New method for Symbol Resolution

Sinisa Milivojevic*

Department of Computer Science, An independent contractor, Larnaca, Cyprus

Review Article

Received: 01-May-2025, **Manuscript No.** GRCS-25-165084; **Editor assigned:** 02-May-2025, PreQC No. GRCS-25-165084 (PQ); **Reviewed:** 12-May-2025, QC No. GRCS-25-165084; **Revised:** 20-May-2025, Manuscript No. GRCS-25-165084 (R); **Published:** 27-May-2025, DOI: 10.4172/2229-371X.16.2.003

***For Correspondence:**
Department of Computer Science, An independent contractor, Larnaca, Cyprus
E-mail: sinisa@cytanet.com.cy

Citation: Milivojevic S. New method for Symbol Resolution. J Glob Res Comput Sci. 2025;16:003.

Copyright: © 2025 Milivojevic S. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution and reproduction in any medium, provided the original author and source are credited.

ABSTRACT

This paper provides the result of an extensive research that resulted in algorithm that provides much more efficient searching through Symbol Tables containing reserved words, keywords, operators, functions and similar. A new (so far most efficient) method has been provided which has been thoroughly tested. That method required the utilisation of the hashing. Many hashes have been tested, of which only few could render required results. This resulted in the the invention and development of new hashes, by the author. All necessary information from this research is presented including characteristics of two known algorithms, and including the efficiency of the utilised hashes and a total speed of execution for the entire symbol table, on two different CPUs.

Keywords: Symbol tables; New search method; Sparse vector; Hashing
CCS CONCEPTS: Computing methodologies, Mathematics of computing, Software and its engineering.

INTRODUCTION

All compilers and interpreters, have their own set of symbols, which comprise reserved words, keywords, function names and similar. In order to recognise that set from other literals, there is a need to have fast method to recognise whether a separate literal used is one of its symbols. That is exactly what is meant by the term “symbol resolution”. There were many methods used in the past, one of which is hashing. The crux of this paper is dedicated to the description of methods that use a family of hashes, which are generated with both byte and bit operations on the symbols. That family of hashes is better known as xxhashes.

DESCRIPTION

Most of the compilers / interpreters are storing their symbols as a packed array of structures, usually written in C programming language. These structures usually contain the string for the symbol literal, its length and the fields for the

attributes which provide more detail on what are the characteristics of the symbol. This paper presents the research that has developed a method that involved the usage of the sparse and imperfect vectors that store indices for the entries in the above described packed symbol vector. This method is called indirect indexing, since valid sparse, imperfect vector whose elements contain valid and invalid values, since this is a sparse vector. Each of the valid values is the index in the packed, perfect vector that stores all symbols.

To put it in a formula, it looks like this:

Symbol= Vector_of_Symbols [Sparse_Vector [x]];

where x is (obviously) the index in the sparse vector.

This approach has necessitated that a method is invented for the calculation of the indices (denoted as 'x' in the above formula) in the sparse vector containing values that point to the valid entries in the packed vector containing symbols. These symbols are reserved words, keywords, function names, operators et cetera.

A very strict criteria has been set that would necessitate that the above indices are calculated only from the string literals of these symbols, but in the manner that the sparse vector contains only one index for each separate symbol. That is the crux of the method that was developed.

This is better visualised in you start from an example. If you look at one of the latest MySQL include file, lex.h, you will notice that there are 845 symbols. However, of these only 803 symbols are relevant for this research. Those 803 symbols include all the reserved words, keywords, operators and functions. The rest are optimiser hints, that can not be treated in the same way, because they contain duplicates, which is incompatible with any symbol resolution. Those are also treated totally differently in the current MySQL code.

The tests were executed on the last 8.0 release available at the time. Since then, new versions and releases have come out, with some changes in the symbol table. All those were tested too and those were very similar to the the original results (presented in the next chapter).

Using imperfect, sparse vector necessitated an algorithm that would find the smallest possible vector that would resolve the symbols. Hence, another objective of this research was to use an optimum-seeking algorithm that would run symbol resolution until they would all fit in the vector, without any duplicates. It was decided that this vector should have no more then 64 K entries. For that purpose a separate array was created, which contains all primes between 16 K and 64 K. Those primes were used for the searching of the smallest vector, where all the entries (in this case 803 of them) would fit, without duplicates. This vector was then generated as an obligatory part of the algorithm results. That also means that the hashing algorithm used should provide a 16-bit value as a final result. This was easily achieved by obtaining a 32-bit hash and calculating the remainder of that value with the size of the sparse vector.

In such a system, each sparse, imperfect vector should have been initialised. In this system, 0 (zero) would be a valid, although highly improbable entry, since the last prime, in the above mentioned array, is definitely smaller then 64K. Hence, UINT16_MAX would be totally impossible entry, as the maximum size of the vector would be smaller. That is a reason why every entry in the sparse vector would be initialised with a value of UINT16_MAX. That has made searching algorithm easier to design. Simply, if calculated index for the sparse vector would contain a value of UINT16_MAX, then that entry is valid and it was written into that spot. If the value found was smaller, then it would

indicate that the algorithm has hit upon a duplicate entry. The code would then proceed with a new, larger vector, until a vector was found that would accommodate all symbols, in this case 803 of them. The sizes of vectors, for the usable hashes, are also included in the results.

It should be mentioned that the results of each successful algorithm included the generated sparse vector containing all entries including the valid ones (in this case 803 of them), vector's size (as number of entries), some info on the successful completion and the speed of the resolution of all 803 symbols, in microseconds. This also means that the runtime code only needed the vector and the xxhash algorithm which returns the symbol. This also means that the size of the product will include additional storage for the array, whose size is circa 100 Kbytes. For the contemporary computers that is a small price to pay. It was discovered during research that those hashes that were successful, managed to do with less then 64 K, while the others could not fit even in much larger vectors. This is due to the fact that the unsuccessful algorithms have produced duplicates even with much larger vector sizes. Some produced duplicates even with 32-bit vector sizes.

It should be pointed out that the speed and efficiency of searching and generating sparse vector, and filing it up with indices in the right places, is not relevant. That is because this step is performed during compiling of the SQL interpreter (or any other interpreter or compiler). The important efficiency and speed is the one in runtime, which deals only with symbol resolution.

At the very start of the research it was clear that classical hashes, that only use byte arithmetics, can not produce desirable results. Hence, this is a reason why it was mandatory that it necessitate usage of hashes which use a combination of both byte and bit operations. Shortly, we use the existing term xxhashes for those hashes that use operations on both bytes and bits.

The research has tested over 30 (thirty) of the existing xxhash algorithms, but found only 4 (four) of those that were successful in symbol resolution.

Further analysis showed that many xxhash algorithms required initialisation with some value. Mostly, the authors suggest the use of the length of the previous symbol for the initialiser. Some other suggest that initialisation is done by the xxhash obtained from the previous symbol calculation. That approach is impossible with compilers / interpreters, since those have their syntax, which is very varied and can combine symbols in many different ways. In that context there is no such thing as a previous or next symbol.

Also, many of the xxhash algorithms required that the final calculation involves a bitwise operation (usually XOR) on the obtained result. This meant that a method had to devise the usage of some seed.

In order to solve both of the problems, two different algorithms were designed for the random generation of two sets of seeds, one for the initialisation of the xxhash and the other for the final bit operation. One algorithm was used for the generation of the array of 7 (seven) random seeds and second, completely different, algorithm was used for the generation of the array of 2099 (two thousand and nineteen nine) random seeds. It is not a coincidence that both of these numbers are primes.

Since only 4 (four) of the existing algorithms lead to the desired results, it was necessary that this research also develops a set of new hash algorithms. In total, 8 (eight) new xxhashes were designed, some of which performed significantly better then the known ones. It must be pointed out that, of those 8 (eight) new xxhashes, 3 (three) were loosely inspired by the existing xxashes. Word "inspired" was used because those algorithms required heavy changes

in order to meet the above described criteria and provide usable results. This practically created new algorithms, with entirely different efficiency .... The original algorithms will be mentioned in the code listings of the new xxhashes.

As already mentioned, the example on which the algorithms were developed and tested was SQL interpreter and more precisely the symbols in MySQL implementation of SQL. One of the characteristics of SQL is that language is case- insensitive. Since symbols are all uppercase in MySQL implementation, that required the addition of the code that converts all letters into uppercase. This could have been achieved by table lookup, but finished by using a simple ternary operator. This code was not included in the presentation of the new xxhashes, nor was it included in the calculation of the algorithm efficiency. The reason for that is that, since SQL is case-insensitive, a programmer or query designer can write symbols in the queries with all possible variants of the letter cases.

In order to test the validity of the final result, many tests were included. First of all, the above described vector and the indices were calculated with upper-case symbols, while testing was done with lower-case symbols. The sparse vector was initialised with UINT16_MAX values, so if the hash returned that value as the valid index, then entire calculus failed. After that test, found symbol length was compared with input symbol length and two strings were then compared for equality.

PRESENTATION OF THE RESULTS

Results presented in this paper contain all of 12 (twelve) xxhashes, but, for the comparison purposes, these also include two other, historically important, methods. First one is used in the current MySQL implementation, and it is based on the digital search method. It can be found in the MySQL source code directory sql/ as a file sql_lex_hash.cc.

Beside the above method, another existing algorithm was included and that is a double hashing algorithm. It was implemented during this research, but again, only for comparing their efficiency and speed with the new method.

All of results are presented in the following table. Table has the following columns:

- Title of the method.
- The algorithm used.
- Reference for the algorithm, for the existing algorithms only. These are numbers corresponding to Bibliography chapter.
- The algorithm eThciency, where ’n’ stands for the number of bytes (characters). Double hashing, although, has difterent eThciency calculus.
- Sparse vector size.
- Total time (in microseconds) for the resolution of all 803 (eight hundred and three) symbols on Intel i5 CPU @ 4.0 GHz, by the method described in the previous chapter.
- Total time (in microseconds) for the resolution of all 803 (eight hundred and three) symbols on ARM, actually M3 CPU @ 4.0 GHz, by the method described in the previous chapter.

A table containing all of the data for those seven columns is presented below, where the title of each column is fully explained in the above list.

Table 1. Presentation of the results.

Method title	Algorithm	Reference	Algorithm efficiency	Sparse vector size in number of entries	Total time on i5 @ 4 GHz in µsec	Total time on M3 @ 4 GHz in µsec
Digital search	Tries	[1]	8*n + 8	-	36	46
Double hashing	hash	[2]	m/n(1/(1-n/m))	m=2753, n= 67	46	44
Bob Jenkins 1	xxhash	[3]	8*n + 10	51347	23	15
Bob Jenkins 2	xxhash	[4]	5*n + 35	43331	22	16
Paul Hsieh (modified)	xxhash	[4]	5*n + 17	54401	21	16
fnv_32a_str (modified)	xxhash	[5]	4*n + 2	49581	16	11
Sinisa 1	xxhash	new	4*n + 9	40823	24	16
Sinisa 2	xxhash	new	5*n + 9	45061	20	13
Sinisa 3	xxhash	new	6*n + 9	45317	25	16
Sinisa 4	xxhash	new	4*n + 7	52951	25	12
Sinisa 5	xxhash	new	4*n + 4	54401	20	13
Sinisa 6	xxhash	new	3*n + 6	42397	19	14
Sinisa 7	xxhash	new	4*n + 1	45751	17	16
Sinisa 8	xxhash	new	5*n + 5	50951	19	17

In the above table, 3 (three) runs were necessary on Intel i5 to provide average results for the speed of resolution in µsec, and with a very small standard deviation. On the M3, however, 12 (twelve) runs were necessary to get results with the approximately same reliability, meaning with the approximately same and acceptable standard deviation.

C at the above table also shows that speed of M3 processor is significantly faster than the one of Intel's i5. That is, actually, expected behaviour. What was unexpected is that rankings in the speed of the algorithms were very different between two CPU's. That could have been explained only with different relative costs of different operations on the bytes and bits for different CPUs.

Intel's CPU has higher costs for division and remainder then other operations, including byte operations like addition and multiplication. ARM CPU has smaller variations among different operations and it proved much trickier to get reliable costs of operation. This started a new research with the aim of establishing more reliable operation costs for both CPUs. When this research is designed and finished, its results will be provided in another paper.

PRESENTATION OF THE NEW HASHES

In this chapter, we shall present hashes in the manner that they were used. They are simply returning the index in the sparse vector, while doing some necessary checks. Hashes that are used in production do not need to do any checks. They should just return the symbol, as in the formula already provided above, which we are repeating here:

Symbol= Vector_of_Symbols [Sparse_Vector [x]];

A hash function can return a pointer to the symbol. Some program opt to check for error, in which case a NULL is

returned for the pointer.

As it is already indicated, hashes presented here return that 'x' from the above formula. You will also note that these hashes return the unsigned 32-bit unsigned integer. So, for the implementation of this method, that value is converted to 16-bit unsigned integer. It is done by calculating the remainder of the 32-bit value by the sparse vector size. This is already explained at the start of this article. All the hashes, as well as many programs written during this research were coded in C standard 2017 [6]. C 2023 was, unfortunately, not finished when the research started. It is my personal opinion that C language, especially it' latest standards, is the best tool for system programming. C++ is not suitable for that purpose, in my humble opinion.

Also, as mentioned above, conversions to uppercase are omitted in the presentation of new hashes.

So, here are the source codes for the new hashes. The code provided here is, from now on, in the public domain.

SOURCE CODE

```
uint32_t sinisa_one_hash (uint16_t length,  char key[length]) { register
   uint32_t                hash=0,i , old_x=seeds[length % 7];
   for (i=0; i<length; ++i)          {
      register int8_t  x= (int8_t) key[i];
      hash= (hash ? hash : my_seeds[(old_x*(uint32_t)x) % 2999]); hash
      += (uint32_t)x  <<  (old_x % 11);
      old_x=x;
   }
   return hash ^ my_seeds[hash % 2999];
}

uint32_t sinisa_two_hash (uint16_t length,   char key[length]) { register
   uint32_t hash=0, i, old_x=seeds[length % 7];

   for (i=0; i < (uint32_t)length; ++i) { register
      int8_t x= (int8_t) key[i];
      hash= (hash ? hash : my_seeds[(old_x*(uint32_t)x) % 2999]);
      old_x=x;
      hash = ((hash << 5) + (hash >> 27)) +  (uint32_t) x;
   }
    hash ^= my_seeds[hash % 2999];
    return hash;
}

uint32_t sinisa_three_hash (uint16_t length, char key[length]) { register
   uint32_t hash= 0, old_x=seeds[length % 7];

   for (register uint16_t i= 0; i < length; i++) { register uint8_t x=
      (uint8_t) key[i];
```

```
        hash= (hash ? hash : my_seeds[(old_x*(uint32_t)x) % 2999]);
        hash+= (uint32_t)x ^ my_seeds[i*x*old_x % 2999];
        old_x= x;
    }
    hash ^= my_seeds[hash % 2999];
    return hash;
}

uint32_t sinisa_four_hash (uint16_t length, char key[length]) { register
    uint32_t hash=0;
    for (register uint16_t i= 0; i < length; i++) { register int8_t  x=
        (int8_t) key[i];
        hash= (hash ? hash : my_seeds[(length*(uint32_t)x) % 2999]) ; hash +=
        (uint32_t)(x)  ^  seeds[x % 7];
    }
    hash ^= my_seeds[hash % 2999];
    return hash;
}

        uint32_t sinisa_five_hash (uint16_t length,  char key[length]) { register
            uint32_t hash, i;
            for (hash= (uint32_t) length, i=0; i < (uint32_t)length; ++i) { register int8_t
                x= (int8_t) key[i];
                hash = ((hash << 5) ^ (hash >> 27)) ^ (uint32_t) x;
            }
            hash ^= my_seeds[hash % 2999];
            return hash;
        }

        uint32_t  sinisa_six_hash (uint16_t length,  char key[length]) {
        /* Loosely inspired by Bernstein hash [4] */
            register uint32_t  hash = 0, old_x= seeds[length % 7]; for
            (register uint16_t i=0; i<length; ++i) {
                register int8_t  x= (int8_t) key[i];
```

```
        hash= (hash ? hash : my_seeds[(old_x*(uint32_t)x) % 2999]);
        old_x=x;
        hash = hash * 55 + (uint32_t)x;
    }
    hash ^= my_seeds[hash % 2999];
    return hash;
}

uint32_t sinisa_seven_hash (uint16_t length, char key[length]) {
/* Loosely based on Zobrist hash [4] */ register
    uint32_t hash, i;
    for (hash=length, i=0; i<length; ++i) {
        register uint8_t x= (uint8_t) key[i]; hash
        ^= x*my_seeds[i];
    }
    return hash;
}

uint32_t sinisa_eight_hash (uint16_t length, char key[length]) {
/* Loosely based on CRC hash [4] */
    register uint32_t hash=0, i, old_x=seeds[length % 7]; for (i=0;
    i<length; ++i) {
        register uint8_t x= (uint8_t) key[i];
        hash= (hash ? hash : my_seeds[(old_x*(uint32_t)x) % 2999]); hash
        += (uint32_t)x;
        hash += (hash >> 11);        hash ^= (hash <<  7);
        old_x= x;
    }
    return hash ^= my_seeds[hash % 2999];
}
```

REFERENCES

1. Donald E. Knuth. The Art of Computer Programming: Sorting and searching, Volume 3. Digital searching. Addison-Wesley. 1998.
2. Sedgewick R. Algorithms in C. Addison-Wesley. 2011.

3. Jenkins B. Jenkins hash function. 1997.
4. Jenkings B. Dr. Dobb's Journal. 1997.
5. Vo P, et al. FNV Hash. 2013.
6. Gusted J. Modern C. Manning. 2023.