

# Trying to outperform a well-known index with a sequential scan

Jan Hentschel, Thomas Meyer, Thomas Rommel

Otto-von-Guericke-University

Universitätsplatz 2

39106 Magdeburg, Germany

{Jan.Hentschel, Thomas.Meyer, Thomas.Rommel}@st.ovgu.de

## ABSTRACT

The string similarity search is an important research area. It enables applications to accept input errors and to detect similarities between strings. This kind of search contains the string similarity search problem. The time to solve this problem depends on the number, the length and the size of the alphabet of the data to search. It is possible to divide the data in data of natural language and data of non-natural language. In data of natural language, this paper analyzes a set of names of cities all over the world. For non-natural language data the paper uses reads from human genome. This paper wants to analyze, if it is possible to outperform an index-based search by a sequential search algorithm. The evaluation shows, that the index-based search has a higher performance on the human genome reads, but not on the geographical names.

## General Terms

Algorithms, Performance

## Keywords

Similarity Search, string similarity, edit distance, string matching

## 1. INTRODUCTION

Beside of the ability of an exact string search, nowadays most applications needs a function for similarity search. Applications search in data of natural language strings or non-natural language strings. Natural language strings are human readable words. An application that searches in this data has to be tolerant against input errors, because the user could make typing errors or errors in the spelling of a word. Nevertheless the application has to find all relevant results in the data [7, 10]. Applications for non-natural languages are for example applications, which search for similar human genome reads [1].

It is necessary to solve the string similarity search problem, to perform the similarity search in both areas. The input for this problem is a query string, a similarity measure and set of data to search. To solve this problem, the application has to determine all the datasets, similar to the query with respect to the similarity measurement [2].

Permission to make digital or hand copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT '13, March 18 – 22 2013, Genoa, Italy

Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

This paper uses the unweighted edit distance as similarity measurement. The used datasets are given by the “String Similarity Search/Join Competition” of the EDBT/ICDT 2013 Joint Conference. The datasets are names of cities as natural language data and human genome reads for non-natural language data. The paper will try several approaches to solve the problem in an efficient way.

The aim of this work is to find out, if a sequential solution of the string similarity search problem on the city-names and the genome reads can outperform an index-based solution in its execution time. Both algorithms will be improved by several approaches.

The paper is structured as follows. The second chapter will explain important terms and introduce several existing approaches from scientific literature. The third and fourth chapter will present approaches to improve the sequential and index-based search algorithms. The fifth chapter will evaluate the implemented applications for sequential and index-based search by measuring the execution time. The sixth chapter summarizes the work and mentions possible future work.

## 2. FOUNDATIONS AND RELATED WORK

This chapter explains important terms and gives a look to related work in scientific literature. At first the string similarity search problem will be explained. After that the edit distance is defined and calculated by a sample. Furthermore the chapter explains two approaches of existing related work. Finally it summarizes the preliminary considerations and forms two hypothesis.

### 2.1 String Similarity Search Problem

The string similarity search problem returns each string of a set of strings that has at least a given similarity for a given query. Let  $q$  be query string,  $X$  a set of strings,  $ed$  a distance function and  $k$  the threshold value for the distance [3, 9]. A string  $x$  is part of the result, if it fulfills the following conditions:

$$x \in X \text{ and } ed(q, x) \leq k \quad (1)$$

To solve this problem, the application has to determine each string  $x$ , that does not exceed the threshold  $k$ . The competition rules give the edit distance as similarity metric. This distance measurement will be explained in the following.

### 2.2 Edit Distance

The edit distance  $ed(x, y)$  of two strings  $x$  and  $y$  is the minimal number of insert, delete or replace operations to transform  $x$  to  $y$ . Two strings are within edit distance  $k$ , if  $ed(x, y) \leq k$  [3, 9]. The following example shows the calculation of the edit distance for two strings. The input are the string “AGGCGT” and “AGAGT”. The operations insert, delete and replace are available to transform the strings. Each execution of an operation has costs of 1. The

computation of the edit distance uses a matrix  $M_{0...l_x, 0...l_y}$  with  $(l_x + 1)$  rows and  $(l_y + 1)$  columns. Let  $l_x$  be the length of string  $x$  and  $l_y$  be the length of string  $y$ . The symbol at position  $i$  of a string  $y$  is  $y_i$ .  $M_{i,j}$  is the entry of the matrix at row  $i$  and column  $j$ . The entry  $M_{i,j}$  is calculated as follows [5]:

$$M_{i,0} = i, M_{0,j} = j \quad (2)$$

$$M_{i,j} = \text{Falls}(x_i = y_j) \text{ dann } M_{i-1,j-1} \quad (3)$$

$$\text{sonst } 1 + \min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}) \quad (4)$$

Figure 1 shows the computation of the edit distance between the two strings. The last step is to calculate the entry in  $M_{l_x, l_y}$ , which will contain the edit distance between string  $x$  and string  $y$ . In this step the calculation uses condition (3), because both strings have the same symbol at the last position. So, it will use the value of  $M_{5,4}$  in entry  $M_{6,5}$ . The calculated edit distance is 2.

This paper will use different approaches to calculate the edit distance, in order to solve the string similarity search problem. It will discuss filters, the use of an index and faster string similarity algorithms. The basis and reference for the implementation of this paper are related publications. These approaches will be explained in the following.

### 2.3 Related Publications

This subchapter will introduce several publications related to the string similarity search problem. The approaches to solve the problem will be explained in a short way.

Human genome reads consist of very long strings. Rheinländer et al. show, that former use of similarity operations is very limited and inefficient. That's why they developed a new efficient algorithm for similarity search named PETER. This algorithm supports Hamming and edit distance. PETER uses a compressed prefix tree for indexing data. Very long suffixes are stored in a file, in order to hold the tree in main memory. The tree stores information about the minimal and maximal length of the strings and the frequency vector, a vector with the number of occurrences for each symbol in the string. These information enable an early filtering of the results. The algorithm can stop searching in a branch, that can not contain correct results [8].

Navarro et al. consider the following problems. The first problem is the size of the index using a suffix tree. The second problem is the exponential dependency of the calculation effort to the length of the strings and the edit distance. To solve the first problem, they use a suffix array. This has the advantage, that the index can only reach a maximum size of four times of the number of strings. Furthermore suffix arrays are with exception of very short strings faster than suffix trees. The second problem is solved by splitting the query string and later integrating the particular results. It is possible to reduce the exponential dependency of string length and edit distance by this method.

All problems and experiences of that approaches are used as basics for chapter 3 and 4.

### 2.4 Preliminaries

Human genome reads and city-names have very specific properties. So it is useful to make preliminary considerations about efficient approaches. Human genome reads have a large string length and a very small alphabet. On the other hand city-names have usually a smaller string length, but a larger alphabet.

The alphabet can vary by using different character sets and languages. For example, adding the Chinese language will enlarge the alphabet by adding all the symbols.

These properties enable the construction of two hypotheses for efficient string similarity search. The paper will analyze these two hypotheses.

- An index-based solution will have a higher performance on human genome reads than a sequential solution, because of the large string lengths.
- An optimized sequential solution can have a higher performance on the city-names than an index-based solution, because of the smaller string lengths.

The next chapters 3 and 4 will discuss approaches to improve the sequential and the index-based solution for the string similarity search.

## 3. RUDIMENTS FOR IMPROVING THE SEQUENTIAL SOLUTION

This chapter will introduce several approaches to improve the performance of the sequential solution. Starting with an initial solution, it focuses on improving the calculation of the edit distance, using value and reference semantic, simple data types and program methods, parallelism and the management of parallelism. Chapter 5 will evaluate these approaches. At first the initial solution is discussed and based on that the improvements will be explained.

### 3.1 Base Implementation

The first implemented solution should work and solve the string similarity search problem without any errors. The used programming language is C++. This language has the advantage to focus on runtime and memory efficiency. Some other programming languages like Java focus on simplicity and clarity of the language instead of performance. So, it is not useful to use such a language [11].

The procedure of the initial implementation consists of reading the query and data sets, calculating the edit distance and writing the results to a file. The results of this solution provide the reference to verify the correct results of the following solutions for the sequential and the index-based search. After verifying the results, the execution times will show, if the implemented approach could improve the previous solution or not. If the implemented approach improved the performance it will be part of the following solutions.

### 3.2 Faster Edit Distance Calculation

As described in chapter 2, the calculation of the edit distance is very costly. Thus, the calculation should be improved by reducing the calculation time.

The first idea tries to avoid the calculation of the edit distance:

- Consideration of string length: Let  $d$  be a delta between the length  $l_x$  of string  $x$  and the length  $l_y$  of string  $y$ , calculated as:

$$d = |l_x - l_y| \quad (5)$$

Delta  $d$  will be compared with the given maximum edit distance. If  $d$  is greater than the given edit distance, the application does not have to compute the edit distance. Because the edit distance will be greater than the given

maximum edit distance. Thus, the current string cannot be part of the correct results.

- Exact search: The given edit distance zero describes the case of exact search. It is not necessary to compute the edit distance. It is more efficient to compare each letter of string  $x$  and string  $y$ , to determine if the string is a result for the query.

If it is not possible to avoid the calculation, the solution should try to abort the calculation early. This approach looks as follows:

- Consideration of diagonal: The diagonals of the edit distance matrix  $M$  have special properties. The diagonal in Figure 1 shows, that the value of the edit distance is rising or stagnating. This is a consequence of the rules for calculating the edit distance described in chapter 2. The calculation depends on one of the three surrounding entries on the left and upper side. So there are overall three diagonals and errors can be reduced by one of the neighbor diagonals. Errors on the diagonal with entry  $M_{l_x, l_y}$  cannot be reduced. At the end of the calculation the entry  $M_{l_x, l_y}$  will contain the edit distance between string  $x$  and string  $y$ . Considering these facts, there are the following two conditions for aborting the calculation:

$$\text{If } (l_x \geq l_y \text{ and } i - d = j \text{ and } M_{i,j} > k) \quad (6)$$

$$\text{If } (l_x < l_y \text{ and } i = j - d \text{ and } M_{i,j} > k) \quad (7)$$

If the value on the diagonal with entry  $M_{l_x, l_y}$  exceeds the given edit distance, the application can abort the calculation. For example in Figure 2 after the algorithm calculated entry  $M_{4,3}$  condition (6) is fulfilled, like that:

$$6 \geq 5 \text{ and } (4 - 1) = 3 \text{ and } 2 > 1 \quad (8)$$

### 3.3 Values and References

In contrast to programming languages like Java, the programmer in C++ has always to decide, if objects are used direct or indirect with pointers or references. The scientific literature uses the terms value and reference semantic. The value semantic is used on stack objects. The size of an object is fixed and cannot be arbitrary changed. The reference semantic is mostly used on heap objects. The size of an object is not fixed and the programmer can change it. Furthermore it is possible to reference an object several times and to manage it by different methods. In addition to the actual object the reference semantic uses a reference, e. g. a pointer. The creation and deletion of such a pointer costs time. Thus, it is useful to use smaller objects direct and to manage larger objects by references [11].

		A	G	A	G	T
	0	1	2	3	4	5
A	1	0	1	2	3	4
G	2	1	0	1	2	3
G	3	2	1	1	1	2
C	4	3	2	2	2	2
G	5	4	3	3	2	3
T	6	5	4	4	3	2

Figure 1. Calculation of edit distance

		A	G	A	G	T
	0	1	2	3		
A	1	0	1	2		
G	2	1	0	1		
G	3	2	1	1		
C	4	3	2	2		
G	5	4	3			
T	6	5	4			

Figure 2. Aborting the calculation of edit distance

The use of the reference semantic has the following advantages:

- Speed: The reference semantic gives the reference of an object to a called method. The value semantic copies the entire object [11]
- Flexibility: For example to order objects, the reference semantic only has to copy the pointer to the correct position, not the entire object. It could use a pointer for each sort key [11].
- Memory Efficiency: The reference semantic uses memory for the current number of objects. The value semantic always allocates memory for the possible maximum number of objects [11].

So, the implementation should use the value semantic for smaller objects and the reference semantic for larger objects, in order to optimize the execution time.

### 3.4 Simple data types and program methods

The area of data types and functions offers some potential for improvements in every programming language. As a basic principle simple data types should be used, because the execution time, due to a lower memory usage, is crucial lower. With these data types it is possible to work faster as with complex data types. For example, the access of an object used in an array is faster than an object used in a vector. The same goal should be achieved with the usage of simple methods. To reduce execution time, it makes sense to implement existing methods in a new way. The comparison of two strings or the minimum of two numbers could be implemented in a new way to reduce some overhead. That the reason why the usage of simple data types and the implementation of simple calculations will be reviewed.

### 3.5 Parallelism

Parallelism is one of the important instruments to gain more performance [14]. Beside the development of an optimal parallelism strategy and its management, the choice of a parallelism library is crucial in C++. C++ itself offers parallelism with C++ 11 [15], but this mechanism in the most common compilers is implement in a dissatisfied way. To choose the right library some core questions must be considered, for example:

- What role plays portability?
- How familiar are the developers with C++ respectively how much time can be invested to become familiar with the appropriate library?
- Does the library fit in the existing implementation?

The point about portability is one of the main exclusion criterions. For a Linux-based implementation all Windows-specific libraries are not valuable. Additionally the portability between different Linux distributions is important. Beside the portability the time investment for the developers to become familiar with a new

library and the integration into the existing implementation should be considered. The time investment can vary from library to library. The same is true for the integration into the existing implementation.

The parallelism feature of C++ 11 will not be covered in any detail, because GCC 4.7 as a compiler does not support most of the appropriate features<sup>1</sup>.

After some research the following libraries were considered:

- Boost<sup>2</sup>
- Qt<sup>3</sup>
- Intel Threading Building Block<sup>4</sup>

The Boost library offers a simple integration and good portability. The time investment is minimal. Because the threading library of Boost was the foundation of the parallelism specification<sup>5</sup> of C++ 11 it is a generic approach. Qt is another library which offers, beside the creation of user interfaces, a parallelism library. Beside the platform independence the time investment for the developers is minimal, because of the great documentation. The Qt library has the disadvantage that its parallelism library is object-oriented and not easy to integrate in a non-object-oriented approach. To use the library classes have to inherit from the QThread<sup>6</sup> object [16]. The last library is the Intel Threading Building Block library. It offers, like Boost and Qt, portability, but a developer has to investment more time to become familiar with it. In contrast it promises good performance and a better parallelism management [13, 17].

Because of the wide spread, the simple integration, and its good documentation Boost will be used for the parallelism implementation. It does not offer the best performance, but it can be used and integrated with no additional effort.

### 3.6 Parallelism management

Beside the usage of the right parallelism library the definition of a parallelism strategy is crucial for the performance. There are three variants how a thread can be closed. In this context the following three hypotheses will be evaluated:

1. Open and close many threads as possible.
2. Open exactly one thread per CPU core.
3. An intelligent management of threads where threads are only opened and closed when it's needed.

The first point has the lowest implementation effort. One thread will be opened per query to calculate the result. This has some disadvantages. On the one hand it is a waste of resources, because every thread object in Boost will be created on the stack, which gives some problems if you open a lot of threads at the same time. On the other hand queries can be executed very fast and open and close a thread can take more time than the execution of the query

itself. The evaluation will show the dimensions of the performance reduction. Open a thread per CPU core is a good alternative to point 1. With this strategy a thread will not be open per query. This guarantees a balance in the work of every CPU core. The implementation effort is a little bit bigger as in the solution of point 1. Crucial for the success of this strategy is a balanced distribution of queries on the different cores. This can be done through a simple partitioning. The most promising approach is an intelligent management of opening and closing threads where threads are only opened and closed when they are needed. This guarantees that resources are not wasted. To use this approach rules have to be defined for opening and closing threads. Two example rules are:

- Open a thread when the average usage over all cores is more than 70%.
- Close a thread when the average usage over all cores is less than 30%.

When following this approach the locking problem has to be considered. An example: There are two threads  $t_1, t_2$ . Thread  $t_1$  likes to open a new thread, because rule number one is matched. Thread  $t_2$  at the same time likes to close a thread, because the average usage is decreased nearly at the same time. There are several solutions to this problem. A well-known is the master-slave principle [12, 18]. In this solution one thread is responsible for opening new threads and closing threads. On the one hand with that solution there is one additional thread which is only responsible for the management of the other threads without executing a single query. On the other hand the locking problem is solved and with an effective management the usage of the resources for the master thread can be neutralized.

Within the scope of this work the best parallelism strategy will be evaluated through several tests.

### 3.7 OVERVIEW ABOUT THE RUDIMENTS FOR IMPROVING THE SEQUENTIAL SOLUTION

All the presented rudiments for improving the sequential solution will be implemented step-by-step. Figure 3 shows a model of the six approaches. The results of the first solution will be used for the comparison in the other approaches. This guarantees the correctness of the results and that the approach improves the performance. There are several measurement points in the implementation to monitor the different modules of the implementation. It is possible to reject an approach if it does not return the correct results or if the execution time is not reduced.

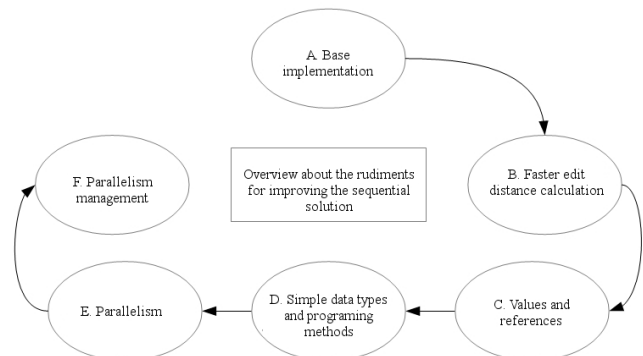


Figure 3. Overview about the rudiments to improve the sequential solution

<sup>1</sup> <http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html>,

Access: 01/12/2013

<sup>2</sup> <http://www.boost.org/>, Access: 01/12/2013

<sup>3</sup> <http://qt.digia.com/>, Access: 01/12/2013

<sup>4</sup> <http://software.intel.com/en-us/intel-tbb/>, Access: 01/12/2013

<sup>5</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2184.html>, Access: 01/12/2013

<sup>6</sup> <http://doc.qt.digia.com/qt/threads.html>, Access: 01/12/2013

## 4. RUDIMENTS FOR IMPROVING THE INDEX-BASED SOLUTION

This paragraph describes the index-based solution. It will describe the different steps to improve the performance of the index-based solution.

### 4.1 Base implementation

At first a solution will be implemented, which solves the string similarity search problem through the usage of an index. From this solution other approach will be evaluated.

As a base implementation the solution of the sequential search will be used. Improvements like the better calculation of the edit distance, the usage of pointers and references, and the usage of simple data types and program methods are already included in this first solution.

A prefix tree will be used for a fast search in the data. This has the advantage that the complexity of query depends on the depth of the tree instead of the number of data sets [6, 8]. The depth is equals to the length of the longest string in the data set.

At first the data sets will be read from the file. After this the prefix tree will be created through adding every single string. Next the query file be read and the queries will be executed. The important time measurement for comparing the sequential with the index-based search is the time needed for calculating the results that means the time frame between reading the files have finished and the end of calculating all results for the given queries. As a specialty compared to the base implementation additional information will be stored in the nodes. This information allows an early cancellation of following the branches who not will return a correct result. To realize that the minimal and maximal length of a data set will be stored in the nodes, which can be reached [8].

In the base implementation for the index-based search a node with the prefix  $y_{0...i}$  must fulfill the following conditions to consider its child nodes for the query  $x$ :

$$ed(x_{0...i}, y_{0...i}) \leq k + d_m \quad (9)$$

$$d_m = \max\{|l_x - \max_l|, |l_x - \min_l|\} \quad (10)$$

The prefix of string  $y$  at the position  $(i + 1)$  is described through  $y_{0...i}$ . Because the edit distance of  $x_{0...i}$  and  $y_{0...i}$  in a prefix tree, based on the step-by-step descent in the tree can only be calculated to a position  $i$  a tolerance value  $d_m$  must be calculated.  $d_m$  compensates the loss of information from not knowing the length of  $y$ . The edit distance may not be greater as the sum of the allowed edit distance  $k$  and the delta  $d_m$ . Delta  $d_m$  means the maximum reachable variance of the length  $l_x$  of the query  $x$  to the maximum length  $\max_l$  respectively the minimum length  $\min_l$  of the result  $y$ . If the value is lower than the variance the current branch of the tree must not be included for the calculation of the result.

The base implementation is required to give a correct result back. If this goal is achieved other steps will be executed to improve performance.

### 4.2 Compression

Another improvement is the compression. The main goal of this approach is to create only as many nodes as needed in the prefix tree. That decreases the memory usage and enables through fewer

calculations of the edit distance a faster similarity search. The following example in Figure 4 shows the compression.

The words “Berlin”, “Bern”, and “Ulm” will be inserted to the prefix tree. After the insertion is completed the prefix tree will be compressed. During that process nodes with only a single child node can be merged. After the compression the sample prefix tree only includes half of the nodes of the normal tree.

### 4.3 Parallelism management

Similar to the sequential search an index-based search can profit from a parallelized execution. Because the parallelization of the index-based search has the same requirements as the sequential solution the same strategies can be used. For more information have a look at chapter 3. In the paragraphs 5 and 6 parallelism and its management is discussed in detail.

### 4.4 Overview about the rudiments for improving the index-based solution

Figure 5 gives an overview about the rudiments to improve the index-based solution.

It has to be mentioned that, beside the improvements discussed in this chapter, the improvements of the sequential search are included in the base implementation.

The process is similar to the one from the sequential solution. The results of the base implementation will be compared in every single step to guarantee the correctness of it. To prove a faster or slower execution time several measurement points are included in the implementation.

## 5. EVALUATION OF THE SEQUENTIAL AND INDEX-BASED SOLUTION

This chapter will evaluate the sequential and index-based solution. At first the evaluation environment will be described. Based on it the presentation, the explanation, and the evaluation of the results of the time measurement will be compared with each other.

### 5.1 Evaluation environment

As an evaluation environment an Ubuntu Desktop 12.10 virtualized in Hyper-V as a standard installation will be used. The environment consists of eight gigabyte RAM and a virtualized Intel i7@2.19 GHz processor with eight processor cores. Because of a better comparison, all of the following measurement results are received from this environment.

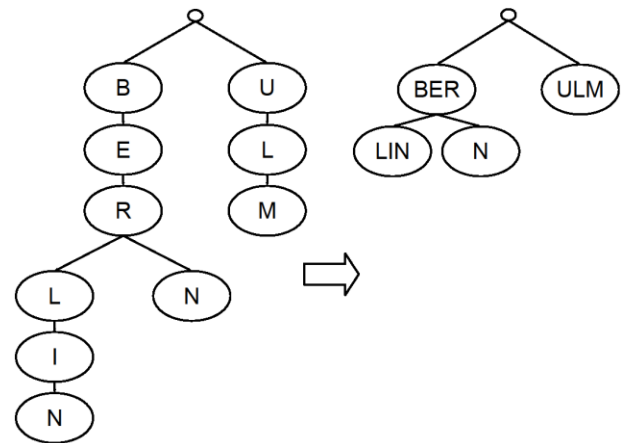
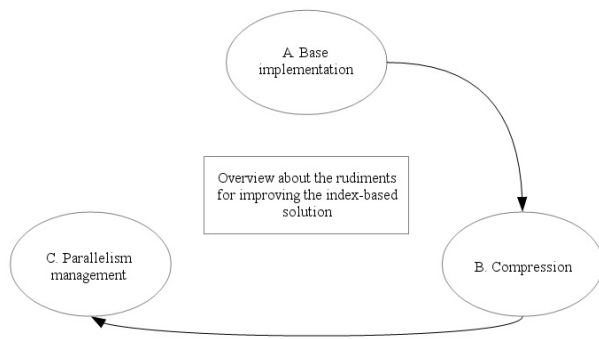


Figure 4. Compression of a prefix tree



**Figure 5. Overview about the rudiments for improving the index-based solution**

## 5.2 Proceeding of the evaluation

The process of the evaluation is as follows. Starting point are the same data sets for the measurement in every single area. The data sets and their properties are listed in Table I.

All in all there are three measurements for every approach on the appropriate data sets. The execution of 100, 500, and 1,000 queries will be measured. All time values are similar to the actual execution and not the CPU time, because the usage of parallelism can lead to other measurement results when using the CPU time. For the sequential and index-based solution only the time will be measured to calculate the results.

If not described in another way the following is true for the following paragraphs: If the execution time of an approach is better than the execution time of the previous approach the new approach will be included in the final implementation.

In the next paragraphs the best sequential and index-based solution for the city names data set will be determined. Based on this the results will be compared with each other to make a decision which solution is better. This proceeding will be repeated for the DNA sequences.

## 5.3 Evaluation of the sequential solution on the city names data set

In this paragraph the sequential solution on the city names data set will be evaluated. The goal of this evaluation is to choose the solution with the fastest execution time for the string similarity search problem. The measurement results of every single step are summarized in Table III of the appendix.

### 5.3.1 Base implementation

The following time measurements are from the base implementation which is used as a reference for the correctness and improvement of the other approaches. This implementation does not use any of the described improvement approaches. To execute 100 queries this implementation takes 16.92 sec., for executing 500 queries 84.80 sec., and for executing 1,000 queries 166.22 sec.

### 5.3.2 Calculation of the edit distance

As explained in section 3, there are different approaches to the calculation of the edit distance implemented. It takes 3.71 sec to execute 100 queries, 17.81 sec to execute 500 queries, and 34.20 sec to execute 1,000 queries. It was possible to reduce the execution time to one seventh.

### 5.3.3 Value or reference

A reduction in the execution time could also be achieved by deciding on a specific use of the value semantic or reference semantic in C++. For example, when passing large objects, they must not be copied completely. This is a C++ specific optimization, which is not available in many other programming languages. To execute 100 queries this implementation takes 2.88 sec., for executing 500 queries 15.13 sec., and for executing 1,000 queries 29.31 sec.

### 5.3.4 Simple data types and program methods

The use of simple data types and implementation of simple program methods promises additional performance gains. It takes 2.20 sec to execute 100 queries, 11.54 sec to execute 500 queries and 21.64 sec to execute 1,000 queries. In comparison to the base implementation, the execution time is only one eleventh.

### 5.3.5 Parallelism

If properly implemented, parallelism can achieve performance gains. In this first implementation of parallelism, the first strategy of parallelism is implemented. A thread will be created for each query and closed after the completion. To execute 100 queries this implementation takes 13.13 sec., for executing 500 queries 64.95 sec., and for executing 1,000 queries 129.35 sec. In comparison to the last implementation, the execution time is much higher. This approach will not be discarded. It should be considered in more detail in the next approach.

### 5.3.6 Management of parallelism

Management of parallelism allows adjustment of parallelism. It prevents the wasting of resources and promises a further performance improvement. Several configurations are tested for their performance gains. Different kind of threads will be opened for all three sets of queries. The use of 4, 8, 16 and 32 threads are considered and examined. The creation of eight threads corresponds to the second strategy and the creation of different kind of threads corresponds to the third strategy. The results of the measurement are shown in Table II of the appendix. The creation of eight threads represents the optimum. This solution is better than the solution of the fourth approach.

## 5.4 Evaluation of the index-based solution on the city names data set

The previous section has shown how the iterative optimizations were able to improve the sequential scan on the city names data set. This section also follows an iterative optimization to improve the index-based search on the city names data set. The measurement results of every single step are summarized in Table V of the appendix.

### 5.4.1 Base implementation

The following results are from the base implementation for the index-based solution on the city names data set. The implementation represents the first implementation of an index based on a prefix tree. It takes 8.14 sec to execute 100 queries, 42.26 sec to execute 500 queries and 77.95 sec to execute 1,000 queries.

**Table I. Overview about the data sets and their properties**

	#Data sets	#Symbols	Length	Edit distance
City names	400,000	ca. 255	max. 64	0, 1, 2, 3
DNA	750,000	5	ca. 100	0, 4, 8, 16



### 5.4.2 Compression

The compression is the first step to improve the index-based solution. As described in Chapter IV, the number of tree nodes is reduced. To execute 100 queries this implementation takes 7.26 sec., for executing 500 queries 38.79 sec., and for executing 1,000 queries 73.43 sec.

### 5.4.3 Management of parallelism

The last step implements the management of parallelism. The implementation of the sequential scan is re-used and adapted according to the conditions. The use of 4, 8, 16 and 32 threads are considered and examined. The results of the measurement are shown in Table IV of the appendix. The creation of 33 threads was not able to reduce the execution time. The use of 32 threads represents the optimal index-based solution for the city name data set.

## 5.5 Comparison of the sequential solution with the index-based solution on the city names data set

The results of the measurement can be stated as follows. The execution time for the sequential scan could be reduced by different kinds of approaches. Considering 100, 500 and 1000 queries, the execution time could be reduced by 91 to 97 percent. The execution time for the index-based solution could be reduced by 81 to 82 percent. In comparison of the sequential solution with the index-based solution on the city names data set, the sequential solution needs between 4 and 58 percent of the time of the index-based solution.

An optimized sequential scan is faster than an index-based solution on a set of short strings. The hypothesis is supported. Figure 6 illustrates the best sequential solution with the best index-based solution. The following sections evaluate approaches to improve the sequential and index-based solution on the DNA data set.

## 5.6 Evaluation of the sequential solution on the DNA data set

The implementation of the sequential solution for the DNA data set is the same as the implementation of the sequential solution for the city names data set. The test cases are similar to those of the sequential solution. The results of measurement for the management of the parallelism are shown in Table VI in the appendix. The optimal number of threads is 32. It is different to the city names data sets.

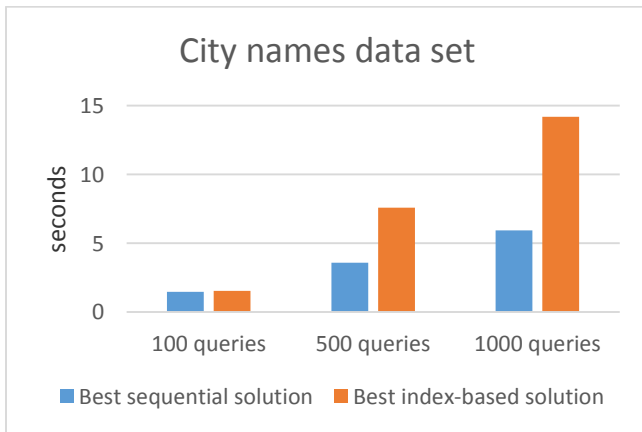


Figure 6. Comparison of the sequential solution with the index-based solution on the city names data set

The measurement results of every single step are summarized in Table VII of the appendix. The last approach with 32 Threads represents the best solution and should be considered for this reason further. This solution will be compared to the best index-based solution.

## 5.7 Evaluation of the index-based solution on the DNA data set

The index-based solution shows that the implementation of a compression and the use of parallelism management can improve the performance of an index-based solution on DNA data sets. The results of measurement for the management of the parallelism are shown in Table VIII in the appendix. The optimal number of threads is 16. This approach represents best solution in comparison to the previous approaches. All results of measurement are shown in Table IX in the appendix.

## 5.8 Comparison of the sequential solution with the index-based solution on the DNA data set

The results of the measurement can be stated as follows. The execution time for the sequential scan could be reduced by different kinds of approaches. Considering 100, 500 and 1000 queries, the execution time could be reduced by 99 percent. The execution time for the index-based solution could be reduced by 91 to 92 percent. In comparison of the sequential solution with the index-based solution on the city names data set, the index-based solution needs between 9 and 19 percent of the time of the sequential solution. An index-based solution is faster than an optimized sequential solution on a set of large strings. The second hypothesis is supported. Figure 7 illustrates the best sequential solution with the best index-based solution.

## 6. CONCLUSION AND FUTURE WORK

The aim of this paper was to outperform an index-based solution with a sequential scan. Different kind of approaches like improving the calculation of the edit distance, using value semantic or reference semantic, using simple data types and implementing simple program methods, implementing parallelism and management of parallelism were considered to implement a sequential solution. The index-based solution is based on this knowledge. Furthermore a compression of the prefix-tree and the management of parallelism were examined. It was determined that the index-based solution takes less time to compute the results on the DNA data set, but it takes more time on the city name data set.

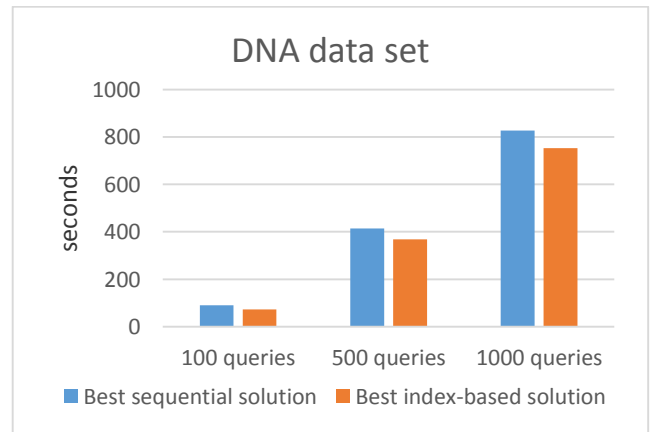


Figure 7. Comparison of the sequential solution with the index-based solution on the city names data set

The topic of this work provides further needs for research. The following area can be examined:

- **Sorting:** Can a pre-sorting by length or alphabet reduce the execution time?
- **Dictionary Compression:** The use of a compression dictionary in the DNA region could achieve improvements. An alphabet of five symbols makes it possible to represent a symbol with three bits. This makes it possible to store symbols more efficiently and to accelerate the computation time of the edit distance, because fewer bits in sum must be compared.
- **Frequency vectors:** Addition information about the number of occurrence could calculate. For the DNA data set, the number of occurrence of the symbols A, C, G, N and T is needed. For the city names data set, the number of occurrence of the symbols A, E, I, O and U is needed. On this basis, it is possible to implement an early filtering.
- **Programming languages:** The same approach could be implemented with a different programming language, to examine their effectiveness in this domain.
- **Library for parallelism:** The parallelism approach could be implemented with a different library. For example, the Qt and the Intel Threading Building Blocks library are two interesting alternatives for parallelism.
- **Management of parallelism:** A possibility of improvement is to implement an intelligent management of parallelism. The opening and closing of threads has to be dependent on the system workload.
- **Number of data records:** Has the number of data records an effect on the best solution?

Finally, this work has produced promising results. There are a lot of other approaches, which could be examined in the future.

## 7. REFERENCES

- [1] J. Buhler, "Provably sensitive Indexing strategies for biosequence similarity search," in Proceedings of the sixth annual international conference on Computational biology, New York, NY, USA: ACM, 2002, pp. 90-99.
- [2] E. Chávez and G. Navarro, "A Metric Index for Approximate String Matching," in Proceedings of the 5th Latin American Symposium on Theoretical Informatics, London, UK, UK: Springer-Verlag, 2002, pp. 181-195.
- [3] D. Fenz, D. Lange, A. Rheinländer, F. Naumann, and U. Leser, "Efficient similarity search in very large string sets," in Proceedings of the 24th international conference on Scientific and Statistical Database Management, Berlin, Heidelberg: Springer-Verlag, 2012, pp. 262-279.
- [4] G. Navarro and R. Baeza-yates, "A Hybrid Indexing Method for Approximate String Matching," *Journal of Discrete Algorithms*, vol. 1, p. 2000, 2001.
- [5] G. Navarro, R. Baeza-yates, Erkki Sutinen, and Jorma Tarhio, "Indexing Methods for Approximate String Matching," *IEEE Data Engineering Bulletin*, vol. 24, p. 2001, 2000.
- [6] E. Hunt, M. P. Atkinson, and R. W. Irving, "Database indexing for large DNA and protein sequence collections," *The VLDB Journal*, vol. 11, no. 3, pp. 256-271, 2002.
- [7] J. Kärkkäinen and J.C. Na, "Faster Filters for Approximate String Matching," in Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007: SIAM, 2007.
- [8] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser, "Prefix tree indexing for similarity search and similarity joins on genomic data," in Proceedings of the 22nd international conference on Scientific and statistical database management, Berlin, Heidelberg: Springer-Verlag, 2010, pp. 519-536.
- [9] E. S. Ristad and P. N. Yianilos, "Learning String-Edit Distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522-532, 1998.
- [10] J. Zhou, J. Sander, Z. Cai, L. Wang, and G. Lin, "Finding the Nearest Neighbors in Biological Databases Using Less Distance Computations," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 7, no. 4, pp. 669-680, 2010.
- [11] H. Helmke, F. Höppner, and R. Isernhagen, *Einführung in die Software-Entwicklung: Vom Programmieren zur erfolgreichen Software-Projektarbeit*. München: Hanser, 2007.
- [12] R. Bündgen, M. Göbel, and W. Küchlin, "A master-slave approach to parallel term rewriting on a hierarchical multiprocessor", *Lecture Notes in Computer Science*, vol 1128, pp. 183-194, 1996.
- [13] G. Contreras and M. Maronosi, "Characterizing and improving the performance of Intel Threading Building Block, " *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, vol., no., pp. 57-66, 2008.
- [14] J. Chen and W.W. Lii, "Multi-Threading Performance on Commodity Multi-core Processors," In Proceedings of International Conference on High Performance Computing in Asia Pacific Region, 2007.
- [15] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: from C++11 to POWER," in Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM, 2012, pp. 509-520.
- [16] K.B. Wheeler, R.C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, vol., no., pp. 1-8, 2008.
- [17] K. Wooyoung and M. Voss, "Multicore Desktop Programming with Intel Threading Building Blocks," *Software, IEEE*, vol. 28, no. 1., pp. 23-31, 2011.
- [18] S. Sahni and G. Vairaktarakis, "The master-slave paradigm in parallel computer and industrial settings," *Journal of Global Optimization*, vol. 9, Issue 3-4, pp. 357-377, 1996.



## 8. APPENDIX

**Table II. Management of parallelism in the sequential solution on the city name data set**

Number of threads	Number of queries		
	100 queries	500 queries	1.000 queries
4 threads	1.29 sec	3.98 sec	7.21 sec
8 threads	<b>1.46 sec</b>	<b>3.57 sec</b>	<b>5.93 sec</b>
16 threads	2.29 sec	3.86 sec	6.17 sec
32 threads	4.56 sec	5.48 sec	6.98 sec

**Table III. Evaluation of the sequential solution on the city name data set**

Approach	Number of queries		
	100 queries	500 queries	1.000 queries
1) Base implementation	16.92 sec	84.80 sec	166.22 sec
2) Calculation of the edit distance	3.71 sec	17.81 sec	34.20 sec
3) Value or reference	2.88 sec	15.13 sec	29.31 sec
4) Simple data types and program methods	2.20 sec	11.54 sec	21.64 sec
5) Parallelism	13.13 sec	64.95 sec	129.35 sec
6) Management of parallelism	<b>1.46 sec</b>	<b>3.57 sec</b>	<b>5.93 sec</b>

**Table IV. Management of parallelism in the index-based solution on the city name data set**

Number of threads	Number of queries		
	100 queries	500 queries	1.000 queries
4 threads	2.39 sec	11.79 sec	20.99 sec
8 threads	1.70 sec	8.17 sec	14.78 sec
16 threads	1.50 sec	7.93 sec	14.31 sec
32 threads	<b>1.53 sec</b>	<b>7.58 sec</b>	<b>14.19 sec</b>

**Table V. Evaluation of the index-based solution on the city name data set**

Approach	Number of queries		
	100 queries	500 queries	1.000 queries
1) Base implementation	8.14 sec	42.26 sec	77.95 sec
2) Compression	7.26 sec	38.79 sec	73.43 sec
3) Management of parallelism	<b>1.53 sec</b>	<b>7.58 sec</b>	<b>14.19 sec</b>

**Table VI. Management of parallelism in the sequential solution on the DNA data set**

Number of threads	Number of queries		
	100 queries	500 queries	1.000 queries
4 threads	126.17 sec	573.94 sec	1,136.40 sec
8 threads	88.94 sec	476.01 sec	841.55 sec
16 threads	83.73 sec	415.25 sec	848.47 sec
32 threads	<b>89.53 sec</b>	<b>413.98 sec</b>	<b>827.32 sec</b>

**Table VII. Evaluation of the sequential solution on the DNA data set**

Approach	Number of queries		
	100 queries	500 queries	1.000 queries
1) Base implementation	≈ half day	≈ 1 day	≈ 2 days
2) Calculation of the edit distance	278.45 sec	1,767.40 sec	3,191.10 sec
3) Value or reference	269.45 sec	1,746.70 sec	3,110.12 sec
4) Simple data types and program methods	267.42 sec	1,512.36 sec	2,833.03 sec
5) Parallelism	88.18 sec	434.66 sec	905.89 sec
6) Management of parallelism	<b>89.53 sec</b>	<b>413.98 sec</b>	<b>827.32 sec</b>

**Table VIII. Management of parallelism in the index-based solution on the DNA data set**

Number of threads	Number of queries		
	100 queries	500 queries	1.000 queries
4 threads	118.31 sec	545.35 sec	1,094.73 sec
8 threads	76.60 sec	419.59 sec	823.76 sec
16 threads	<b>71.78 sec</b>	<b>367.95 sec</b>	<b>753.01 sec</b>
32 threads	72.62 sec	370.21 sec	768.96 sec

**Table IX. Evaluation of the index-based solution on the DNA data set**

Approach	Number of queries		
	100 queries	500 queries	1.000 queries
1) Base implementation	876.48 sec	4,355.42 sec	8,686.65 sec
2) Compression	352.24 sec	1,737.44 sec	3,450.47 sec
3) Management of parallelism	<b>71.78 sec</b>	<b>367.95 sec</b>	<b>753.01 sec</b>