Linköping Studies in Science and Technology

Thesis No. 723

Indexing time-series data using text indexing methods

by

Henrik André-Jönsson

Submitted to the School of Engineering at Linköping University in partial
fulfillment of the requirements for the degree of Licentiate of Technology

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 1999

Indexing time-series data using text indexing methods

by

Henrik André-Jönsson

March 1999
ISBN 91-7219-292-5
Linköping Studies in Science and Technology
Thesis No. 723
ISSN 0280-7971
LiU-Tek-Lic-1998:58

ABSTRACT

This thesis analyzes a new approach to indexing time series. Indexing time series is more problematic than indexing text since the data is more compact. We investigate the use of signature files to index the time series and we also propose a new method to index the signature files to speed up the search of large time series. We have conducted a large number of tests with signature files and we feel confident that signature files provide a working solution. We also shown that it is possible to get high precision from a signature-based system. The proposed index structure, the signature tree, will also provide good search performance. We also compare the performance of signature files and our proposed signature tree to an ordinary linear search of the data and a B-Tree index. Our conclusions are that the signature files might be a good solution for some applications but the B-Tree will always beat the signature-file-based systems at search speed. The implementation of the signature tree will be further analyzed in future work.

This work has been supported by the Swedish Foundation for Strategic Research

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Acknowledgements

I would like to thank my supervisors, Prof. Tore Risch at Linköping University and Prof. Dushan Badal, University of Colorado in Colorado Springs for their support and input on this interesting subject.

I would also like to thank the members of EDSLAB for valuable input on subjects in the database area, Monika for always beeing there, Malin for her help with some diagrams and Joyce and Damon for just beeing good friends.

I would also like to thank Ivan Ranking for checking the English in this thesis.

Henrik André-Jönsson
Linköping March 1999.

List of Figures

List of Tables

1 Introduction

This thesis addresses the problem of automatic indexing and retrieval of digitized one-dimensional (1D) data. Although we propose to investigate indexing and retrieval of any sequence of real numbers, we use the term time series to be consistent with the literature. A time series is a sequence of real values generated and stored in many applications. Examples include voice, histories of stock prices, histories of product sales, histories of engine testing, seismic data, aircraft flight recordings, weather data, environmental data (pollution levels for various chemicals), satellite sensor data and astrophysics data.

1.1 The need for 1D servers

All database systems today have efficient means for storing very large amounts of data and for quickly accessing elements stored. The first databases stored records. The personnel department at a company could keep records of all employees, how much they earned and where each person worked. Every record could easily be found if the system were supplied with the employee's employment number, social security number or perhaps even the employee's name. When these databases were created, the system was told exactly what it should store about each person, and how the data should be organised so that the records could be quickly and easily found, even when there were thousands of employees at the company.

As time moved on, electronic libraries emerged, especially when the Internet gained in popularity. Now the database people face another problem. How should one store a book, and even more difficult, how does one index a book? It is possible to store books and articles in the same system used by companies to keep track of all their employees but when we want to search for a book it becomes more difficult. In a traditional system we are limited to searching for the book or article title, and if we want to search the contents of the book we are limited to searching for keywords that someone has found descriptive of the book/article.

If one were to search the contents of a text and used a word or synonym that was not entered in the list of keywords, the search would fail even if the words were present in the actual text. This is where free text search becomes relevant.

In free text search we supply text to the database and the database builds an index over all words used in the text. This allows us to search for any words that are present in the text. This method is today used in most systems where the user presents the system with a query and the system returns all text that is relevant to the query. Unfortunately this method has its problems. In the older systems the system administrator specified exactly what the system should store and how it should be stored. If, in the personnel example above, the system creator decided that it should only be possible for the system to find a record if the employee's social security number was provided, this was the only index created. In these free text search systems the system has, in some way, built an index for each possible word since the user can ask for every word present in the text. The indexes become enormous.

It was for these free text systems that the signature file was first suggested. The signature file is, compared to other indexing methods, very small. Usually around 10% to 20% of the target file size. One problem with signature files is that the method used is probabilistic. It can never tell if the queried words are present in the text, just if they may be present in the text. Because of this signature files never gained in popularity, until now. Now the need for a small index supersedes the need for an accurate recall mechanism. There are, however, still problems with signature files. The signature size grows linearly with the data set as well as the search time.

It is for applications that need "free text" search for time series data that this research targets, i.e., it would be possible for a system to answer queries like "When has this signal showed this behaviour".

If this research is successful, then its results should be useful in many applications requiring storage and retrieval of time series data. Such applications cover all branches of engineering, science (weather, geology, environment, astrophysics, medicine), stock market analysis, sales data analysis (data mining), etc. Time-series data in these applications is generated by instruments or sensors or the data may reflect behaviour of stock market or other entities. Typical queries may be "find past days in which seismic recording in USGS Palo Alto station showed patterns similar to today's pattern" or "find companies whose stock prices move similarly". Currently there is no database system technology that provides such capability.

Very little work has previously been done so far in the area covered by this thesis. Although we discuss related work in more detail in the second chapter of the thesis, we briefly review it here in order to introduce the approach to automatic indexing and retrieval of time series investigated in this thesis.

We found just few very recent papers dealing with time-series data indexing [FALO94, AGRA95, AGR295]. Time-series data indexing described in [FALO94] is based on using the first few DFTs (Discrete Fourier Transform) coefficients to represent part of time series within a window moving along the signal one step at a time. At each step the DFT is applied and this generates a one-point representation in 2D space consisting of the first two DFT coefficients. In this way time-series data is represented by a trail in 2D space. Such a trail is then divided into subtrails that are subsequently represented by their Minimum Bounding Rectangles in an R*-tree [BECK90], that is used as index. As pointed out in [AGRA95] the work reported in [FALO94] is not readily applicable as it ignores several problems such as amplitude scaling and offset translation. The indexing described in [AGRA95] is based on the shape similarity of atomic subsequencies (windows) of two signals, on stitching similar windows to form pairs of large similar subsequencies (long subsequence matching) and on finding a non-overlapping ordering of subsequence matches having the longest match (sequence matching). Both papers [FALO94, AGRA95] are deficient in that they tested their proposed indexing methods on very small collection of stock market histories. In another words it is not clear whether the indexing they propose will work on large real-life time-series database. Second, neither indexing method is invariant as regards data amplitude, offset or translation, or rotation. In other words, to use their indexing one has to perform amplitude scaling, offset translation, etc. This makes both methods unsuitable for real-life applications.

A related paper dealing with multimedia data indexing [FALO95] avoids the hard problem of feature extraction by assuming that a “domain expert” will somehow provide feature vectors or at least the easier to provide dissimilarity/distance of two objects. The problem of feature extraction is central to this area of research. It is by choosing how to extract features that we can later decide if two sequences are similar. Given this assumption the paper then describes an algorithm that maps objects into k dimensional space so that distances (dissimilarities) are preserved and possible. Another paper reporting related work [AGRA95] presents a shape definition language, SDL, for retrieving objects based on shapes contained in the histories associated with these objects. A novel feature of the language is its ability to perform blurry matching when the user cares about the general shape but does not care about specific details.

This thesis investigates a time-series data indexing method that avoids the drawbacks of published methods. In particular, the proposed method is amplitude, translation (offset) and rotation invariant. In other words, the proposed time-series indexing method can be used to index time-series data that have different amplitudes, are shifted or translated, and are rotated as well. The principle idea of the proposed time-series data indexing method is to encode the shape of time series into an alphabet of characters that can then be further identified as “words” describing time-series events like various shapes. Thus, we propose to transform time-series data into strings of characters or words and then to treat them as text. In other words, time-series data will become a text file and time-series data indexing problem will become a text indexing problem. Recent work on efficient full text indexing for terra-size text databases (digital libraries) has been performed that deal with a promising approach called context signature files [BADA95].

Context signature files are based on superimposed coding signature files and on the notion of spatial distance or co-occurrence of words in the text. The basic idea is to encode a block of text (by hashing words after passing them through a stopper and a stemmer) into a bit vector (signature). When signatures are stored in the sequential file, such a file is called a signature file. If the size of the text blocks is chosen well, it can be taken as the smallest unit of word distances, i.e., words occurring in the text block represented by one signature have the distance one. The words occurring in different signatures have a distance given by the distance of their signatures. In this way one can restore spatial proximity (the distance between words in the original file) into the signature files and use it to rank the text query results by comparing the word pair distances in the query with the same word pair distances in the signature file representing various documents. The documents that have the spatial proximity of words closest to the spatial proximity of the same words in the query are ranked as the most similar to the query. Tests on the context signature files show that the context signature file index was substantially smaller than the inverted file index or context vector index. Also, the recall and the precision of text retrieval using context signature files index was the same as for the inverted file and context vector indexes [BADA95].

This thesis investigates the use of signature files for automatic indexing and retrieval of time-series data.

2 Terminology

Feature Extraction

In order to index the time series we have to extract some information from the signal that is important for our view of the data. Feature extraction is a very crucial step. If we extract too much data from the time serie the index can become too large, but if we extract too little data we will not be able to search the time serie in all the ways we would like to. In several methods the result of the feature extraction is an N-dimensional vector that is said to be in feature space.

Similarity Between Shapes

When we search the time series we need to know when two segments match each other. We need to know if it is an exact match or if they just are similar. How do we define exact matches and similarity matches.

Indexing

The extracted features must be indexed so that the stored data can be searched in an efficient manner. The index should preferably be small compared to the signal.

False drop

A false drop is a false hit. The term is used when discussing signatures. If the signature indicates a match but the signature block does not contain the query pattern we have a false drop.

Precision

Precision is a way to measure the false drop in a system. The precision is the percentage of all returned results to a query that were relevant, i.e. they were not false drops. It is used to test the performance of signature files and other probabilistic indexes.

Signal

A signal is the unprocessed data stream we receive from a sensor. In this thesis we usually mean, when it is not clear from the context, time series.

Time-series

Time series are a sequence of values, ordered by time, e.g. a sampled sequence of data stored on a computer.

3 Related Work

In this section we take a closer look at related work. This section is divided into four parts. The first part serves as an introduction to indexing and in the second part we explain some of the problems involved in indexing 1D signals. The third part describes some related work in indexing 1D signals and finally the fourth part explains about signature access methods.

In the text we talk about text indexing methods and an important item to bear in mind is that these indexes in many cases can be used to index other things than text, but then we always have to associate each value with a keyword.

Much research has been done in the area of indexing text. In the last few years, however, there has been an increase in research on indexing signals. The problem of indexing 1D signals is more complicated than indexing text and so far nobody has been able to propose a generally applicable solution to the problem.

3.1 Indexing

Indexing is a technique used to speed up access of stored data. By storing the data in a clever way or by saving additional information about the data it is possible to find elements without having to scan the entire data set. Without indexing techniques all of today's database systems would be more or less useless since it would take far too much time to scan the entire data set every time we wanted to extract some data.

There are several indexing techniques for textual data since this has been a highly important research topic in data access research and a brief introduction to the most common techniques will be presented here.

3.1.1 Ordering

The simplest approach to index textual data is to order the keywords alphabetically. If we consider a set of records with information about persons we might order the elements according to the names of the persons in a list. If we then want to see if we have a "John Smith" in the data set we just have to start scanning the list from the start and as soon as we encounter an element that is alphabetically placed after "John Smith" i.e. "Mick Jones" (the names are ordered on the entire name, not just the family name) we can say that there are no records of "John Smith" in our data without having to scan the rest of the data. This method still involves a sequential scanning of the data and the later in the ordering the data we wish to find, the more of the data set we have to scan. In the worst case we might still be forced to search the entire data collection, i.e. if the person we are looking for is placed last in the ordered data set.

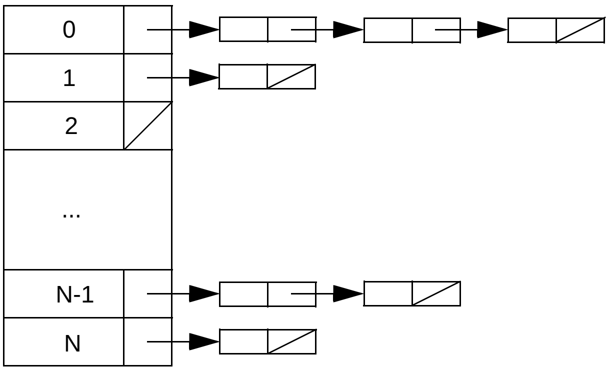
3.1.2 Hashing

A better solution would be to determine right away if a person was in the data set. One way to do this is to use hashing. There are two kinds of basic hashing, open hashing and closed hashing. If the index is properly configured we can find any element in constant time. The problem with hash indexes is that the size of the bucket array is fixed. If the performance of the index degrades because of the lists in the open hashing grows too large or we have many collisions in closed hashing a new, larger, bucket array has to be created and the data have to be moved to the new index one item at a time; unlike search trees that can grow dynamically, these hashing methods are called static hashing. These problems have been addressed using linear hashing [LITW80].

3.1.2.1 Open Hashing

In open hashing we have an array of N, initially empty, lists. When an item is added to the collection we use a hash function to find out which list in the array we should add the item to. See Figure 1. The item is added to the list at the position in the array specified by the hash function. The hash function should spread the data as evenly as possible over the array so ideally all lists should be small and no list should be much larger than any other list.

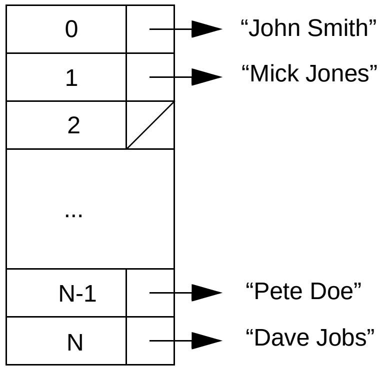
FIGURE 1. Open Hashing



3.1.2.2 Closed Hashing

In closed hashing each array entry holds only a single element instead of a list of elements. Since several elements can hash into the same entry of the array we need some way of handling these collisions. A very simple way of doing this can be to just try the next entry in the array. If the next entry is empty we store the element there and if it is occupied we just continue to advance one step forward in the array.

FIGURE 2. Closed Hashing



3.1.3 Linear hashing

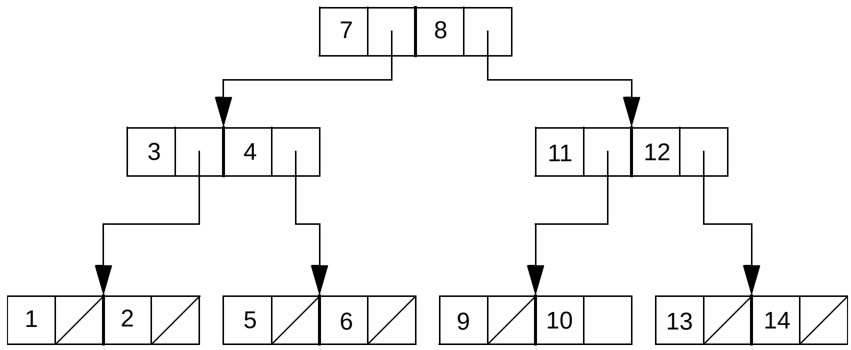
In linear hashing the system starts out with a number of buckets, where each bucket has a certain size, i.e. a page on secondary storage. When a bucket is full, the linear hashing algorithm splits the bucket into two thus allowing the amount of data to grow without lowering the performance of the system.

At startup the algorithm divides all possible hash values $[A, B)$ into binary intervals of size $\frac{B-A}{2^k}$ or $\frac{B-A}{2^{k+1}}$ for some $k \geq 0$. Each interval corresponds to a bucket. $t \in [A, B)$ is a pointer that separates smaller intervals from larger ones: all intervals of size $\frac{B-A}{2^k}$ are to the left of t, and vice versa. If a bucket reaches its maximum capacity due to an insertion the interval $\left[t, t + \frac{B-A}{2^k}\right)$ is split into two subintervals of equal size and t is advanced to the next large interval remaining. When $t = B$ the file has doubled and all intervals have the same length, $\frac{B-A}{2^{k+1}}$. In this case we reset the pointer t to A and resume the split procedure for the smaller intervals.

3.1.4 Search Trees

Trees are one of the most basic search structures in computer science. A search tree is a special kind of tree that is used to guide the search for a record. Each node in the search tree contains a (fixed) number of <pointer, value> pairs. The pointer points to a subtree and the value is a value larger than all entries in the subtree pointed to by the pointer. The <pointer, value> pairs are ordered by the value field. When the search tree is searched, we start at the root of the search tree. By comparing the different values stored at the node with the value to be searched for we can determine which sub tree the value must be stored in. We move to that sub-tree and the search process is continued until we reach the leaves of the tree. If the search tree is balanced, i.e. all branches of the tree are of the same, or of similar, length the search tree can be searched in a time logaritmically proportional to the size of the data set we are indexing with the search tree. A nice feature of the trees is that they can be easily balanced at runtime. Figure 3 on page 10 shows an example of a basic search tree.

FIGURE 3. A Search Tree



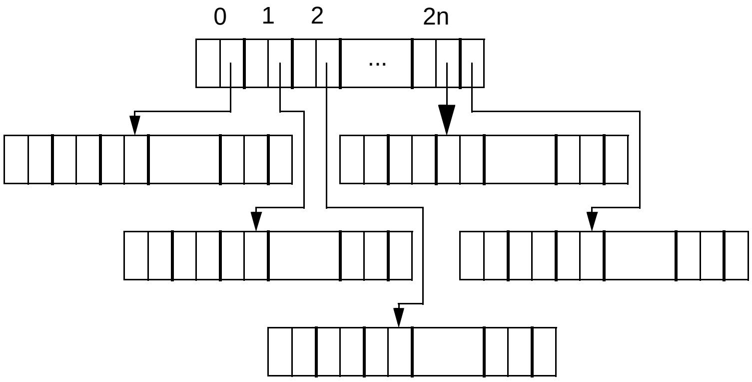
Since trees can be searched very efficiently there are a great number of variations of search trees, the most popular being B-Trees, R-Trees and modifications of these.

3.1.4.1 B-Trees

The B-Tree was first proposed in [BAYER72]. A B-Tree is an index developed to be very efficient on secondary storage and is based on the assumption that it is much more costly to fetch a page from secondary storage than it is to search in main memory.

Each node corresponds to a page on the secondary storage and contains between n and $2n$ items and $n+1$ to $2n+1$ pointers to other nodes (see Figure 4 on page 11). The <value, pointer> pairs in each node are ordered so that (from left to right) all values in the sub-tree pointed to by the leftmost pair is smaller than the value in that pair. All nodes in the sub-tree pointed to by the next pointer contain values between the value in this pair and the value stored in the next pair to the right and so in. In the sub-tree pointed to by the left most pointer, a pointer not associated with any value, all values are greater than the value in the rightmost <value, pointer> pair. During inserts and deletions nodes that have reached their maximum capacity are split. Splits can propagate up through the tree. For uniformly distributed data extendible [FAGIN79] and linear hashing will outperform the B-Tree on the average for exact match queries, insertions and deletions.

FIGURE 4. A B-Tree



3.1.4.2 R-Trees

R-Trees are based on B-Trees and correspond to a hierchary of nested d-dimensional (hyper) rectangles, first suggested in [GUTT84]. As with B-Trees each node in the tree corresponds to a page on secondary storage. Each node contains between m and M entries. Each entry contains a bounding (hyper) rectangle and a pointer to another node whose bounding rectangles are all overlapped by the stored hyper rectangle. The leaf nodes contain pointers to the actual data instead of pointers to other nodes.

As with B-Trees the R-Tree is height balanced to optimize the search speed.

3.2 Problems with 1D data compared to textual data

In many engineering applications or medical applications sensor data is stored in a database to be examined later. Unfortunately traditional databases do not have efficient indexes to index this kind of data. Over the last few years these kinds of data have become more and more important and much research has been done on the subject.

To index a 1D signal we are faced with many problems unknown to regular textual databases. The first problem is how information from the signal is to be extracted from the signal. This problem can be attacked in two ways. Either we design the system for a specific application. We can then learn what the 1D signals look like, what features the users are interested in and we can construct our system to extract exactly those features. This approach allows us to construct a system that will perform very well, as long as the users state exactly the questions the system was built to manage. The problem with this approach is, of course, that if the users suddenly decide that they are interested in a new feature of the signal, we won't be able to process those queries since the new feature was not extracted from the signal. The other way to attack the problem is to try and extract features that will describe the signal so we will be able to answer as many queries as possible, without prior knowledge of them.

The next problem we are faced with is the similarity problem. In most cases the user will not be interested in just retrieving exact matches to a query for a signal. Since signals can be very complex it is more likely that the user will give the system an example of a signal and ask the system to retrieve all signals that are similar to the example signal. What do we mean when we say that two signals are similar? If we return too many signals to the user, the system will be useless to him since he will miss the interesting signals among all the non-interesting signals. If, however, we return too few signals it is possible that some of the signals the user was interested in have not been retrieved by the system which is to say we can not find data in the system. There are two major approaches to this problem as well. Either the similarity mechanism is built into the system and the user has very little or no control over it or the user will have to specify the fuzziness of the query while stating it to the system. It it better to provide the user with some control over how many results a query should provide.

Then we need to address the index. The index should allow us to search the collection of signals in a fast and efficient way but it is also preferable that it is small in comparison to the signal being indexed and that it grows appropriately compared to the signal. The most common indexes are some sort of tree structures that usually offers very good search performance as well as growth performance. There are exceptions: the signature file search time for instance grows linearly with the data set but the size of the index is usually very small and the inclusion test is so simple compared to the data being indexed so that in many cases the signature file still is a good index compared to a tree structure. The size compared to the simplicity of the index can offer a fast and simple search. But it is worth noting that since the signature files search times grow linearly and the tree structure search times grow logaritmically there will always be a break-even size where the tree structure will start to perform better than the signature file. Since both index types grow linearly the signature file might still be preferable because of its smaller size. It is desirable that the index is small compared to the signal since a collection of 1D signals easily can become very (Terabyte sized) large.

3.3 Fast Subsequence Matching in Time-Series Databases

In [FALO94] the authors propose an effective indexing method to locate 1D subsequences within a collection of sequences matching a given query pattern. Each sequence is mapped to a small set of multidimensional rectangles in a feature space. The rectangles are then stored in a traditional spatial index, for example an R*-tree.

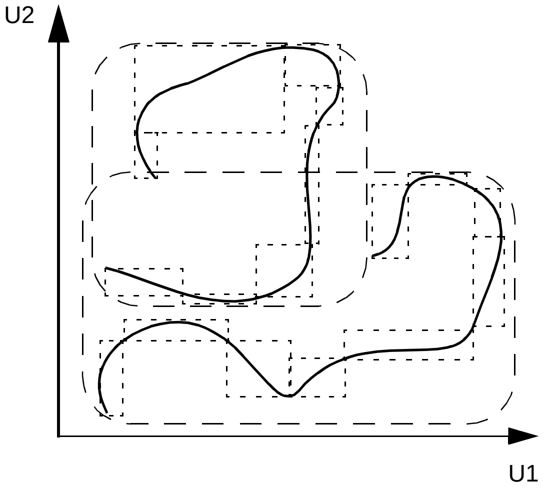
3.3.1 Feature Extraction

This method uses a distance preserving transform, for example the Discrete Fourier Transform (DFT) to extract features from the signal to feature space. But in order to accomplish subsequence matching, the feature extraction process is as follows. A small window is introduced. The window slides along the signal, one position at a time, and for each position of the window, the content of the window is mapped to a feature space vector with DFT. To make the index smaller, only the first few DFT coefficients are saved. The signal will now be represented as a sequence of vectors in feature space, or as a trail in feature space. The dimension of feature space will be fairly small, 2 or 3 dimensions, so that the trails can be efficiently indexed.

3.3.2 Indexing

The trail is then stored in an R*-tree. But storing each point in the trail creates a very ineffective index so instead the trail is recursively divided into minimum bounding (hyper-)rectangles (MBRs), see Figure 5 on page 13. In order to search the index all sequences whose MBRs intersect the query, MBR is returned.

FIGURE 5. Trail with MBRs that are Stored in the Index



3.3.3 Similarity Between Shapes

As mentioned in the previous section two sequences are considered similar if their MBRs intersect. The difference between the stored sequence and the query sequence can be controlled by how deep in the R*-tree the intersecting MBRs are checked.

3.3.4 Positive features

The method relies upon already existing and well known indexes.

3.3.5 Negative features

When the index is searched not only all sequences matching the query will be returned but also some sequences whose MBRs intersect the query MBR but whose sub-trails do not intersect the query trail. If the index is to be efficient, the number of false returns have to be minimized otherwise the results from a search have to be searched a second time to dismiss all false hits.

The method of similarity between shapes suggested in this paper is very sensitive to scaling and translation.

3.4 Fast Similarity Search in the Presence of Noise, Scaling and Translations in Time-Series Databases

In [AGRA95] an improved version of the method suggested in [FALO94] is presented. The improvements in this method over the one proposed in [FALO94] are the following:

1. No problems with amplitude scaling
2. No problems with offset translation.
3. No problem of ignoring unspecified portions of sequences while matching others (avoiding noise).

The general idea is the same. Each sequence is mapped to a small set of multidimensional rectangles in a feature space. The rectangles are then stored in a traditional spatial index, for example a R*-tree.

3.4.1 Feature Extraction

Like [FALO94] this method uses a distance preserving transform, for example the Discrete Fourier Transform (DFT) to extract features from the signal. It also uses a window that slides along the signal and then maps the content of the window to a point in feature space, but here the similarities end. Before the feature vector is calculated, the content of the window is normalized.

3.4.2 Indexing

The data points are stored in an R-Tree (the authors tried implementing the index as both a R*-tree and a R+-tree). To find a result to a query the following three steps are performed.

1. Atomic matching; find all possible pairs of matching windows.
2. Window Stitching; combine returned window pairs so that longer matching sequences are obtained.
3. Subsequence matching; find a non ovelapping ordering of sequence matches having the longest match length.

3.4.3 Similarity Between Shapes

The process of determining whether two signals are similar is performed in two steps. First all windows that are similar to the windows in the query are fetched from the index. Here the degree of similarity is checked by the distance of the windows vectors in feature space. The system can control the fuzziness by stating that all stored windows with vectors within a distance e of each query window's feature vector is similar to the query window. The next step in the similarity process is to determine whether two sequences are similar. This is done by trying to find a maximum number of non-overlapping time-ordered pairs of windows from the query and the stored windows returned from the index.

This is formally defined for two sequences S and T as follows (S_i and T_j are sub sequences i and j for each sequence respectively):

1. $S_i < S_j$ and $T_i < T_j$, $1 \le i < j \le m$, the sequence of sub sequences should be time ordered.
2. There exists some scale λ and some translation Θ so that EQ 1 is valid.

$$\forall i|_1^m \Theta(\lambda(S_i)) \cong T_i \qquad \text{(EQ 1)}$$

3. There exists a fraction of match length, ξ, greater than or equal to a specified threshold so that EQ 2 is valid.

$$\frac{\sum_{i=1} length(S_i) + \sum_{i=1} length(T_i)}{2 \cdot min(length(S), length(T))} \geq \xi \qquad \text{(EQ 2)}$$

3.4.4 Positive features

The two-level similarity mechanism offers this system the advantage that two sequences can be similar even though parts of the sequences are dissimilar. The user can control both levels in the similarity process. In the first level the user can specify the maximum distance between two vectors that should be considered similar and on the second level the user can specify the minimum length of sequences that must be similar and the maximum length of a dissimilar sequence that is accepted by the system.

3.4.5 Negative features

Since vectors for all windows are stored in the R-Tree the index will be very large and grow linearly with the size of the stored signal.

3.5 Querying Shapes of Histories

In [AGR295] the authors address the shape-querying problem from an entirely new direction. They propose a shade definition language that is a description of the signal. The language is made up of a few elementary shape descriptors and operations to build more complex systems out of these. They also propose a data structure that also serves as a fast index for searching the stored sequences.

3.5.1 Feature Extraction

The idea behind the feature extraction is to extract the same features that humans extract when they see a 1D signal. A 1D signal can be seen as a transition sequence and its these transitions that we extract. The process can be described like this. In each position, calculate the amplitude difference from the previous position and depending on the this difference, add a symbol from the alphabet of elementary transitions to the feature extraction string. The alphabet used in the paper is shown in Table 1 on page 17.

The result of the feature extraction process is a string of symbols from the alphabet.

TABLE 1. The SDL Alphabet

Symbol	Description
Up	Highly increasing transition
up	Slightly increasing transition
down	Slightly decreasing transition
Down	Highly decreasing transition
appears	Transition from a zero value to a non-zero value
disappears	Transition from a non-zero value to a zero value
zero	Both the initial and final values are zero

3.5.2 Similarity Between Shapes

The similarity concept of this model is very simple. Two sequences are similar if they map to the same symbol string.

3.5.3 Indexing

The storage structure the authors propose not only stores the symbol strings but also provides an efficient hierarchical index. The index is a four-layer index. The first layer is an array indexed by the symbols in the alphabet. Each entry points to an object in the second level. The second level is a collection of arrays. Each array is indexed by the start period of the first occurrence of the symbol, the symbol in the first level that points to this array. Each entry that corresponds to an occurrence of the symbol in the symbol string then references an object in the third level. The third level is also a collection of arrays. Each array is indexed by the maximum number of occurrences of the symbol and points to a list of objects ids that point to the original data.

3.5.4 Query Processing

Much of the work in [AGR295] concerns querying. The authors propose a language, Shape Definition Language (SDL), so that the user can combine the elementary shape symbols in the alphabet to more complex shapes. The symbols of the alphabet, the most elementary shapes in the system, are defined like this:

```
(alphabet (symbol lower_value upper_value
incomming_constrain outgoing_constraint))
```

Table 1 on page 17 shows the alphabet used in the paper and each symbols definition.

To combine these symbols into shapes the user can use any of the following functions: concat, any, exact, atleast, atmost, in, precisely, noless, nomore and inorder.

Each sequence can then be named as a shape with the shape keyword. SDL can even create parameterized shapes.

A shape description created with the shape command can then be applied to a storage structure and the result consists of references to all sequences in the structure that match the shape.

3.5.5 Positive features

SDL offers a very powerful and yet simple way of describing signals that feels very natural for the user. Also the similarity concept is very intuitive for a user.

3.5.6 Negative features

The precision of the search is determined by the size of the alphabet and it is difficult to be able to offer a simple way for the user to query the signals and at the same time let the user search for very specific sequences. The problem is that to make the symbol string correspond to very few signals the alphabet has to be very large, but if the alphabet is too large it will be very difficult for the user to state imprecise queries.

3.6 HierarchScan: A Hierarchical Similarity Search Algorithm for Databases of Long Sequences

In [LI96] a method to effectively search a collection of 1D signals for a pattern is suggested. The main idea is that each sequence is first normalized with respect to energy, amplitude or some other suitable aspect of the application, . The sequences are then transformed into a series of feature space vectors that are stored in a database.

3.6.1 Feature Extraction

Feature extraction can be performed with DFT, DCT, subband coding or wavelet transform. First the signal is segmented into shorter sub-sequences with a sliding window. Each sub-sequence, of length k, is then normalized with respect to its energy, amplitude or some other aspect. The sub sequence is then mapped to a feature space vector of size k. The feature space vector is segmented into non overlapping groups and the segmented vector is then stored in a database.

3.6.2 Indexing

This paper does not suggest a way to index the feature space vectors but relies instead on an existing database engine to store and index the vectors in an efficient way. This makes it difficult to evaluate the effectiveness of the method since the performance depends heavily on the performance of the underlying database.

3.6.3 Similarity Between Shapes

As a similarity concept the authors propose to use several correlation functions. A segmented feature space vector is obtained from the query sequence and then the most prominent segment of the query vector is checked against all stored feature vectors' same segment with the first correlation function. The correlation functions should be chosen so that we get a fast but maximal filtering in each step. The result is a set of stored vectors whose first checked segment correlates to the query vector's most prominent segment. Then the secondmost prominent query segment is checked against the remaining stored vectors' same segment with the second correlation function and so on.

3.6.4 Positive features

The method proposed by the authors is a simple but yet quite powerful similarity search mechanism that can quite easily be implemented on top of an existing database system.

3.6.5 Negative features

This method only suggests a mapping method from signals to feature space vectors and a concept of similarity but do not offer any advice on how this should be efficiently searched. They rely on the fact that an existing database system will index the vectors so that storing and fetching will be fast. One major drawback with this is that many database systems lack efficient indexes to be used with vectors.

3.7 Approximate Queries

In [SHAT96] the authors wish to introduce a new method that have a very simple and powerful concept of similarity. By making the method more of a framework and relying very heavily on knowledge from experts, it should be possible to achieve good system performance in some applications. On the other hand the heavy use of experts will make it infeasible for many applications.

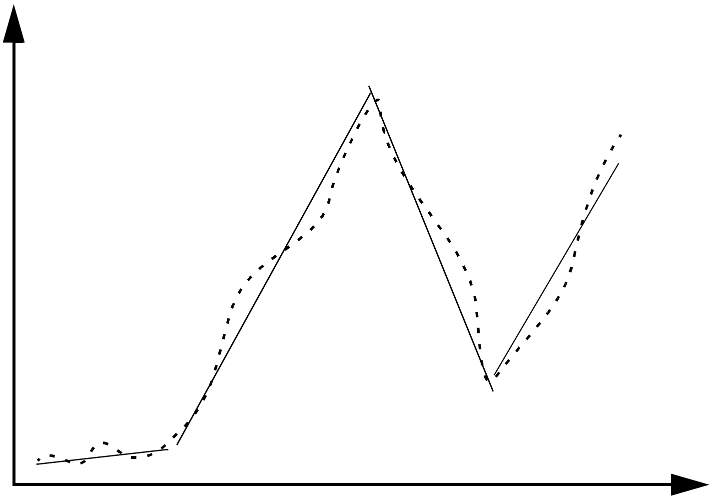
3.7.1 Feature Extraction

The idea is to break the 1D signal into smaller segments and to represent each segment as a polynomial. Under the assumption that $size \gg 1$ is true for each segment a certain amount of compression will be achieved by this. Functions have the following desirable features:

1. Significant compression
2. Simple lexicographic ordering
3. Continuity allows interpolation of unsampled points.
4. Behavior of functions is captured by derivatives, inflection points, extremes, etc.

The points where the 1D signal should be broken are very important since all important searchable features must exist inside a segment. Figure 6 on page 20 shows a signal broken into segments and each segment's polynomial approximation of the first degree.

FIGURE 6. A Signal and its Polynomial Approximation



3.7.2 Similarity Between Shapes

A query to the system is considered as a set of sequences, S. If we take a sequence r from S we can construct a new sequence, r', that is also a member of

S by applying a number of transformations to r. Examples of such transformations include:

1. Translation in time and amplitude.
2. Dilation and Contraction.
3. Deviations in time, amplitude and frequency.
4. Any combination of the above.

All resulting sequences in the stored 1D signal that are members of S are considered to be exact matches to the query sequence. Approximate matches are all matches that deviate from the query in any of the dimensions that correspond to the specified features (for example the number of peaks or the steepness of the slopes) within a certain error tolerance.

3.7.3 Indexing

In order to be an effective indexing method for time series a good index is needed so that the extracted features can be efficiently searched. The authors mention the importance of this but do not suggest a suitable index in their paper but instead note it as future work. This makes it very difficult to compare with other existing methods since the index is one of the most important features of a system.

3.7.4 Positive features

Since the system is tailored for a specific application it is possible to tweak it to perform very well in that specific environment. The major contribution of this paper is a new similarity and feature extraction process.

3.7.5 Negative features

The authors mention several problems with this approach; for example it is important that the break points are chosen so that all “interesting features” will end up in the same segments. This focus on the incoming data’s features of interest makes it possible to state very complex queries to the system but it also makes the system more difficult to adapt to different applications. Another problem is to find suitable function representations for the segments. These problems have to be solved by a human expert who has to examine the 1D signals and determine which features that are important and how the should be represented in the system. The fact that the method is so dependent on a human expert makes it difficult to adapt to new domains. Another problem lies in the fact that an expert first has to decide which features are of interest and then the system is designed for these features. The system will be designed to answer certain queries but will not be able to answer others. The final drawback with this paper is the lack of a suggested index. Since the index is a very important aspect of the system it is hard to say anything about the system without any ideas about how the index should be constructed.

3.8 Signature Access Methods

The signature approach is an access path for text, employing both a data structure, the signature file, and an access method for transforming queries into search keys. Generally, access paths are divided into either key transforming (hashing) or key comparisons (indices). The signature method is a key comparison when in its customary sequential file. Even though hashing is used to create compressed indices, it does not provide a transformation from a key term to its address. Rather, during a query the key terms are transformed into signatures and then searched by comparison to the signature file, the index.

3.8.1 History of Signature Files

Typically, signature files have only been used as a probabilistic filter for initial text search, but even then only in cases where the queries have conjunctive Boolean types. This is because the nature of signature files makes the indexed retrieval of disjunctive Boolean queries very complex. On the other hand, Boolean retrieval methods are less favored than relevance-based retrieval methods because of the added search expertise needed to use Boolean queries. Signature access paths are unsuitable with relevance ranking methods because of the document properties lost in the signature file processing.

3.8.2 Signatures Files

The signature file is a file that contains hashed relevant terms from documents. Such hashed terms are called signatures. There are several ways to extract the signatures from the documents - four basic methods being WS (Word Signatures), SC (Superimposed Coding), BC (Bit-Block Compression) and RL (Run-Length Compression).

3.8.2.1 Word Signatures

To encode a text using WS (first introduced in [FALO84]) each word in the text is hashed into a bit pattern of some fixed length. To form the document signature all these bit patterns are concatenated (see Figure 7 on page 22 for an example). Searching is done by hashing the query terms into a query signature and then matching it with the document signature.

FIGURE 7. Example of Word Signatures

Text:	example	of	word	signatures
Word Signature:	0010	1011	1100	0100
Document Signature:	0010 1011 1100 0100			

3.8.2.2 Superimposed Coding

In the Superimposed Coding, SC, the text database is divided into a number of blocks. A block B_i is associated with a signature S_i, that is a fixed length bit vector. S_i is obtained by hashing each non-trivial word - articles and other ‘‘small’’ words are ignored - in the text block into a word signature and OR-ing them into the block signature (see Figure 8 on page 23 for an example). The query is hashed, using the same signature extracting method used for the documents, into the query signature S_q. The document search is then performed by matching the signature file and retrieving a set of qualified signatures S_i such that Equation 3, on page 23 is satisfied.

$$S_i \wedge S_q = S_q . \qquad \textbf{(EQ 3)}$$

FIGURE 8. Example of Superimposed Coding

Text:	**Signature:**
example	0100 0011 1001 0000
of	1000 0010 1100 0010
superimposed	0100 0110 0100 0010
coding	1001 0000 0101 0001
Block Signature:	1101 0111 1101 0011

3.8.2.3 Bit Block Compression

One problem found in both WS and SC signatures is that when optimal false drop (false detection) reduction is obtained with larger signatures, a fairly sparse vector is formed that increases the search overhead. To handle this BC was introduced in [FALO87]. BC speeds up the search by splitting the sparse vector into groups of consecutive bits (bit blocks) and encoding each bit. For each bit block a signature is created that is of variable length and consists of at most three parts. The first part is one bit long and indicates whether there are any 1s in the bit block (bit is set to one) or if the bit block is empty (bit is set to zero). If the block is empty, the signature stops at this part. The second part indicates the number of 1s. It consists of $S - 1$ 1s (S being the number of ones) and a terminating 0. The third and final section contains the offset for each of the ones reported in the second offset, given from the beginning of the bit block. One offset is given for each 1. See Figure 9 on page 24.

FIGURE 9. Example of Bit Block Compression

Sparse Vector:	0000	1001	0000	0010	1000
Part 1:	0	1	0	1	1
Part 2:		10		0	0
Part 3:		00 11		10	00

The signatures are then stored in the signature file in one of two concatenated ways shown in Figure 10 on page 24 and Figure 11 on page 24. In the first method, (Figure 10 on page 24), each group in the sparse vector is stored together, first part 1, then part 2 and finally part 3. In the second method, (Figure 11 on page 24), all part 1 bits are first stored, then part 2 bits and finally the part 3 bits.

FIGURE 10. Method 1 for Storing BC Signatures

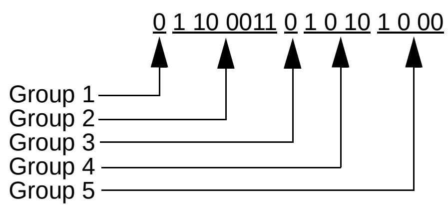
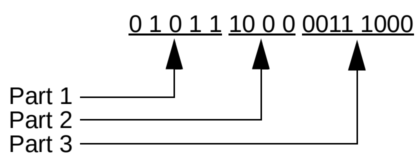


FIGURE 11. Method 2 for Storing BC Signatures



In some applications we do not want the text to be split into several different signature blocks. This is possible by using RL. The RL method is a slight modification of the BC method that is insensitive to the number of words hashed per block. This eliminates the need to split the text into several logical blocks. There is no need for remembering whether some of the terms of the query were found in any of the previous blocks of the text. This is achieved using a different bit-block size vector for each text fragment being encoded according to the number of bits set in the sparse vector that is formed. The steps in extracting a signature using RL are as follows:

1. Hash each non-common word of the text to a number in the range of $[1,B]$ and find the number of distinct 1s in the resulting sparse vector, W.

2. Compress the sparse vector using the BC method. The size of all the sparse vectors is the same but the bit block size of each encoded vector becomes $B\ln\left(\frac{2}{W}\right)$, resulting in variable sizes in the stored signatures.

3. Both the signature and the number of 1s in the sparse vector, W, are stored, uniquely identifying the value of the bit block size.

Although signature files were first proposed by C. Mooers in 1949 they have not been of much interest until the last decade. This is attributed to two things. First, signatures are a probabilistic method and only return whether a pattern may be in a target or not. Secondly, early implementations were very susceptible to false drops, i.e., they reported a hit when the pattern was not present in the target. These points are still valid to some extent, but the need to have search methods for text volumes where inverted files are far too cumbersome has superseded these points. Also, new methods for creating signatures have reduced the occurrence of false drops.

The main advantage of signature files is their low storage overhead. The signature file size is proportional to the text database size and the size of the signature file averages from 10% to 20% of the target text size. However, since the entire signature file has to be scanned every time a pattern is searched for, the search times will also grow linearly and this becomes a problem for massive text databases like digital libraries. As mentioned above there will always be a break-even size between a signature file and an inverted file if the search time is considered. Since the signature file can be so small compared to the data file, this break-even size can be very large but at some size the inverted file will always start performing better. Since both the signature file size and the inverted file size grows linearly, the signature file will always be more efficient if storage space is considered.

3.8.3 Signature Files Today

The signature file methods above all use one level sequential file schemes for their query search. As a consequence, these methods become slow for large data files since the entire signature file has to be searched during a query. Most recent research has therefore been performed with the aim of finding faster and more efficient ways to store signature files. Thus, there are designs for signature file storage structures or organizations such as a transposed file organization or bit-slice organization [ROBE79], single and multi level organizations [OTOO86, SACK87] such as S-trees [DEPP86]. The most recent signature file organization is called the partitioning approach by which the signatures are divided into partitions and then the search is limited to relevant partitions. The motivation for the partitioning approach is a reduction in search space as measured either by the signature reduction ratio (ratio of the number of signatures searched to the maximum number of signatures) and by the partition reduction ratio (ratio of the number of partitions searched by the maximum number of partitions [LEE89]. Two approaches to the partitioned signature files have appeared [ZEZU91, LEE89]. One uses linear hashing to hash a sequential signature file into partitions or data buckets containing similar signatures [ZEZU91]. The second approach [LEE89] uses the notion of a key, that is a substring selected from the signature by specifying two parameters - the key starting position and the key length. The signature file is then partitioned so that the signatures containing one key are in one partition. The published performance results on the partitioned signature files are based mostly on simulations of small text databases and are not conclusive. There was no attempt to address the scalability of the partitioned signature files to massive text databases. The partitioned signature files grow linearly with the text database size and thus they exhibit the same scalability problem as other text access structures.

4 Proposed Approach

We propose to see if it is possible to find a novel approach to the problem of indexing signals by combining the two problem areas of indexing signals and indexing text. By trying to find a suitable way of transforming the sequence of values in the signal to a stream of symbols from an alphabet, we hope to be able to use text indexing methods to index the symbol stream.

In our preliminary work we have chosen to use a very simple way of mapping the signal to text string by calculating the time derivative in each point, calculated as the difference between two nearby samples, and to map this value to a symbol. By introducing a discretisation of the time derivative values we obtain a certain fuzziness in the textual description of the symbol. This will also be our concept of similarity: if two signals map to the same symbol string they are similar; this process is described in more detail in Section 4.1.1, “Feature Extraction in our System,” on page 28.

To index the symbol string we chose to use signature files. Signature files will preserve important features of the signal like order, they offer a very fast similarity test and they are, in the text case, very small compared to the text file they index. One potential problem with signature files is that the file has to be scanned linearly. For text signatures this is usually not a major problem since the number of signatures is so small compared to the data size but in our case the number of signatures in each signature file threatens to be so large that this could be a problem. We have found a novel way of indexing the signature file itself that allows us to scan the file in logaritmic time instead of linear time. This work is still preliminary but appears promising.

We have also implemented a prototype version of this system and conducted a large number of tests. A selection of the results is presented in Section 6, “Experiments,” on page 61, and this new approach looks promising.

4.1 Feature Extraction

Before the signal can be indexed we need to decide what features the user should be able to search. This step is an adaption of the input data, the signal, to the rest of our system. The idea behind feature extraction is to filter out unimportant features of the signal, i.e. high frequencies or very low frequencies, and just store items that in a simple way describe the behavior of the signal. There are several techniques available for extracting information from a signal but before a technique is chosen, a decision about how the system will be used has to be made.

One important aspect of the feature extraction process is that we just save the “important” features and skip everything that is deemed unimportant. Usually the description of the signal after feature extraction is much smaller than the actual signal. This also introduces a certain fuzziness. While working with signals it is usually not a good idea to index absolute values and distances in the signal. If the system is queried for the sequence “1.1 1.4 1.0 0.8” the user probably wants to find the sequences “1.1 1.4 1.0 0.9” and “1.2 1.4 0.9 0.8” if they are present in the indexed material even though they do not match the query signal exactly. This is the same problem found in text indexing. If a verb is included in the query it is usually not important if it is in the present or past tense, it is the action described by the verb that is the important thing. The same applies for signals. The query is usually not for the exact sequence of value but for all sequences that are some sense are similar to the search sequence.

Calculus gives us several tools to describe the behavior and shape of a signal that helps to extract features from a signal and here is a short summary of the most common approaches.

A common method to capture the basic behavior of a signal is to calculate the discrete Fourier transform (DFT) of the signal. The signal is then viewed not as a sequence of values in the time domain but a sequence of frequencies. If we want to extract the overall shape of the signal and avoid noise and small variations of the value, we can drop all high frequency components of the DFT and just keep the first few frequency components. These components might give us a reasonably good approximation of the shape of the signal. The more frequency components we save, the better approximation we get.

Another way of capturing the behavior of the signal is to approximate shorter segments of the signal with polynomials. If we are just interested in whether the general trend of the signal is increasing or decreasing, we can approximate the signal with a polynomial of the first degree. This is a very rough approximation but it can easily be refined by using polynomials of the 2nd or 3rd degree. The problem with this approach is that it might be difficult to segment the signal so that the polynomial approximations are as good as possible.

Yet another way to describe the signal is to describe the time derivative of the signal.

4.1.1 Feature Extraction in our System

To extract features from the incoming signal we use the process suggested in [AGR295]. Informally the process can be described as follows: construct the time derivative of the signal and then quantify it. The process is done in the following steps. We scan the signal, this can be done off-line or on-line while the signal is generated and for each point we calculate the difference between the value in the point and the value in the next point. This value is then mapped to a symbol from the alphabet. The resulting string describes the shape of the incoming signal in a very intuitive way.

We call the alphabet used for describing the signal a Shape Description Alphabet, SDA. The size of the SDA is also important since it determines the granularity of the fuzziness in the system. When we talk about the SDA in general we refer to it as SDA, but if we refer to a specific version of SDA with, for instance, 5 symbols we refer to an SDA(5). Since we map a continuous value, the signal time derivative in each point, to a disrete value, a symbol from the alphabet (see Table 2 on page 30 for an example of an alphabet), we introduce a certain amount of fuzziness in the system. A signal described as a symbol string will have an envelope and all signals within that envelope will map to the same symbol string. The envelope is not described as a simple delta value around the signal since it is the signal time derivative and not the signal that is mapped to the alphabet. The envelope for three different sequences can be seen in Figure 12 on page 29, Figure 13 on page 29 and Figure 14 on page 30. All examples use the SDA(5) described in Table 2 on page 30.

FIGURE 12. Envelope of a sinus wave "abcdeedcba"

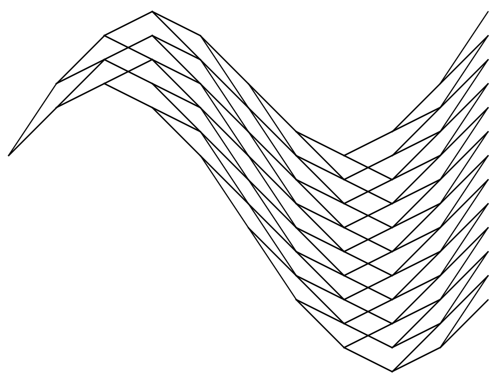


FIGURE 13. Envelope of a stable sequence "cccccccc"

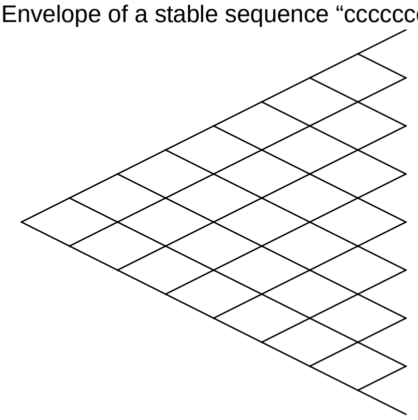


FIGURE 14. Example of a triangular sequence “bddbbddbbddb”

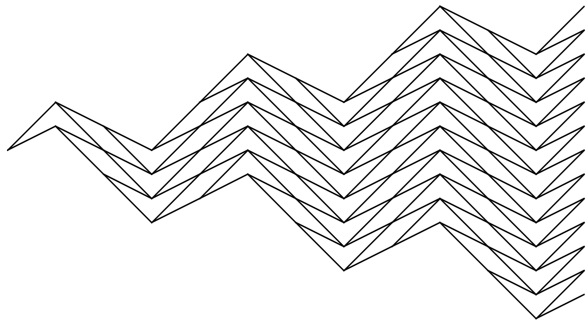


TABLE 2. An Example of a SDA(5)

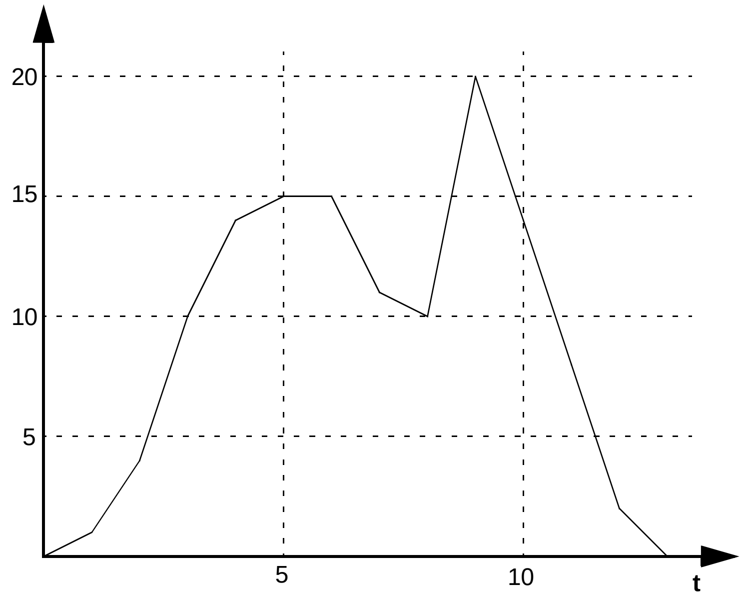
Symbol	Meaning	Definition
a	Highly increasing transition	$\frac{d}{dt} > 5$
b	Slightly increasing transition	$5 \geq \frac{d}{dt} > 2$
c	Stable transition	$2 \geq \frac{d}{dt} \geq -2$
d	Slightly decreasing transition	$-2 > \frac{d}{dt} \geq 5$
e	Highly decreasing transition	$\frac{d}{dt} < -5$

In this mapping a certain level of fuzziness is introduced. If the text strings are used as a concept of similarity, one can say that two sequences are similar if they map to the same text string. If we consider the sequence in Figure 15 on page 31, “0 1 4 10 14 15 15 11 10 20 14 8 2 0”, it will map to the string “cbabc-cdcaeeec” (using SDA(5) in Table 2 on page 30). But we can se that the sequence shown in Figure 16 on page 32, “-10 -11 -6 1 6 4 2 -1 -3 15 9 3 -3 -1” also will map to the same string. These two sequences are therefore considered to be similar by the system. So the granularity of the fuzziness is controlled by the size of the alphabet.

This concept of similarity is very attractive since it is very simple, intuitive and close to how humans think of similarities. It has also the desirable property of being amplitude invariant.

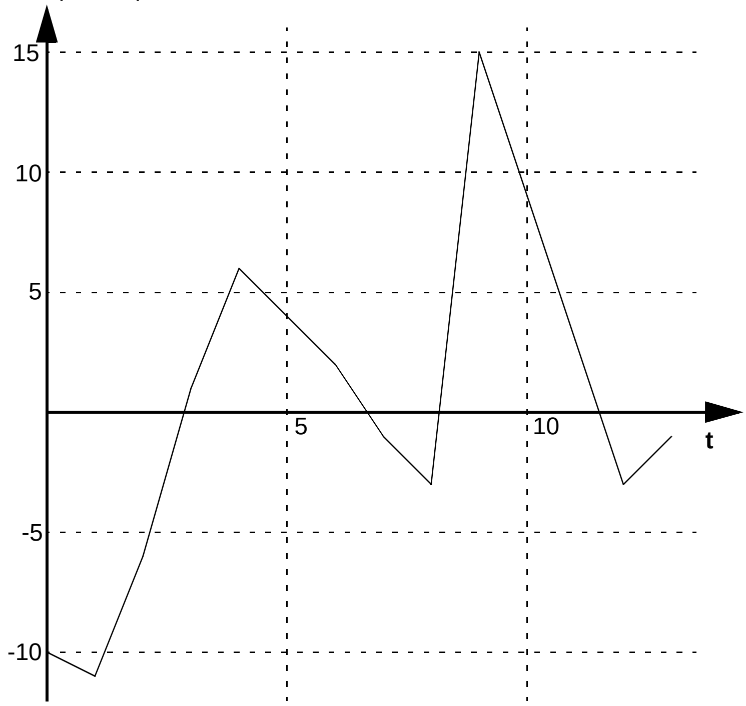
But this approach is not ideal. If this is the only similarity concept usedand the size of the alphabet is small, each query will result in a large amount of hits. Each hit will map to the same textual string but the similarity between the signals might be too vague for a human to accept the sequences as similar, the granularity is too big. To remedy this the alphabet size can be increased. Using the same similarity concept as before the number of hits received to a query will now be fewer since there are now fewer signal segments that map into the same string. But suppose that the user wishes to state a very vague query to the system. It can not be easily done. The precision of the system is controlled by the size of the alphabet and is determined when the index is built. It would be favorable to have some way of controlling the level of fuzziness at query time. We have investigated a mechanism to control the fuzziness of a query at query time, see Section 4.4, “Similarity in Signatures,” on page 41 for a discussion of the issue.

FIGURE 15. Example Sequence 1



To conclude the discussion about feature extraction, this is the part of our work that has the most potential for growth. Since our entire approach is so novel we have concentrated our efforts on other aspects of the system and not emphasised the feature extraction process. Future work in this area will include a more thorough examination of different extraction techniques and investigations of how the concept of similarity appears in each model.

FIGURE 16. Example Sequence 2

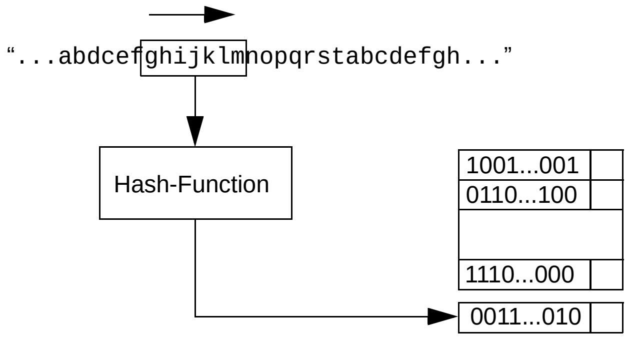


4.2 Generating Signatures

As described in Section 3.8.2, “Signatures Files,” on page 22 there are several different types of signatures. We have decided to use superimposed coding, SC, signatures since this allows us to use a simple inclusion test, see Section 3.8.2.2, “Superimposed Coding,” on page 23.

When signatures are created from text, words are fed to the signature generator. Each signature represents some disrete parts of the text, a paragraph, a chapter or an entire book. When the system is queried for a phrase or a word the system can say that the phrase or word is in those specific parts. A signal lacks any disrete parts and we can not break up the signal. The signal is one huge continuous stream of data and each point is related to its predecessor and its successor. It was decided early on that a minimal atomic unit had to be introduced. This unit is the smallest entity we can search for in the signal, a window. The window creates disrete units in the signal, it groups a number of values together in the signal and treats them as a unit, a word. But if the signal is just divided into word a query from the user will most probably not map directly to those words, a query might consist of the last three characters from one stored word combined with the first few characters from the next. In this is solved by sliding the window along the signal, one position at the time. This way all possible words will be created from the signal so whatever word is searched for, it will be in the index.

FIGURE 17. Creating Signatures from the Symbol String



Furthermore, a way to break up the signal into sections or blocks has to be found. If one signature represents the entire signal the only result that could be obtained from the system would be if a given subsequence was present in the signal, not where it could be found. Secondly the signature needed to map an entire signal would be very large. Signatures used to map texts are more than 500 bits long, and since a signal can be very large, the signatures would also have to be very large.

This is solved by breaking the signal into smaller sequences, signature blocks. Each signature block is represented by a signature and the system will be able to find all signature blocks that contain a certain subsequence. [DAVI95] points out that if the signatures are 50% full, 50% of the bits in the signature are set to 1, the false drop is minimized with respect to the fill grade. A signature block is considered full when it is 50% full and then the next signature block is started.

This introduces another problem. If the user queries for a sequence and half of the sequence is stored in one signature block and the other half is stored in the next signature block, the system will not find it by just searching the signatures.

4.2.1 Different signatures

Two approaches to solving this problem were examined. The first approach introduces overlapping signature blocks and the second approach introduces a modified search mechanism.

4.2.1.1 Overlapping Signatures

The idea behind overlapping signatures is that each subsequence should be stored in two signature blocks so that all subsequences can be found in at least two signature blocks. To produce overlapping signatures the following method is used: A signature block is produced as before, the window slides along the signal and for each window position the contents of the windows are hashed to the signature. When the signature is full, the next window position is calculated with EQ 4, i.e. when a signature block is full we start the next signature block from the middle of the previous one. This will assure us that each subsequence we are searching for will be found by searching the signatures.

$$NewStartPos = \frac{OldEndPos - OldStartPos}{2} \qquad \textbf{(EQ 4)}$$

Unfortunately this approach has two drawbacks. First of all the signature file will grow since we have to use many more signatures to map the file. Secondly, and more importantly, this approach puts a maximum length on queries. Each query can be at most the length of the overlap between signature blocks, otherwise the same problem as before will occur.

4.2.1.2 New Search Mechanism

The other approach to solving the problem is to introduce a new search mechanism that searches for possible matches between signature blocks. The window slides along the signal and each window content is hashed to a signature. When a signature is full, the first window position in the next signature will be one position to the right of the last window position in the previous signature, that is the next window position as the window slides along the signal.

When the signatures are searched, each signature is searched twice. First the stored signature S_N is tested. If the test indicates that there is no match between the query signature Q and the stored signature, we test the query signature against the next stored signature, S_{N+1}. If this test also fails then the combined signature $S_N \vee S_{N+1}$ is tested. If this combined signature matches the query signature the stored signature S_N is reported as a hit. If the combined stored signature does not match the query signature, we continue to test the next stored signature in the sequence, S_{N+2}. The advantage of this method is that each word is only stored in one signature, no overlap is necessary.

This new approach introduces a new problem. A signature is, as mentioned before, considered full when 50% of the bits are set to one. If we logically OR two such signatures together we will in a worst case scenario end up with a combined signature with all positions set to ones. A signature like that will match all possible signatures and is therefore very likely to produce a false hit. To minimize the risk of such useless signatures a signature needs to have fewer bits set. We found that lowering the fill factor to 30% instead of 50% gave us the same performance as the old search mechanism did with a 50% fill factor. There are two drawbacks with this method. First of all the search process is more time-consuming since we have to test each signature twice and OR signatures together. Secondly the size of the signature file grows since each signature block will be smaller. But most important, this method does not put an upper limit on the query length.

4.2.2 Conclusion

When the two approaches were compared, we decided to go with the second one since an upper limit to the query length would be a very serious limitation.

Finally, we summarize how the signatures are created from the sequence string. We let a window slide along the signal, one position at a time and the contents of the window are fed to a hash function. The output of the hash function is a signature with a number of bits set. This signature is logically OR-ed together with the already existing block signature. If the block signature if full it is appended to the end of the signature file and a new empty block signature is created. The process is shown in Figure 17 on page 33.

4.2.3 Hashing the windows to Signatures

To hash the window contents, the words, to the signature we need an efficient hash function. Three hash functions have been tested. The first hash function tested was EQ 5, presented in [DAVI95]. EQ 5 was quickly dismissed. This hash function needs a large alphabet to function properly and since we do not want to be forced to introduce a larger alphabet, it was substituted with the second hash function.

$$BitPos = mod\left(\left(\sum_{i=0}^{length(S)} GetPos(S, i)\right), SizeOfSignature\right) \tag{EQ 5}$$

This hash function performs quite well with smaller alphabets but has some problems if the small alphabet is combined with a very small window. But for small alphabet sizes combined with small window sizes even this function performs poorly. The third hash function tested was a hash function found in the public domain on the internet. The hash function is called MD5 and is used in encryption programs like PGP. This function is much more complicated than the previous two but provides very good performance. The algorithm is described in [RIVE92].

4.2.3.1 Longer Queries

Since the search mechanism does not put a restriction on the query length as long as we only have one signature the next step is to see if the process can work for queries of arbitrary length. Let us assume that the query maps into M signatures. Is it then possible to find a match in the signature file? As can easily be shown, a very slight modification of the search mechanism totally removes the upper query length boundary.

The search mechanism has only to be modified accordingly: When the stored signature S_N is tested and if the test signals a hit we test the next stored signature S_{N+1} to the second query signature, Q_2. If this is a hit, we test query signature 2, Q_2, to the stored signature S_{N+2} and so on until we test query signature Q_M to the stored signature S_{N+M}. If all these tests are hits, then we report a hit starting in signature block S_N. The same procedure is done with signature S_{N+1}. When both stored signatures S_N and S_{N+1} have been tested, and failed, then the combined signature $S_N \vee S_{N+1}$ is tested. If this combined signature matches the first query signature then the next stored signature tested against query signature 2, Q_2, will be signature $S_{N+1} \vee S_{N+2}$, and so on. If all these combined signatures match the query signatures, the stored signature S_N is reported as a hit.

With this simple modification it is possible to remove the upper query length boundary.

4.2.3.2 The Size of the Signature Files

One reason why signature files were attractive in the first place was that for texts the signature files are a very compact index, around 10% of the original text. An important question is if that property has been lost now that we have modified the generation process so much. It is possible to set up an expression that gives us a the size of the largest possible signature file for a given input file. It will be a worst case scenario since we assume that all window contents in any given signature block will map to previously unused bits in the signature. In reality the contents of some windows will map to bits already set by other windows in the same signature and so the actual signature block size will be larger than the calculated one and as a consequence the signature file will contain less signature blocks than calculated.

We start by introducing the following symbols:

- SB_{size}, the number of tokens in a minimal signature block.
- N_{ws}, the window size in tokens.
- N_{sb}, number of bits is the signature.
- F_g, the signature fill grade.
- w, number of bits set in the signature by each window.
- n, number of values in the original time sequence.
- N_{sig}, the maximum number of signature blocks needed to index the data.
- p, the size of a pointer in a generic unit.
- b, the size of a generic unit in bits (8 bits in a byte).
- R, the size of the signature file compared to the size of the time sequence.
- n_s, the number of generic units per data point in the time sequence.
- SF_{size}, the size of the signature file.

The first step is to calculate the minimum signature block size. The first window will add N_{ws} tokens to the signature block and set w bits in the signature. Each additional window will add 1 token to the signature block and add another w bits to the signature so the size of the signature block when the signature is full will be (EQ 4)

$$SB_{size} = N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{w} \quad \text{(EQ 6)}$$

The number of signatures with this size needed to index the whole data file will be

$$N_{sig} = \frac{n}{SB_{size}} = \frac{n}{\left(N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{2}\right)} \quad \text{(EQ 7)}$$

The size of the signature file will be the number of signatures multiplied by the size of each signature. Each signature will not only consist of the bit sequence but will also contain a pointer to the segment in the data file that is represented by the signature. The size of the signature file will therefore be

$$SF_{size} = N_{sig} \cdot \left(\left\lceil \frac{N_{sb}}{b} \right\rceil + p\right) = n \cdot \frac{\left\lceil \frac{N_{sb}}{b} \right\rceil}{N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{2}} \quad \text{(EQ 8)}$$

Finally the ratio between the original data file and the signature file can be calculated as

$$R = \frac{SF_{size}}{n \cdot n_s} = \frac{\left\lceil \frac{N_{sb}}{b} \right\rceil}{\left(N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{2}\right) \cdot n_s} \quad \text{(EQ 9)}$$

The ratio for an input file with 400,000 4-byte data points have been calculated for different configurations of the signature file, see Figure 18, “Signature File Size, 30% Fill grade, 1 bit per window,” on page 38 and Figure 19, “Signature File Size, 30% Fill grade, 4 bits per window,” on page 38.

FIGURE 18. Signature File Size, 30% Fill grade, 1 bit per window

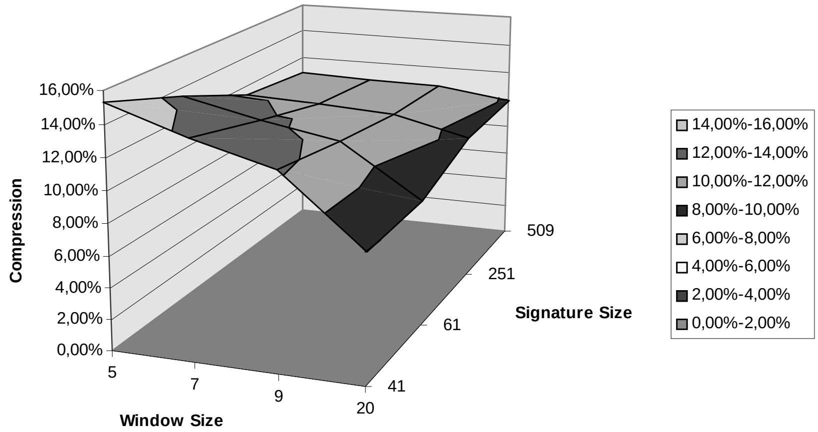
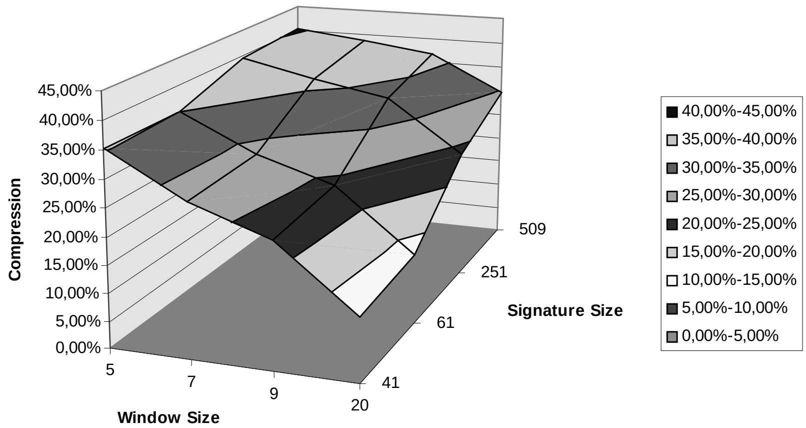


FIGURE 19. Signature File Size, 30% Fill grade, 4 bits per window



In these calculations the following assumptions were made: The number of values in the original sequence, n, was set to 400000. The pointer size, p, was set to 4 bytes. The size of a byte, b, was set to 8 bits and finally the size of each data point in the original data sequence, n_s, was set to 4 bytes (a long value).

As shown in Figure 18 and in Figure 19 the ratio is always smaller than 100%, i.e. the signature file is always smaller than the data file used creating the signature file. It is also easy to see that for most configurations of the signature file the signature file is not as compact as when dealing with text. If we only use 1 bit per window, the ratio achieved is good, between 18% and 6%. In Figure 18 we can see that for smaller window sizes the signature file size decreases with the signature size. But if the word size starts to approach 20 characters then the signature file size will increase instead of decreasing with growing signature sizes. In Figure 19 each window sets 4 bits instead of 1 bit. As we show in Section 6.2.1, “The Tests,” on page 66 we can show that this improves the search precision dramatically. In this figure the signature file size always increases with signature size. For all configurations the word size has a very large influence and if a small signature file is important then large windows should be used.

During performance testing we noticed that the performance did not improve if we increased the number of bits set by each window beyond 4.

It might be possible to improve the size if the signature file especially for signatures with a low fill grade by using signature compression as described in Section 3.8.2.3, “Bit Block Compression,” on page 23 but we have not looked into that possibility.

4.3 Searching the Signature File

In this section the search performance of the signature file is compared to the straight forward linear search of the string. Both the original string and the signature file grow linearly with the data set and we have to scan all data in both cases so what are the benefits of the signature file? There are two benefits. First the signature file is, as we have shown in Section 4.2.3.2, “The Size of the Signature Files,” on page 36, smaller than the original data file so we can scan the signature file with fewer disk accesses than the data file. Second the inclusion test performed on signatures are very simple and are effectively performed on a computer.

Before we discuss the costs of scanning the data vs. the cost of scanning the signature file, we make the following assumptions. If the cost of a disk access is 1, then the cost of a memory access is in the order of 10^{-4}. The number of disk accesses will, by far, be the most dominant factor of calculating the search cost and therefore we will say that the cost of a search is equal to the number of disk accesses needed to perform the search.

4.3.1 Cost of Scanning the Data

To store the data file on disk we need $\frac{n \cdot n_s}{P_s}$ disk pages, where n is the number of elements in the original data file and n_s is the size of each element. If each disk access has a cost of 1, the cost of scanning the data can be calculated with EQ 10.

$$C = \left\lceil \frac{n \cdot n_s}{P_s} \right\rceil \qquad \text{(EQ 10)}$$

The cost of scanning the data for a few different page sizes and for a few different sizes of the data is shown in Table 3, the size of each element in the file is set to 4.

TABLE 3. Cost of Scanning Data

Number of Elements	4kb	8kb	16kb	32kb
10^3	1	1	1	1
10^6	977	489	245	123
10^9	$9, 8\times10^5$	$4, 9\times10^5$	$2, 5\times10^5$	$1, 3\times10^5$
10^{16}	$9, 8\times10^{12}$	$4, 9\times10^{12}$	$2, 5\times10^{12}$	$1, 3\times10^{12}$

It feels relevant to point out that the cost will be the same even for smart search algorithms, like the Boyer-Moore algorithm, since these algorithms speed up the search of each individual page, but we still have to get all pages from secondary storage. Another thing is if we are not interested in retrieving all occurences of a queried pattern but just the first occurence or if we just want to check whether the queried pattern is present or not, the search will, on average, be half the maximal time we have calculated above with EQ 10.

4.3.2 Cost of Scanning the Signature File

To calculate the cost of scanning the signature we calculate the number of disk pages needed to store the signature file. Equation 9, on page 37 is used to calculate the size ratio of the signature file and the original data file. If we multiply this ratio with the size of the original data file and then divide this size with the size of each disk page we get the number of disk pages we need to scan. In Table 4 we have assumed a ratio of 30%. We show in Section 4.5, “Probability Of False Hits in the Signature File,” on page 42 and Section 6.2, “Measuring the False drop,” on page 65 that by carefully choosing our parameters it is possible to achieve a precision of 1 we also assume that we have very few or no false drops in the signature. Table 4 shows the cost of scanning the signature file.

TABLE 4. Cost of Scanning Signature File

Number of Elements	4kb	8kb	16kb	32kb
10^3	1	1	1	1
10^6	294	147	74	37

TABLE 4. Cost of Scanning Signature File

Number of Elements	4kb	8kb	16kb	32kb
10^9	$2,9\times10^5$	$1,5\times10^5$	$7,5\times10^4$	$3,9\times10^4$
10^{16}	$2,9\times10^{12}$	$1,5\times10^{12}$	$7,5\times10^{11}$	$3,9\times10^{11}$

If we compare the costs in Table 3 and Table 4 we see that the signature file is cheaper to scan. Since the signature file is 30% smaller than the original data file, it will also be 30% cheaper to search than the original data. Both costs grow linearly with the size of the data. Note that it will always be cheaper to search the signature file than performing a linear search of the original data, even if we use a smart scanning algorithm for the linear scan. Note that in all other cases than the linear search of the original data, the result of the search will be a number of pointers into the data and not the data itself. To get an exact cost the additional cost of fetching the data will have to be added to the cost of scanning the index structure. If we are only interested if a subsequence is present or not in a larger sequence we do not have to add this time. If we are only interested in the first occurence, we have to add the cost of one disk access and in the general case we have to add one disk access for each expected answer to a query. Note that in the signature file search, as well as in the linear data search, the average search time is halved if we just need to find the first occurence instead of all occurences.

4.4 Similarity in Signatures

It is tempting to assume that two similar sequences will hash into two similar signatures. The argument for this is the following: If we have a string X it will map to signature S. If we now introduce a small change in the string so we get string X' this string will generate signature S' that is "similar" to signature S in that they only differ in very few positions.

We make this assumption our hypothesis; similar sequences map to similar signatures.

If our hypothesis holds we will be able to retrieve similar sequences from the database by just examining the signatures. If a stored signature only differs in very few bit positions from the query signature then we can say that the stored signature corresponds to a sequence that is similar to the query sequence.

Unfortunately we can show that this hypothesis does not hold. If we change one position in the string, this change will be reflected in up to N_{ws} windows. These windows will have a different content than the original windows and will therefore hash into up to $w \cdot N_{ws}$ different bits. Even for small window sizes we can se that two similar sequences can hash into two dissimilar signatures.

So unfortunately our hypothesis does not hold and we can not use similarity in signatures to control similarity in signals.

4.5 Probability Of False Hits in the Signature File

A false drop is when a signature is returned as a match but the data searched for is not present in the signature block. To calculate the probability of a false drop we start by defining the false drop mathematically (EQ 11).

$$F_d = \{SignatureQualifies|BlockDoesntQualify\} \qquad \textbf{(EQ 11)}$$

The probability that a bit in a signature with m bits and fill grade g is 0 can be calculated with (EQ 12).

$$P_0 = \left(1 - \frac{1}{m}\right)^{gm} \qquad \textbf{(EQ 12)}$$

To calculate the risk of a false drop we can calculate the probability that the k bits in the query signature is set in a stored signature (EQ 13). So given a number of bits that are set in the query signature we can calculate the risk of finding a signature with the same bits set

$$p(k) = \left[1 - \left(1 - \frac{1}{m}\right)^{gm}\right]^{k} \qquad \textbf{(EQ 13)}$$

In Figure 20, Figure 21, Figure 22, Figure 23 and Figure 24 the probability for a false drop with different signature sizes is shown. The fill grade for each signature is 30% but since we OR signatures together, the fill grade used in these calculations is 60%. It is worth noting that the probability of a false drop does not change very much with different signature sizes.

If these values are compared to the experimental data in Section 6.2.1, “The Tests,” on page 66 we can see that the theoretical values are quite accurate. To compare the results we have to convert the probability of false drop into precision. To do this we can take the probability of false drop and multiply this probability with the number of signatures used to index the signal. We then get the number of signatures that should match a given query signature by coincidence. If this value is inverted we get a number that we can compare with the actual precision.

FIGURE 20. Probability of False Drop with a 16-bit signature

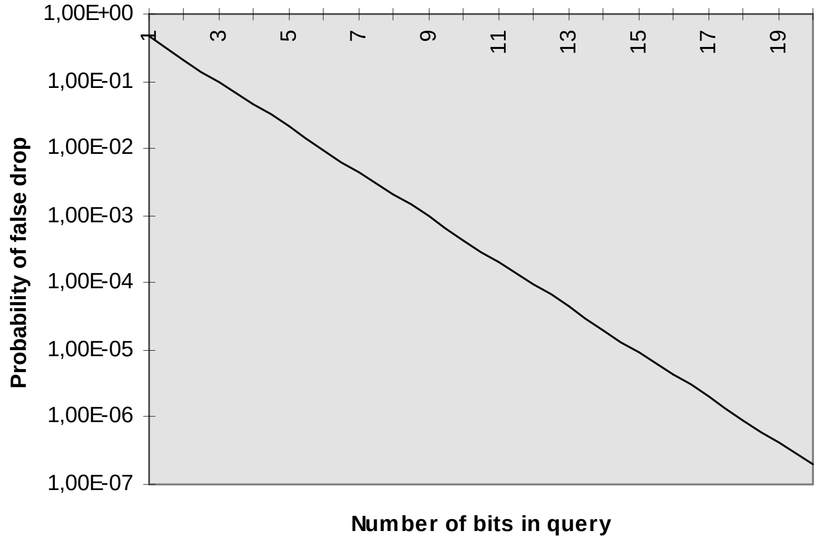


FIGURE 21. Probability of False Drop with a 24-bit signature

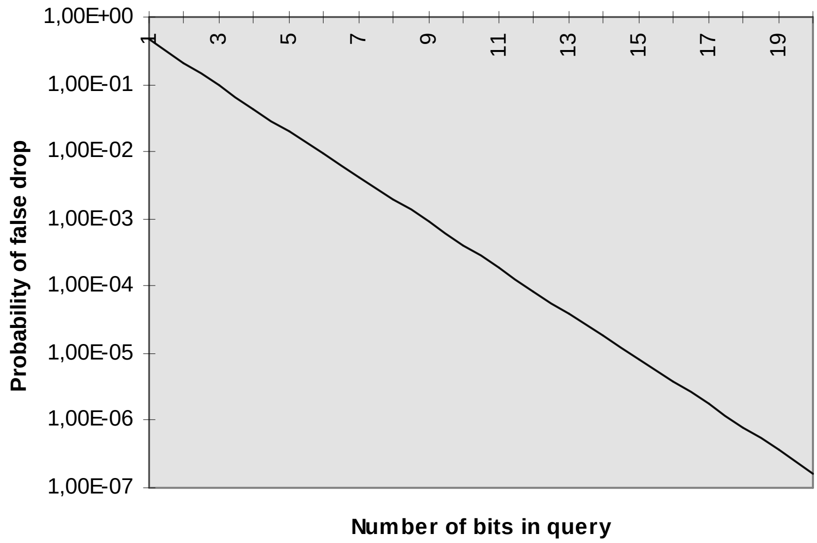


FIGURE 22. Probability of False Drop with a 61-bit signature

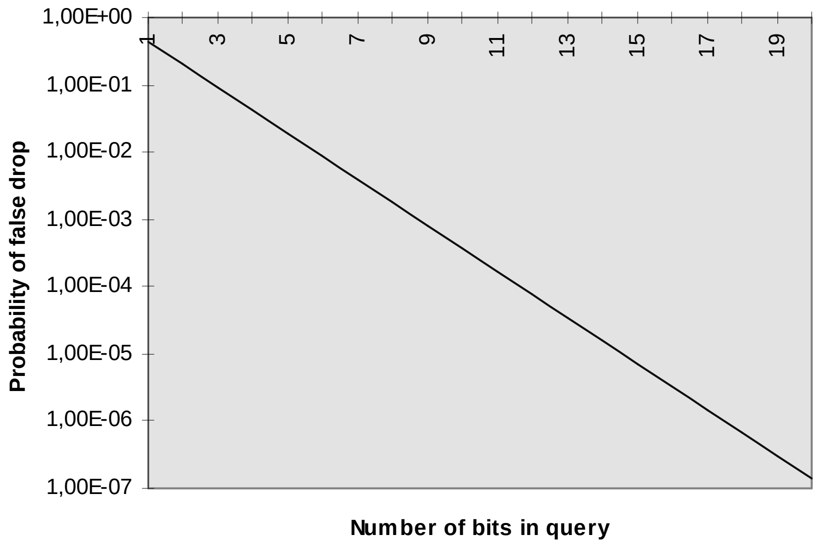


FIGURE 23. Probability of False Drop with a 251-bit signature

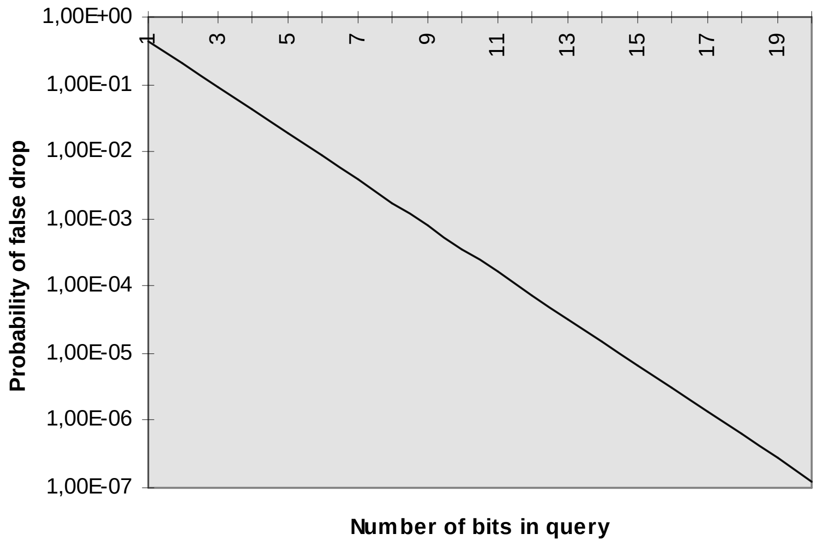
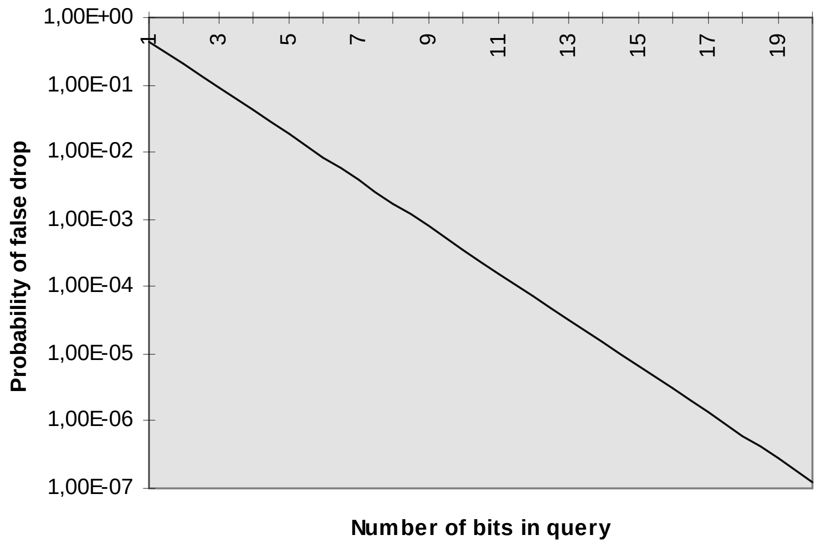


FIGURE 24. Probability of False Drop with a 509-bit signature



4.6 Our work

In all the experiments the data structure used has been a simple signature file. The signature file is a just a sequence of signatures. For a description of how the signature file is created see Section 4.2, “Generating Signatures,” on page 32. This storage structure is not optimal since we have to search the entire collection of data to see if a subsequence is present. This naive index is used as our reference index, a worst case scenario when it comes to search times. The signature file is still faster to search than the actual signal since it is much easier, and faster, to test the signature for inclusion than scanning the actual signal.

When this project was started we had two goals for the index. It must be fast to search and offer a good compression ratio compared to the original signal. The first goal is self explanatory but the second needs a comment. The goal is to develop a system that can be used to index large, Terrabyte sized, collections of 1D data therefore it is important that the size of the index does not become too large.

4.7 Indexing the Signature File

To improve the performance of the system it is vital to find a more effective index. Since trees have very good search characteristics, we have aimed for some search tree suitable for the signature file. While working with signatures we have found that the search structure needs two further criteria, it must be searchable with incomplete keys and we must preserve the ordering of the signatures. We must be able to search with incomplete keys since the signatures themselves act as keys and it must be possible to find all signatures that match a very sparsely populated signature, not only fully populated signatures. The second criterion is that since the signal is ordered, we must always be able to determine if one event comes before or after another event. The behavior and some aspects of the ST index are similar to another index proposed in [LIN98].

4.7.1 The Signature Tree, ST

Since the signatures are binary words and we use the regular Boolean operations on them one structure that matches all these criterias is a Boolean algebra. The Boolean algebra is a lattice. The topmost node is the empty binary word, a word that contains only zeros. The top node has as many children as there are bits in the word and each child differs from its parent in one bit position. All nodes at the first level from the top have one bit set to one and all other set to zero. The second level consists of all possible binary words with two bit set to one. Each node in the second level is connected to those two nodes in the first level that when they are logically OR-ed together will form the node in the second level. The third layer is constructed in the same way as the second layer and this is continued until we reach the bottom node, the node consisting of only ones. All Boolean operations are performed by moving up or down in the lattice. A logical AND operator between two nodes takes us towards the empty top node to the node's common predecessor, the first common node that can be reached from both nodes. A logical OR on the other hand will take us towards the full bottom node to the nodes common successor, the first common node that can be reached from both nodes. The algebra is symmetric and a negation will be represented as a jump from the node we wish to negate to its opposite in the algebra. All Boolean operations can be seen as traversal of the algebra. Since the signatures are binary words and we use the Boolean operators AND and OR to find matches, it would be possible to store the data in an Boolean algebra. In the text I will refer to the Boolean algebra as a signature tree, ST.

Each node in the tree will also hold a reference to a segment list. When data is inserted into the tree, a reference to the data segment will be stored in the segment list. This makes it possible to have several segments of the signal being represented by the same signature. Our analysis of the signature files generated in our experiments shows that as long as there are empty signatures they will in most cases be filled before the lists at any other node starts to grow. Since each node in the ST represents a signature and the total number of possible different signatures for a certain signature size is fixed, the index will eventually saturate. As signatures are entered into an empty index, the index will grow very quickly but as the index starts to saturate the growth will slow down until the index is saturated. At that point all possible signatures are already in the ST so we do not have to create more nodes as we continue to add signatures to the index. The new data pointers will just be appended to the already created nodes so the index will grow linearly.

4.7.2 The size of the ST

In this section we will calculate the size of a fully populated ST search tree. The number of nodes at each level can be calculated as $\binom{x}{y}$ where x is the number of bits in the signatures and y is the level. The top node is at level 0. To index a signature file we need $F_g \cdot N_{sb} \cdot 2$ levels where F_g is the fill grade and N_{sb} is the number of bits in the signature. Each signature will only hold $F_g \cdot N_{sb}$ bits but since we OR signatures together the tree needs to hold $2 \cdot F_g \cdot N_{sb}$ levels. The total number of nodes in the ST will therefore be

$$Nodes = \sum_{y=0}^{N_{sb} \cdot 2 \cdot F_g} \binom{N_{sb}}{y} \quad \text{(EQ 14)}$$

For each level the number of children of each node will decrease by one, the top node maximum N_{sb} children, and the number of parents of each node will increase by 1 for each level, the top node has 0 parents. Therefore each node, regardless of level, will have to store pointers to N_{sb} relatives. If we know the size of a pointer, p , and then add a number of bytes of extra information that has to be stored in each node, i_b , we can now calculate the total maximum size of the ST as:

$$MaxSize = \sum_{y=0}^{N_{sb} \cdot 2 \cdot F_g} \binom{N_{sb}}{y} \cdot (N_{sb} \cdot p + i_b) \quad \text{(EQ 15)}$$

Table 5 on page 48 shows the number of nodes a full ST requires using four different signature sizes. We can clearly see in this table that the ST grows very rapidly as the signature size increases, so if the ST should be a possible index structure, the signatures have to be small. To calculate the values in Table 5 on page 48 we have used EQ 14 and assumed that $F_g = 0, 3$.

TABLE 5. Number of Nodes in the ST Using Different Signature Sizes.

Signature Size	12	16	20	61
Total number of Nodes:	3 797	58 651	910 596	2×10^{18}

Using Equation 15, on page 47 it is also possible to calculate the actual size of the ST. If we assume that F_g is 30%, p is four bytes and we need to store 5 extra bytes in each node ($i_b = 5$) we can calculate the size of a full ST with different signature sizes, Table 6 shows the ST sizes for 12-, 16-, 20- and 61-bit signatures. We can clearly see that to use the ST we have to use small signatures. Even for a signature size of 61 bits the ST will be enormous.

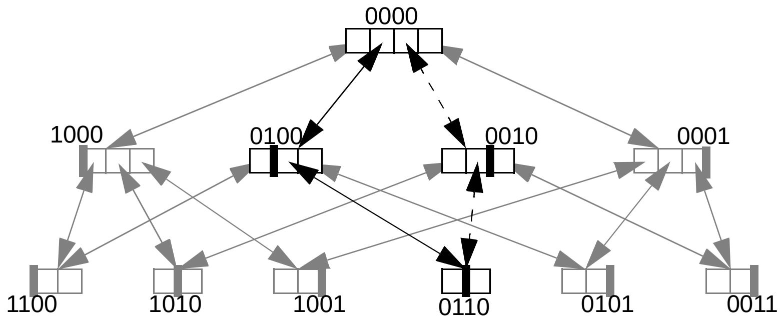
TABLE 6. ST sizes for Different Signature Sizes

Signature Size	12	16	20	61
Total Size (in bytes):	$2, 0\times10^{5}$	$4, 0\times10^{6}$	$7, 7\times10^{7}$	$5, 5\times10^{20}$

4.7.3 Building the ST

When a new signature is to be added to the ST we do as follows. We start by positioning our marker at the top node. The position of the first 1 from the left will determine which of the top nodes children we should mode the pointer to i.e. if the first 1 is in position 5, the pointer to the segment will be placed somewhere under node 5. We move our marker to the fifth child from the left of the top node. Then we continue by examining in which position the second 1 is. This determines which child of the current node we should place the node. When there are no more bits set to 1 in the signature we store a pointer to the signal segment in that node. We then have to backtrack through the tree and insert all parent nodes that the inserted node is dependent on. As we go back up through the structure additional nodes will be inserted until we are back at the top node.

FIGURE 25. An ST after the signature "0110" has been inserted



In Figure 25 the two first levels of a 4-bit signature ST str shown after a segment with the signature "0110" has been inserted. The pointer to the segment has been stored in the node labeled "0110". All black nodes and arcs have been created by the action and the gray ones have not been created yet and are just shown for illustration. The arcs represented with solid arrows show the path we follow when we go down into the ST and the dotted arrows show the path followed when we backtrack after the insertion. Note that the signature is not stored anywhere but is given by the position of a node in the ST. In the figure each node has a number of pointers. Some to the left and some to the right of a vertical bar. The pointers to the right of the bar are the nodes we can access while going scanning a signature and moving down into the ST. The ones to the left of the bar are pointers that can be followed after we have reached a node as a result of a search with an unfull signature or pointers accessed while backtracking the ST after an insertion. Note that the pointers on each side of the bar are sorted so that the ones with bits set in lower positions (positions to the left) are stored to the left of the other pointers. This allows us to find the correct pointer immediately without having to scan.

As can be seen, the tree will grow very fast as long as there is very little data stored in it but as the amount of data increases the probability that nodes and arcs that have to be created already exist increases and the rate of growth decreases. This continues until the ST is fully populated. When the ST is full the only information we need to add to the ST is a new reference to the new segment at the indicated node. Therefore the index will grow linearly after the ST is full.

Note that the insertion algorithm is not complete. In future work we intend to investigate how the ST should be populated so that as few nodes as possible will have to be inserted without losing any functionality.

4.7.4 Searching the ST

At this stage in our work we are only considering simple searches, i.e. the maximum query size is one signature. The search time is logarithmically proportional to the size of the signature and since the size of the signature is fixed in the index the search time will be constant. When the index is saturated, each leaf node will contain a list of pointers to signature blocks, but since each signature block in the list has the same signature the list does not have to be searched, the entire list is just returned as a result of the search. The index search time will still be constant. We have scanned a signature file containing over 2,000 signatures and no signature occurred more than twice, i.e. no leaf node list would contain more than two elements. The average leaf node list in our preliminary test would be very close to 1.

To search for simple queries, queries that only consist of one signature, it would be possible to perform very fast searches. A simple search through a fully populated ST would need approximately as many steps as a full signature has bits set to 1. The cost of a real search will be higher since when we have scanned the entire signature we will have to move through the ST to find data that might span two signatures. We have not examined this algorithm yet.

If we assume a signature size of 61 bits and a fill grade of 30%, then we would have at most 19 bits in a full signature, so a simple search can at most be performed in approximately 19 steps. If we make the (somewhat crude) assumption that each step results in a disk access, we can calculate the approximate cost of searching such a ST, see EQ 16.

$$Cost = 19 \cdot 10^3 \approx 2 \cdot 10^4 \qquad \text{(EQ 16)}$$

Note that when a signal is indexed it will result in a large number of full signatures and at most one signature that is not full. For practical reasons we can therefore say that all segment pointers will be stored at the same distance from the top node. So a simple search will always have the same cost regardless of the amount of data stored in the ST. Note that we have to add the cost of actually fetching the data, see the discussion in Section 4.3.2, “Cost of Scanning the Signature File,” on page 40.

How many signatures will a *full*, all leaf nodes present, ST hold? Since the size of each level in the ST is given by the binomial distribution, at the 19th level, where all leaves will be stored, the tree will have $\binom{61}{19}$ nodes = $2,97 \cdot 10^{15}$ nodes. Note that this is the number of signatures, the size of the corresponding data file would be larger (see Equation 6, on page 37 and Equation 7, on page 37, as we calculate in Section 4.8, “Comparison with a B-Tree Structure,” on page 52 this corresponds to a data file with approximately 10^{16} elements). Searching $2,97 \cdot 10^{15}$ nodes stored in a signature file will take in the order of $\frac{10^{15}}{PageSize}$ disk accesses. If we assume a page size of 8kb we can see that it will take approximately 10^{11} disk accesses or 31 years if each disk access takes 10 ms whereas the ST can be searched in approximately 200 ms. But can the ST be searched with unfull signatures? Yes. If the search process does not take us all the way to a leaf, the result of the search is all children of the node the search terminated in.

The search algorithm will have to be expanded since the algorithm described here only can do simple searches. The first thing we have to do is to expand the algorithm to allow us to find data between signatures. One way to do this is to store a pointer with each signature block pointer so that we can find the successor of this block in the ST. Then we can check the combined signatures for a hit by making sure that the query signature is a parent to the combined node. Another way to implement this might be to store the combined signatures in the ST. The next extension we have to make is to expand the algorithm so that queries of any length can be stated to the ST. All these expansions will, of course, make the algorithm more complicated and slower.

Please note that our work with the ST is very preliminary and that is the reason we show the time spent searching a full ST compared to searching the same amounts of data in a signature file or in a B-tree. Future work will give us better, and more realistic, cost models of the ST and the possibility of finding the data sizes where each data structure is optimal.

4.7.5 Conclusions

A major problem with the signature file is that it has to be scanned sequentially. With the ST this will no longer be necessary and it will be possible to search the signature file efficiently in constant time. Note that the search time for a window is constant relative to the signature size so that for an implementation the search time is expected to be constant. The problem with the ST is the size. The ST grows very fast when data is inserted, so for small sequences the ST will not be a good solution because of the massive overhead. Since the ST grows slower and slower, it will reach a break-even point for very large sequences where it will provide very good performance searching large sequences. It might be possible to make a smarter implementation of the ST but this is also future work.

Another problem not discussed here is the problem of actually storing and retrieving data from the disk. We have assumed that the sequence is stored as a sequential array on the disk but for very large sequences this might be a very bad solution. One solution suggested in [LIN99] is to use a B-Tree to create a dynamic array on the disk.

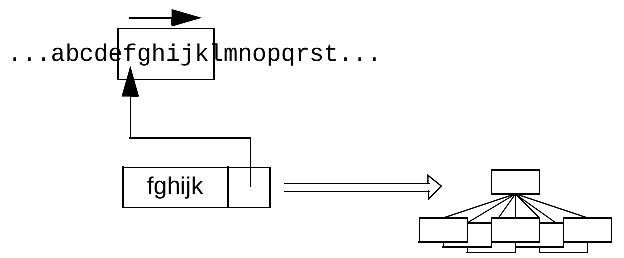
4.8 Comparison with a B-Tree Structure

To show some of the benefits of our proposed index structure we compare it to the B-Tree structure (see Section 3.1.4.1, "B-Trees," on page 10). We use the B-Tree for comparison since it is one of the most common, and successful index structures used today.

4.8.1 Test System

The test system uses the following procedure to index the sequence. As in our system, we slide a window along the sequence. The the first characters of the window contents are used as a key value for the sequence and then a pointer back to the sequence is inserted into the B-Tree.

FIGURE 26. System Using a B-Tree index



When the system is used to retrieve a sequence, the following algorithm is used:

1. Retrieve all sequence segments that match the first segment of the query. Move these segments to a result list.
2. Go through the result list and match each subsequent segment of the query to the segments found in step 1. The assumption made here is that the result list retrieved in step 1 is sufficiently cheaper to scan linearly than it is to make several B-tree searches.

We use the same data in this experiment as we did while we were discussing the ST (see Section 4.7 on page 46). Using Equation 6, on page 37 and Equation 7, on page 37 we can give an approximate size of the original data file. We set $N_{ws} = 16$, $N_{sb} = 61$, $F_g = 0,3$ and $w = 4$. This gives us an approximate size of $n \approx 6 \times 10^{16}$ elements. To simplify calculations we can assume we have approximately 10^{16} elements in the original data file. To address this amount of data it is not sufficient to use 4-byte pointers so we use 8-byte pointers instead.

We can calculate the number of keys we can fit on each page of the secondary storage. We calculate that for each key the element has to be stored together with two pointers. The number of keys per page is calculated with EQ 17. In Table 7 on page 53 we can see the number of keys per page for four different page sizes. In the calculations we have, as mentioned above, assumed a pointer size, p, of 8.

$$K_p = \left\lfloor \frac{PageSize}{KeyLenght + 2 \cdot p} \right\rfloor \quad \text{(EQ 17)}$$

TABLE 7. Number of keys per secondary storage page for 4 different page sizes.

Key Length	4kb	8kb	16kb	32kb
4	204	409	819	1638
5	195	390	780	1560
6	186	372	744	1489
7	178	356	712	1424
8	170	341	682	1365
9	163	327	655	1310
10	157	315	630	1260

The average number of page accesses, P_a, needed to searching the B-Tree can be calculated with EQ 18. n is 10^{16} elements.

$$P_a = \left\lfloor \frac{\ln(n)}{\ln(K_p \cdot 0,63)} \right\rfloor \quad \text{(EQ 18)}$$

The cost of searching the B-tree is approximately equal to the number of pages needed to fetch from secondary storage. Note that the cost should only be thought of as a *very* approximate time estimate and that we have to add the cost of retrieving the data to the cost of scanning the index, see Section 4.3.2, "Cost of Scanning the Signature File," on page 40.

TABLE 8. Cost of searching the B-Tree

Key Length	4kb	8kb	16kb	32kb
4	7	6	5	5
5	7	6	5	5
6	7	6	5	5
7	7	6	6	5
8	7	6	6	5

TABLE 8. Cost of searching the B-Tree

Key Length	4kb	8kb	16kb	32kb
9	7	6	6	5
10	8	6	6	5

In Table 8 we can see that larger page sizes are slightly better since they result in a cheaper B-tree search cost, but the differences are very small. However if the key length is too small, we will have a large number of occurrences of each key that we have to scan if the query sequence is larger than the key and this will increase the cost. A further analysis of this has to be made in the future.

For smaller sequences the B-Tree will perform better not only because the data set is smaller but also since each entry will be smaller. In most cases we can use 4-byte pointers instead of 8-byte pointers and this will improve the B-Tree performance slightly.

TABLE 9. Cost of searching a smaller B-Tree

Key Length	4kb	8kb	16kb	32kb
4	3	3	2	2
5	3	3	2	2
6	3	3	2	2
7	3	3	2	2
8	3	3	2	2
9	3	3	2	2
10	3	3	2	2

If we use Equation 17, on page 53 and Equation 18, on page 53 to calculate the cost of searching the B-Tree index if the B-Tree only contains 10^8 elements and we use 4-byte pointers. In Table 9 we can see that the B-Tree is very fast for this smaller sequence. We can also see if we compare Table 8 and Table 9 that the B-Tree scales very nicely.

If we compare these results with the cost of searching the signature file, the signature file search is much slower. The cost of searching the signature file is, with this cost model, approximately 10^{15}, so the B-Tree is much faster. The cost of an ST search would be approximately 10^4 (see Equation 16, on page 50), much faster than the signature file search but approximately a factor 10^3 slower than the B-Tree. Our tests show that the ST will be evenly populated; as long as there are unpopulated leaf nodes, they will fill up before the lists at other nodes start to grow. This could still make the ST more interesting than the B-tree since we will not have to scan any lists, but this is future work. We will also have to make a further analysis of the ST to see if we can find efficient algorithms for it.

It is important to point out that these are just preliminary figures and they are just intended to provice the reader with some idea of what could be expected in future tests.

5 Test Data

The system has been tested on two different sets of data. The first set is a synthetically generated signal and the second set is sensor data from a thermometer at the Department of Computer Science, Linköping University.

We decided to use both a synthetically generated signal as well as a real signal for two reasons. First of all we had problems in finding signals that were long enough. A popular test for time series indexing methods is to use stock exchange data but in the sources we found the files were to small to be interesting. The other reason was that we could easily control the shape of the synthetically generated signal so that we could test our system with different variations of the signal.

5.1 The Synthetically Generated Signal

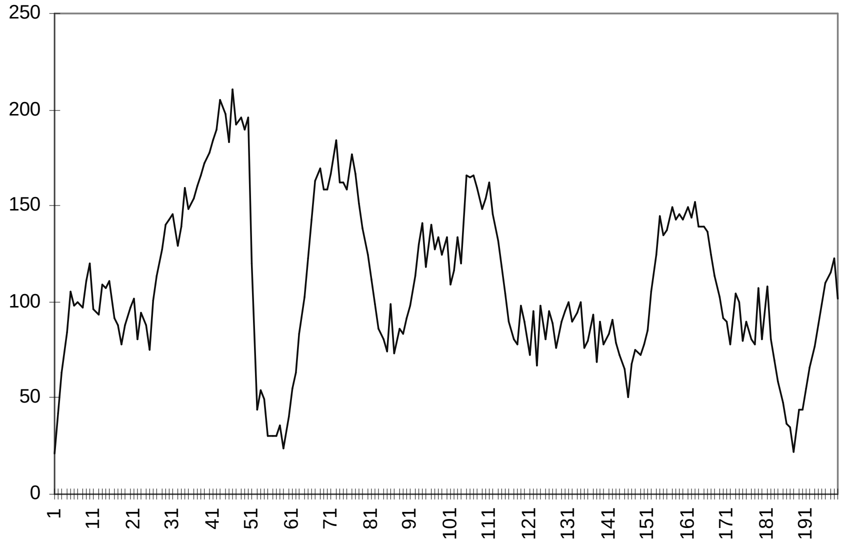
The synthetically generated signal is generated with random numbers to mimic a signal produced by a sensor. This signal can easily be generated at an arbitrary length. The signal used in testing our system has consisted of more that 400,000 data points.

The signal is constructed along these steps. We start at an initial value, α, that is set to 0 in the first iteration:

1. Generate a random number in the range $-A$ to A. This is the next level, β, we should move to.
2. Generate a random number in the range 1 to B. This is the number of steps it will take to reach β from α.
3. Add B steps to the signal, each step $\frac{\beta - \alpha}{B}$ long.
4. Generate a random number in the range 0 to C. This is the number of steps where the signal will be stable at level β.
5. For each step that the signal is stable at level β, generate a random number in the range $-D$ to D and add it to β so that some smaller variations around β are introduced.
6. Set $\alpha = \beta$ and go back to step 1.

This method gives us a signal that contains slopes, stable areas and smaller, high frequency variations. A small example of a synthetically generated sequence can be seen in Figure 27 on page 58.

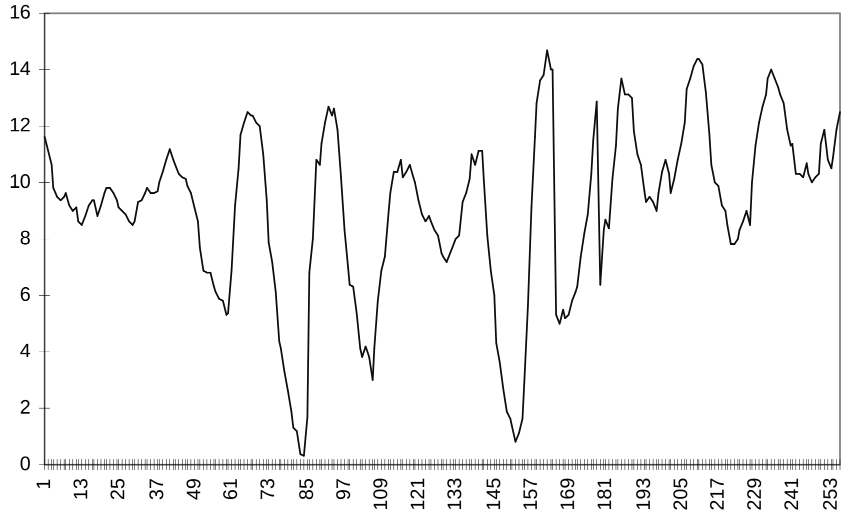
FIGURE 27. Example of a Synthetically Generated Signal



5.2 The Real Sensor Data.

The real sensor data is a collection of temperatures from the computer science department. An automatic system measures the outdoor temperature once every hour 24 hour a day and adds the data to a large temperature file. This file contains temperature data for more than three years but due to hardware and software errors the file only contains valid data corresponding to approximately two years. Figure 28 on page 59 shows a small sample from this data sequence.

FIGURE 28. Example of Temperature Data



6 Experiments

During the course of our work we have conducted a number of tests. The main objective of the tests has been to measure the false drop for different queries of different lengths under varying circumstances. The false drop is an established method of measuring the performance of probabilistic search methods such signature files and is defined as the ratio between the number of returned relevant documents and the total number of returned documents. We have also conducted tests of the hash function to ensure that the signatures generated were of high quality.

We have changed several variables during the experiments to see how they influence the false drop. The variables are the window size, the signature size, the alphabet size and the number of bits set by each window in the signature. The window size controls the size of the sliding window we use to construct the signatures. The window size also has a direct impact of the system since the content of a window will become an “atomic” entity in the system. Sequences smaller than window size characters can not be searched for, i.e. the window size decides the minimum query length in the finished system. Since we do not want to design the system so that it can only find large subsequences we want the minimum query length to be small (smaller than 7 values). However, a very small window size together with a very small alphabet size will lead to problems while generating the signatures (the hash function will not be able to perform properly). The possible minimum alphabet size was primarily tested with the hash function but we also did some tests were we measured the false drop rate with a few different alphabet sizes.

The importance of the signature size is also something we wish to test. Recent work on text files shows that the system performance increases with larger signatures [BADA95]. In work with text files very large signatures are often used. But since we wish to index the signature file with an ST (see Section 4.7, “Indexing the Signature File,” on page 46) we would like to have as small signatures as possible. The different signature sizes used are 41, 61, 251 and 509 bits.

During our first set of tests we saw that the precision was often very poor when we searched the signature file for short sequences. An attempt to improve the false drop when searching for short sequences each window was permitted to set more than one bit in the signature when the signatures were generated. Experiments were made with 1, 4 and 6 bits but since no difference was achieved by using 6 bits instead of 4, only the first two categories are shown here.

6.1 Signature Quality

The quality of the signatures was measured by generating a signature file from a large test file (see Section 5.1, "The Synthetically Generated Signal," on page 57). When the signature file was produced, all signatures were scanned and the bit probability of each bit in the signature was calculated.

This data was then used to calculate a mean bit probability, a standard deviation of the mean bit probability for each configuration. The minimum and maximum bit probability was also stored. The figures in this section show the results from signatures created with the MD5 hash function and both 1 and 4 bits set per window in each signature. Since the same data file was used in all tests, the number of signatures created varies. With the 16-bit signatures the actual number of created signatures were around 55,000. This number decreased as the signature size increased and in the last test with a 509-bit signatures the number of created signatures were around 2,000. This is the reason that the performance seem to slowly degrade as the signature size increases.

FIGURE 29. Signature Quality: 16-bit signatures

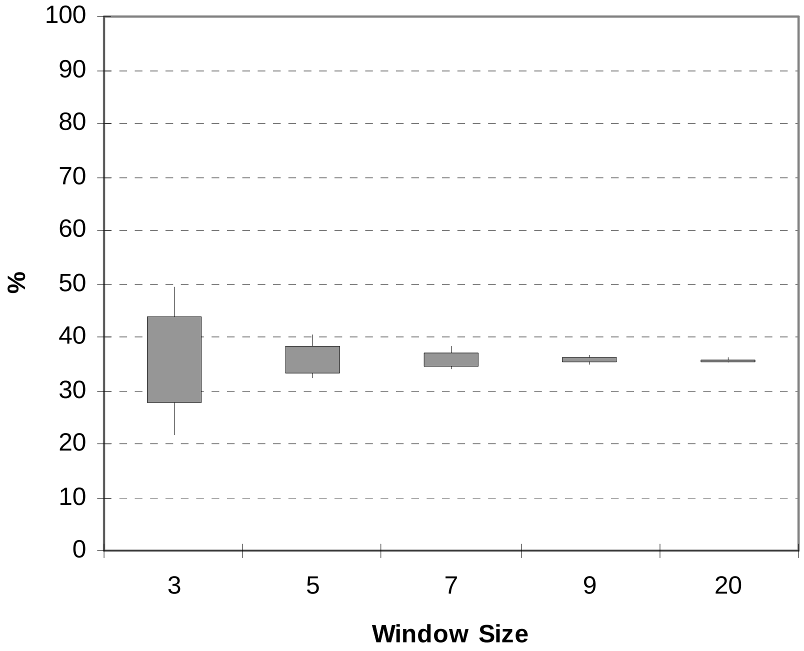


FIGURE 30. Signature Quality: 24-bit signatures

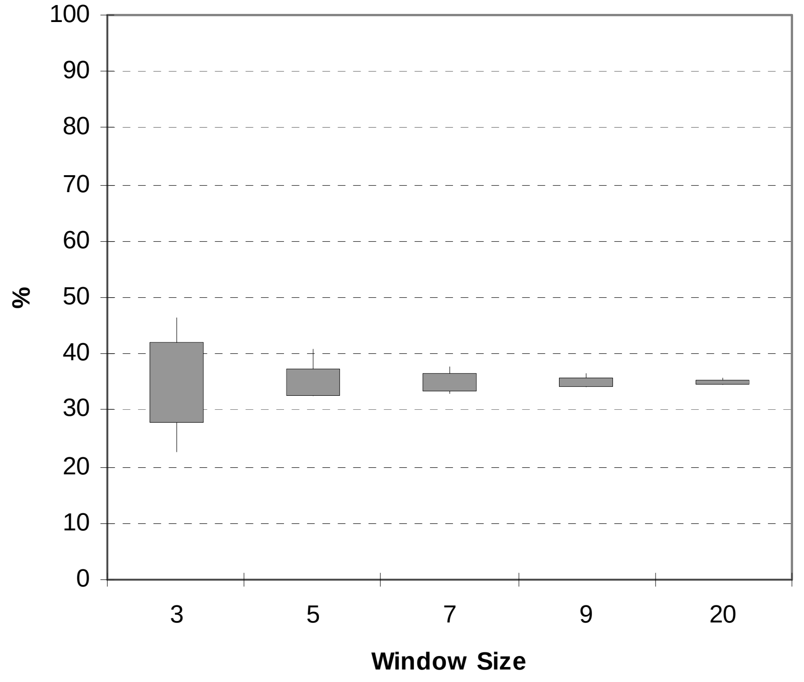


FIGURE 31. Signature Quality: 41 bit signature

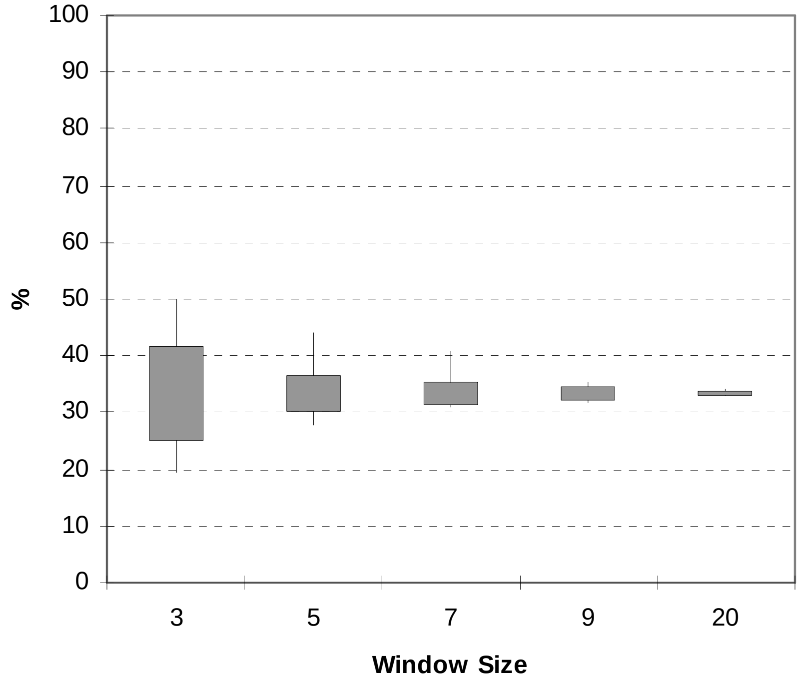


FIGURE 32. Signature Quality: 61-bit signatures

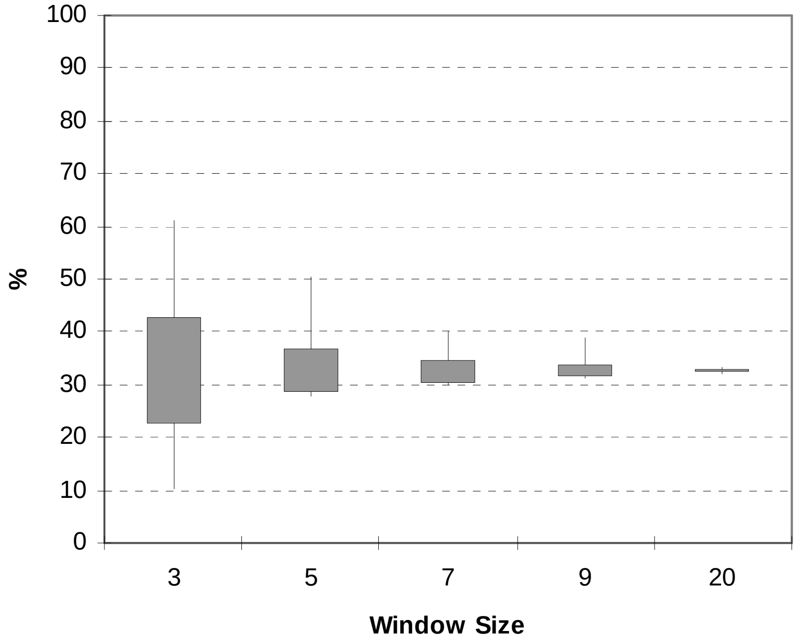


FIGURE 33. Signature Quality: 251-bit signatures

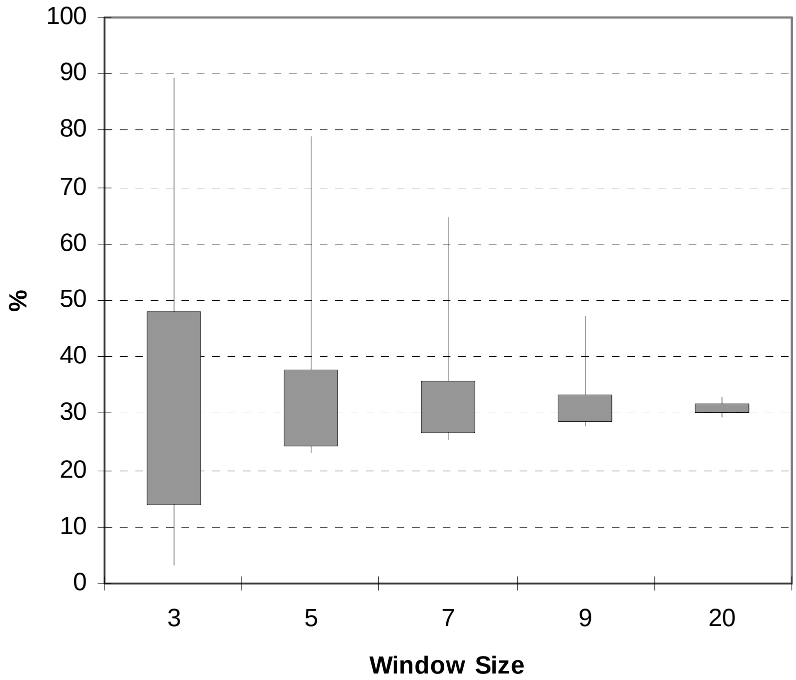
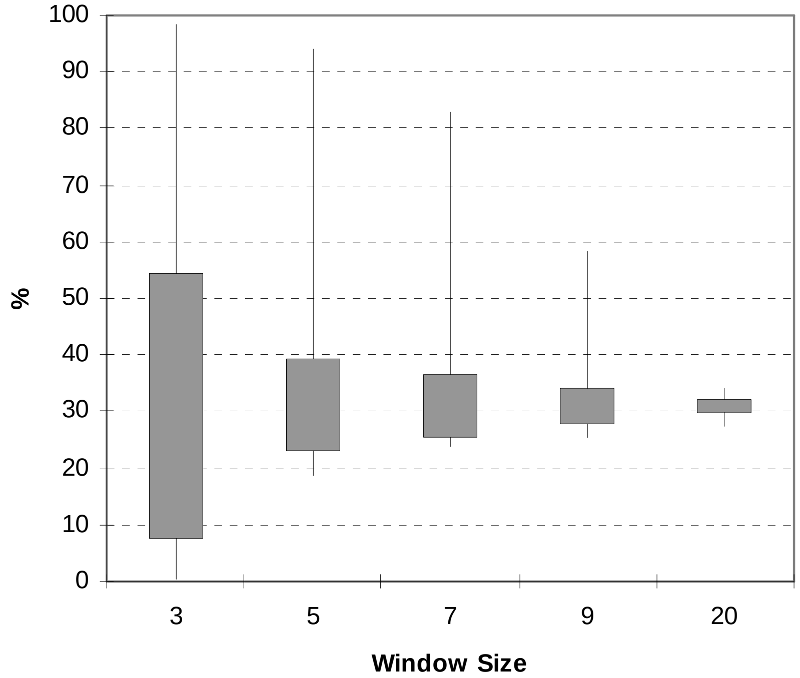


FIGURE 34. Signature Quality: 509-bit signatures



6.2 Measuring the False drop

To measure the false drop we generated a signature file from one of our test files. Then we searched the signature file for a number of subsequences from the test file. We counted the number of results returned by the signature file scan and then we scanned the file for the sequence and counted the number of actual occurrences. The precision was then calculated with EQ 19.

$$Precision = \frac{ActualOccurences}{ReturnedOccurences} \quad \text{(EQ 19)}$$

To see how the false drop was influenced by the size of the subsequences we created several different queries of varying length. The first sub-subsequence we searched for was a sequence consisting of the first window size values from the subsequence, then we searched for the window size(N_{ws})+1 values and so on up to a sequence length of N_{ws} +20 values. By doing this we can observe how the false drop rate changes with varying search sequence lengths.

6.2.1 The Tests

In this section we present some of the test results we have obtained from the testing we have done. Each diagram represents one test: the signature size, the window size and the number of bits set by each window.

Each line in the diagrams is one test run. A test run is created as follows. The test system fetches a subsequence from the stored sequence 20+window size characters long. Then it starts testing the signature file. The first window size characters from the subsequence forms the query pattern. The signature file is used to find all occurrences of the query pattern. The stored sequence is also scanned linearly for the query pattern so that the precision can be calculated. Then the next character from the subsequence is added to the query pattern and the procedure is repeated. This is continued until the query pattern consists of the entire subsequence.

6.2.1.1 The Results

We have tested each configuration with 7 test runs.

In the first 24 diagrams (Figure 35 to Figure 59) each window sets only one bit in the signature and in the next 24 diagrams (Figure 60 to Figure 84) each window sets 4 bits in each signature.

In the first 5 diagrams (Figure 35, Figure 36, Figure 37, Figure 38 and Figure 39) a 16-bit signature was used with varying window sizes. As we can see the precision of the system does not start to improve until we use a 9-character window, Figure 38, or a 20-character window, Figure 39, but it is still not good. A 16-bit signature is not useful with this setup unless we search for very large sequences.

In the next 5 diagrams (Figure 40, Figure 41, Figure 42, Figure 43 and Figure 44) a 24-bit signature was used. The precision here is comparable with the 16-bit signature precision, not very useful unless we search for very large sequences.

In Figure 50, Figure 51, Figure 52, Figure 53 and Figure 54 a 251-bit signature is used. With a very small window size, Figure 50, the precision is not good, but as the window size increases the signature file starts to work properly. Already with a 7-character window, Figure 52, the precision starts to come up to tolerable levels for mid sized sequences and for larger window sizes, Figure 53 and Figure 54, we get a precision of 1 when we search for sequences that consist of 13 to 14 windows (window size+13 characters).

In Figure 55, Figure 56, Figure 57, Figure 58 and Figure 59 a 509-bit signature is used. Here we can see that we get a very good precision for very small window sizes. When a 5-character window is used, Figure 56, we get a precision of 1 already at a query size of 12.

When we increase the number of bits set by each window to 4 we see a remarkable increase in precision. In Figure 60, Figure 61, Figure 62, Figure 63 and Figure 64 we use the same configuration as in Figure 35, Figure 36, Figure 37, Figure 38 and Figure 39 with the exception of number of bits set by each window. As we can see here we get a precision of 1 for very short queries. With a query length of 9 we have a precision of 1 for all queries in the test run.

For the other configurations we can see the same effects but an interesting phenomenon is that for larger signatures we get worse precision with very small window sizes, but for larger window sizes the precision improves with larger window sizes.

FIGURE 35. 16-bit signature, 3-character word, 1 bit per word

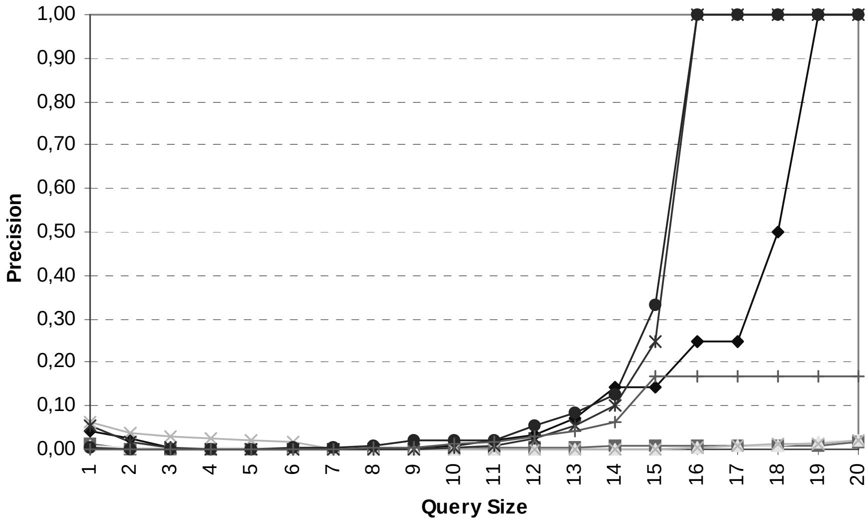


FIGURE 36. 16-bit signature, 5-character word, 1 bit per word

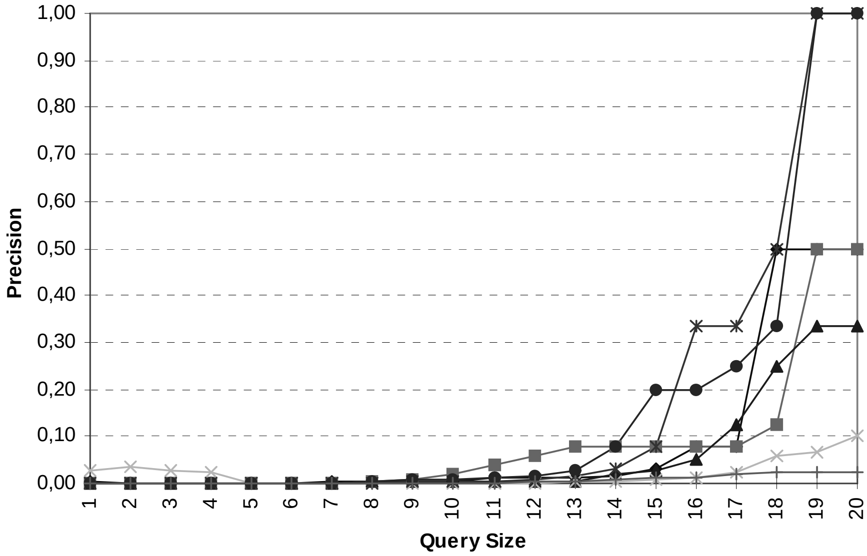


FIGURE 37. 16-bit signature, 7-character word, 1 bit per word

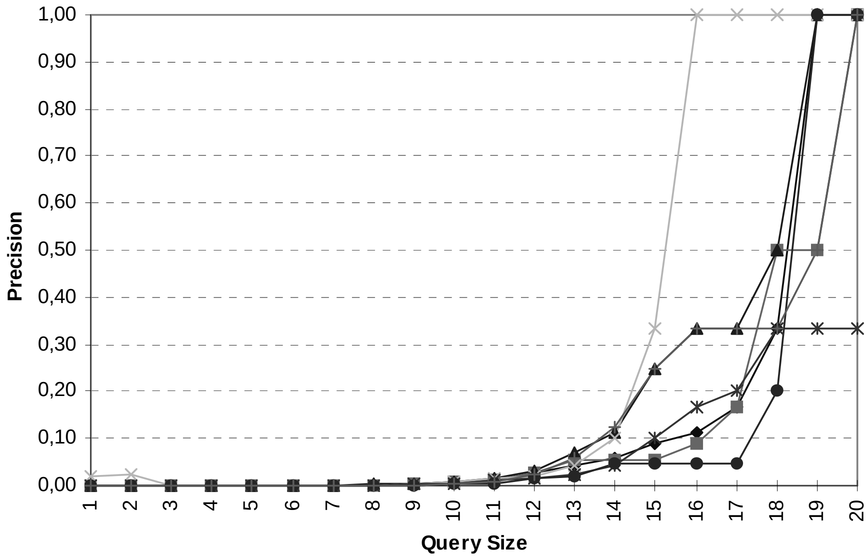


FIGURE 38. 16-bit signature, 9-character word, 1 bit per word

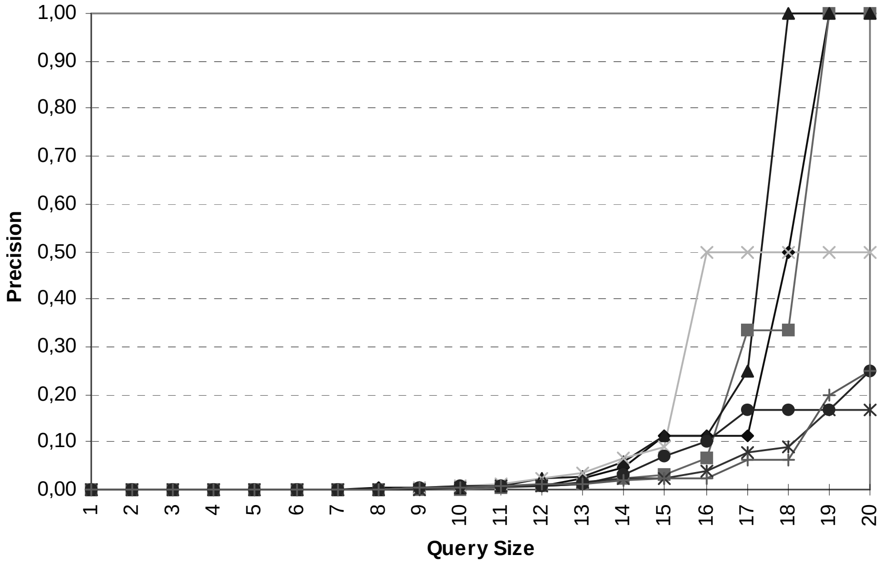


FIGURE 39. 16-bit signature, 20-character word, 1 bit per word

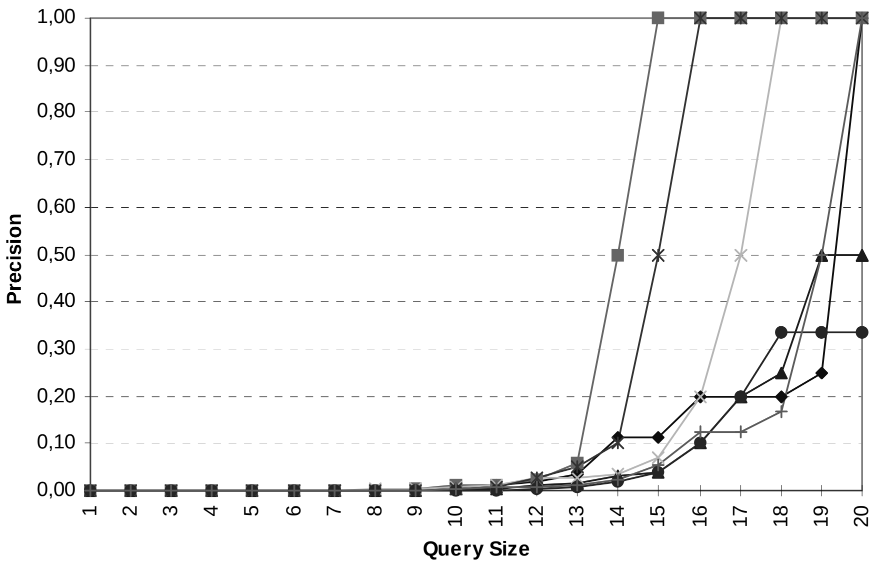


FIGURE 40. 24-bit signature, 3-character word, 1 bit per word

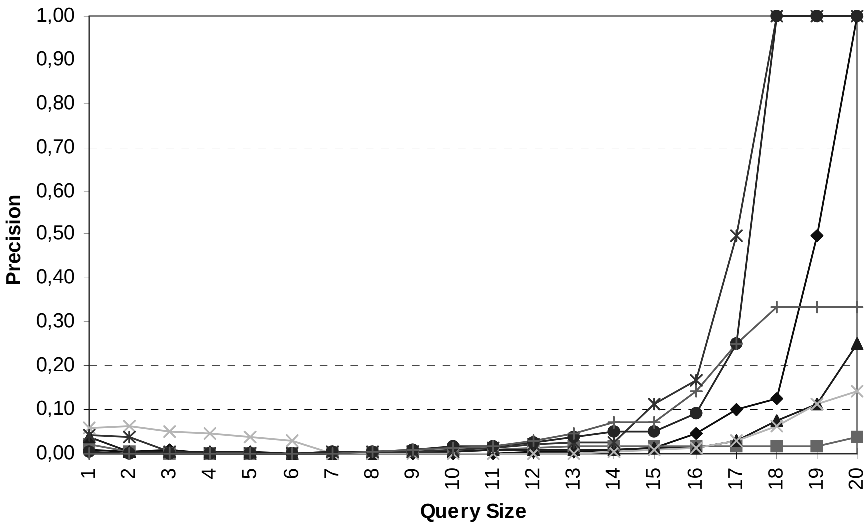


FIGURE 41. 24-bit signature, 5-character word, 1 bit per word

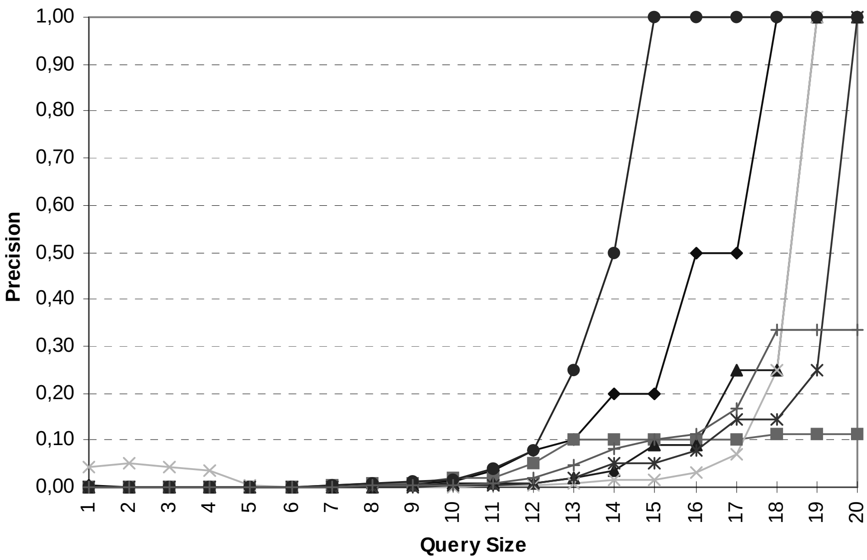


FIGURE 42. 24-bit signature, 7-character word, 1 bit per word

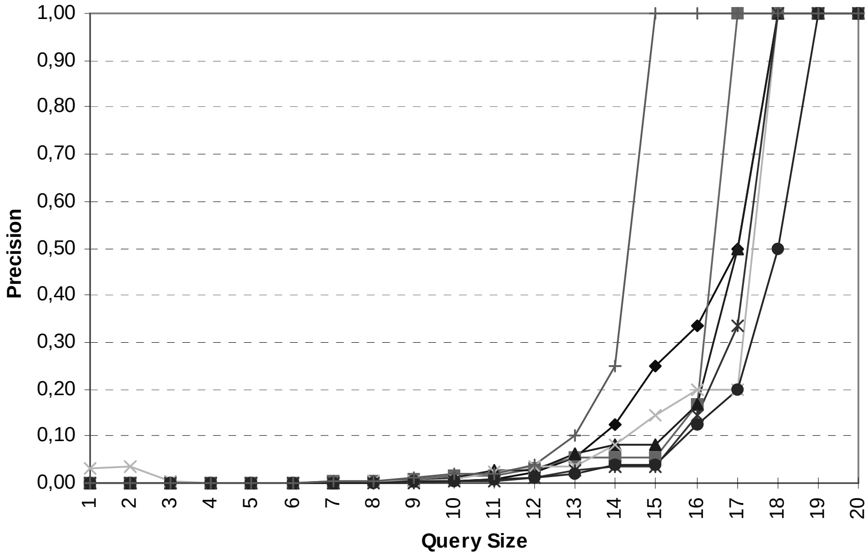


FIGURE 43. 24-bit signature, 9-character word, 1 bit per word

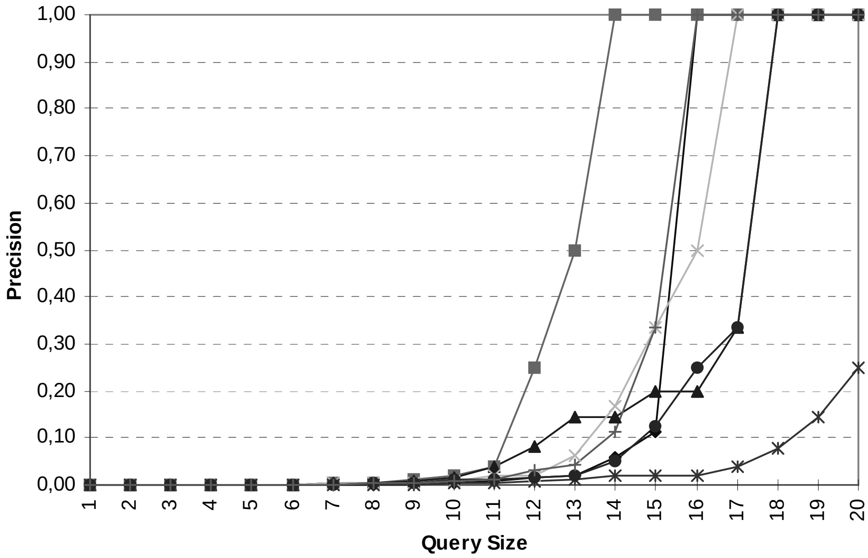


FIGURE 44. 24-bit signature, 20-character word, 1 bit per word

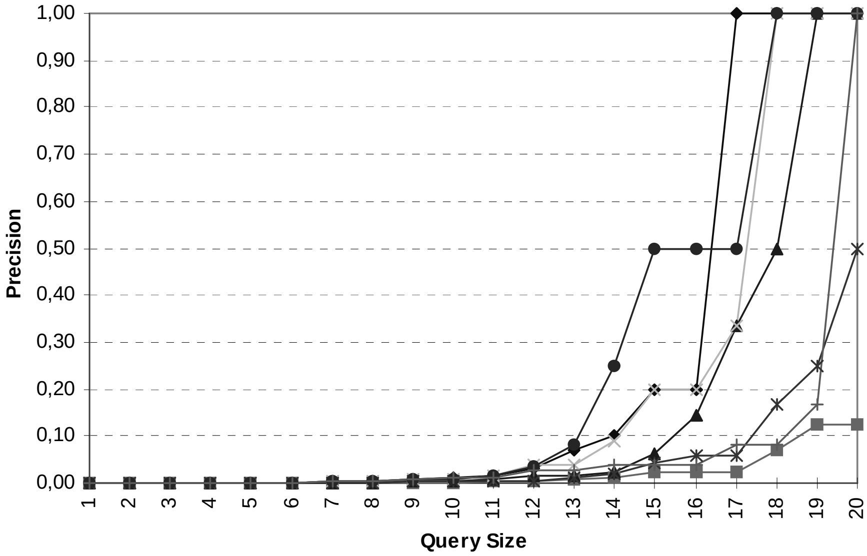


FIGURE 45. 61-bit signature, 3-character word, 1 bit per word

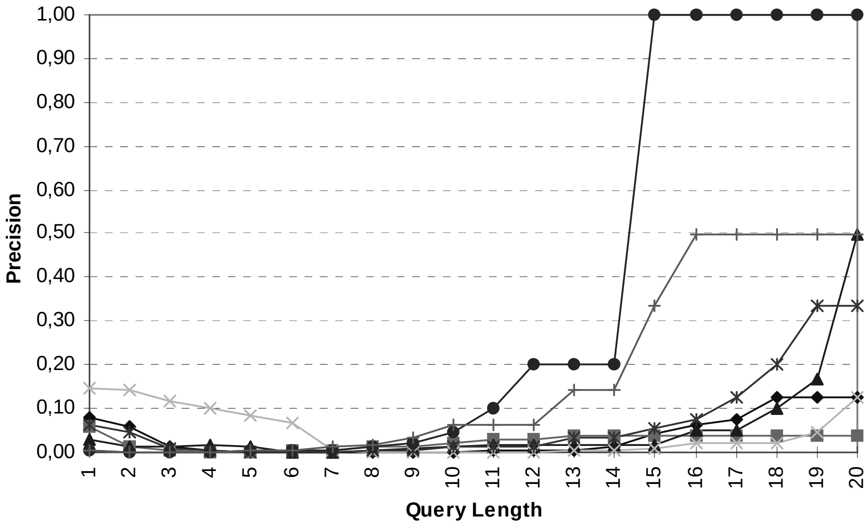


FIGURE 46. 61-bit signature, 5-character word, 1 bit per word

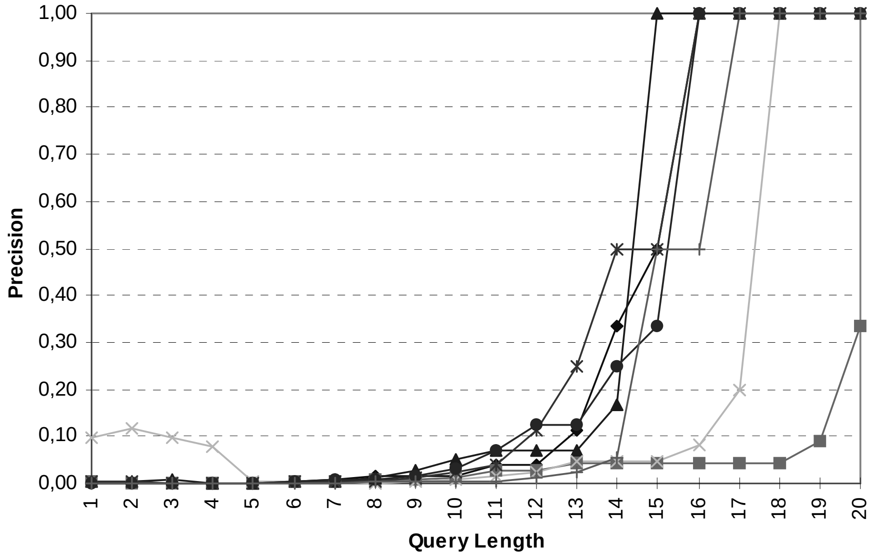


FIGURE 47. 61-bit signature, 7-character word, 1 bit per word

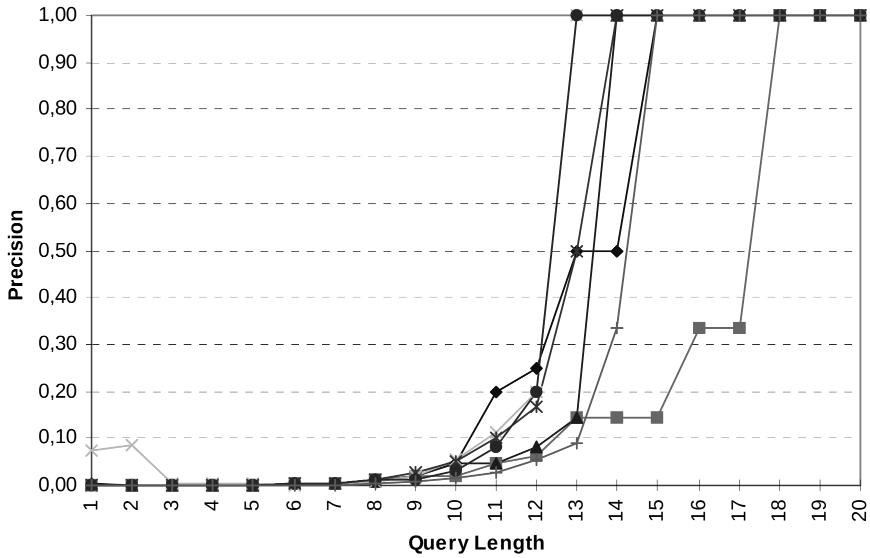


FIGURE 48. 61-bit signature, 9-character word, 1 bit per word

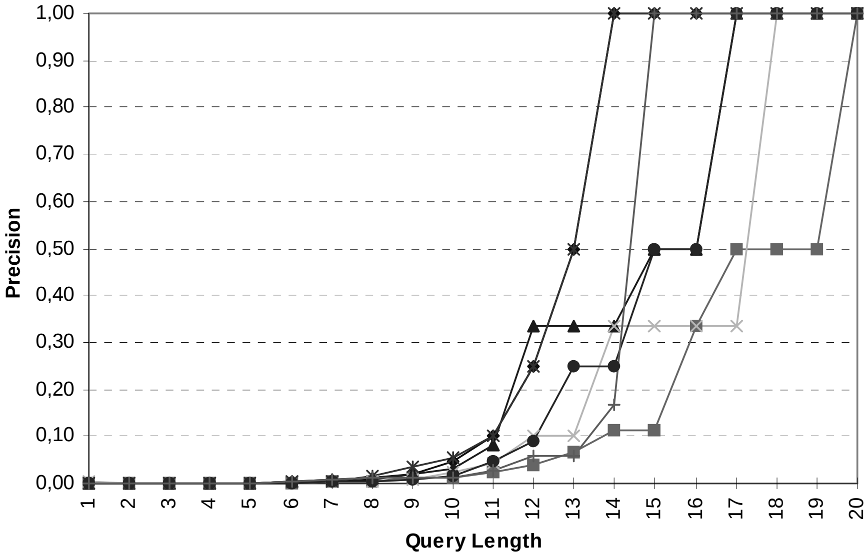


FIGURE 49. 61-bit signature, 20-character word, 1 bit per word

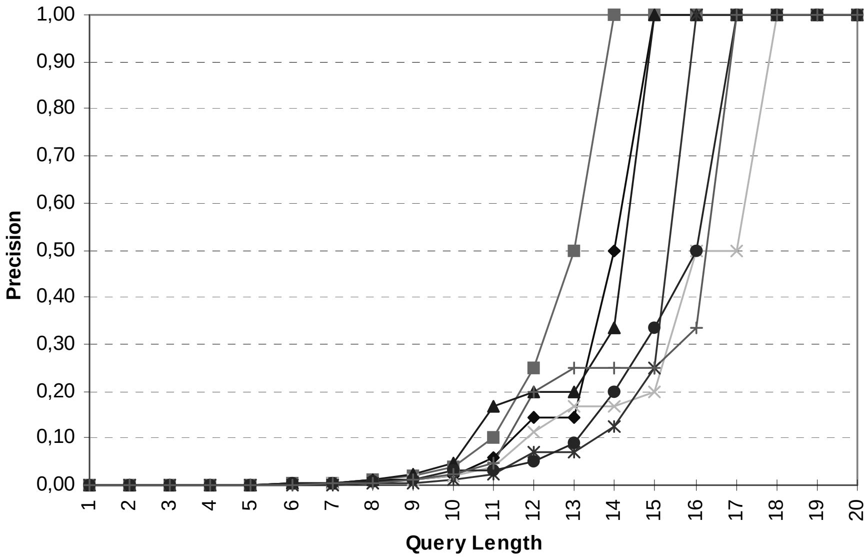


FIGURE 50. 251-bit signature, 3-character word, 1 bit per word

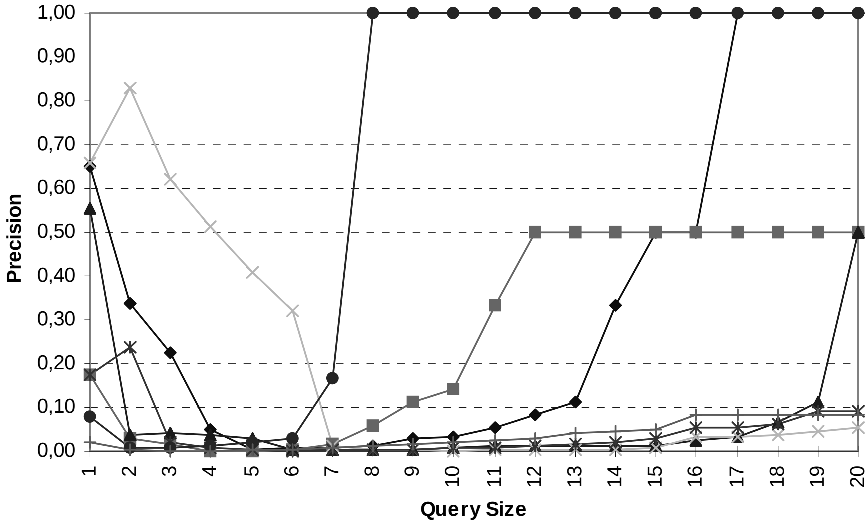


FIGURE 51. 251-bit signature, 5-character word, 1 bit per word

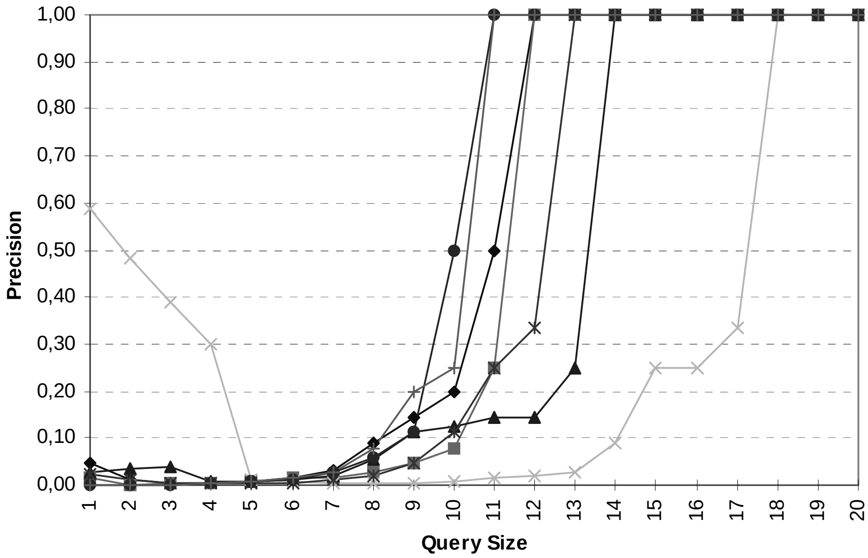


FIGURE 52. 251-bit signature, 7-character word, 1 bit per word

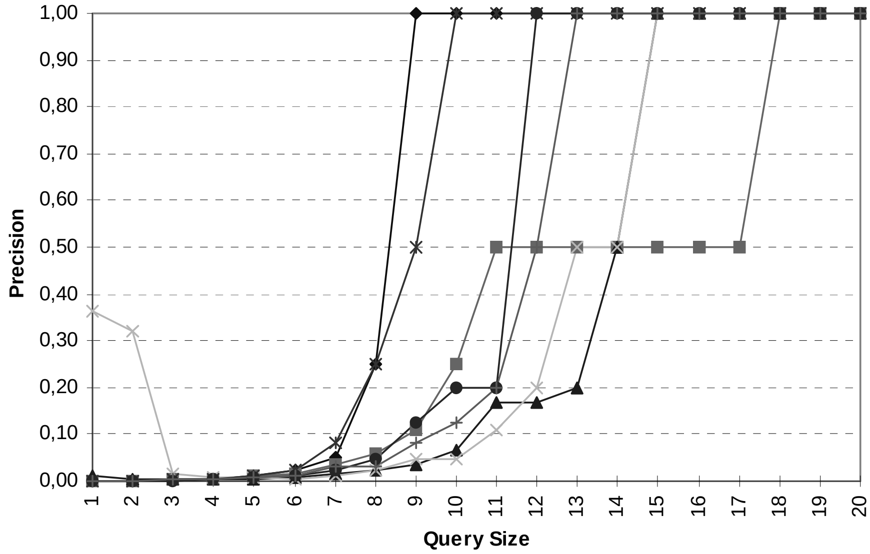


FIGURE 53. 251-bit signature, 9-character word, 1 bit per word

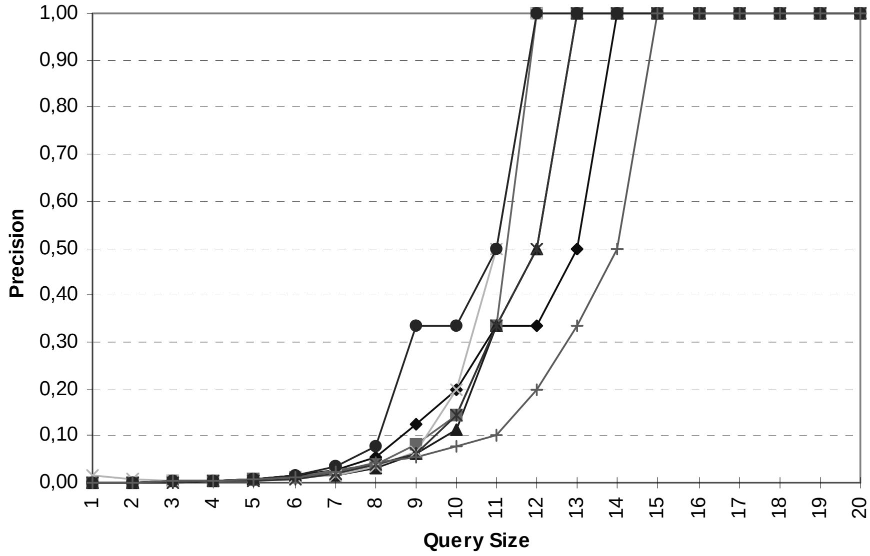


FIGURE 54. 251-bit signature, 20-character word, 1 bit per word

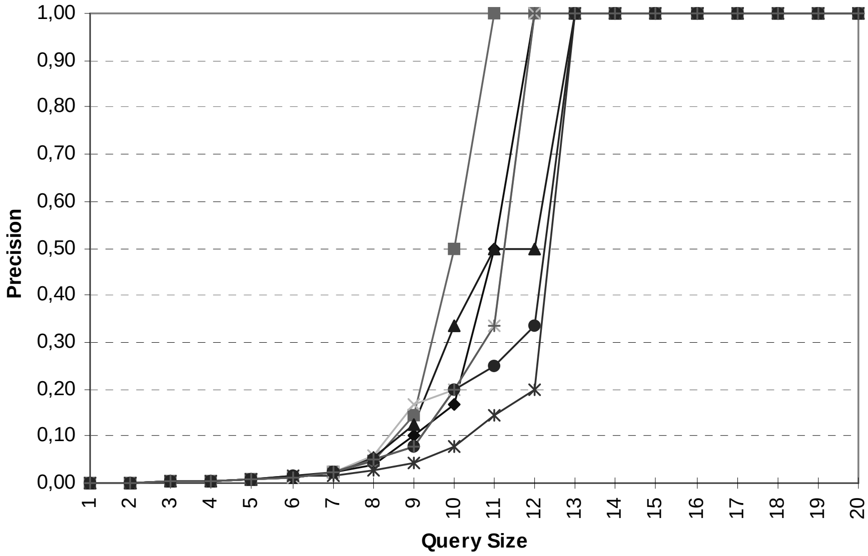


FIGURE 55. 509-bit signature, 3-character word, 1 bit per word

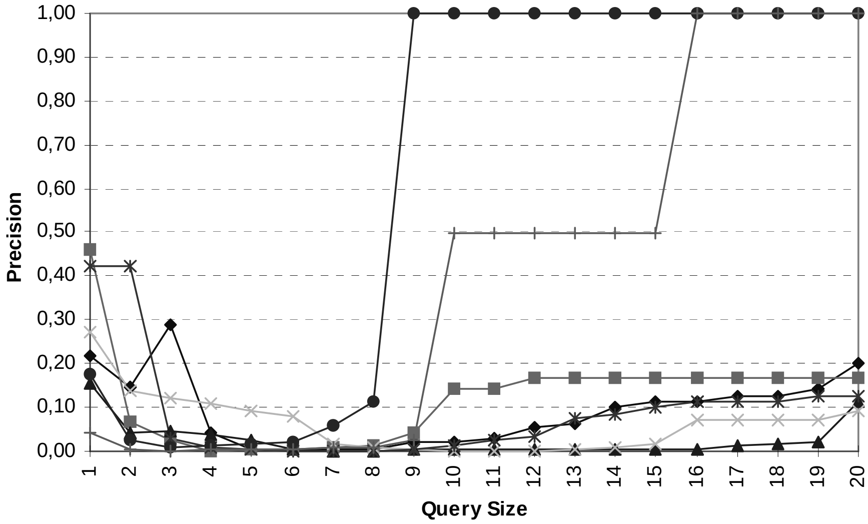


FIGURE 56. 509-bit signature, 5 character word, 1 bit per word

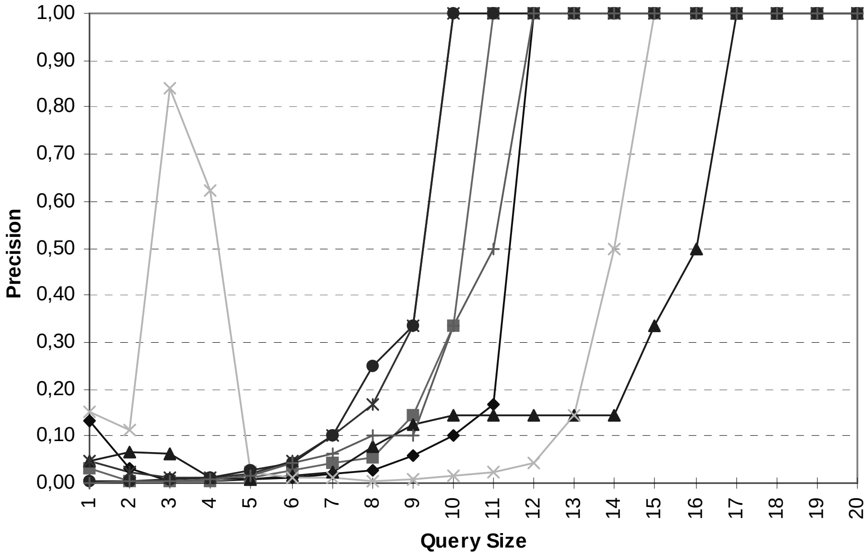


FIGURE 57. 509-bit signature, 7-character word, 1 bit per word

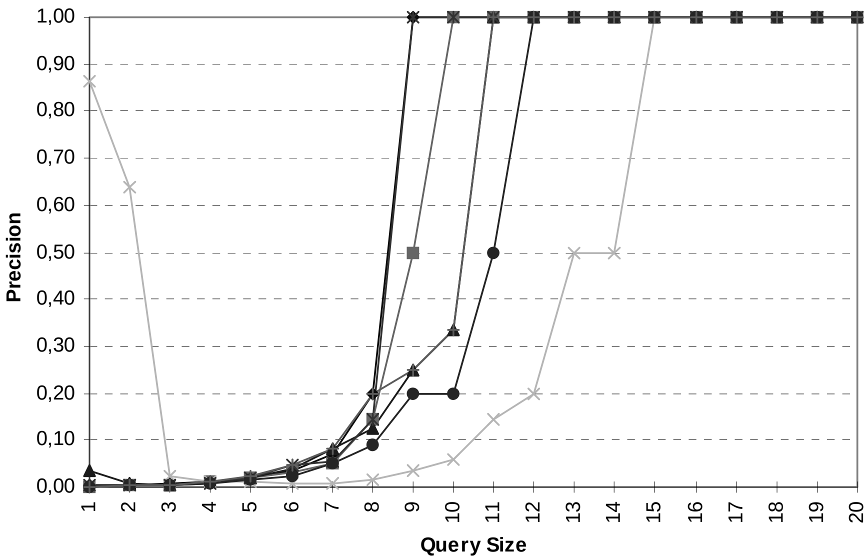


FIGURE 58. 509-bit signature, 9-character word, 1 bit per word

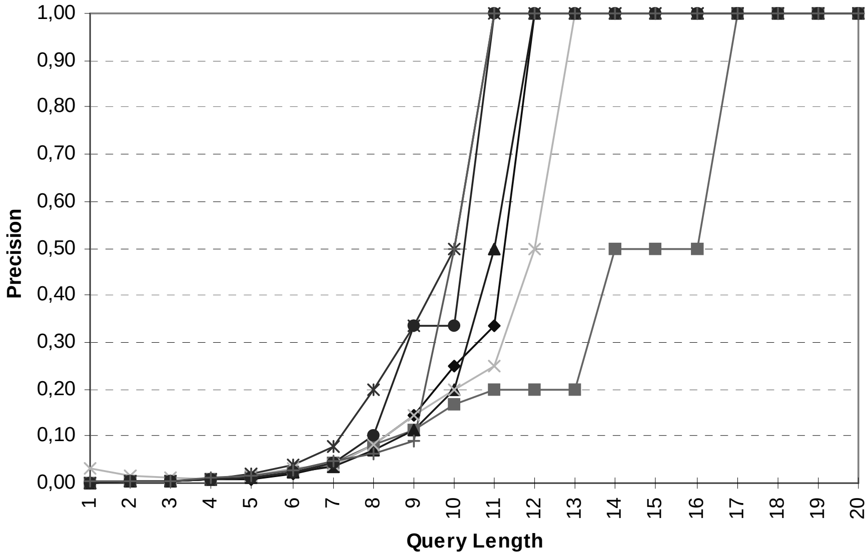


FIGURE 59. 509-bit signature, 20-character word, 1 bit per word

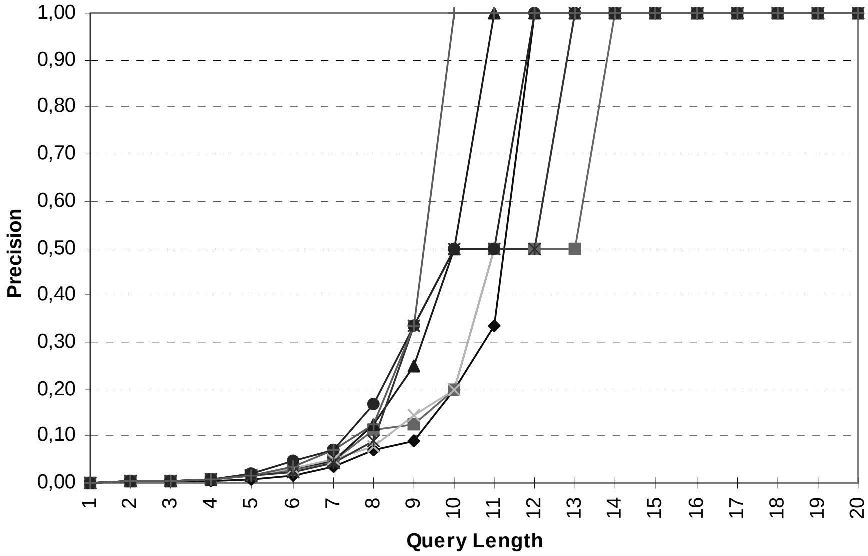


FIGURE 60.

16-bit signature, 3-character word, 4 bits per word

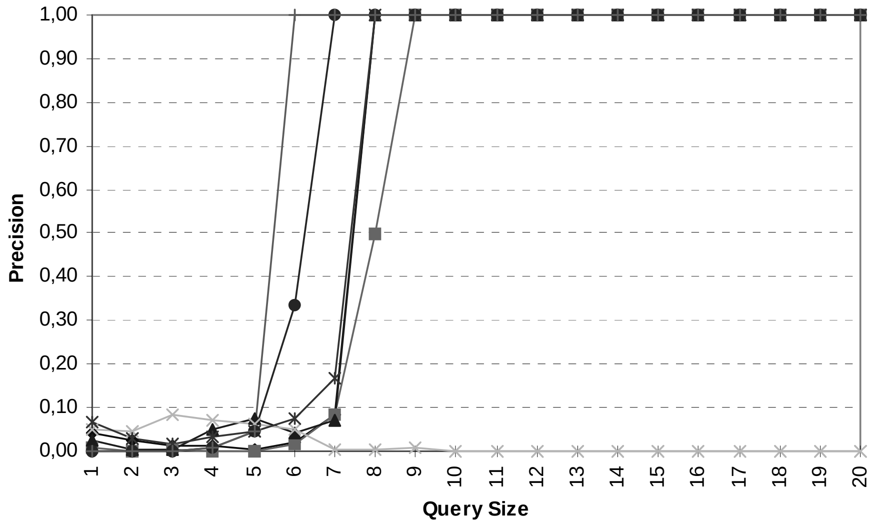


FIGURE 61.

16-bit signature, 5 character word, 4 bits per word

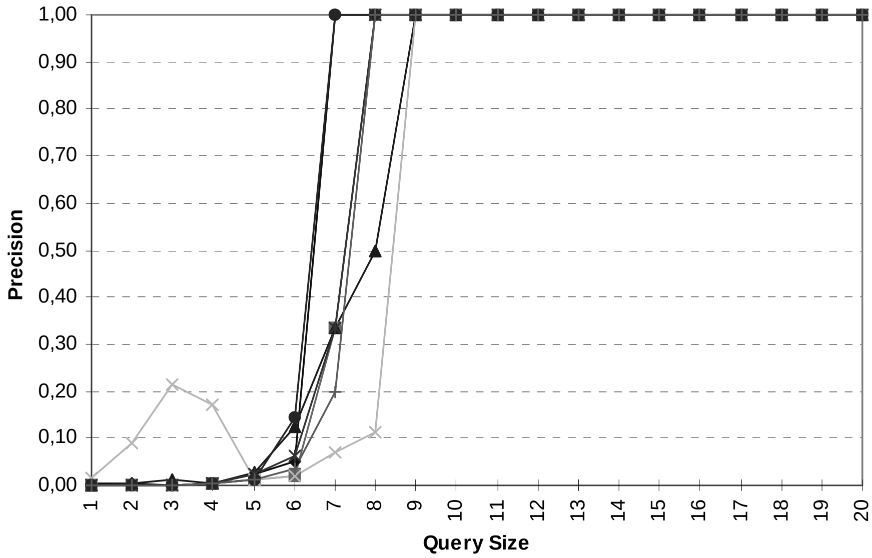


FIGURE 62. 16-bit signature, 7-character word, 4 bits per word

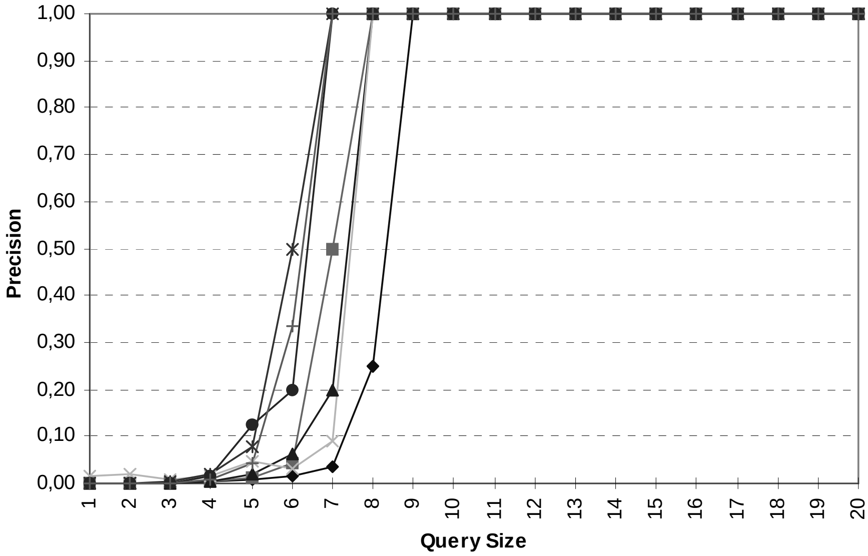


FIGURE 63. 16-bit signature, 9-character word, 4 bits per word

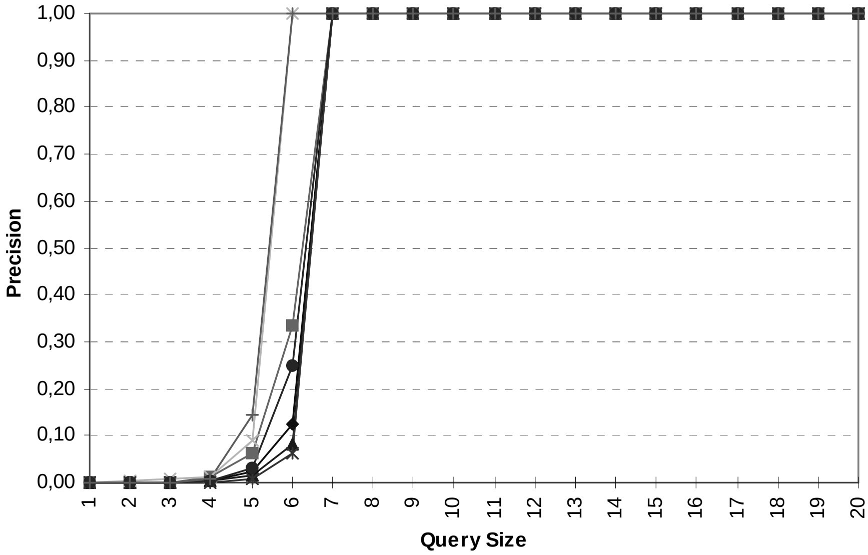


FIGURE 64. 16-bit signature, 20-character word, 4 bits per word

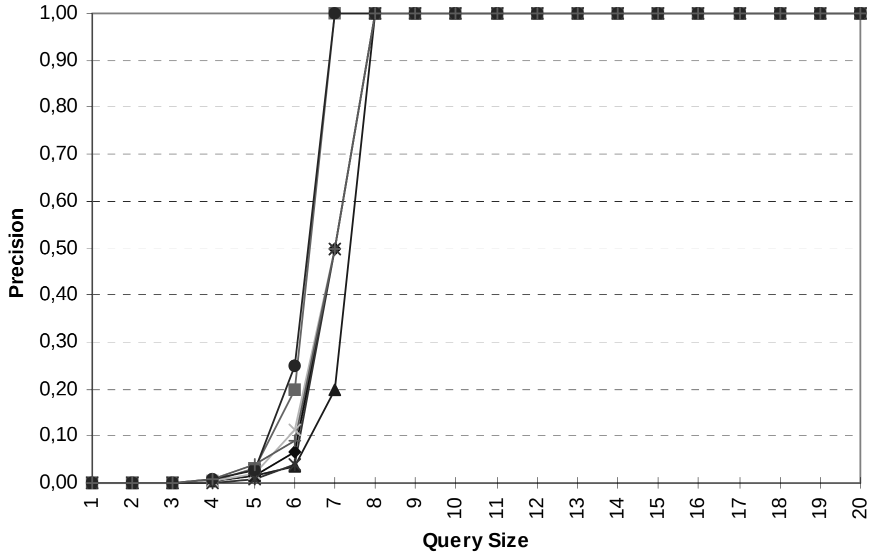


FIGURE 65. 24-bit signature, 3-character word, 4 bits per word

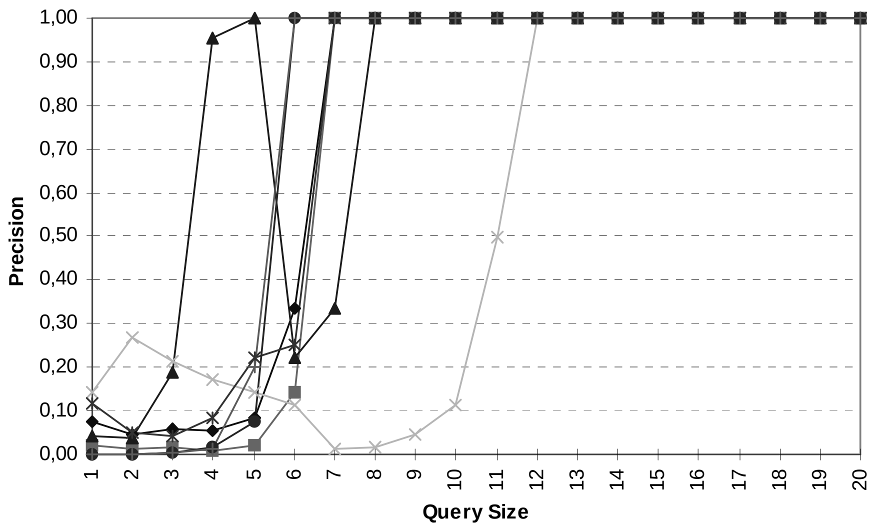


FIGURE 66. 24-bit signature, 5-character word, 4 bits per word

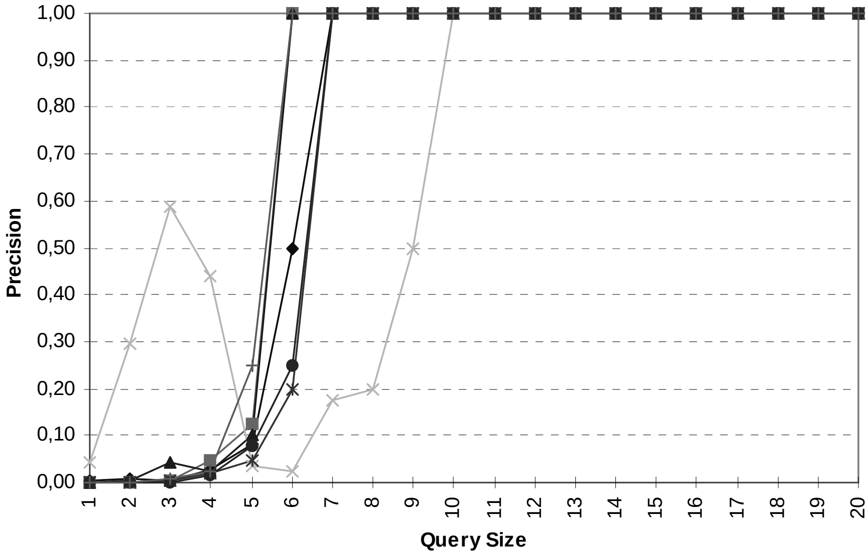


FIGURE 67. 24-bit signature, 7-character word, 4 bits per word

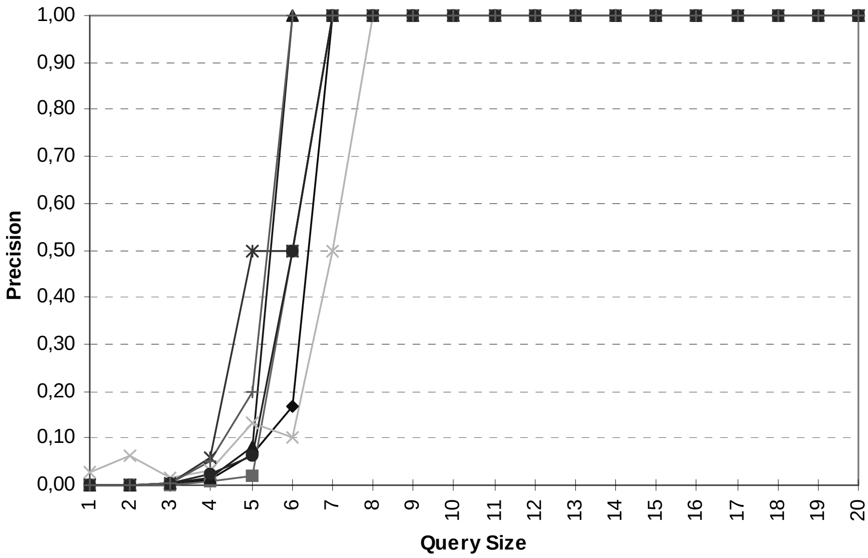


FIGURE 68.

24-bit signature, 9-character word, 4 bits per word

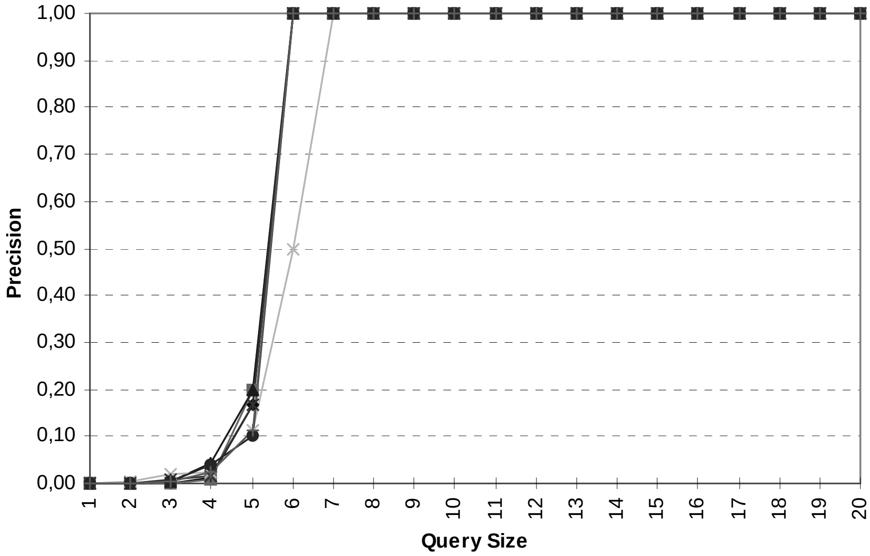


FIGURE 69.

24-bit signature, 20-character word, 4 bits per word

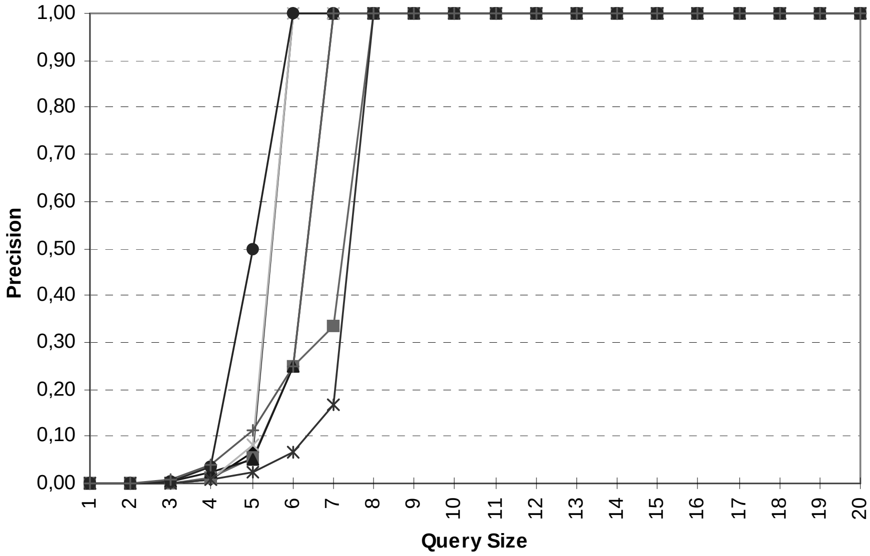


FIGURE 70. 61-bit signature, 3-character word, 4 bits per word

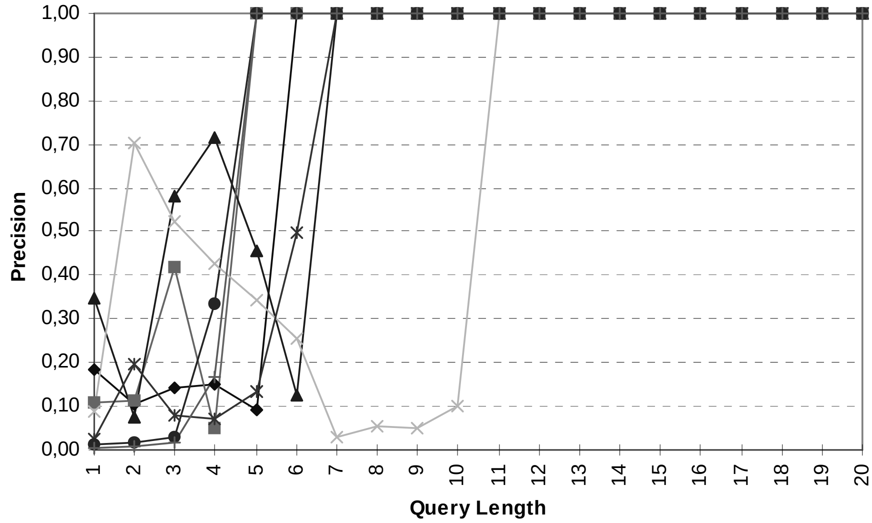


FIGURE 71. 61-bit signature, 5-character word, 4 bits per word

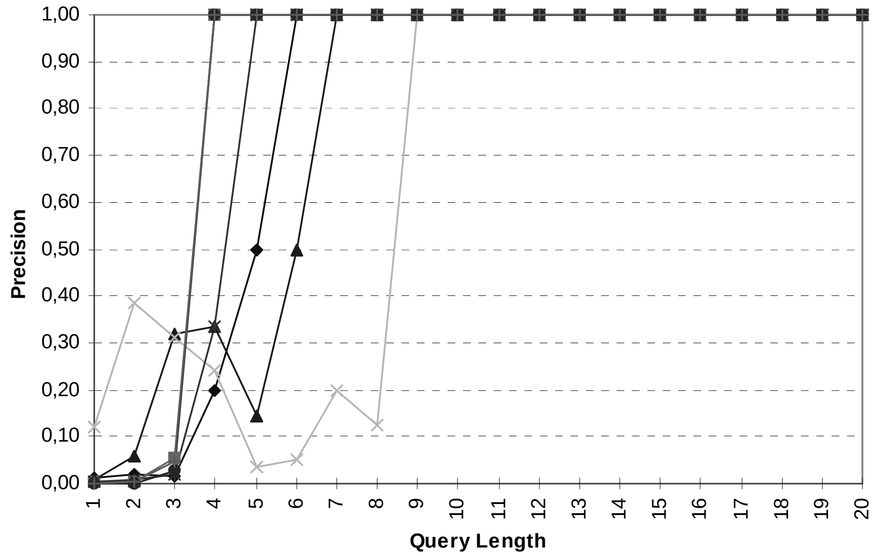


FIGURE 72.

61-bit signature, 7-character word, 4 bits per word

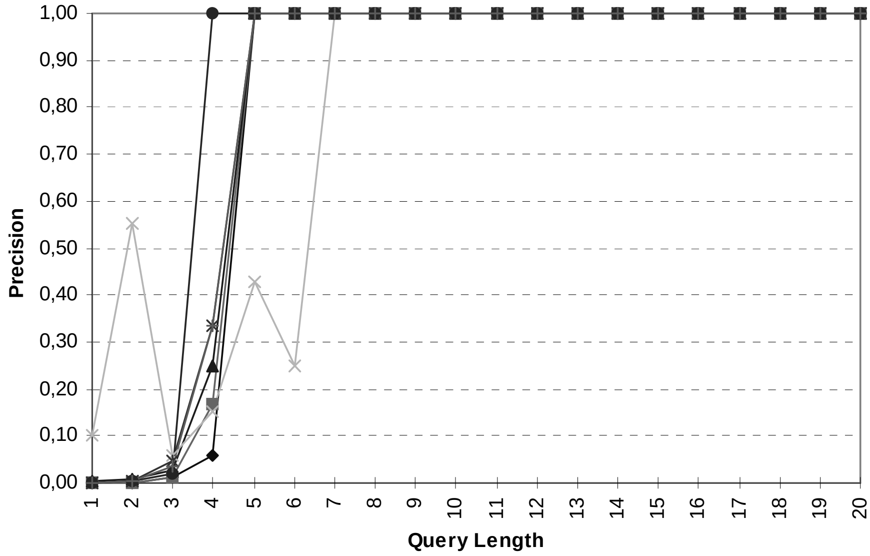


FIGURE 73.

61-bit signature, 9-character word, 4 bits per word

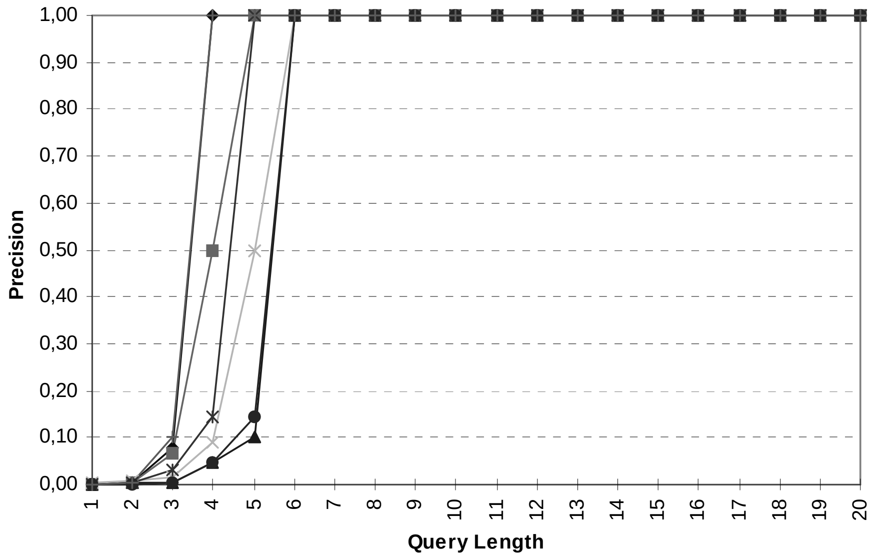


FIGURE 74.

61-bit signature, 20-character word, 4 bits per word

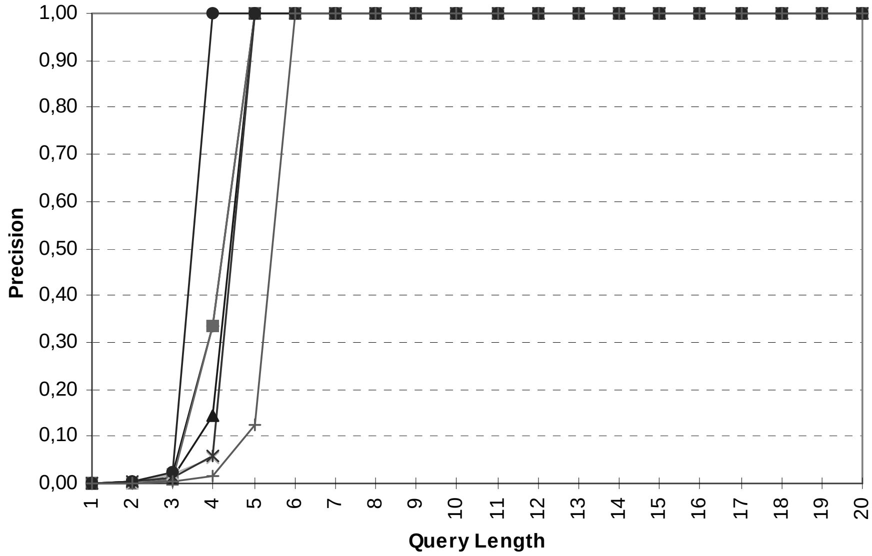


FIGURE 75.

251-bit signature, 3-character word, 4 bits per word

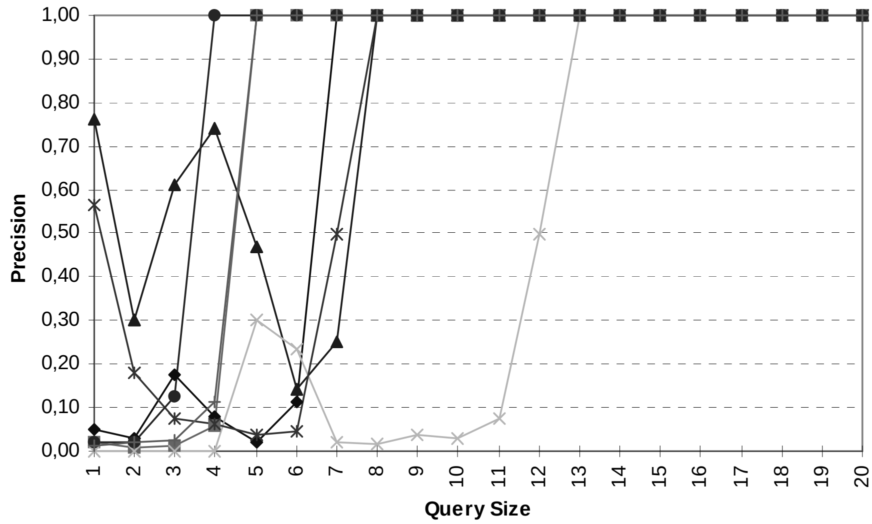


FIGURE 76. 251-bit signature, 5 character word, 4 bits per word

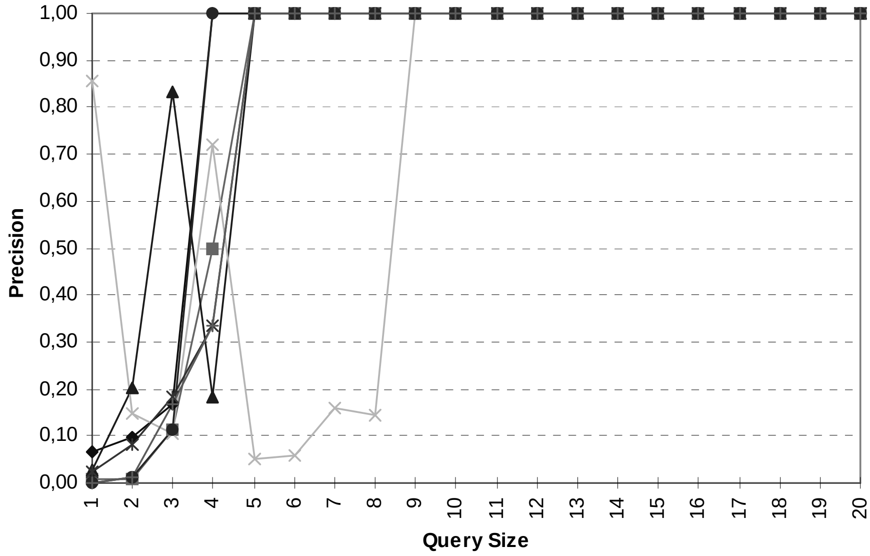


FIGURE 77. 251-bit signature, 7-character word, 4 bits per word

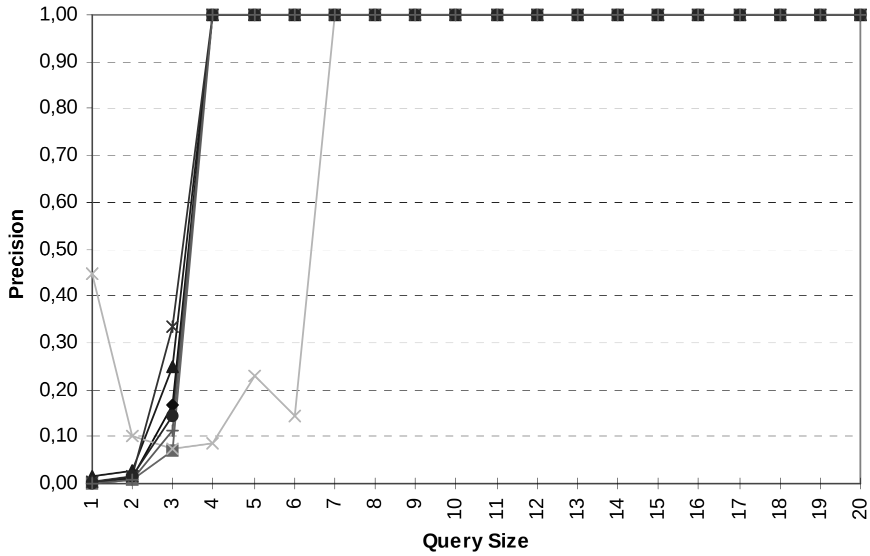


FIGURE 78. 251-bit signature, 9-character word, 4 bits per word

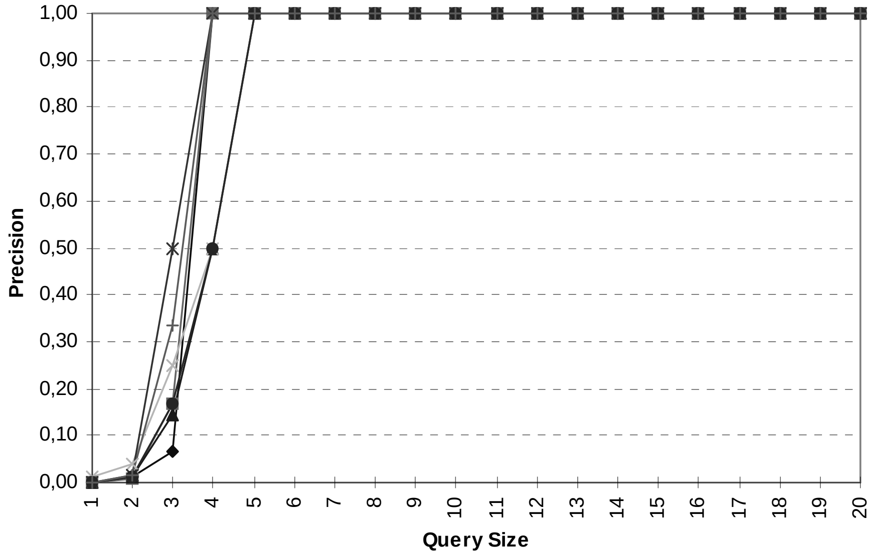


FIGURE 79. 251-bit signature, 20-character word, 4 bits per word

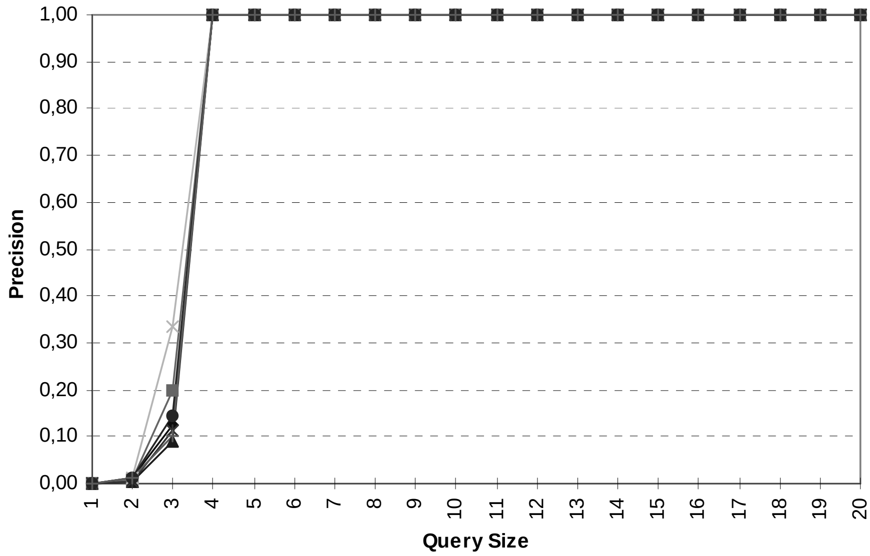


FIGURE 80. 509-bit signature, 3-character word, 4 bits per word

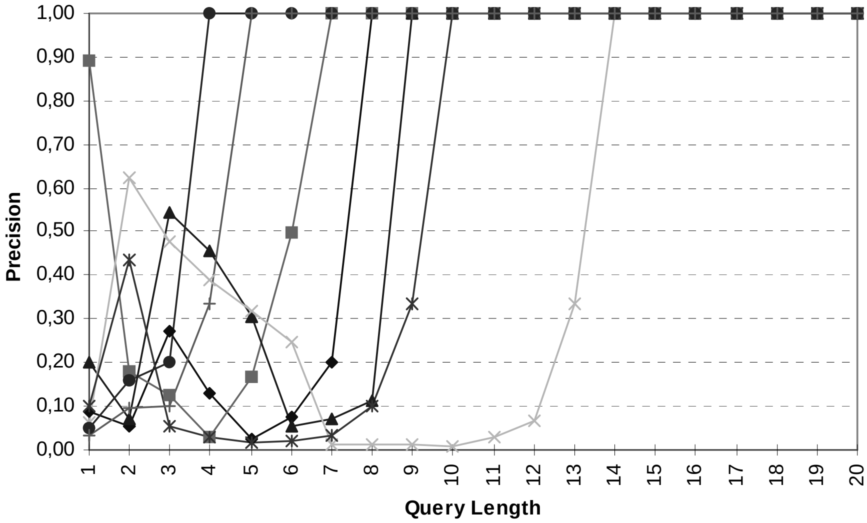


FIGURE 81. 509-bit signature, 5 character word, 4 bits per word

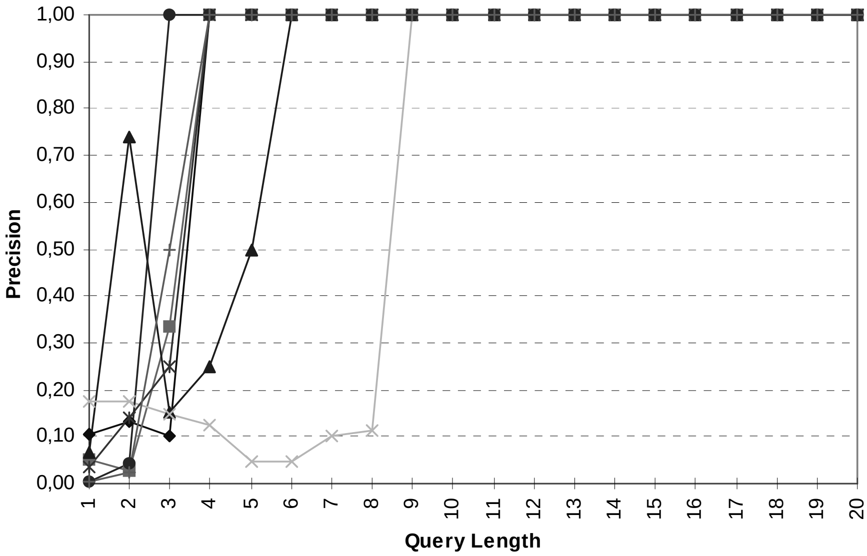


FIGURE 82. 509-bit signature, 7-character word, 4 bits per word

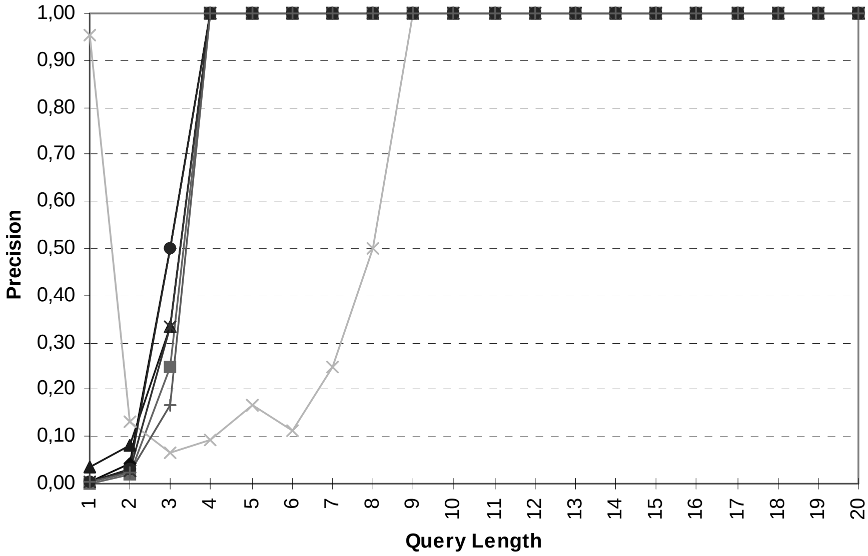


FIGURE 83. 509-bit signature, 9-character word, 4 bits per word

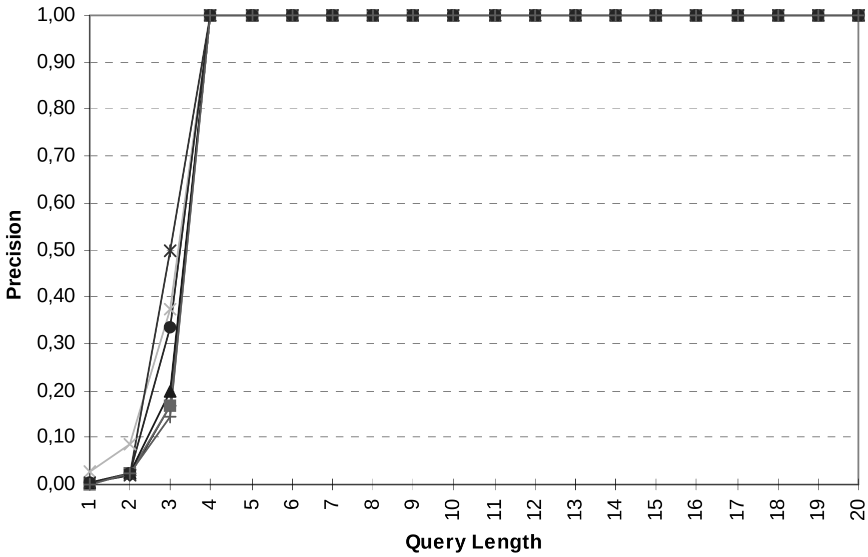
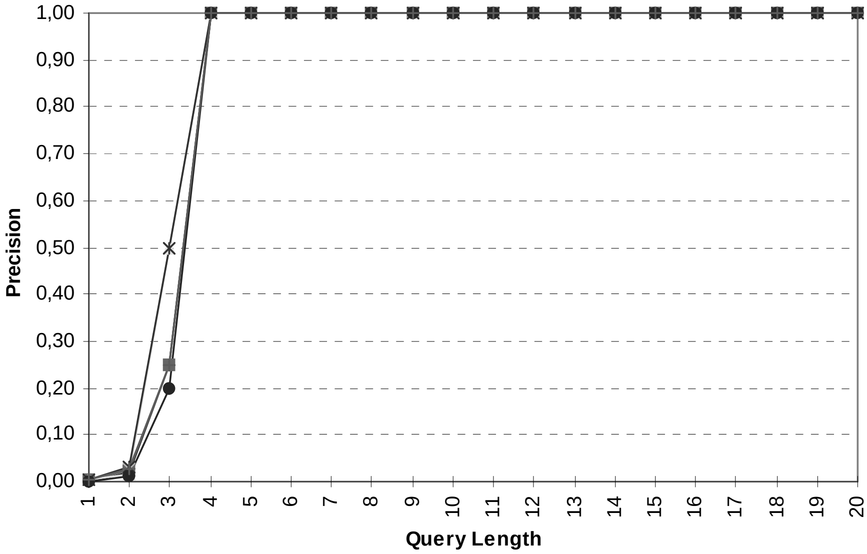


FIGURE 84. 509-bit signature, 20-character word, 4 bits per word



7 Conclusions

In this section we conclude our preliminary work in finding a new suitable index for time series. We have made an extensive survey of related work in this area and we have not found any existing method to be so general that it will work in a general application. We have started working on a new indexing method and so far our approach seems promising.

7.1 Feature Extraction

The feature extraction process is one of the areas least explored. We have chosen to use a method suggested in [AGR295], see Section 3.5, “Querying Shapes of Histories,” on page 16, because of its simplicity and that it initially offered us a way to transform the time series into text, something that we know how to index efficiently. The SDA has many benefits but as can be seen in Section 4.1, “Feature Extraction,” on page 27 there are also some problems with the method.

The concept of similarity suggested in this work is very attractive since it is very intuitive and close to how humans think of similarities. It has also the desirable property of being amplitude invariant.

If this is the only used similarity concept and the size of the alphabet is small, each query will result in a large amount of hits. Each hit will map to the same textual string but the similarity between the signals might be too vague for a human to accept the sequences as similar. To remedy this the alphabet size can be increased. Using the same similarity concept as before the number of hits received to a query will now be fewer since there are now fewer signal segments that map into the same string. But suppose that the user wishes to state a very vague query to the system. It can not be easily done. The precision of the system is controlled by the size of the alphabet. It would be favorable to have some way of controlling the level of fuzziness at query time. We have investigated a mechanism to control the fuzziness of a query at query time, see Section 4.4, “Similarity in Signatures,” on page 41 for a discussion of the issue.

7.2 Signature Files

Most of our work so far has been to see if signature files were suitable for this kind of data and on the basis of the results we have obtained it is our belief that signatures can be well suited for this kind of data. The signature files always offer better search performance than a sequential scan of the original data and the small size compared to the B-Tree might make signature files a good solution.

We introduced a new signature file search algorithm in this work. We have experimented with two different extensions to the signature file search algorithm, see Section 4.2.1, “Different signatures,” on page 33. Since the second version proposed does not limit our queries, we chose this as our search algorithm.

7.2.1 The ST

A major problem with the signature file is that it has to be scanned sequentially and therefore it does not scale well. With the ST this will no longer be necessary. Since the search time of a tree structure roughly corresponds to the height of the tree and the height of the ST will be a function of the signature size and not the data set size, it will be possible to search the signature file efficiently in constant time as long as the ST has not saturated. The biggest problem with the ST is the size. The ST grows very fast when data is inserted, so for small sequences the ST will not be a good solution because of the massive overhead. Since the ST grows slower and slower, it will reach a break-even point for very large sequences were it might offer good performance searching large sequences. Our preliminary work shows that a B-Tree still will perform a factor of 10^3 faster than the ST. We will have to investigate if it is possible to make a better implementation of the ST and find better algorithms so that the performance gap between the B-Tree and the ST closes.

7.2.2 The B-Tree

We have shown a way of using B-Trees to index 1D data. Our preliminary calculations shows that it should be possible to get very good performance if the translated string is indexed by a B-Tree. The B-Tree should not only offer very good search performance but should scale very nicely.

7.3 Test Data

We have not seen any differences in precision on our different sets of data. Future tests with different test data will have to show if this is true for all various signal or just a coincidence with the signals we have tested with.

7.4 The Experiments

As we show in Section 6.2.1, “The Tests,” on page 66 it is possible to achieve good precision in almost all configurations. The actual choice of implementation has to be decided by other parameters, the size of the signature file, the size of the queries we wish to use and the signature size. Since the size of the ST is related to the size of the signatures we are interested in using small signatures so that the ST will not be too big. We are also interested in using small windows so that we can search the index for small subsequences and from our tests we can see that it is possible to use this combination and still get good precision.

It is also worth noting that when we let each window set four bits in the signature, the precision does not differ too much between the different window sizes, and when we use a window of size 7 or larger the improvements in the precision is very small.

7.5 Conclusions

In our experiments we have shown that it is possible to achieve good precision even for small signatures and small windows. The biggest problem with our approach today is the search speed of the signature file. Even if the signature files are faster to search than just sequentially searching the original time series it is still a sequential scan and for some data size other indexing methods, e.g. B-Trees, will scale much better. We have also shown a theoretical way to index the signature file, the ST. The ST improves the search speed of the signature file dramatically but does not offer the same performance as a B-Tree. We will continue work on this and see if the ST can be improved.

We feel that this approach of transforming a signal into a text string and then using traditional text indexing methods to index it looks very promising and it is our belief that the remaining issues will be clarified in our future work.

8 Future Work

Since this is work in progress there are several loose ends and phenomena that have not been thoroughly investigated yet, but in this section we will provide a short summary of the most pressing issues.

The feature extraction is the part of our work that has the most potential for growth. Since our entire approach is so novel we have concentrated our efforts on other aspects of the system and not emphasized the feature extraction process. Future work in this area will include a more thorough examination of different extraction techniques and investigations of how the concept of similarity is in each model and how the blurry matching can be controlled.

The precision results shown in Section 6.2.1, “The Tests,” on page 66 should be presented as statistical values. For each configuration a mean precision value has to be calculated and a standard deviation together with min/max values of the precision. This could have been done in this work but we felt that it would be unfair to the reader to present the material in that way since the number of tests with each configuration is too small. Future tests has to be conducted where each configuration is tested with more queries so the data can be presented with statistical data instead of actual precision for each test query.

The ideas and thoughts behind the ST are given in this work but the ST has to be implemented and shown to be faster than existing index structures. We will also have to continue our theoretical investigation of the ST and show at what data sizes the ST will start to outperform existing indexing methods.

Another phenomenon that has to be looked into is how this index is to be used by users and applications. Some sort of query language has to be proposed and when we start to look into the query language we have to deal with query optimization: can we combine “signal”-queries with SQL queries?

9 References

[AGRA95] Fast Similarity Search in the Presence of Noice, Scaling and Translations in Time-Series Databases, *In Proc. of the 1995 VLDB Conf, pp. 490-501*, R. Agrawal, K.-I. Lin, H.S. Sawhney, K. Shim.

[AGR295] Querying Shapes of Histories, *In Proc. of the 21st VLDB Conf., pp. 502-514,* R. Agrawal, G. Psaila, E.L. Wimmers, M. Zaït.

[BADA95] Investigation of Unstructured Text Indexingm, *In Proc. of DEXA95 Int. Conf./Workshop on Database and Expert Systems, pp. 387-396*, D.Z. Badal, M.L. Davis.

[BAYER72] Organization and maintenance of large ordered indices. *Acta Informatica 1(3) 1972 pp. 173-189*, R. Bayer, E.M. McCreight.

[BECK90] The R*-tree: an efficient and robust access method for points and rectangles, *In Proc. of ACM SIGMOD, pp. 322-331*, N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger.

[DAVI95] Signature Based Ranked Retrieval in Unstructured Text, *Master of Science Thesis, University of Colorado, May 1995*, M.L.Davis.

[DEPP86] S-tree: A Dynamic Balanced Signature Index for Office Retrieval, *In Proc. of ACM Conf. on Research and Development in Information Retrieval, pp. 77-87*, U. Deppisch.

[FAGI79] Extendible hashing: A fast access method for dynamic files. *In ACM Trans. Database Systems 4(3) 1979, pp. 315-344*, R. Fagin, J. Nievergelt, N. Pippenger, R. Strong.

[FALO84] Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation, *In ACM Transactions on Office Information Systems, Vol 2, No. 4, pp.237-257*, C. Faloutsos, S. Christodoulakis.

[FALO87] Description and Performance Analysis of Signature Methods, *In ACM Transactions on Office Information Systems, Vol 5, No. 3, pp. 267-288*, C. Faloutsos, S. Christodoulakis.

[FALO94] Fast Subsequence Matching in Time-Series Databases, *In SIGMOD 94 5/94, pp. 419-429*, C. Faloutsos, M. Ranganathan, Y. Manolopoulos.

[GUTT84] R-trees: A dynamic index structure for spatial searching. *In Proc. ACM SIGMOD Int. Conf. on Management of Data 1984 pp. 47-54*. A. Guttman.

[LEE89] Partitioned Signature File: Design and Performance Evaluation, *In ACM Transactions on Office Information Systems, Vol. 7, No. 2, pp. 158-180*, D.L. Lee, C.W. Leng.

[LI96] HierarchScan: A Hierarchical Similarity Search Algorithm for Databases of Long Sequences, *In Proc. of the 12th International Conference on Data Engineering, pp. 546-553*, C.-S. Li, P.S. Yu, V. Castelli.

[LIN98] Querying Continous Time Sequences, *In The 24th International Conference on Very Large Databases (VLDB'98)*, *pp. 170-181,* L. Lin, T. Risch.

[LIN99] Management of 1-D Sequence Data - From Discrete to Continous, Dissertation No. 561, Linköping University, L. Lin.

[LITW80] Linear hashing: A new tool for file and table addressing.*In Proc. 6th Int. Conf. on Very Large Databases, pp. 212-223*, W. Litwin.

[OTOO86] Balanced Multidimensional Extendible Hash Tree, *In Proc. of the 5th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, pp. 100-113*, E.J. Otoo.

[RIVE92] R.L. Rivest, rivest@theory.lcs.mit.edu, *http://www.columbia.edu/~ariel/ssleay/rfc1321.html.*

[ROBE79] Partial Match Retrieval via the Method of Superimposed Codes, *In Proc. IEEE 67, pp. 1624-1642*, C.S. Roberts.

[SACK87] Multikey Access Methods Based on Superimposed Coding Techniques, *In ACM Transactions on Database Systems, Vol 12, No. 4, pp.655-696*, R. Sacks-Davis, A. Kent, K. Ramamohanarao.

[SHAT96] Approximate Queries and Representations for Large Data Sequences, *In Proc. of the 13th International Conference on Data Engineering, pp. 536-545*, H. Shatkay, S.B. Zdonik.

[ZEZU91] Dynamic Partitioning of Signature Files, *In ACM Transactions on Information Systems, Vol. 9, No. 4, pp. 336-369*, P. Zezula, F. Rabitti, P. Tibero.

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.
No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.
No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.
No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.
No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.
No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.
No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.
No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.
No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.
No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.
No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.
No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.
No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.
No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.
No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.
No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.
No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.
No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
No 318 **Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.
No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.
No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.
No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.
No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.
No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.
No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997.
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.