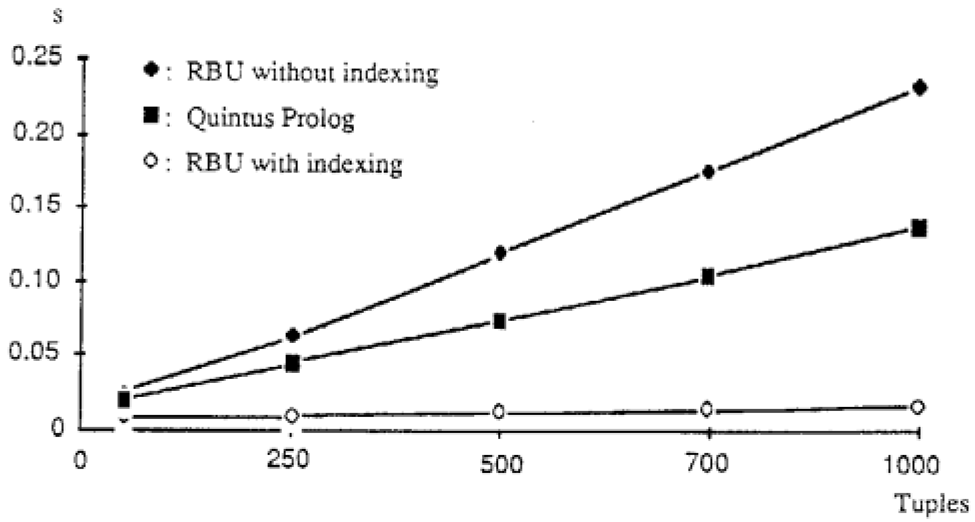

Figure 10. Search speeds for Type *B*

Figure 10 shows the effect of hashing for this case. The search speed of unification-restriction with indexing hardly changes, regardless of the number of tuples. The combination of the trie structure and hashing is very effective.

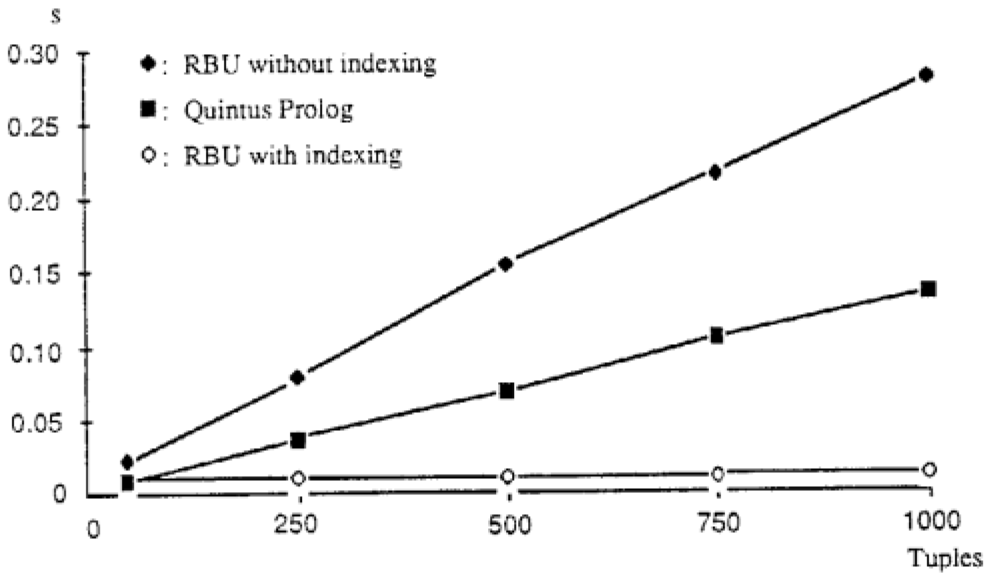

Figure 11. Search speeds for Type *C*

Figure 11 shows the effect of the trie structure for type *C*, in which hashing does not work. Because backtracking is reduced by branching, the search speed is almost independent of the tuple count *N*. Hashing does not significantly improve the search speed (Figure 12).

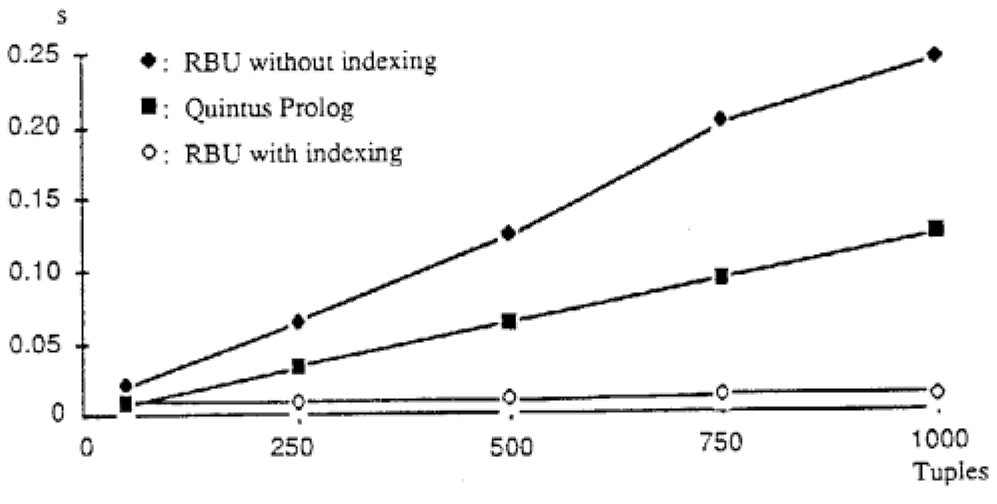

Figure 12. Search speeds for Type *D*

Figure 13 compares the tuple insertion speeds of the two systems. Tuple insertion using RBU takes only about one sixth the time of a Prolog *consult* operation. The overhead for making an index for a term relation is small compared to that the insertion time.

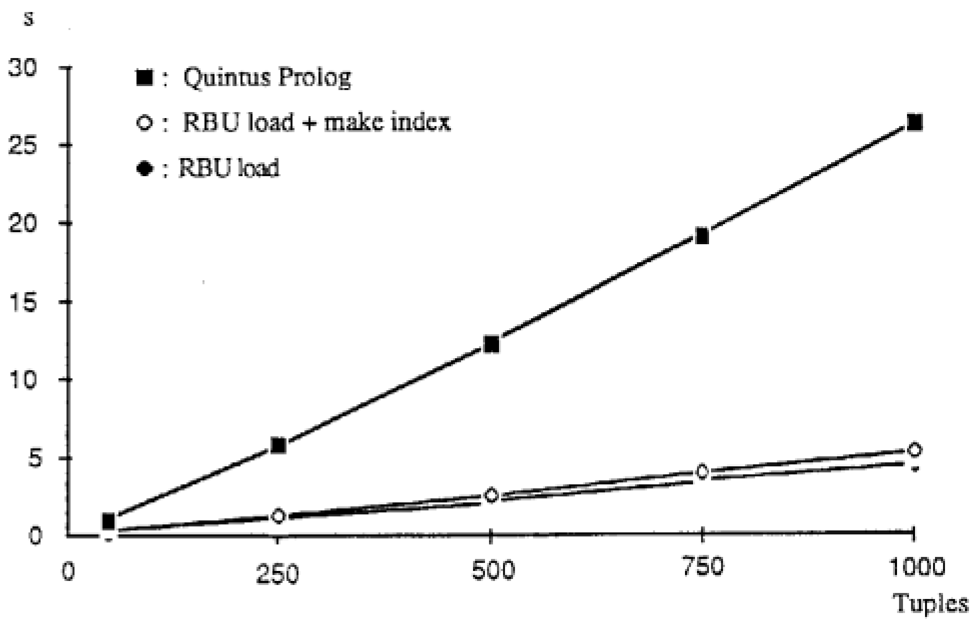

Figure 13. Insert speed comparison

5. Conclusions

As indexing for RBU systems, we proposed using trie structures for indexing of term relations and showed that the trie structure reduced comparisons between elements and backtrackings in the RBU operation. Combining the trie structure with hashing is most effective for terms in both varied and similar forms. The LOSR of terms in the trie structure can be applied to denote differences of term structure after checking as few nodes of the trie structure as possible. We also proposed a unification algorithm for the LOSR using FIFOs and trie structure traversal with a stack.

The search and updating speeds of our prototype were evaluated and compared with those of a Prolog system. Clearly, indexing using hashing and trie structures is very effective in speeding up term retrieval and requires little overhead for maintaining indexes in updating.

Acknowledgments

We thank Dr. Hidenori Itoh, manager of the Third Laboratory of ICOT, and the laboratory staff for their constructive discussion and Mr. Hiromu Hayashi, manager of the Artificial Intelligence Laboratory, Fujitsu Laboratories Ltd., for his helpful suggestions. We also thank Mr. Mark Feldman of the artificial intelligence laboratory for his helpful advice on the refinement of the paper.

References

[Chang and Lee 73] C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Itoh 87] H. Itoh, T. Takewaki, and H. Yokota, "Knowledge Base Machine Based on Parallel Kernel Language," *Proc. of 5th International Workshop on Database Machines*, pp. 15-28, 1987.

[Knuth 73a] D. E. Knuth, *The Art of Computer Programming*, 3, Sorting and Searching, Addison-Wesley, 1973.

[Knuth 73b] D. E. Knuth, *The Art of Computer Programming*, 1, Fundamental Algorithm, Addison-Wesley, 1973.

[Lloyd 84] J. W. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, 1984.

[Morita 86] Y. Morita, H. Yokota, K. Nishida, and H. Itoh, "Retrieval-By-Unification Operation on a Relational Knowledge Base," *Proc. of 12th International Conference on VLDB*, pp. 52-59, 1986.

[Ohmori 87] T. Ohmori and H. Tanaka, "An Algebraic Deductive Database Managing a Mass of Rule Clauses," *Proc. of 5th International Workshop on Database Machines*, pp. 291-304, 1987.

[Ullman 82] J. D. Ullman, *Principals of Database Systems*, 2nd ed., Computer Science Press, Potomac, Md., 1982.

[Wada 87] M. Wada, Y. Morita, H. Yamazaki, S. Yamashita, N. Miyazaki, and H. Itoh, "A Superimposed Code Scheme for Deductive Databases," *Proc. of 5th International Workshop on Database Machines*, pp. 569-582, 1987.

[Yokota 86] H. Yokota and H. Itoh, "A Model and Architecture for a Relational Knowledge Base," *Proc. of the 13th International Symposium on Computer Architecture*, pp. 2-9, Tokyo, 1986.

[Yokota 88] H. Yokota, H. Kitakami, and A Hattori, "Knowledge Retrieval and Updating for Parallel Problem Solving," submitted to the International Conference on Fifth Generation Computer Systems 1988.