Indexing Strategies for Time Series Data

by

Henrik André-Jönsson

Linköping Studies in Science and Technology
Dissertation No. 757

ISBN: 91-7373-346-6

ISSN: 0345-7524

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

Linköping 2002

Abstract

Traditionally, databases have stored textual data and have been used to store administrative information. The computers used, and more specifically the storage available, have been neither large enough nor fast enough to allow databases to be used for more technical applications. In recent years these two bottlenecks have started to disappear and there is an increasing interest in using databases to store non-textual data like sensor measurements or other types of process-related data. In a database a sequence of sensor measurements can be represented as a time series. The database can then be queried to find, for instance, sub-sequences, extrema points, or the points in time at which the time series had a specific value. To make this search efficient, indexing methods are required. Finding appropriate indexing methods is the focus of this thesis.

There are two major problems with existing time series indexing strategies: the size of the index structures and the lack of general indexing strategies that are application independent. These problems have been thoroughly researched and solved in the case of text indexing files. We have examined the extent to which text indexing methods can be used for indexing time series.

A method for transforming time series into text sequences has been investigated. An investigation was then made on how text indexing methods can be applied on these text sequences. We have examined two well known text indexing methods: the signature files and the B-tree. A study has been made on how these methods can be modified so that they can be used to index time series. We have also developed two new index structures, the signature tree and paged trie structures. For each index structure we have constructed cost and size models, resulting in comparisons between the different approaches.

Our tests indicate that the indexing method we have developed, together with the B-tree structure, produces good results. It is possible to search for and find sub-sequences of very large time series efficiently.

The thesis also discusses what future issues will have to be investigated for these techniques to be usable in a control system relying on time-series indexing to identify control modes.

Acknowledgments

I would like to start by thanking my two main supervisors, Prof. Tore Risch, for accepting me as a doctoral student at EDSLAB and for teaching me what makes a database tick, and Prof. Nahid Shahmehri, for accepting the role as my main supervisor when Prof. Tore Risch moved to a new position at Uppsala University. Prof. Nahid Shahmehri has helped and guided me through the long process of putting my work together and creating a thesis out of the fragments. I would also like to thank the other members in my supervision committee, Prof. Lennart Ljung who has given me a completely different view on my work, Assoc. Prof. Patrick Lambrix for his valuable input, especially in the latter stages this work, and Prof. Tore Risch for staying in my supervision committee and for his valuable input on this thesis. Additionally I would like to thank Dr. Dushan Badal who, during a sabbatical at EDSLAB, got me started on this project. His ideas and visions started it all.

A project like this would not have been fun without colleagues to discuss the topics with; the former members of the EDSLAB and the current members of the ADIT/IIS-LAB, Prof. Robert Sebasta at the computer science department at UCCS, Jens-Olof Lind, Rich, and Darleene, a big thank you to you all.

Brittany Shahmehri's help with proof-reading has been very valuable. Anne Eskilsson, Kit Burmeister, Jeanette DeCastro, Britt-Inger Karlsson, and Helene Wigert have all helped with different administative tasks over the years. A special thank you to Lillemor Wallgren for all her help and advice.

On a more personal note I would like to thank my current colleagues at Xelin R&D for their support, Anders Törne, Peter Loborg, Hans Olsén, and Anders Henriksson. May we continue to have fun in the future...

Finally I would like to save my biggest thank you to the one person that has been living in the shadow of this huge project, my beloved wife, Monika. I hope we will be able to spend more time together now.

This work has been supported by the ECSEL (Excellence Center in Computer Science and Systems Engineering in Linköping) graduate school and the ISIS (Information Systems for Industrial Control and Supervision) competence center at Linköpings Universitet.

Henrik André-Jönsson, April 2002

Table of Contents

Chapter 1

Introduction

In this chapter we will introduce the reader to our work, show how it fits into the database research field, and then discuss what this thesis will focus on. For readers not familiar with the field we will also give a crash course in the relevant areas so that the reader will have access to our underlying definitions and terminology.

Imagine the cozy environment of a library, surrounded by books and friendly personnel, maybe with a steaming cup of coffee at your side. The purpose of a library is to store large amounts of data, provide the possibility to ask questions of the personnel, and to provide an organized structure to easily find the data. This picture of a library can be applied to today's picture of digitized storage, called a database. A database is data storage and a database management system is the digitized personnel, which provides structures for fast retrieval, and methods for accessing the data stored in the database. The database management system also provides a language that enables the database user to query the data in the database.

In the books, as in the usual digitized storage, it is mostly text that is being stored. It is easy to search for textual data and to verify the result, but what happens if you want to find data that cannot be described textually, like finding all the pictures with a red house? It is possible to try to describe all of the characteristics of the house with text, but that would require a long textual description, and the result may not contain the red house that was searched for.

Imagine instead that we could search for a picture that shows a red house, and then the database would actually search for all pictures matching your description. These kind of possibilities are what we hope to be able to do with multimedia databases. They do not just store text and let the user search for textual data or keywords, but actually let the user search other kinds of data like pictures and sound.

Data like a picture contains many kinds of features you could query on: areas, boundaries, and shapes, etc. In order to query on these types of features, some mechanism is necessary to index and do fuzzy matching on large quantities of non-textual data. That is the focus of this thesis. For simplicity, we use one-dimensional sequences of data (time series).

This thesis focuses on different indexing strategies for a specific type of data and subsequences of very large time series. These indexing strategies are useful for the final step in locating the actual data in the database. We wish to find data that is similar to the query as fast as possible, with respect to other properties, e.g. memory consumption, added storage space etc. The rest of the database system will not be discussed in this thesis.

The next section will continue with a more thorough introduction to the database field. Since time series databases are traditionally considered to be temporal databases, we will continue our introduction with a brief overview of temporal databases. This will eventually lead us to a discussion of time series, and finally we will focus on different index structures.

The rest of the introduction will describe the actual problem and the proposed solutions and conclude with a description of the contributed work.

For those unfamilar with basic indexing techniques, please consult Appendix D, “Indexing,” on page 189.

1.1 Introduction to Database Systems

A database system is a software layer built on top of a database management system (DBMS), which includes applications that allow user interaction. A DBMS is a collection of programs that enables a user to maintain a database. The database is the actual data collection; the DBMS provides facilities for defining, constructing, and manipulating the database. See Figure 1.1, “A database system,” on page 3 for a graphical view of a database system.

When we think informally about a database we think about the data collection and do not think about all the software that helps us to model the data, to keep the data consistent, enables several people to work with the data simultaneously, and above all, to access the data. When all these components work together the users will not be aware of the many layers between them and the actual data. Another important aspect of the DBMS is that it has to scale well. A user should not notice if the amount of data increases a hundred times. Neither should the user notice if the number of simultaneous users increases. The system performance will almost always degrade, but the degradation should be so small that the difference to each user is negligible.

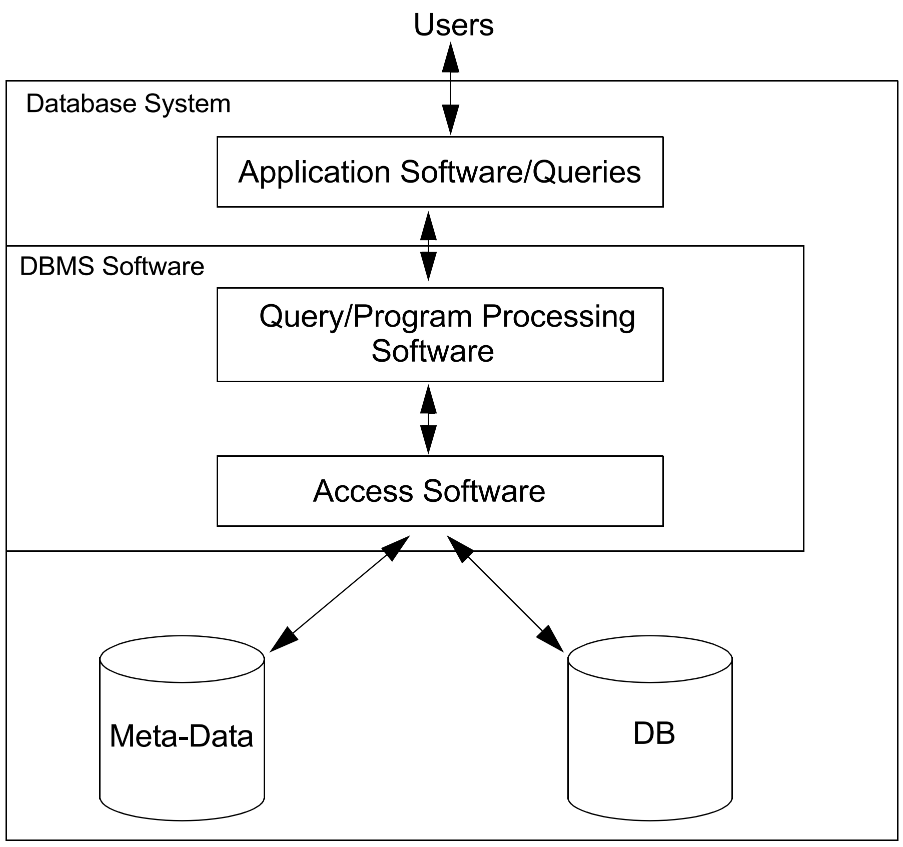

Users
Database System
Application Software/Queries
DBMS Software
Query/Program Processing Software
Access Software
Meta-Data
DB

Figure 1.1: A database system

In Figure 1.1 the user is at the top and the database system is everything below the user. If we take a look inside the database system, the first layer we encounter is the application software and application queries layer. This layer is responsible for giving the user a view of the data that corresponds to the user's needs. E.g. if the database is a system that keeps track of airplane bookings for a travel agency, the clerks are probably not interested in having to explicitly query the database. They enter a flight number, press a button, and get the number of seats available on that flight. It is the application program that is responsible for giving them this view, and when the user clicks on the button the application program collects the information it needs from the user interface and constructs one or several queries to the database expressed in a high level query language e.g. SQL.

These queries are then received by the next layer, which contains software to process the queries. This layer transforms the high level query into something that is useful internally in the database and passes the information along to the next layer. This layer is also responsible for making sure that several users can simultaneously access the data.

The next layer contains the access software. This layer is responsible for retrieving the data as efficiently as possible. By reading meta-data from the database definition, using several parameters that have been entered by the database administrator or gathered statistically from previous data accesses, an optimized way of retrieving the data, i.e. the fastest way of retrieving the data based on how the data is stored in the database, is produced. Using this optimized access plan, it then finds the actual information in the database and retrieves the information and sends it back up through the different layers, to the user.

The DBMS supplies functionality that lets a database administrator define what the database should look like and how the different data are related. It also supplies an application programmer with functionality so that data can be entered into the database. Finally, it provides the user with a high level interface to the data so that efficient searches of the data can be performed and the data can be updated so that it stays relevant. For a discussion of different data models we refer the reader to [ELMA00, DATE00].

1.2 Temporal Databases

A temporal database is a database that contains historical data instead of, or as well as, current data, i.e. not only the current value is stored but also the historical values. Such databases have been researched since the mid 70s. A temporal database can use any data model, but it needs some additional functionality compared to an ordinary non-temporal database to handle time, as will be discussed below. There is a broad spectrum of temporal databases, from databases where data is only inserted, never deleted or updated, to databases that only contain current data, and where data is deleted or updated as soon as it ceases to be true.

The first example mentioned above, an append-only database, leads to an extreme database that only contains historical values. The other example, a database that only contains current values, is the opposite extreme, a normal, non-temporal database.

In a temporal database, not only the data values, but the time the value was entered, as well as all previous values are stored. E.g. if we have a non-temporal database containing salaries of employees, we can only see the current salary of a given employee.

But in a temporal database we can also see the history of salaries for each employee and at which points in time their salaries were changed. This is usually referred to as versioned data.

1.2.1 Time

Since time is continuous and computers use discrete time we have to decide what granularity of time we should use. In temporal databases the time granularity is referred to as the chronon. If two different data are modified in the same time step, it is often very useful to say that they were changed simultaneously. In the example above with employee salaries it is probably meaningless to store the exact time the salary increased. It does not matter what time of the day the salary was changed; it might not even be important to know the exact day it was changed, perhaps only the month is relevant.

In technical applications the chronon is often finer. If we monitor a very fast process we might need a chronon of seconds or milliseconds. The chronon has to be determined by the application.

Time is always present in one way or another in a temporal database. The two extremes are that the time can be implicit or explicit. If the time is implicit we are not interested in the actual time stamps of each value, but it is important to know the order of the values, i.e. one value comes before another in the sequence.

If the time is explicit, then each value is stored with a time stamp so we can see at what time a certain value became true, and we can find information about the time span during which a certain value was valid.

There is another way of handling time. Whenever the time advances one step, a snapshot of the entire database is taken, and all modifications to the data are recorded in the “new” database, while the old is saved as history.

There are several problems associated with all of these approaches. Since a copy of the data is made whenever it changes, or in some cases every time the time-marker is advanced, a temporal database can quickly become very large. One way to avoid this is to introduce life spans, i.e. when a historical value gets old enough it is deleted.

The interested reader is referred to [TANS93, ELMA00, DATE00].

1.2.2 Temporal Data

It was not until quite recently that temporal databases started to becoming more common and today most modern extensible DBMS:es have some form of support for temporal data. There is, however, not yet a fully established standard for temporal data. There is a standardisation proposal, TSQL3, but as of this writing, it has not yet been passed. There might be several reasons for the slow growth of temporal databases. Some possible reasons stated by [DATE00] are:

- It is only recently that disk storage has become cheap enough to make the storage of large volumes of historical data a practical proposition.
- There are difficulties in creating a temporal extension to SQL or some other existing database query language.
- There is still no consensus from the research community on the best way to approach the problem.

There is still an emerging interest in temporal databases, especially since the concept of data warehousing has become more popular.

In this research we have used a specific kind of temporal data, time series. For a description of time series see Section 2, "Time Series," on page 15.

1.3 Problem Description

This thesis addresses the problem of automatic indexing and retrieval of sub sequences 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 data, 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.

We have focused on very large time series and made an assumption that the time series will be too large to store in a main memory database. Our focus has therefore been to find an indexing strategy that will minimize the number of accesses to a paged slower secondary memory, e.g. a hard disk. If we take a look at modern computer hardware architecture we will see that these strategies will work well on main memory implementations as well since the main memory of modern computers is divided into several different categories. Closest to the processor we have a small amount of very fast memory, caches, that can bee accessed very quickly, and then the vast majority of main memory is of a slower variety. The speed difference of these two memory types might not be as large as the speed difference between traditional primary and secondary memory but it is still very real. So even though we only discuss disk-based storage the same principles are valid for modern main memory systems as well.

All database systems today have efficient means for storing very large amounts of data and for quickly accessing stored elements. There is however no established way of performing shape queries or queries for the time points at which a time series had a certain value. We propose to investigate a system that allows us to perform shape queries on the time series. Shape queries are queries on the shape of the time series where sequences that have a similar shape to the query pattern should be found regardless of their absolute values. When we investigate a suitable feature extraction process, it is important to find such a process that is both amplitude and translational invariant, i.e. since it is the shape of the time series we are interested in, we do not care if the shape is shifted along the amplitude or time scale.

The feature extraction process is closely related to the kind of queries we intend to support. Feature extraction is a process in which we extract some describing features we are interested in, in our case about the shape of the time series. These features are then indexed to make it possible to search for a certain behaviour in the time series. Some possible feature extraction methods available are:

- Time derivative
- Curvature
- Discrete Fourier Transform coefficients
- Polynomial approximation

All of these can capture the behaviour of the time series.

A problem with time series is that they can easily become very large. Storage of very large sequences is a complicated problem. Even though this problem is relevant, it will not be addressed in this thesis. For a further examination of the storage problem see [LIN99].

1.4 Motivation of Time Series Research

If the research on time series databases is successful, then the 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 are generated by instruments or sensors, or the data may reflect behaviour of the 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”. Today we see that systems with these capabilities are starting to emerge, but for large time series there is still no good solution for shape queries available [ORAC97, INFO97].

1.5 Related Work

Surprisingly little work has been done in the main area covered by this thesis, automatic indexing of time series data. It is important to note that even though we briefly mention temporal databases to offer the reader an introduction to this area we have not done any work in the traditional field of temporal databases. Although we discuss some current strategies for indexing time series data in Section 4, “Current Approaches to Time Series Indexing,” on page 45 of the thesis, we briefly review a wider range of related work here in order to introduce the approach to automatic indexing and retrieval of time series investigated in this thesis.

The most recent and most relevant papers we have found that deal with or describe time series indexing related subjects are [AGRA95, AGR295, BECK90, BERC96, BOHA01, FALO94, GUTT84, GÜNT97, HOEL92, INFO97, JAGA00, KAHV01, LI96, LIN98, LIN99, ORAC97, RONS98, SAME84, SELL87, SHAT96, WANG01]. Time series data indexing described in [FALO94] is based on using the first few DFT (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 two-dimensional space consisting of the first two DFT coefficients. In this way time series data is represented by a trail in two-dimensional space. Such a trail is then divided into sub-trails that are subsequently represented by their Minimum Bounding Rectangles in an R*-tree [BECK90], that is used as an 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 sub-sequences (windows) of two signals, on stitching similar windows to form pairs of large similar sub-sequences (long sub-sequence matching), and on finding a non-overlapping ordering of sub-sequence matches having the longest match (sequence matching). Both papers [FALO94, AGRA95] are deficient in that they tested their proposed indexing methods on a very small collection of stock market histories. In other words, it is not clear whether the indexing method they propose will work in a large, real-life time-series database. Second, neither indexing method is invariant with regard to data amplitude, offset, 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 multi-media data indexing [FALO95] avoids the difficult problem of feature extraction by assuming that a “domain expert” will somehow provide feature vectors, or at least provide dissimilarity/distance of two objects.

A very interesting approach to searching a time series for patterns is proposed in [LI96]. Each time series is divided into shorter sub-sequences and each sub-sequence is then normalized before it is mapped to a feature space vector. The method allows a simple but powerful concept of similarity when the index is searched.

Another interesting solution to this problem is presented in [SHAT96]. Here the time series is divided into smaller segments and each segment is then approximated by a polynomial. This allows the authors to treat the time series as a continuous signal and not as a traditional sequence of discrete values. Unfortunately the suggested method relies heavily on human experts and manual work for setting up a system.

In [LIN98] the IP-index is introduced. It is an index structure that support queries for interpolated data in time series. The idea is to treat the time series as a sampled signal and assume that the original time series can be approximated by interpolation between the sample points. This allows the system to give more precise answers. Imagine that we created a time series out of temperature readings and we sampled the temperature every six hours. Assume that the temperature was $10°C$ at 6 in the morning and $20°C$ at noon and that we stored only these two values. We could then use the IP-index to search for the point of time the temperature was $15°C$ and we would get the result 9 in the morning, assuming the index was using a linear approximation. In [LIN99] the $\sigma*$ -operator is introduced. It is an extended select operator that retrieves time intervals from a time series.

A paper that does not deal with indexing of time series, but is still relevant for our approach to the problem is [KAHV01]. This paper deals with index structures for strings. The paper addresses the problems you have when you wish to find similarities between two DNA strings. A sequence of wavelets is constructed from the strings and then this information is stored in an index structure proposed by the author called a Multi Resolution String (MRS) Index structure. [JAGA00] also introduces an indexing strategy for strings, but since the paper focuses on prefix queries, it is not well suited for time series indexing.

The different ways of indexing time series often have to index feature space vectors of a dimension of two or higher, so even if we are only trying to index a one dimensional sequence, the underlying index structure often has to support higher dimensions. [BECK90, BERC96, GUTT84, SAME84, SELL87, WANG01] all introduce index structures that are more or less fundamental for indexing spatial data. Others, e.g. [GÜNT97] suggest a modification to traditional disk based spatial access methods, to minimize the redundancy in the structures and improve search speeds. [HOEL92] compares how different spatial indexing methods perform. [BOHA01, RONS98] both focus on main-memory databases. From these papers we learn that different page-based index structures are becoming more important for main-memory applications since modern computer hardware architecture uses several different kinds of main-memory that all work at different speeds. This causes us to have the same IO problems in main-memory applications that we have in disk based applications.

The problem of feature extraction is central to this area of research. It is in 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.

At the time of writing most major database systems have some support for time series. Informix has the time series datablade [INFO97], Oracle has a time series DataCartridge [ORAC97], and IBM has a time series DataExtender.

We also based some of our work upon [BADA95], which describes context signature files. 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].

1.6 Contributions of This Work

The main contributions of this work are:

- We demonstrate that it is possible to create a very fast index structure for time series.
- We demonstrate an indexing structure that scales very nicely with very large time series.
- We demonstrate a feature extraction method that transforms the time series into a text sequence.
- We demonstrate a method that makes it possible to perform shape queries on time series.
- We show how our indexing methods could fit into a solution to a problem from the control area.

If the data set is sufficiently small, it is always possible to come up with some indexing method that offers very good search speed and different search options. It is a much more challenging problem to find an indexing method that offers very efficient searches and that scales well with the size of the indexed data. Our goal has been to attack the problem of indexing large time series that contain at least 10^6 elements. We demonstrate that these time series can be searched with a low search cost, and that the index structure scales nicely.

Several index structures exist for time series, but none are as efficient and compact as text indexes. This is partly because when we index a text, we can often use our knowledge about the text, e.g. the fact that some words in the text only work as "glue" for the information, that the same word can appear differently depending on the context it appears in etc., and the fact that we have a limited number of words. This knowledge makes it possible to pre-process the text before we index it without, usually, losing any important information, but still making the text we actually index much smaller that the original text. When we index time series we have a much harder time pre-processing the time series. There are several similarities between text indexing and time series indexing. It is possible to compare the problem of finding sub-sequences within a large time series to the problem of finding words within an English text. We demonstrate that it is possible to map the problem of indexing time series into a text indexing problem. That makes it possible for us to take advantage of research done in the text indexing area and apply it to time series indexing.

The first step in these methods is to convert the time series into a text. With this approach it is possible to find a feature extraction process that is amplitude and translation (offset) 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 are translated and we will still be able to find similarities between these different time series. 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 the time series data indexing problem will become a text indexing problem.

We have examined some index structures designed for indexing text files, signature files (see Section 5, “The Signature File,” on page 59), and B-trees (see Section 7, “The B-Tree Structure,” on page 95). We have also created and investigated an extension to the signature files (see Section 6, “Indexing the Signature File,” on page 85) and an extension to the Trie structure (see Section 8, “The PTrie Structure,” on page 109).

We will also demonstrate that the feature extraction method we have selected to transform the time series into a test allows us to perform efficient shape queries on the time series. Each feature extraction process introduces a different concept of similarity, and we show that it is possible to obtain a very intuitive concept of similarity.

Our conclusion is that, for time series, the B-tree offers the best overall performance benefits. The B-tree performance also scales very nicely with respect to the data size. We also show that even for small time series the B-tree is the most cost-efficient index structure. If the size of the B-tree is considered, we show that signature files can be an efficient alternative to B-trees. We also demonstrate that the signature file will always be faster than any other sequential scan method.

We have also developed the PTrie structure. This structure can not be as efficient as the B-tree for data stored on a slower, paged, storage, but the PTrie has support for performing fuzzy queries that can make it an attractive replacement for the B-tree in some main memory applications.

We have also examined how a problem from the automation and control area could benefit from the functionality of a database system with the ability to perform searches on time series. The application we have examined also has several non-database related problems, and we have identified some of the biggest problem areas.

1.7 Outline of the Thesis

The thesis has the following structure:

Chapter 1 (this chapter) provides background information that places the work presented in this thesis into perspective. This chapter also contains background information for readers not familiar with this research area. This chapter also contains short descriptions of the relational and object-oriented data models so that the readers will be familiar with them when they are used in the thesis as examples.

Chapter 2 introduces the reader to time series and describes different kinds of temporal data. This chapter also includes a description of the time series we have used while testing our index structures, as well as the motivation for our choices.

Chapter 3 describes the concept of feature extraction. In this chapter we introduce the reader to the concept of feature extraction, and show different methods of extracting features from signals or time series.

Chapter 4 is a review of some of the current approaches to time series indexing. All of the papers we have looked at try to solve the problem of automatically indexing time series so that we can search for sub-sequences.

Chapters 5 to 8 present our different approaches to the problem of indexing time series.

Chapter 9 summarizes the different approaches we have examined in this work.

Chapter 10 shows how this work can be applied to a control problem and what further problems we have to investigate before a complete control solution can be constructed.

Chapter 11 is a summary of our conclusions.

Chapter 12 outlines future work.

Chapter 13 contains references.

Appendix A contains precision diagrams obtained from our signature file implementation.

Appendix B contains test results obtained with the B-tree structure.

Appendix C contains test results obtained with our PTrie structure.

Appendix D is a summary of basic indexing structures.

Chapter 2

Time Series

This chapter introduces the concept of time series and describes what they are and what they are not. We will also introduce the reader to some important factors: the search cost and the transfer time. Finally we will show two examples of test data we have used in our work.

Timed data, also called time sequences, were first introduced in [SHOS86], and, as the research area of temporal databases gained momentum, were further analyzed in [SEGE93].

A time sequence is a sequence of values obtained from some process over time. All values in the sequence are ordered in time and stored together with a time stamp. These values have been obtained by sampling a signal. There is no required sampling frequency. For example, we might supervise the signal with a low sampling frequency when the rate of change is low. When the rate of change in the signal increases, the sampling frequency also increases. Since the sampling frequency can change over time, it is important to save time stamps with each value so that we can see when the value was obtained. Such a time sequence is called an irregular time sequence.

If we use a constant sampling frequency it is no longer necessary to store time stamps with each value, since we can calculate the time a certain value was obtained if we know the time the first sample was collected and the sampling frequency used. Such a time sequence is called a regular time sequence or time series.

An example of an irregular time sequence might be a temperature log from a machine where the temperature is stored every minute when it is above a certain threshold. Examples of time series are stock prices, collected at the end of every trading day, or temperatures from a boiler sampled every second. In the stock price example it is important that you only consider non-trading days, otherwise the sequence would become an irregular time sequence.

In literature one can find references to time sequences and time series. Sometimes the two terms are used interchangeably, but to avoid confusion we use the definition used by [LING99]. The term time sequence is a more general term than time series. A time sequence can be either regular or irregular, but time series have to be regular. This thesis only deals with time series.

In order to find a suitable index structure for time series it is important to know what properties they have.

- Time series are ordered by time, i.e. the order of the values is important and must be preserved.
- Time series are usually very long [SHOS86].
- Time series are usually append-only structures [SHOS86].

2.1 Measuring the Search Cost

Throughout this thesis we assume that the data are stored permanently on disk. Data are moved from disk storage into main memory for processing and then moved back to disk when they are no longer needed.

Space on a disk is allocated in blocks, or pages. The size of a block can range from as low as 512 bytes up to 64 kb. Since each block can be fetched very quickly from the disk it is preferable to have large disk blocks. But, since a block is the smallest allocation unit on the disk, storing small data sets that do not fill up an entire block wastes a lot of disk space. From this point of view it is desirable to have small blocks, so some kind of trade off has to be made.

In modern systems the time to access a random block on disk is in the range of 10 ms, whereas the time to access a random main memory address is in the range of 0.1 μs.

Accessing data from disk is approximately 100,000 times slower than accessing data from main memory. Therefore we use the number of necessary disk accesses to estimate the cost of searching an index and ignore the main memory accesses.

It is important to point out that this is a simplification of the real world, since it would imply that searches in a main memory application would be instantaneous, which they clearly are not. However, for systems where the data resides on disks this simplification offers a reasonable cost estimate.

2.2 Transfer Time

When discussing the search cost we must also mention the transfer time. The transfer time is the time it takes for the data to be transported from the disc, once it has been found, into main memory. In the case of small files the transfer time is usually negligible. However, when the size of the files increases, so does the transfer time, and for very large files the transfer time can be much greater than the search time.

To understand the influence of the transfer time we have to take a look at how the data is transported from the disc surface to the correct memory area. From the disk surface the data is transported to the disk cache, then it is transported over some kind of bus from the disc unit to the computer and then finally the data is transferred to the correct memory area.

The surface of the disc is divided into tracks and sectors. Each sector is a circular track around the rotational centre of the disc and each track is the divided into a large number of sectors. When we wish to load data from the disc we first have to move the read head, placed on a small mechanical arm, to the correct track of the disc. Once the head has reached the correct track we have to wait for the correct sector to appear below it. If we are unlucky we might have to wait for the disc to rotate one full revolution. As the data passes under the head it is transferred to the internal disc cache. When the cache is full, or the read operation is completed, the content of the cache is moved from the disc cache to the host computer over a bus. Two popular bus technologies today are SCSI and IDE. Once the data have left the disc cache it usually ends up in an internal disc cache in the computer primary memory controlled by the operating system before it can be accessed by the application that requested the data.

If a large amount of data is read from the disc and the data is stored in consecutive blocks, the transfer time will become important. In that case, the time to read the data will not be determined by the block search time but by the transfer time. To estimate the difference between these two let's assume that data is stored in 64 kb blocks on the disk. A modern disk will manage to transfer somewhere between 10Mb and 100Mb data per second. This means we will read between 160 and 1600 blocks per second. If we once again take a look at data sheets for modern discs we see that a typical search time is around 10ms. So we can load 100 randomly placed blocks per second. It is somewhere between 1.6 and 16 times faster to load consecutive disk blocks that to load randomly located blocks.

All of these different steps take a small amount of time. If we then load very large amounts of data so that the disc head has to move repeatedly, and we have to wait for the disc to rotate in place repeatedly, the transfer time can be the limiting factor of a system.

We have not examined the transfer times. For a more in depth discussion on transfer times see [LIN99].

2.3 Test Data

Our index strategies have been tested on two different sets of data. The first set consists of synthetically generated time series and the second set consists of sensor data from a thermometer at the Department of Computer Science, Linköping University.

We decided to use both synthetically generated time series as well as a real time series for two reasons. The goal of our research has been to find an index structure suitable for large time series. We had problems finding real time series that were long enough. A popular test for time-series indexing methods is to use stock exchange data, but the files were too small to be interesting in the sources we found. Also, we could easily control the shape of the synthetically generated time series so that we could test our system with different variations.

We decided that the synthetically generated time series was sufficiently good when we could not distinguish the synthetically generated time series from the real time series by examining the results we obtained when we indexed them.

2.3.1 Synthetically Generated Time Series

When we generate time series, it is important that they have similar properties to real data. This is important so that the synthetic data do not trick us into making changes to the index structures that would not be necessary with real data. As we were searching for sufficiently long time series, we gathered several smaller time series from around the internet, e.g. stock market data, sensor data from a process industry, and temperature data from the Computer Science Department at Linköping University. We then used these time series to gain an understanding of how we could generate synthetic time series.

How should synthetically generated time series mimic real time series? It depends on the application in which it will be used. Since we are interested in “shape queries” we decided that it was important that the shape of the synthetically generated time series mimic the shape of the real time series. We examined all our existing time series and constructed an algorithm that would produce time series of any length that would “look like” real time series. We verified the synthetically generated time series by visually comparing them to the “real” time series, and when we could not distinguish the synthetic time series we accepted the algorithm as “good enough.” The most important aspect of the time series we have chosen is that they have a shape with some features that we can perform searches on. A truly random time series does not qualify as a good time series from our point of view, since it is only noise and it does not contain recognizable features. For time series to be interesting for this application they have to contain shape features, e.g. large and small slopes and stable areas of varying length.

It is, however, still very important to verify all results we have obtained with the synthetic time series and make sure that we get the same results with real data. We have done that consistently throughout our work.

Synthetic data must be good enough to reliably test the system. The performance of most index structures varies depending on the input data, in our case the generated time series. Many index structures work very well for some input data, but it is also possible to find input data that make the same structures perform poorly. A simple example might be the use of a hash table to index a collection of synthetically generated object names. If all names for some reason hash to the same value, the hash table will not work – we will end up having to scan all of the data anyway. There are two possible reasons for this. One is that the algorithm is bad, and needs to be adjusted. The second is that the data is not appropriate for the algorithm. In this case, it is possible that our object name synthesizer generated names that are more similar than the names used to develop the algorithm. In our test case the hash table will not work but in a real application the names would not be as similar and the hash table would work fine.

As this example illustrates, it is important, as we mentioned earlier, that the synthetic data do not trick us into making changes to the index structures that would not be necessary with real data.

2.3.2 The Time Series Algorithm

Our algorithm is designed to generate random time series of arbitrary length that mimic the time series produced by a sensor. In our testing we have used time series consisting of as many as 400,000+ data points.

We have developed the following algorithm to generate time series. The signal is constructed as follows: the algorithm uses four constants, A, B, C and D, and for each iteration, four random numbers, β, γ, δ and ε. We start at an initial value, α, that is set to 0 in the first iteration:

1 Generate a random number, β, in the range $-A$ to A. This random number is the next level we should move to.

2 Generate a random number, γ, in the range 1 to B. This random number is the number of steps it will take to reach level β from level α.

3 Add γ steps to the signal, each step $\frac{\beta - \alpha}{\gamma}$ long.

4 Generate a random number δ 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 δ that the signal is stable at level β, generate a random number ε 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. An example of a synthetically generated sequence can be seen in Figure 2.1. The x-axis represents the time and the y-axis represent the value. If the time series in Figure 2.1 is compared to the time series in Figure 2.2 (real data collected from a temperature sensor), we can see that both contain shapes we can search for.

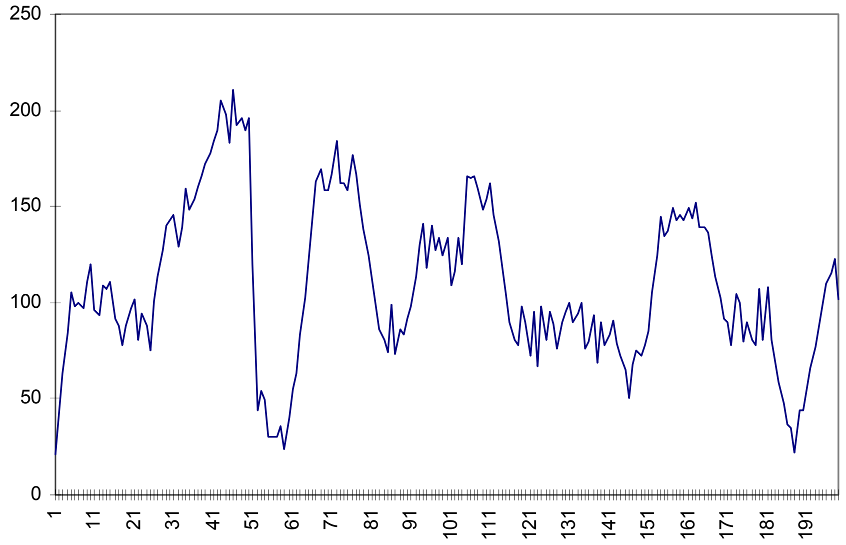


Figure 2.1: Example of a synthetically generated signal

2.3.3 The Real Sensor Data

For a real time series, we chose to use the temperature data collected from a sensor at the computer science department in Linköping. We did this because it is more consistent than our other options e.g. the stock market data. The temperature varies over time but the changes are not very drastic whereas stock market and process data, as well as our synthetically generated data, might be more "erratic" in its behaviour.

An automatic system measures the outdoor temperature once every hour 24 hours 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 2.2 shows a small sample from this data sequence. The x-axis represents the time and the y-axis represents the temperature.

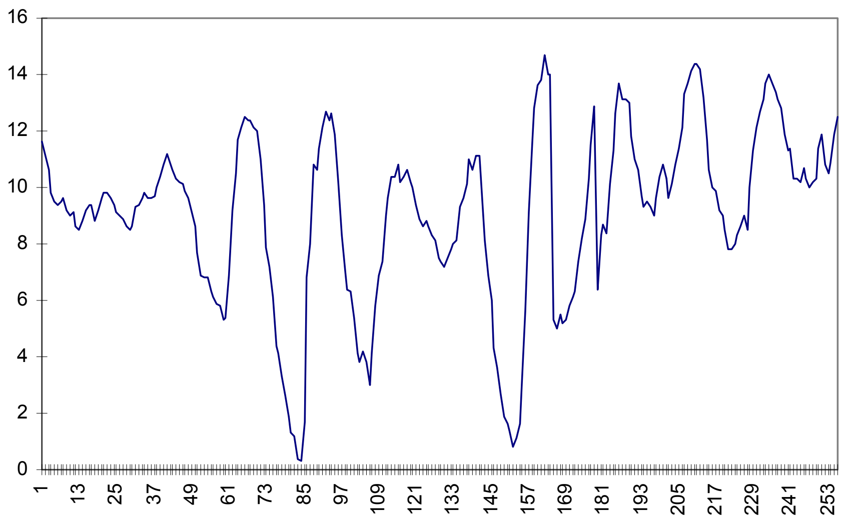


Figure 2.2: Example of temperature data

Chapter 3

Feature Extraction

This chapter will introduce the reader to the concept of feature extraction. We will describe what it is and give examples of a few different feature extraction methods found in related work. We will also discuss our approach of creating "words" from time series.

Before the time series can be indexed we need to decide what features the user should be able to search for. In our work we have focused on shape queries. Shape queries are queries on the shape of a signal/time series, e.g. "find all sub-sequences of the stored time series whose shape is similar to this given query shape". These are not the only queries it is possible to state on time series. Other types of queries include value queries, e.g. "at what times did the time series have a value of '5'?" or time queries, e.g. "what value did the time series have at time '7'?". The kind of queries you intend to run against the system is important. The reason for this is that the feature extraction process has implications for several parts of the system. Since, during feature extraction, we only extract certain information about the time series, how we choose this information will have a great impact on other parts of the system. To exemplify the feature extraction process, imagine that the time series is a stream of cars on a highway and that feature extraction is done by a person standing next to the highway. The person responsible for feature extraction will note features of the passing cars that are important. One feature he can note is the color of the passing cars. We will then be able to, at a later time, ask him if e.g. any blue cars passed. But if this is the only feature he extracts from the stream of cars, we will not be able to ask him if there were any trucks in the car stream.

The same thing holds true for time series. We have to decide what aspect of the time series we intend to query on in order to decide on a feature extraction process. The natural question then is why we do not save all different aspects of the time series. Another reason for performing feature extraction is to remove "unimportant" information in order to reduce the amount of information we have to index. Usually the description of the signal after feature extraction is much smaller than the 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. In many cases the user is only interested in finding sequences that are similar or have the same shape as the query sequence. 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 similar to a 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, the action described by the verb is the important thing. The same applies for signals. The query is usually not for the exact sequence of values but for all sequences that are in some sense similar to the search sequence. There is also another important issue when we talk about signals. The time series are constructed by sampling a continuous signal. If we sample the same signal twice, with the same sampling frequencies, but with slightly different offsets, the resulting time series will be different even though they are generated from the same signal. Two identical signals could result in two slightly different time series, and this is one of the reasons that it is important for a query mechanism to find them both, since the user has no control over the sampling process.

In the rest of this chapter we will use the term “signal” and the term “time series” interchangeably. Since these feature extraction processes are carried out in a computer the signal will always be sampled and can therefore be seen as a time series.

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

A common method used to capture the basic behaviour 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. If the approximation has to be improved, more frequency components have to be saved.

Another way of capturing the behaviour 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 is difficult to segment the signal so that the discontinuities introduced by breaking up the signal won't interfere with our ability to query the sequence of polynoms.

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

3.1 Feature Extraction Using Time-Derivative

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. First we scan the signal. This can be done off-line or on-line while the signal is read into the system. For each point in the sampled signal, the time series, 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 an 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 real value, the signal time derivative in each point, to a discrete value, a symbol from the alphabet (see Table 3.1 on page 27 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, see Figure 3.1, but a slightly different envelope since it is the signal time derivative and not the signal that is mapped to the alphabet. If a time series fits within the envelope of another time series, they are considered similar. The envelope for three different sequences can be seen in Figure 3.2 on page 26, Figure 3.3 on page 26 and Figure 3.4 on page 27. All examples use the SDA(5) described in Table 3.1 on page 27.

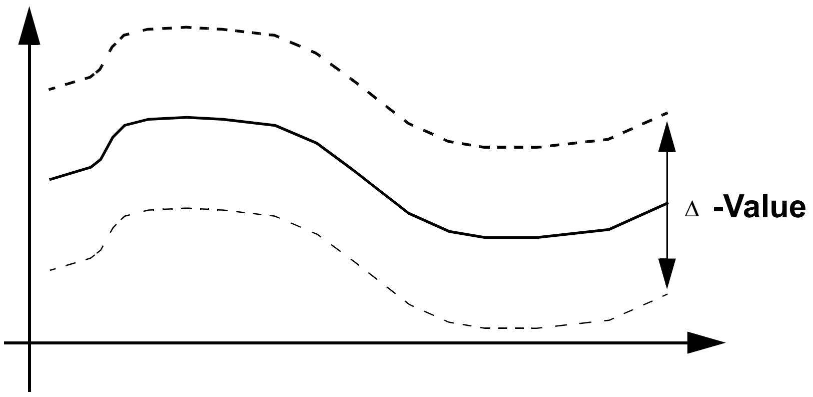


Figure 3.1: Description of delta envelope

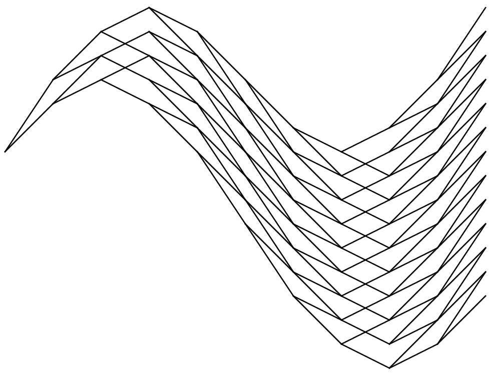


Figure 3.2: Envelope of a sinus wave “abcdeedcba”

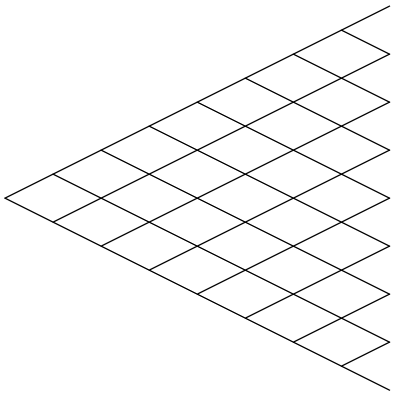


Figure 3.3: Envelope of a stable sequence “cccccccc”

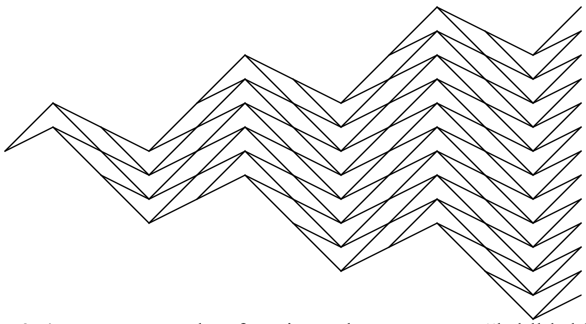


Figure 3.4: Example of a triangular sequence “bddbbddbbddb”

Table 3.1: An Example of a SDA(5)

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

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 3.5 on page 28, “0 1 4 10 14 15 15 11 10 20 14 8 2 0”, it will map to the string “cbabccdcaeeec” (using SDA(5) in Table 3.1 on page 27). But we can see that the sequence shown in Figure 3.6 on page 28, “-10 -11 -6 1 6 4 2 -1 -3 15 9 3 -3 -1” will also map to the same string. These two sequences are therefore by the system considered to be similar. So the granularity of the fuzziness is controlled by the size of the alphabet.

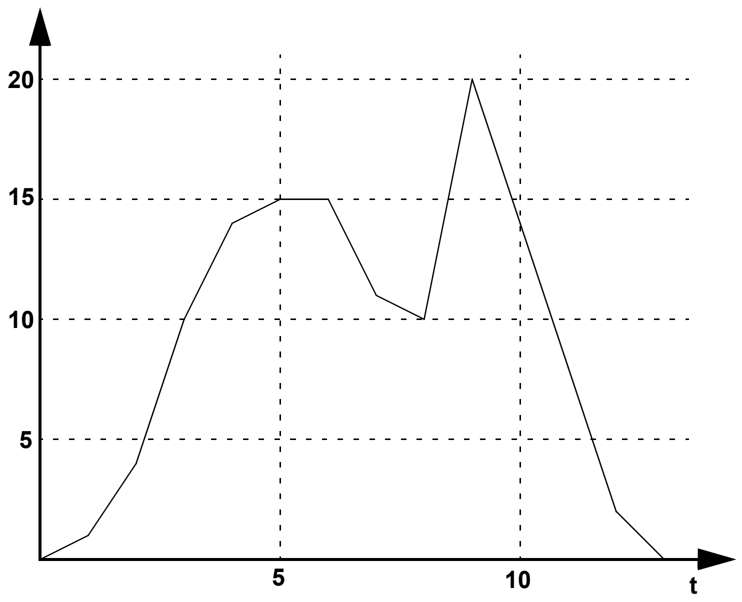


Figure 3.5: Example sequence 1

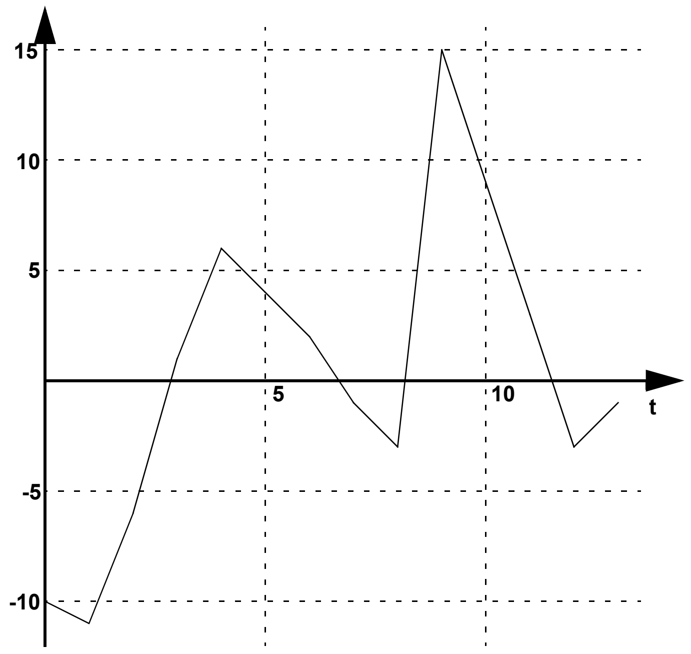


Figure 3.6: Example sequence 2

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

But this approach is not ideal. If this is the only similarity concept used 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 as similar, the approximation is too vague. To remedy this the alphabet size can be increased. Using the same similarity concept as before the number of hits yielded by 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 done easily. The precision of the system is controlled by the size of the alphabet and is determined when the index is built. It would be favourable 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 5.6, “Similarity in Signatures,” on page 73 for a discussion of the issue.

3.2 Feature Extraction Using Curvature

A slight modification of the method described in Section 3.1, “Feature Extraction Using Time-Derivative,” on page 25, is to use the curvature instead of the time-derivative. From the point of view of our indexing structure this method is identical to feature extraction using time-derivative described above in Section 3.1, “Feature Extraction Using Time-Derivative,” on page 25, since the result of the feature extraction process is a sequence of characters that can be treated in the same way as if it had been generated using time derivative.

The difference between these two methods is that the concept of similarity will change. If we base the concept of similarity on the same principle as before, that two sequences are similar if they map to the same symbol sequence, two time series that are similar using the time derivative feature extraction process might not be similar using the curvature feature extraction process and vice versa.

3.3 Feature Extraction Using DFT

This is a commonly used feature extraction method. If the problem is to index collections of smaller sequences, the transform, e.g. the discrete Fourier Transform, is applied to each sequence, and two to four components are selected to represent the entire sequence. These components form our feature vector.

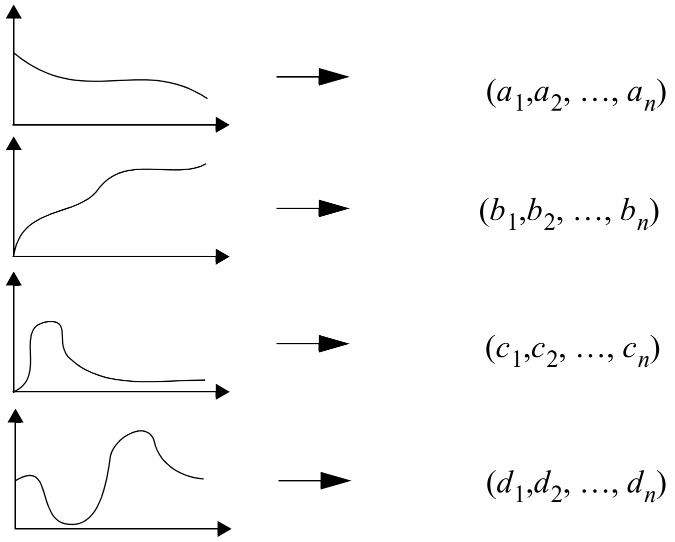


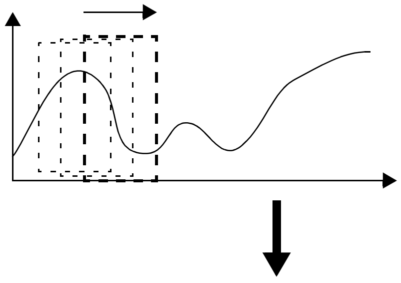
Figure 3.7: Extracting information with DFT, example 1

Since we extract very little information from each sequence the compression will be very good. This method will allow us to search a collection of sequences for a given sequence, but if the sequences are longer it might be interesting to find sub-sequences within a long sequence. See Figure 3.7 for an example.

If we are interested in finding sequences from within larger sequences we have to extract more information from each sequence. To accomplish this we introduce a sliding window of width n. At the start the window is placed over the first n values of the time series.

1 Transform the n values in the window using DFT to obtain a feature vector.
2 Store x components from the DFT, usually two or three.
3 Move the window one step forward in the time series.
4 Repeat from step 1.

The entire time series after this process has been mapped into a sequence of feature vectors, see Figure 3.8 for an example. It will then be possible to find any sub-sequence of length n in the sequence. It is not possible to find shorter sequences, but by dividing a query of length m, $m > n$, into $m - n + 1$ queries, each of length n, it is possible to find all sequences longer than n.



$(a_{11}, \ldots, a_{1n}), (a_{21}, \ldots, a_{2n}), \ldots, (a_{k1}, \ldots, a_{kn})$

Figure 3.8: Extracting information with DFT, example 2

3.4 Feature Extraction Using Polynomial Approximation

It is also possible to break up the sequence into smaller segments and then approximate each segment with a polynomial. For each segment we then have to store the start and end position together with the polynomial coefficients. If the segments are sufficiently large it is possible to achieve a very good size ratio between the original data file and the extracted data.

One advantage with this method is that the extracted approximation is continuous and it is very simple to determine if two sequences are similar or if a longer sequence contains a shorter query sequence.

A problem with this approach is that if the sequence is long it might be very difficult to approximate the entire sequence with one polynomial. The solution is to break up the signal into several smaller segments and then to approximate each segment with a polynomial. The problem is to decide where to break the large sequence so that no interesting features “get lost” in the break, and that the resulting sequences can be approximated with a high precision.

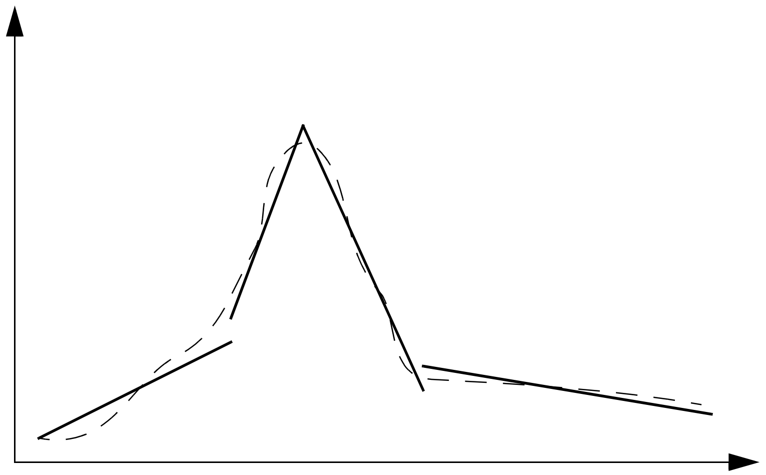


Figure 3.9: Sequence and polynomial approximation

In Figure 3.9 a sequence is shown together with a simple first degree approximation of the sequence. Each segment will be described by four parameters, first the segment start and end position and then the k and m values from the line equation $y = kx + m$ since in this example the sequence is approximated with a first degree polynomial.

3.5 Feature Extraction Using Events

If we see the sequence as a number of interesting events with uninteresting sequences in between, and if we assume that the number of events is low compared to the length of the sequence, another approach might be to try and find those interesting events and extract them together with their position in the original sequence.

If the time series consist of relatively few events that are repeated a large number of times it should be possible to use these events as words and then use a non-modified text indexing method to index the time series.

Event extraction can, of course, be combined with several of the feature extraction methods described above. We first translated the sequence into a text string as described in Section 3.1, “Feature Extraction Using Time-Derivative,” on page 25, but it is quite possible to first find all events and then calculate the DFT for each found event as described in Section 3.3, “Feature Extraction Using DFT,” on page 29.

When we talk about finding events in a sequence it is important that we first decide what an event is. Informally it is quite simple, an event is when something happens in the sequence. In reality it becomes a little more complicated. We have three primary parameters that control the search for events: the look-a-head (l-a-h), the fuzziness, and the minimum event size.

3.5.1 Event Fuzziness

The fuzziness describes how tolerant we should be before we start a new event. With "exact match" we start an event as soon as we find a variation in the sequence, i.e. two neighbouring elements in the symbol sequence are different, and with higher fuzziness we allow larger variations to take place before a new event starts. E.g. if we have the sequence "aaabccdfaeee..." and use exact matching (and l-a-h 2, see Section 3.5.2, "Look-a-Head," on page 33) the first event start we will find is triggered by "abc". If we increase the fuzziness to 2 the first event start we will find will be "cdf", since this is the first segment found where the maximum distance between the characters in the sequence is larger than 1.

3.5.2 Look-a-Head

The look-a-head describes how many steps ahead the function looks to determine if an event starts or ends. E.g. assume that we have the sequence "aaabccdfaeee...", and say that we allow a fuzziness of 1 without starting an event. If we set the look-a-head to 1 and start looking for an event, the first event we will find is "dfae...", since "aa", "ab", "bc", "cc" and "cd" only differ by one. If we set the look-a-head to 3 the first event we will find is "abc..." Since "aab" only differs in one position it will be skipped, but the next step, "abc", differs by 2 steps.

Once we have found the start of an event we remove the first characters that are not part of the event. In the example above with look-a-head set to 2 and exact match, if "aab" triggers the start of a new event, the stored new event will start with just "ab". We then continue until we find a new sequence that does not contain any variations. In the example mentioned above (look-a-head set to 2 and exact match), if "aab" triggered a new event the event will not terminate until "eee" is found, and the stored event will be "abccdfae".

3.5.3 Minimum Event Size

Another important control parameter is the minimum event size. Since we can obtain a huge amount of very small events we might want to put a lower limit on the length of events we accept as interesting events. This is a simple filter mechanism. An event is detected as described above in Section 3.5.1, "Event Fuzziness," on page 33 and

Section 3.5.2, “Look-a-Head,” on page 33, and if the detected event size is smaller than the preset value it is discarded and the event search continues.

3.5.4 Experiments

In order to see if it is possible to find a small set of events that describes the sequence and might correspond to words in an English text, we have conducted a large number of tests trying to identify such “words” in time series. We would like to be able to identify a relatively small number of sequences that make up the majority of the entire sequence. These “words” should occur frequently. There should only be a small number of words that occur infrequently.

We have conducted several different experiments trying to extract events from a sequence. We have used different sequences – randomly generated sequences and real sequences of varying length, up to 400,000 elements long. The goal of our experiments has been to see if there are such events, and if it is possible to express a longer sequence using a small number of smaller events.

The results presented here are from a synthetically generated sequence with 400,000 elements.

In Figure 3.10, Figure 3.11, Figure 3.12, Figure 3.13, Figure 3.14, Figure 3.15, and Figure 3.16 we show the frequency of events of a certain length from the file. We have used a look-a-head of 1 and exact match. The shorter events are more frequent but as soon as we start looking for larger events we do not have to look at events larger than 6 elements before most of the events only occur once.

In the diagrams, the events are sorted by the number of occurrences. The actual event sequence is printed above each bar.

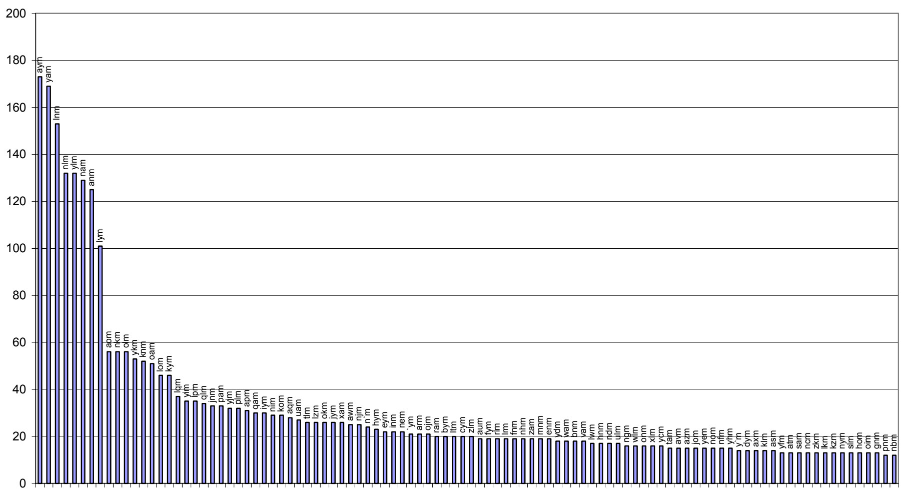


Figure 3.10: Events of length 3

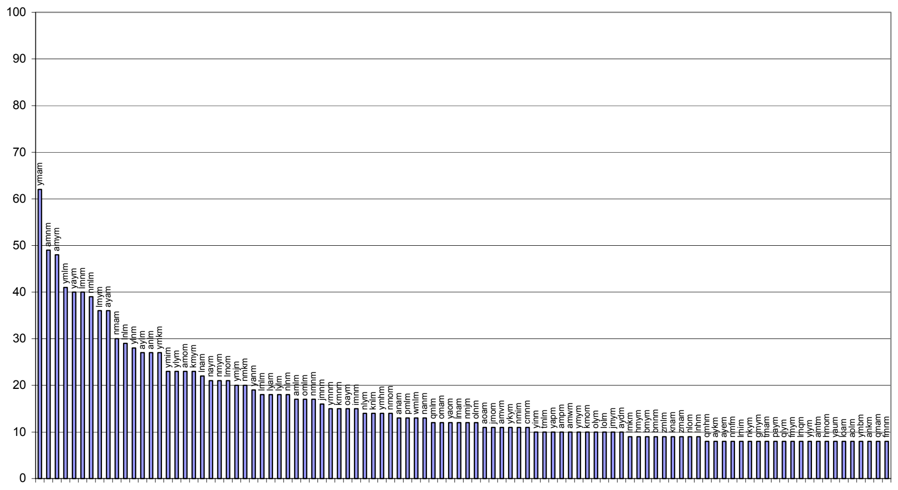


Figure 3.11: Events of length 4

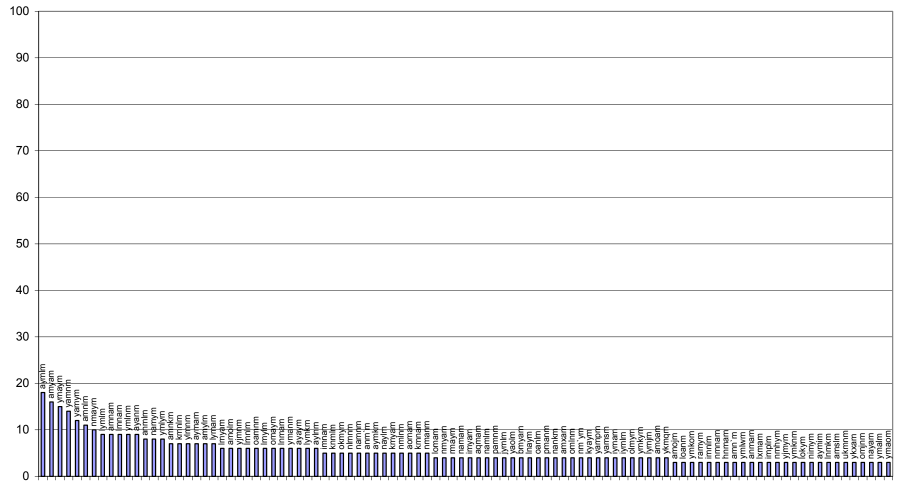


Figure 3.12: Events of length 5

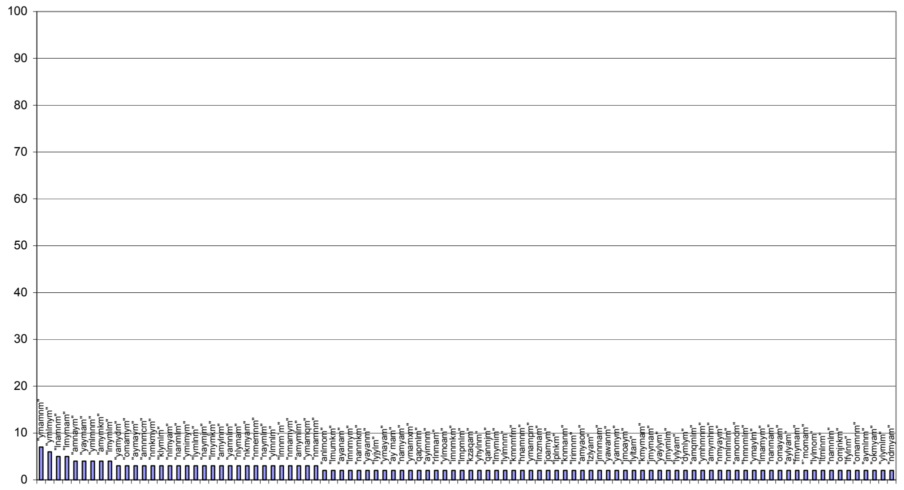


Figure 3.13: Events of length 6

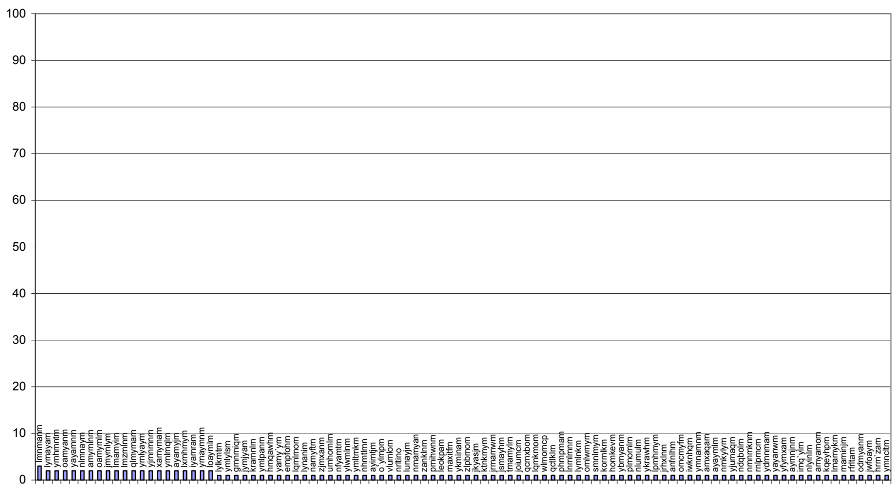


Figure 3.14: Events of length 7

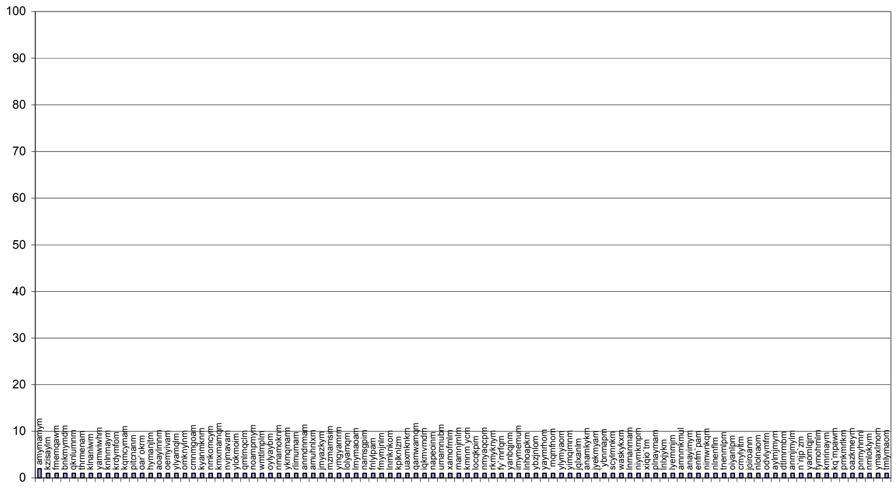


Figure 3.15: Events of length 8

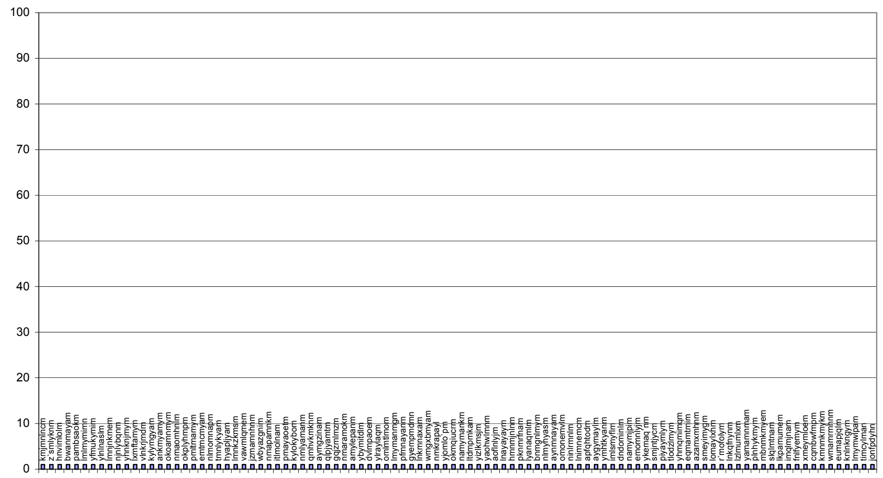


Figure 3.16: Events of length 9

The next tests show the total number of events of all lengths found, using different combinations of look-a-head and fuzziness. In Figure 3.17 we found a total of 22,380 different event types. There are a few very short events that occur rather frequently, but then the frequency quickly falls to below 300 events found.

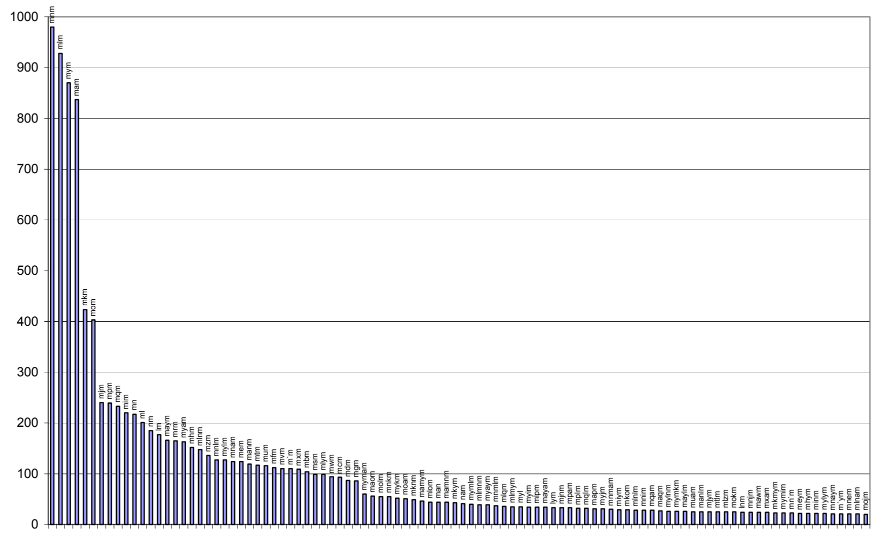


Figure 3.17: Events found in sequence, l-a-h 1, exact match

One way to try to address the problem of so many infrequent events is to increase the fuzziness. Figure 3.18 shows the results from the same sequence if the fuzziness is increased to 3. In this case 15,636 different event types were found in the sequence. We can see that the occurrence for a few events are much higher but little else differs.

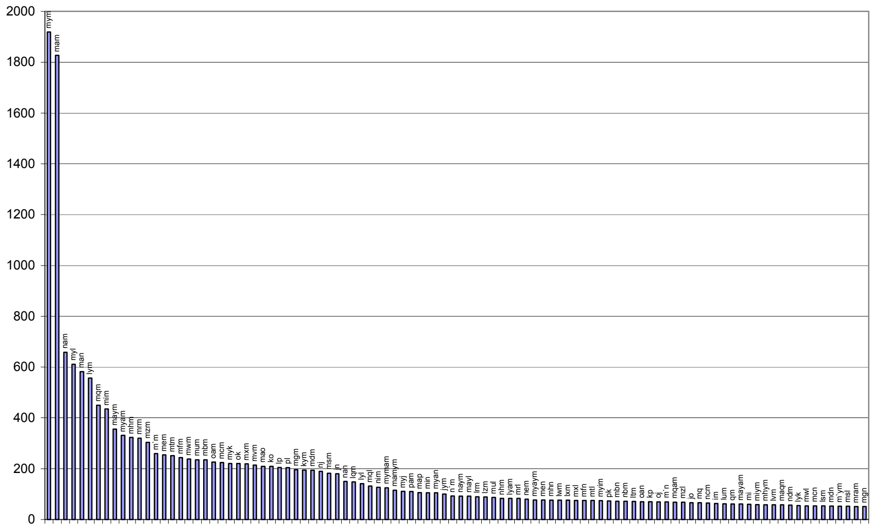


Figure 3.18: Events found in sequence, l-a-h 1, fuzzy match 3

What happens if we modify the look-a-head instead? In Figure 3.19 we can see what happens when the look-a-head is increased to 3. The number of events detected is greatly reduced. The total number of events found is 15,416 different event types.

In Figure 3.20 the fuzziness is again increased and we see the same kind of difference between Figure 3.19 and Figure 3.20 as we see between Figure 3.17 and Figure 3.18. The increased fuzziness only increases the occurrence of a small number of events and does not have a major impact on the majority of events. With this configuration 19,845 different event types were detected in the sequence.

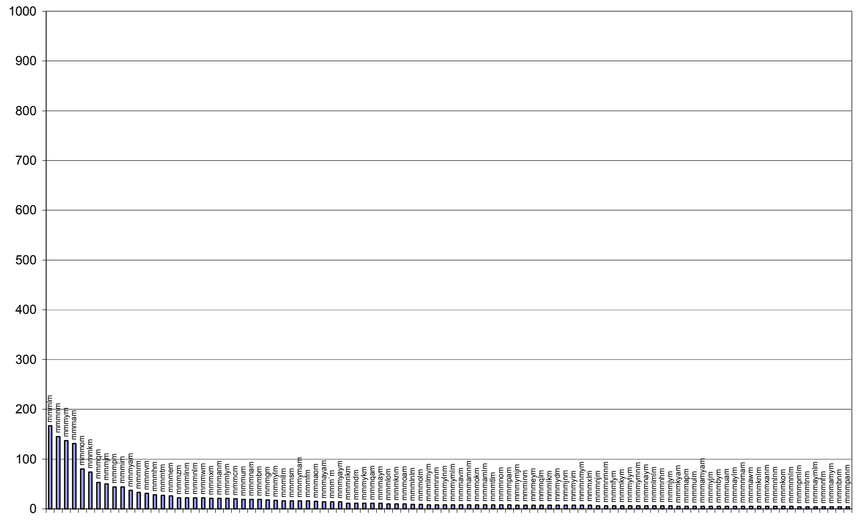


Figure 3.19: Events found in sequence, l-a-h 3, exact match

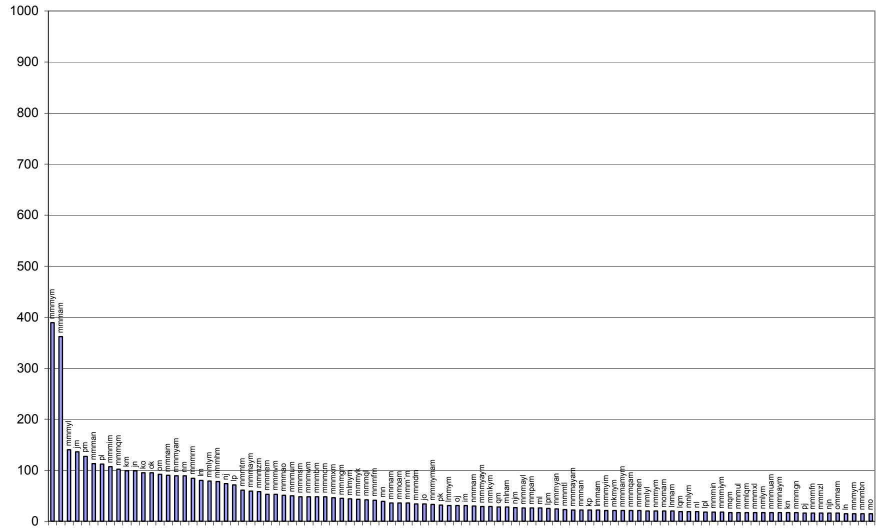


Figure 3.20: Events found in sequence, l-a-h 3, fuzzy match 3

If this event-finding approach is to be used as a basis for extracting information from the time series, and later indexing the time series, we would like to have a smaller number of events extracted and a higher occurrence frequency. The situation described above, in which a few events occur repeatedly but the vast majority of events only occur once, is not desirable.

One solution might be to try and group similar events together. We introduce another parameter, *event distance*. Once an event is found, it is compared to all other events of equal length. If the new event is within a certain distance from a stored event, it will be stored together with the already stored event, even though they are not exactly equal. This will result in similar events being stored together.

We calculate the event distances as the number of steps by which one sequence would have to be changed in order to become another sequence. If e.g. we have the two sequences “aabc” and “abbc” it is possible to change sequence one into two by changing the second ‘a’ into a ‘b’ (one step), therefore the distance between these two sequences is one. If we instead had the two sequences “aabc” and “adbd” the distance between them would be four since it takes three steps to change the second ‘a’ into a ‘d’ and one step to change the ‘c’ into a ‘d’.

In Figure 3.21 we rerun the experiment shown in Figure 3.17, but now we allow an event distance of 1. In this case we find 19,921 different event types in the sequence. The four most frequent events now occur much more frequently, which means that some events are now grouped together. Unfortunately, the overall event distribution has not changed very much.

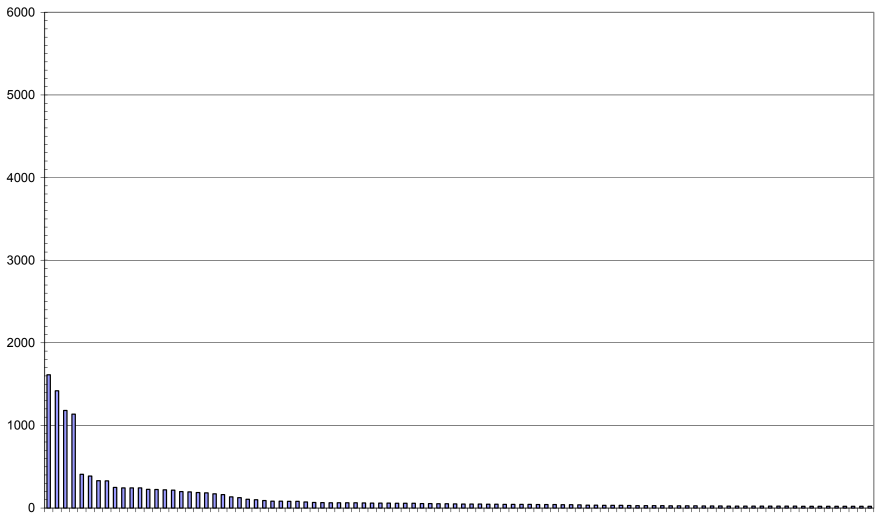


Figure 3.21: Events found in sequence, l-a-h 1, exact match, ev. distance 1

In Figure 3.22 the event distance is increased to 3 and we can see that the most frequent event type now occurs more than 3,000 times, a vast improvement, but unfortunately we do not see a similar increase in the other event occurrences. The total number of events found is now 16,173 different event types. So from the experiment in Figure 3.17 the total number of events is greatly reduced, but the vast majority of events are still unique events that only occur once in the sequence.

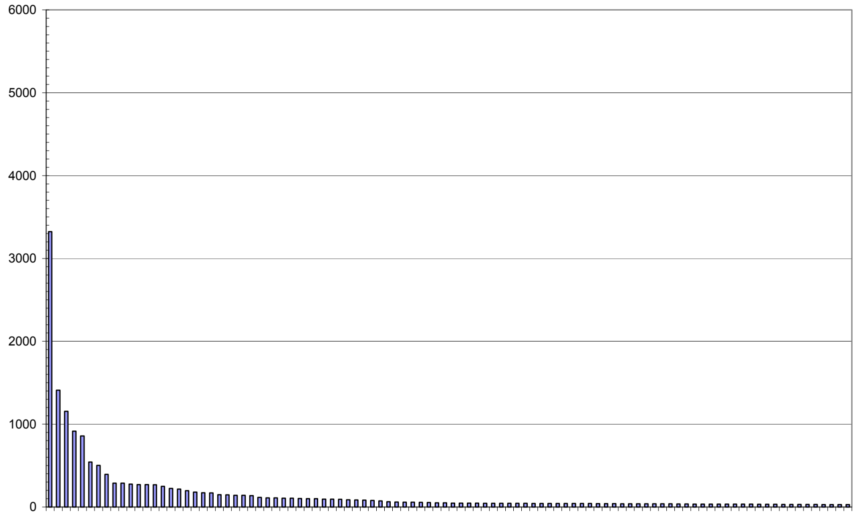


Figure 3.22: Events found in sequence, l-a-h 1, exact match, ev. distance 3

Finally, in Figure 3.23 the fuzziness is increased to 3. The total number of events detected is now only 7,721, compared to 16,173 different event types in the previous experiment, a great improvement. We can also see that the most frequent events now occur over 5,000 times, and for a small number of events the occurrences are significantly increased, but the vast majority of events are still unique and only occur once.

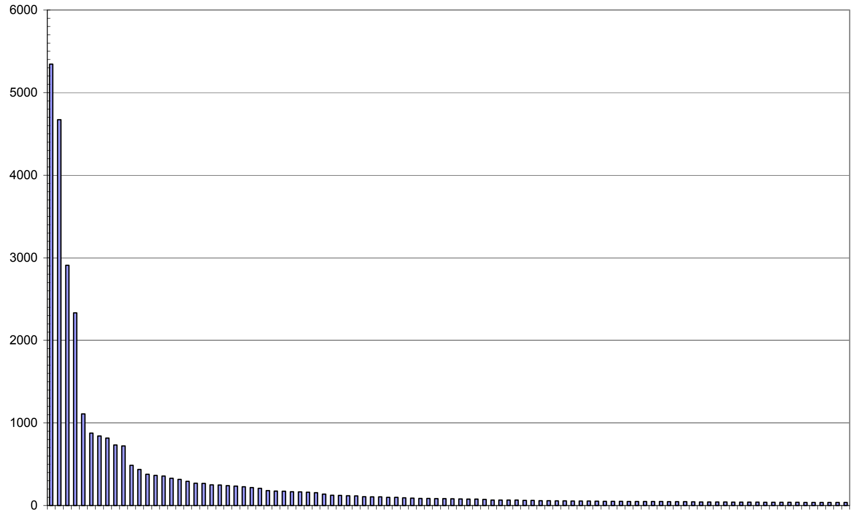


Figure 3.23: Events found in sequence, l-a-h 1, fuzzy match 3, ev. distance 3

3.5.5 Conclusion

It is tempting to see how far we are able to carry the text analogy, but trying to identify words, as we did, is taking it too far.

We have been able to identify a few events that occur frequently in the time series, but for general time series there also exist a very large number of unique events. It is possible to extract events if we know what to look for while doing the extraction, but it is not possible to do this in a general case.

Based on our test results we do not believe that it is possible to use this approach for creating a generic feature extraction technique.

There is, however, one very important issue here. What if the sequence was very "predictable" and contained few exceptional events? It is easy to imagine sequences that fit this description, e.g. a time series describing the hight of a trajectory or a time series describing the flight speed of an air line freighter. If we have time series like this, this method might work better.

In the example above with the trajectory a far better way of storing the time series might even be to approximate the time series with a mathematical expression. If the time series describes a very smooth predictable sequence this might be a very efficient way of storing the sequence.

Chapter 4

Current Approaches to Time Series Indexing

This chapter introduces the reader to other work related to ours.

In this chapter we take a closer look at a few different approaches to time series indexing. This section is divided into two parts. In the first part we explain some of the problems involved in indexing 1D signals. In the second part we describe some related work in indexing 1D signals. For an introduction to index structures and text indexing methods see Appendix D, "Indexing," on page 189.

In this chapter we talk about text indexing methods. It is important to bear in mind that these indexes in many cases can be used to index things other than text, e.g. images. In the case of images, however, each object must be associated with a keyword, e.g. it is possible to store images in a database but in order to search the images someone would have to associate keywords with each image.

There has been a great deal of research 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 one-dimensional signals is more complicated than indexing text, and so far nobody has been able to propose a generally applicable solution to the problem of indexing time series.

The related work we discuss in the second part of this chapter has been selected to represent five different approaches to this problem. Different refinements to these methods exist, but they can still be classified as belonging to one of these original types of approaches.

The first class of methods is represented in Section 4.2, "Fast Sub-Sequence Matching in Time Series Databases," on page 48. This class uses a distance preserving transformation to extract features from the time series. This transformation is then used to map a sequence of the time series to an N-dimensional point in feature space and a traditional index structure, e.g. R-trees, is used to index the data.

The second class is a slightly more advanced version of the previous one, presented in Section 4.3, “Fast Similarity Search in the Presence of Noise, Scaling and Translations in Time-Series Databases,” on page 50. The big difference between these two approaches is that the latter has a mechanism that makes it possible to find sequences that almost match the query sequence, fuzzy matching.

The third class is presented in Section 4.4, “Querying Shapes of Histories,” on page 52. The paper discussed here approaches the problem from a completely different angle by transforming the time series into texts and then defining a shape definition language that can be used to define queries on the textual representation of the time series.

The fourth class is presented in Section 4.5, “HierarchScan: A Hierarchical Similarity Search Algorithm for Databases of Long Sequences,” on page 54. Here everything is normalized before it is stored in the index, and the similarities can be measured by the number of transformations needed to transform the query sequence into the stored sequence. This class represents a more elaborate way of controlling the fuzziness of the query.

The final class is presented in Section 4.6, “Approximate Queries,” on page 56 where the entire time series is treated as a continuous signal and approximated by a mathematical approximation. Discarding the original data and replacing it with a continuous approximation opens up a new set of tools that we can use to store and search for similarities but it also introduces new problems.

4.1 Problems with Time Series 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 index structures to index this kind of data. Over the last few years this kind of data has become increasingly important.

In indexing a time series we are faced with many problems unknown to regular textual databases. The first problem is how information from the time series is to be extracted. This problem can be attacked in two ways. Either we design the system for a specific application, or we try to design a very generic system. In the first approach we learn what the time series looks like, and what features the users are interested in, then 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 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 in the time series, we will not be able to process those queries since the new feature was not previously extracted from the time series.

The second way to attack the problem is to try and extract features that will describe the time series 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 simply retrieving exact matches to a query for a time series. Since time series can be very complex it is more likely that the user will give the system an example of a time series and ask the system to retrieve all time series that are similar to the example time series. What do we mean when we say that two time series are similar? If we return too many time series to the user, the system will be useless to him since he will miss the interesting time series among all the non-interesting time series. If, however, we return too few time series, it is possible that some of the time series the user was interested in will not have been retrieved by the system, which is to say we can not find data in the system. There are two common ways of approaching to this problem. Either the similarity mechanism is built into the system or the user is given some control over it. If the mechanism is built into the system then the user has very little or no control over the degree of similarity. The second approach will allow the user to specify the fuzziness of the query while stating it to the system. We feel that it is better to provide the user with some control over how many results a query should provide.

Next we need to address the index structure. The index structure should allow us to search the collection of time series in a fast and efficient way, but it is preferable that it is small in comparison to the time series being indexed, and that it grows in proportion to the time series. The most common index structures are some sort of tree structures that usually offer very good search performance as well as growth performance. There are exceptions, e.g. the signature file. The signature file's search time grows linearly with the data set, but the size of the index is usually very small and the inclusion test is simple compared to the data being indexed. In many cases the signature file still is a good index structure compared to a tree structure. The size compared to the simplicity of the index structure 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 time series, since a collection of time series easily can become very large (Terrabyte-sized).

The papers that we have selected as representatives of the different classes of approaches have all been analyzed according to the following template so that we can more easily compare them:

- **Introduction.** This is a short description of the proposed approach.
- **Feature Extraction.** A description of the method used to extract the features of the time series. For a more detailed discussion on feature extraction methods see Section 3, “Feature Extraction,” on page 23.
- **Indexing.** A short description of the indexing structure proposed by the authors.
- **Similarity between shapes.** A short description of the similarity concept being used in the proposed method.
- **Positive features.** A short summary of the positive features of the proposed method.
- **Negative features.** A short summary of the negative features of the proposed method.

4.2 Fast Sub-Sequence Matching in Time Series Databases

This way of solving the problem is represented by [FALO94]. In [FALO94] the authors propose an effective indexing method to locate time series sub-sequences 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. For a description of an R*-tree see Appendix D.7.4, “R*-Trees,” on page 204.

4.2.1 Feature Extraction

This method uses a distance preserving transform, for example, the Discrete Fourier Transform (DFT), to extract features from the time series to feature space. In order to accomplish sub-sequence matching, a simple process is used:

- A small window is introduced.
- The window slides along the time series, one position at a time.

- 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 time series 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.

4.2.2 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 4.1. In order to search the index, all stored sequences, whose MBRs intersect the query MBR, are returned.

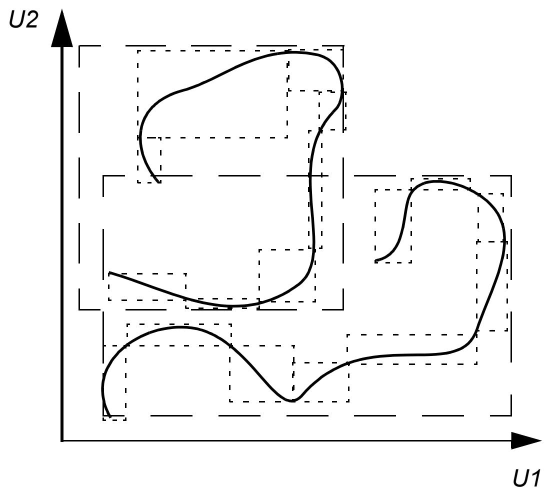


Figure 4.1: Trail with MBRs that are stored in the index

4.2.3 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 one checks for the intersecting MBRs.

4.2.4 Positive Features

The method relies upon already existing and well-known indexes. It uses a well-known simple method of extracting features from the time series, then stores these feature vectors in a well-known, multi-dimensional index structure.

This makes it possible to add sub-sequence searching facilities to an existing system without too much extra work.

4.2.5 Negative Features

When the index structure is searched, not only will all sequences matching the query 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 has 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 [FALO94] is very sensitive to scaling and translation.

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

This class of more advanced solutions is represented by [AGRA95]. 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 multi-dimensional rectangles in a feature space. The rectangles are then stored in a traditional spatial index, for example an R*-tree.

4.3.1 Feature Extraction

Like [FALO94] this method uses a distance preserving transform, for example the Discrete Fourier Transform (DFT) to extract features from the time series. It also uses a window that slides along the time series, 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.

4.3.2 Indexing

The data points are stored in an R-Tree (the authors tried implementing the index as both an R*-tree and an 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 Sub-sequence matching – find a non-overlapping ordering of sequence matches having the longest match length.

4.3.3 Similarity Between Shapes

The process of determining whether two time series 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 ε 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 both 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 Equation 4.2 is valid.

$$\forall i\big|_1^m \Theta(\lambda(S_i)) \cong T_i \tag{4.2}$$

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

$$\frac{\sum_{i=1}^{m} length(S_i) + \sum_{i=1}^{m} length(T_i)}{2 \cdot min(length(S), length(T))} \ge \xi \tag{4.3}$$

4.3.4 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 of the similarity process. On the first level the user can specify the maximum distance between two vectors that should be considered similar. 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 to be accepted by the system.

4.3.5 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.

4.4 Querying Shapes of Histories

This approach to the problem is represented by [AGR295]. This is a very novel approach and we have not found any other papers suggesting this solution. 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 time series. The language is made up of a few elementary shape descriptors and operations. To build more complex shapes, the operations can be applied to the elementary shapes to get new shape descriptors. They also propose a data structure that serves as a fast index for searching the stored sequences.

4.4.1 Feature Extraction

The idea behind the feature extraction is to extract the same features that humans extract when they see a one-dimensional signal (a time series). A one-dimensional signal can be seen as a transition sequence, and it is these transitions that we extract. The process can be described like this:

- In each position, calculate the amplitude difference from the previous position and the current position.
- depending on this difference, add a symbol from an alphabet of elementary transitions to the feature extraction string.

The alphabet used in the paper is shown in Table 4.1 on page 53.

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

Table 4.1: 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

4.4.2 Similarity Between Shapes

The similarity concept of this model is very simple. Two sequences are similar if they map to the same symbol string. For an example of this similarity concept see Section 3.1, "Feature Extraction Using Time-Derivative," on page 25.

4.4.3 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 id's that point to the original data.

4.4.4 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, which are the most elementary shapes in the system, are defined like this:

(alphabet (symbol lower_value upper_value incomming_constrain outgoing_constraint))

Table 4.1 on page 53 shows the alphabet used in the paper and each symbol's definition.

To combine these symbols into shapes, the user can employ 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.

4.4.5 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.

4.4.6 Negative Features

The precision of the search is determined by the size of the alphabet, and it is difficult 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.

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

This class of solutions is also a very small class since we have only found this approach in one paper, [LI96]. In [LI96] a method to effectively search a collection of time series 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.

4.5.1 Feature Extraction

Feature extraction can be performed with DFT, DCT, sub-band coding, or wavelet transform. First the time series 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.

4.5.2 Indexing

The method proposed by this paper does not suggest a way to index the feature space vectors, instead it relies 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.

4.5.3 Similarity Between Shapes

The authors propose to use several correlation functions, as a similarity concept. A segmented feature space vector is obtained from the query sequence, then the most prominent segment of the query vector is checked against all stored feature vectors' same segments with the first correlation function. The correlation functions should be chosen so that we get 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. The second most prominent query segment is then checked against the remaining stored vectors' same segments with the second correlation function, and so on.

4.5.4 Positive Features

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

4.5.5 Negative Features

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

4.6 Approximate Queries

This is also a very novel approach and this class is represented by [SHAT96]. In [SHAT96] the authors wish to introduce a new method that has a very simple and powerful concept of similarity. By making the method more of a framework and relying heavily on knowledge from experts on how to perform certain key steps in the process, it should be possible to achieve good system performance in some applications. On the other hand, the heavy dependence on experts will make it impractical for many applications.

4.6.1 Feature Extraction

The idea is to break the time series 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. These functions have the following desirable features:

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

The points where the time series will be broken are very important since all important searchable features must exist inside a segment. Figure 4.4 shows a time series broken into segments, and each segment's polynomial approximation of the first degree.

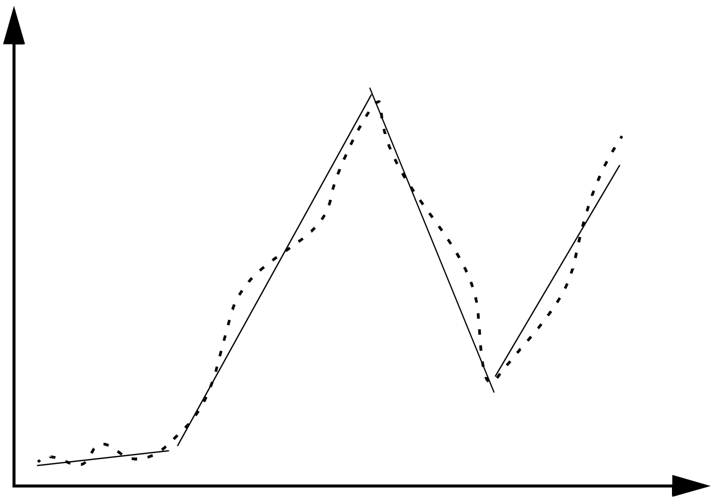


Figure 4.4: A signal and its polynomial approximation

4.6.2 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 time series 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 corresponding to the specified features (for example the number of peaks or the steepness of the slopes) within a certain error tolerance.

4.6.3 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, instead they note it as work to be done in the future. This makes it very difficult to compare this method with other methods, since the index is one of the most important features of a system.

4.6.4 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.

4.6.5 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 features of the incoming data 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 time series and determine which features that are important and how they 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 method is the lack of a suggested index structure. Since the index structure is a very important aspect of the system, it is hard to say anything about the system without any ideas about how the index structure should be constructed.

Chapter 5

The Signature File

This chapter will introduce how we have used signature files to index time series and how we use them to search the time series. We will show the theoretical sizes, probability of falsedrops, and search costs for different signature file parameters.

Time series have been used to offer a very small index structure that allows free text searches. This is done by hashing the words in the text to a binary vector, then searching the binary vector instead of the original text. For a more in-depth description of signature files, see Appendix D.4, "Signature Access Methods," on page 193.

This chapter begins with a brief introduction to the modifications we had to make in order to use signature files to index time series. We also describe some of the major problems we faced, and how we solved them. We then continue with a short evaluation of different hash functions. Since the alphabet we are using is much smaller than the alphabet used in English text, it is important to find a hash function that still works well.

We will then take a look at the size and search performance of the signature file. Finally, we will conclude this section with a summary of the signature file structure.

5.1 Introduction

One indexing method that produces small indexes for large texts is signature files. Since one of our initial goals was to find an index structure for large time series (see Section 1.6, "Contributions of This Work," on page 11) we saw the signature file as a good candidate for our index structure.

As described in Appendix D.4, “Signature Access Methods,” on page 193, there are several different types of signatures. We decided to use superimposed-coding, SC, signatures since this allows us to use a simpler inclusion test than the other signature types, see Appendix D.4.3, “Superimposed Coding,” on page 194.

When signatures are created from text, words are fed to the signature generator. Each signature represents some discrete 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 present in a certain paragraph, chapter, or book, depending on what the signature represents. A signal on the other hand lacks any discrete parts and cannot be broken up. The signal is one huge continuous stream of data and each point is related to its predecessor and its successor. We decided early on that a minimal atomic unit had to be introduced. Without this limitation we would have to index each individual symbol in the time series. To compare such an undertaking to text indexing, this would mean not only indexing every word in the text, but also every letter in every word. The question we asked ourselves was whether allowing searches for arbitrarily small sequences would be meaningful.

The minimal unit we introduced is the smallest entity that can be searched for in the time series, a window. The window creates discrete units in the time series. It groups a number of values from the time series together and treats them as a unit, which we refer to as a word. But if the time series is divided into words, a query from the user will be unlikely to 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. This is solved by sliding the window along the time series, one position at a time. In this way each possible word is created and stored in the index, see Figure 5.1. Unlike words in an English text, all these words will be of the same length.

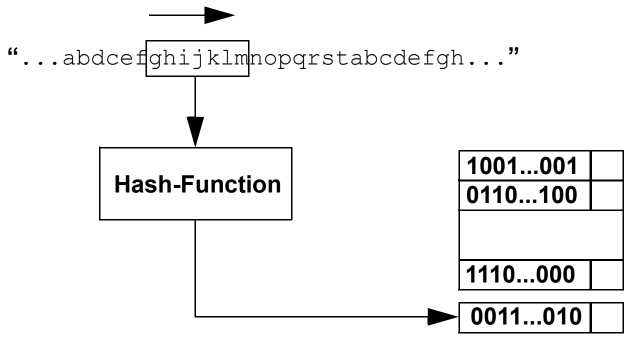


Figure 5.1: Creating signatures from the symbol string

Furthermore, a way to break up the time series into sections or blocks has to be found. If one signature represents the entire time series, the only result that could be obtained from the system would be if a given sub-sequence was present in the time series. The location of the sub-sequence would not be available.

Additionally, the signature needed to map an entire time series would be very large. Signatures used to map texts are more than 500 bits long, and since time series can be very large (usually much larger than regular English texts), the signatures needed to map such a time series would also have to be very large.

This is solved by breaking the time series into smaller sequences, which we refer to as 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 sub-sequence. [DAVI95] points out that if the signatures are 50% full, i.e. 50% of the bits in the signature are set to 1, then the false drop is minimized with respect to the fill grade (how much of the signature that contains 1's). A signature block is considered full when it is 50% full and then the next signature block is started.

The false drop is a measurement of how many extra, false, hits the index will return as answer to a query. Since signature files is a probabilistic index it will always return results that have a possibility of containing the query pattern, but it is not certain, see Section 5.7, "Probability of False Drops in the Signature File," on page 74.

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 searching only the signatures.

5.2 Different Signatures

We found and examined two approaches that solve the problem of finding sequences between signature blocks. The first approach introduces overlapping signature blocks and the second approach introduces a modified search mechanism.

5.3 Overlapping Signatures

The idea behind overlapping signatures is that each sub-sequence should be stored in two signature blocks so that all sub-sequences will be present 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 time series 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 Equation 5.4, i.e. when a signature block is full we start the next signature block from the middle of the previous one, see Figure 5.3.

This will assure us that each sub-sequence we are searching for will be found by searching the signatures.

Figure 5.2 and Figure 5.3 show the difference between normal signatures/signature blocks and overlapping signatures/signature blocks. In Figure 5.2 a time series is indexed using normal signatures and in Figure 5.3 the same time series is indexed with overlapping signatures. If we are searching for the sequence “mnopqrs”, present at the end of signature block n, we will find it in signature block n in both examples. If we instead were searching for the sequence “nopqrst”, we would not be able to find it using the normal signatures, since that sequence is not present in any signature block. Using overlapping signatures we are able to find the sequence in signature block $n + 1$.

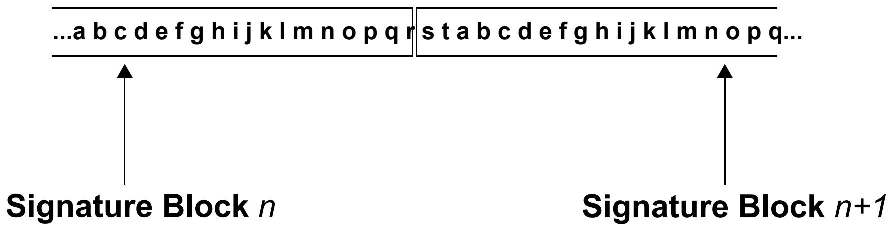

...a b c d e f g h i j k l m n o p q r s t a b c d e f g h i j k l m n o p q...
Signature Block n
Signature Block n+1

Figure 5.2: Normal signature blocks

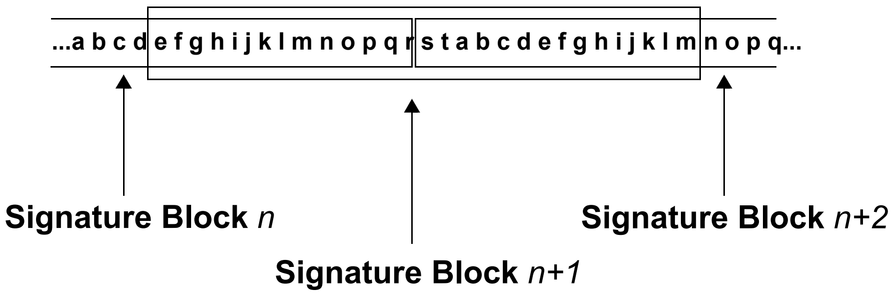

...a b c d e f g h i j k l m n o p q r s t a b c d e f g h i j k l m n o p q...
Signature Block n
Signature Block n+2
Signature Block n+1

Figure 5.3: Overlapping signature blocks

$$NewStartPos = \frac{OldEndPos - OldStartPos}{2} \tag{5.4}$$

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 a search might fail to find a sequence which is present in the time series, leaving the initial problem unsolved.

5.3.1 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 sequence and each window's 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. The next window position will be the same as with normal signatures.

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. See Figure 5.5 for the search algorithm.

```
result searchSignatureFile(List SignatureFile,Signature Query){
  List searchResult = EmptyList;
  Signature CurrSig = SignatureFile.getFirst();
  Signature NextSig;
  if(CurrSig != EndOfList && Query & CurrSig == Query){
    searchResult.add(CurrSig.BlockPosition);
  }
  while( CurrSig != EndOfList ){
    NextSig = SignatureFile.getNext();
    if( NextSig != EndOfList ){
      if(Query & NextSig == Query){
        searchResult.add(NextSig.BlockPosition);
      } else {
        if(Query & (CurrSig | NextSig) == Query){
          searchResult.add(CurrSig.BlockPosition);
        }
      }
    }
    CurrSig = NextSig;
  }
  return searchResult;
}
```

Figure 5.5: Signature search algorithm

The algorithm can be written as:

1 Check the current block, if we find a match advance the current block pointer and restart on step 1.
2 Advance current block pointer and check again. If we find a match, advance block pointer and restart on step 1.
3 Logically or the current and previous signatures and check them for a match. Advance current block pointer and continue on step 1.

Informally we can say that we first check whether the sequence we are looking for is present in the current signature block, then we check for it in the next signature block. If both these tests fail we combine the two signature blocks into one, and then test whether the sequence is present in the combined signature block. One important thing to remember here is that the actual signature blocks are never searched, we only search the signatures. Since the signatures are bit vectors obtained by hashing the contents of the signature block, we loose some information in the process.

This new approach introduces a new problem. A signature is, as mentioned before, considered full when 50% of the bits are set to 1. If we logically OR two such signatures together, in a worst case scenario we will end up with a combined signature in which all positions set to 1's. 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 to 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.

Most important, however, this method offers the benefit of searching effectively without putting an upper limit on query length when the query only consist of one signature.

5.3.2 Conclusion on Search Mechanism

If we decided to go with the first approach, overlapping signatures, we would get a system with both a minimal and a maximal query length. By choosing the second approach, a new search mechanism, we could avoid the upper query length limitation as long as the query only consists of a single signature.

5.4 Hashing the Windows to Signatures

In this section 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 is 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 5.1 on page 60.

When we say that a hash function performs well, we mean that all bits in the signatures created using the hash function have a similar probability of being set.

To hash the window contents, the words, to the signature we need an efficient hash function. We have tested three different hash functions, one very simple, one moderately complex and one complex. The first hash function tested was Equation 5.6, presented in [DAVI95]. In Equation 5.6, S is the word we are calculating a hash value for, and B is the size of the signature. Equation 5.6 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 replaced with the slightly more complex hash function, Equation 5.7.

$$BitPos = mod\left(\sum_{i=0}^{length(S)} S[i], B\right) \tag{5.6}$$

In Equation 5.7, S is the word we are calculating the hash value for, B is the size of the signature, and C is a constant that satisfies Equation 5.8. K is the range of the keys we wish to generate.

This hash function performs quite well with smaller alphabets, but has some problems if the small alphabet is combined with a very small window.

$$BitPos = mod\left(\frac{\left(\sum_{i=0}^{length(S)} S[i]\right)^2}{C}, B\right) \tag{5.7}$$

$$C = \left|\sqrt{\frac{K^2}{B}}\right| \tag{5.8}$$

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 [PGP00]. This function is much more complicated than the previous two, but provides very good performance. The algorithm is too long to reproduce here, so the interested reader can find a detailed description in [RIVE92].

5.4.1 Longer Queries

Since the search mechanism does not put a restriction on the query length as long as the query maps to only 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 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, we then 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, 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.

5.4.2 The Size of the Signature Files

One reason 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. It is important to consider whether 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 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 one calculated. Consequently the signature file will contain fewer 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. Equation 5.9 gives us the number of set bits in a full signature.

$$bits = N_{sb} \cdot F_g \tag{5.9}$$

If this value is divided by the number of bits each word set, w, we get the number of words that correspond to a full signature, see Equation 5.10.

$$\frac{words}{signature} = \frac{N_{sb} \cdot F_g}{w} \tag{5.10}$$

Since the words are created by sliding a window over the time series, this number of words corresponds to the same number of characters plus the number of characters in a word minus one. This gives us the minimum signature block size, in tokens, see Equation 5.11.

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 (Equation 5.11)

$$SB_{size} = N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{w} \tag{5.11}$$

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

$$N_{sig} = \frac{n}{SB_{size}} = \frac{n}{\left(N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{w}\right)} \tag{5.12}$$

The size of the signature file will be the number of signatures multiplied by the size of each signature. Each signature will consist not only 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:

$$SF_{size} = N_{sig} \cdot \left(\left\lceil \frac{N_{sb}}{b} \right\rceil + p \right) = n \cdot \frac{\left\lceil \frac{N_{sb}}{b} \right\rceil}{N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{w}} \qquad (5.13)$$

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

$$R = \frac{SF_{size}}{n \cdot n_s} = \frac{\left\lceil \frac{N_{sb}}{b} \right\rceil}{\left(N_{ws} - 1 + N_{sb} \cdot \frac{F_g}{w} \right) \cdot n_s} \qquad (5.14)$$

The ratio for an input file with 400,000 4-byte data points has been calculated for different configurations of the signature file, see Figure 5.15, “Signature file size, 30% fill grade, 1 bit per window,” on page 69 and Figure 5.16, “Signature file size, 30% fill grade, 4 bits per window,” on page 70.

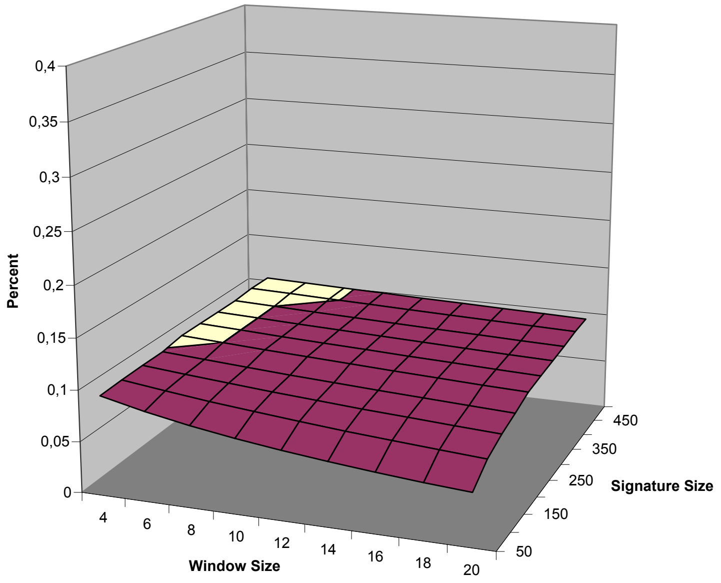


Figure 5.15: Signature file size, 30% fill grade, 1 bit per window

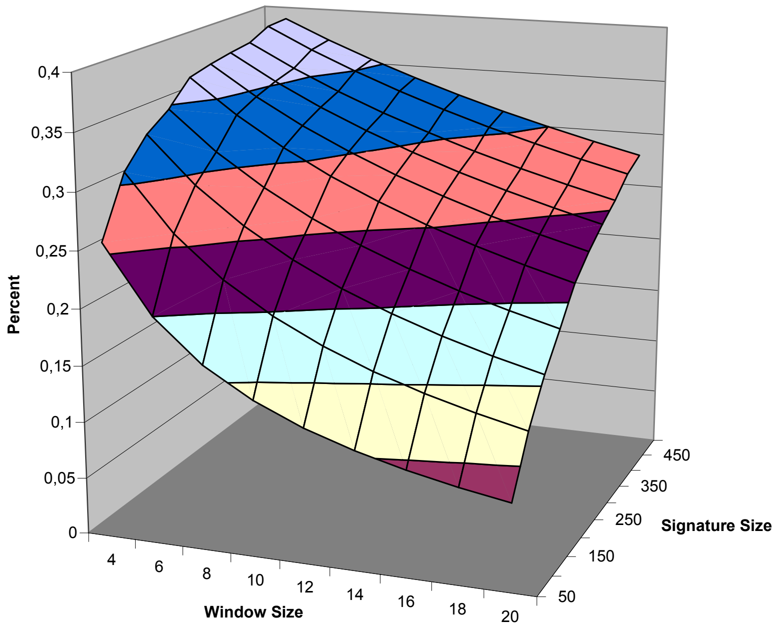


Figure 5.16: Signature file size, 30% fill grade, 4 bits per window

In these calculations the following assumptions were made:

- The number of values in the original sequence, n, was set to 400,000.
- The pointer size, p, was set to 4 bytes.
- The size of a byte, b, was set to 8 bits.
- 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 5.15 and in Figure 5.16, the ratio is always smaller than 100%, i.e. the signature file is always smaller than the data file used to create the signature file. It is also easy to see that for most configurations of the signature file, the 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 5.15 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, the signature file size will increase instead of decreasing with growing signature sizes. In Figure 5.16 each window sets 4 bits instead of 1 bit. As shown in Section 5.8.3, “Precision Diagrams,” on page 82, 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 great influence. 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. Since the signature file size increases linearly with the number of bits used to create each signature, it is not meaningful to use more than 4 bits per window.

It might be possible to improve the size if the signature file, especially for signatures with a low fill grade, using signature compression as described in Section D.4.4, "Bit Block Compression," on page 194, but we have not looked into that possibility.

5.5 Searching the Signature File

In this section the search performance of the signature file is compared to a simple 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 5.4.2, "The Size of the Signature Files," on page 67, 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, a few assumptions must be made. 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 in calculating the search cost. Therefore we will say that the cost of a search is equal to the number of disk accesses needed to perform the search.

5.5.1 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 Equation 5.17.

$$C = \left\lceil \frac{n \cdot n_s}{P_s} \right\rceil \tag{5.17}$$

The cost of scanning the data for a few different page sizes and for a few different data sizes is shown in Table 5.1. The size of each element in the file is set to 4.

Table 5.1: 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 is relevant to point out that the cost will be the same even for smart search algorithms, like the Boyer-Moore algorithm [BOYE77]. These algorithms only speed up the search of each individual page, something we have disregarded in all our tests, but we still have to get all pages from secondary storage. This is the only cost we regard.

Another issue relevant to cost is that if we are not interested in retrieving all occurrences of a queried pattern, only the first occurrence, or if we only want to check only whether the queried pattern is present or not, the search will, on average, be half the time we have calculated above with Equation 5.17.

One thing we must mention here is that since we perform a sequential scan of the data, the search cost will be lower. This is because when we search the original data and when we describe the signature file in the next section, the performance will be determined by the transfer time and not by the block search time. As described in Section 2.2, “Transfer Time,” on page 17, if we can place the data in consecutive sectors on the disk, we will be able to read the data at least 1.6 times faster. This does not matter when we compare brute force scanning to signature file scanning but it will be important when we later compare these methods with e.g. B-trees.

5.5.2 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 5.14, on page 69 is used to calculate the size ratio of the signature file and the original data file. If we multiply this ratio by the size of the original data file and then divide this size by the size of each disk page we get the number of disk pages we need to scan. In Table 5.2 we have assumed a ratio of 30%. We show in Section 5.7, “Probability of False Drops in the Signature File,” on page 74 and Section 5.8.3, “Precision Diagrams,” on page 82 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 5.2 shows the cost of scanning the signature file.

Table 5.2: Cost of scanning signature file

Number of Elements	**4kb**	**8kb**	**16kb**	**32kb**
10^3	1	1	1	1
10^6	294	147	74	37
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 5.1 and Table 5.2 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. As we can see, it will always be cheaper to search the signature file than to perform a linear search of the original data, even if we use a smart scanning algorithm for the linear scan. In all cases other than the linear search of the original data, the result of the search will be a number of pointers to 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 in whether or not a sub-sequence is present or not in a larger sequence, we do not have to add this time. If we are only interested in the first occurrence, 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 need to find the first occurrence.

5.6 Similarity in Signatures

It is tempting to assume that two similar sequences will hash into two similar signatures. The argument for this is that if a string X map to signature a S. Then if we introduce a small change to the string 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.

5.6.1 Hypothesis

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 simply by 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.

5.6.2 Result

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 see that two similar sequences can hash into two dissimilar signatures.

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

5.7 Probability of False Drops in the Signature File

A false drop occurs 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 (Equation 5.18).

$$F_d = \{SignatureQualifies | BlockDoesntQualify\} \tag{5.18}$$

The probability that a bit in a signature with m bits and fill grade g is 0 can be calculated with (Equation 5.19).

$$P_0 = \left(1 - \frac{1}{m}\right)^{gm} \tag{5.19}$$

To calculate the risk of a false drop we can calculate the probability that the k bits in the query signature are set in a stored signature (Equation 5.20). 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:

$$p(k) = \left[1 - \left(1 - \frac{1}{m}\right)^{gm}\right]^{k} \tag{5.20}$$

In Figure 5.21, Figure 5.22, Figure 5.23, Figure 5.24 and Figure 5.25 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 5.8.3, “Precision Diagrams,” on page 82, we can see that the theoretical values are quite accurate. To compare the results, we have to convert the probability of a false drop into a precision. To do this we can take the probability of a 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.

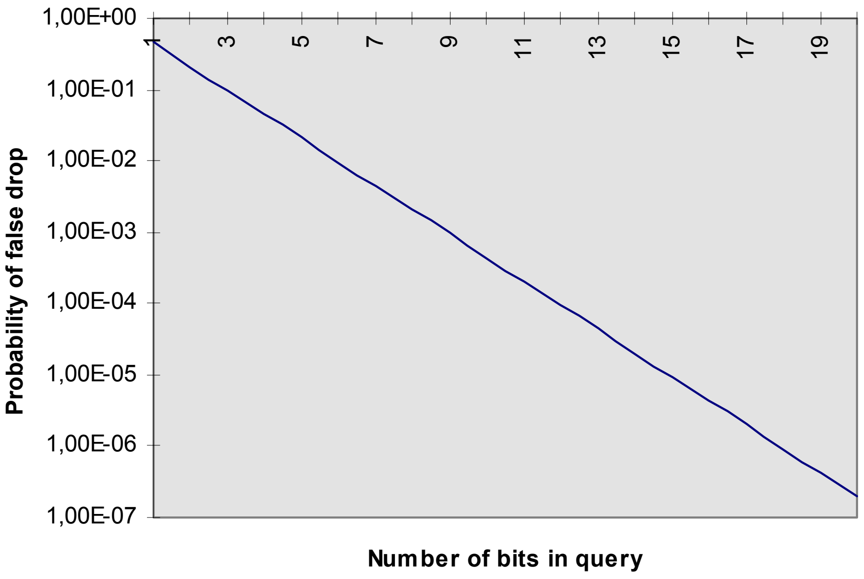


Figure 5.21: Probability of false drop with a 16-bit signature

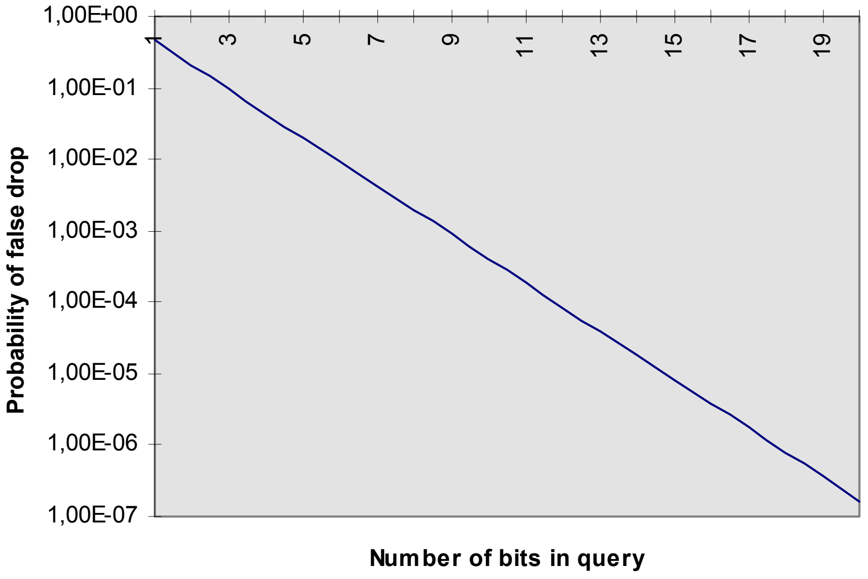


Figure 5.22: Probability of false drop with a 24-bit signature

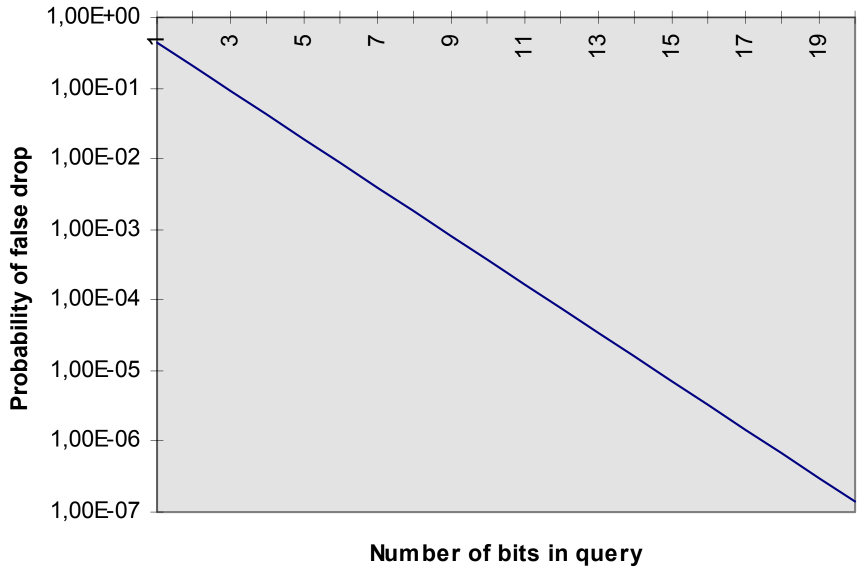


Figure 5.23: Probability of false drop with a 61-bit signature

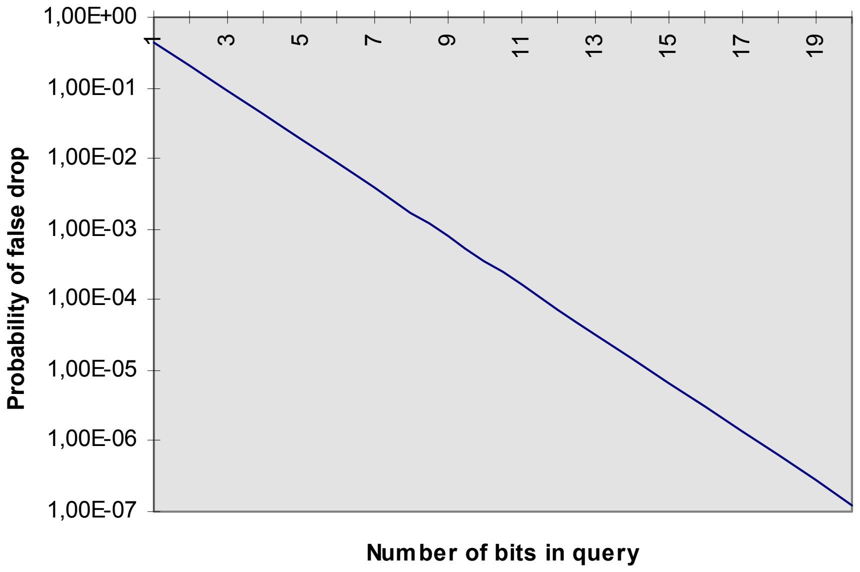


Figure 5.24: Probability of false drop with a 251-bit signature

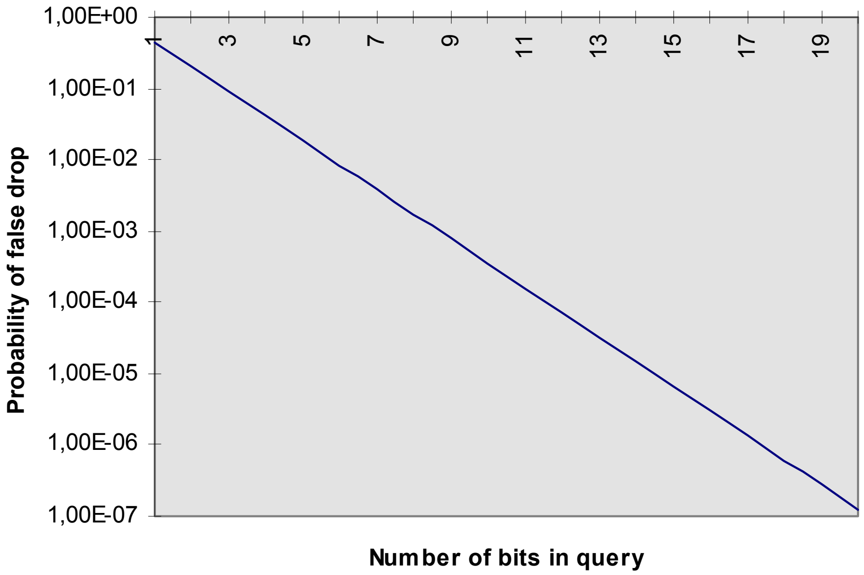


Figure 5.25: Probability of false drop with a 509-bit signature

5.8 The Tests

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 as signature files, and is defined as the ratio between the number of returned relevant documents and the total number of returned documents, see Equation 5.30. We have also conducted tests of the hash function to ensure that the signatures generated were of high quality, see Section 5.8.1, “Signature Quality,” on page 79. It is important that the generated signatures are of high quality, since otherwise it would be impossible to determine if problems in the search algorithm stem from badly generated signatures or problems with the search algorithm.

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
- 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 on 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 determines the minimum query length in the finished system. Since we do not want to design the system so that it can only find large sub-sequences, 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 minimum possible alphabet size was primarily tested with the hash function, but we also did some tests where 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 6, “Indexing the Signature File,” on page 85) we would like to have the signatures as small as possible. The different signature sizes used are 41, 61, 251, and 509 bits, to give us some information about how the precision varies with the signature size.

During our first set of tests, see Section 5.8.2, “Measuring the False Drop,” on page 82, we saw that the precision was often very poor when we searched the signature file for short sequences. In 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 done with 1, 4, and 6 bits, but since no difference was achieved by using more than 4 bits, only the first two categories are shown here.

5.8.1 Signature Quality

The quality of the signatures was measured by generating a signature file from a large test file (see Section 2.3.1, “Synthetically Generated Time Series,” on page 18). When the signature file was produced, all signatures were scanned and the bit probability of each bit in the signature was calculated. Each bit in the signature should have the same probability of being set, and if this is not the case it is an indication that the hash function does not work with the data set.

This data was then used to calculate a mean bit probability, and a standard deviation of the mean bit probability for each configuration. The minimum and maximum bit probabilities were 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 was around 55,000. This number decreased as the signature size increased, and in the last test with a 509-bit signature, the number of created signatures was around 2,000. This is the reason that the performance seems to slowly degrade as the signature size increases.

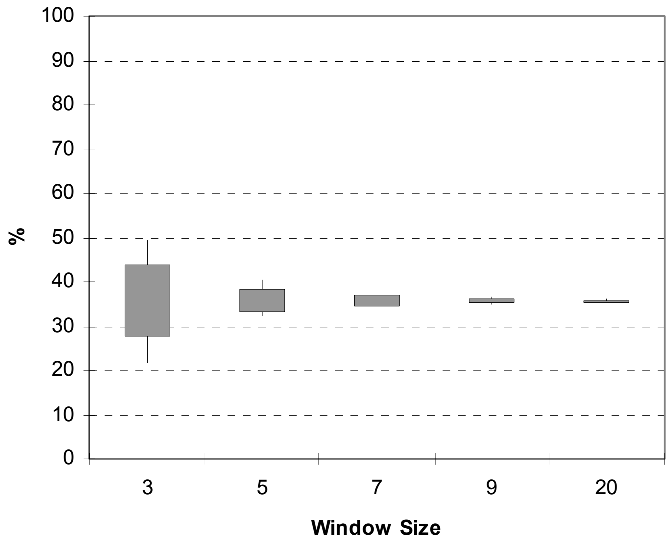


Figure 5.26: Signature quality: 16-bit signatures

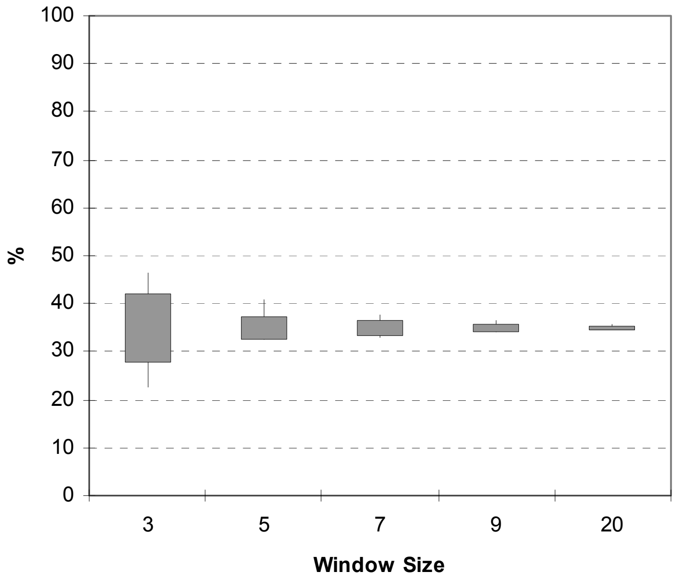


Figure 5.27: Signature quality: 24-bit signatures

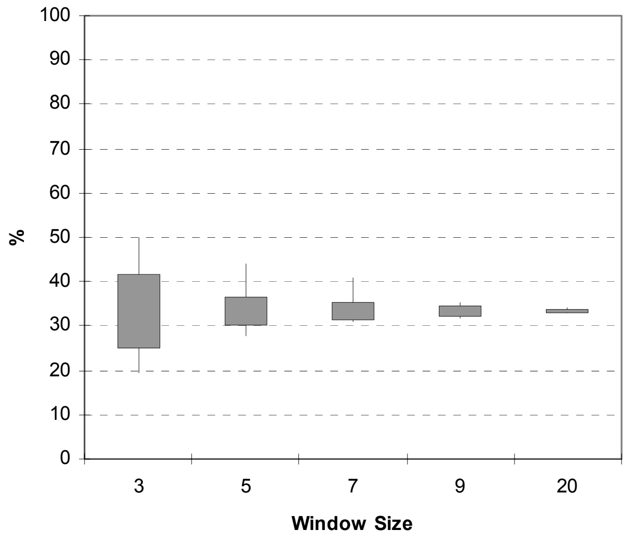


Figure 5.28: Signature quality: 41 bit signature

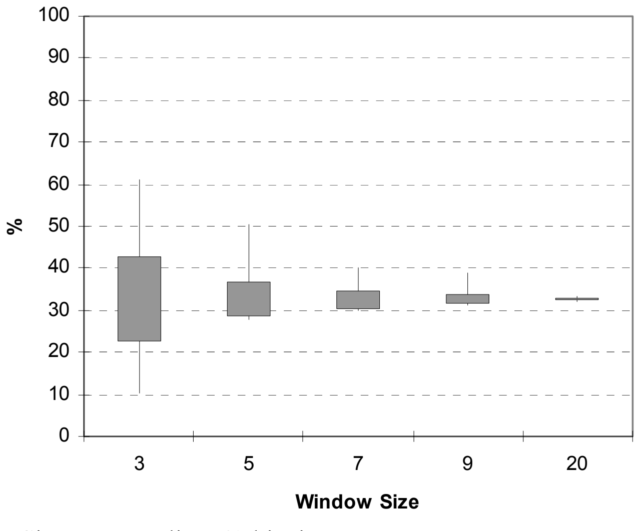


Figure 5.29: Signature quality: 61-bit signatures

The average bit probability should be equal for all bits, and should be close to the fill grade. In this case the fill grade is 33%. By looking at these diagrams we can see that the average bit probability is close to 33%, and as the window size increases, the deviation of the bit probability diminishes. If the window size is five characters or more, the signatures produced are of high quality.

5.8.2 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 sub-sequences 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 Equation 5.30.

$$Precision = \frac{ActualOccurences}{ReturnedOccurences} \tag{5.30}$$

To see how the false drop was influenced by the size of the sub-sequences, 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 sub-sequence, 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.

5.8.3 Precision Diagrams

We have done extensive testing on how the precision is influenced by the signature size, the window size, the number of bits set by each window in the signature, and the query length. These diagrams can be found in Appendix A, “Signature File Precision Diagrams,” on page 143

Our conclusion from these experiments is that it is possible to choose the signature size, the window size, and the number of bits per window, so that the precision is very high even for very short queries.

5.9 Conclusion

Signature files are a simple probabilistic index structure for files. We have shown that it is possible to use signature files to index time series as well. The difference between time series and text introduces some additional problems that are not present in text signature files. We have shown that these problems can be solved, but that the solution makes the ratio between the signature files and the original data file higher, from 5% to 10% for text files, up to around 30% for time series – see Section 5.4.2, “The Size of the Signature Files,” on page 67 for a thorough discussion of this issue.

Even though the size of the signature files created from time series are larger than signature files created from text, we show that it is still more cost effective to search the signature file than to scan the original data file, no matter what kind of search algorithm you use. See Section 5.5.1, “Cost of Scanning the Data,” on page 71 and Section 5.5.2, “Cost of Scanning the Signature File,” on page 72.

If we use an advanced hash function, it is possible to construct signatures of high quality even if the alphabet used is very small. For a more thorough discussion on this topic see Section 5.8.1, “Signature Quality,” on page 79.

Finally we have shown that by choosing the parameters available, it is possible to achieve a very high precision, so that in many cases a secondary search to remove the false drops will not be necessary – see Section 5.8.3, “Precision Diagrams,” on page 82.

Chapter 6

Indexing the Signature File

In this chapter we will examine the signature tree. The signature tree, or ST, is a data structure we developed to address the shortcomings of the signature file, namely that the search time grows linearly with the size of the data set.

We start by giving a short introduction to the problem description and then we go on to describe the data structure. After the overall description, we give a theoretical analysis of the size and performance of the structure. These are then verified with tests, and we conclude the chapter with our conclusions regarding the signature tree structure.

6.1 Introduction

To improve the performance of the system it is vital to find an efficient index structure. Our goal was to find a structure that would allow us to quickly discard uninteresting signatures and leave us with the ones in which we are interested. This is how search trees behave, and since search trees are a well-researched area, some kind of tree structure seems promising. While working with signatures we have found that the search structure needs to fulfil two further criteria: it has to be searchable with incomplete keys, and the ordering of the signatures must be preserved.

We have to 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 because the signal is ordered, we always have to be able to determine if one event comes before or after another event. The behaviour and some further aspects of the ST index are similar to another index proposed in [LIN98].

6.2 The Signature Tree

Since the signatures are binary words, and we use the regular Boolean operations on them, one structure that matches all these criteria is a Boolean algebra. The Boolean algebra is a lattice. Figure 6.1 shows a simple Boolean Algebra.

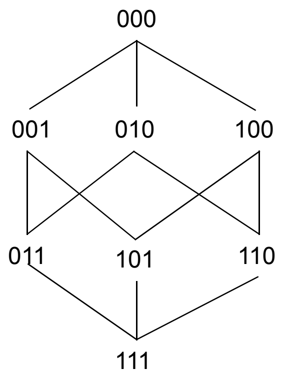


Figure 6.1: A Boolean algebra

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 others set to zero. The second level consists of all possible binary words with two bits 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 nodes' 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 a Boolean algebra. In the text we will refer to this 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 start 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 previously created nodes, so the index will grow linearly.

6.3 The Size of the ST

In this section we will examine the size of the ST structure. First we will take a look at the theoretical size of the structure, and then we will compare these results with our experimental implementation of the structure.

6.3.1 The Theoretical Size of the ST

To find out how large this structure can become 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 described by Equation 6.2.

$$Nodes = \sum_{y=0}^{N_{sb} \cdot 2 \cdot F_g} \binom{N_{sb}}{y} \qquad (6.2)$$

For each level the number of children of each node will decrease by one, the top node having a 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 using Equation 6.3.

$$MaxSize = \sum_{y=0}^{N_{sb} \cdot 2 \cdot F_g} \binom{N_{sb}}{y} \cdot (N_{sb} \cdot p + i_b) \qquad (6.3)$$

Table 6.1 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 6.1 on page 88 we have used Equation 6.2 and assumed that $F_g = 0,3$.

Table 6.1: Number of nodes in the ST using different signature sizes

Signature Size	**12**	**16**	**20**	**61**
Total number of Nodes:	3797	58651	910596	2×10^{18}

Using Equation 6.3, 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 6.2 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 6.2: ST sizes for different signature sizes

Signature Size	**12**	**16**	**20**	**61**
Total Size (in bytes):	$2,0 \times 10^5$	$4,0 \times 10^6$	$7,7 \times 10^7$	$5,5 \times 10^{20}$

6.4 Building the Signature Tree

When a new signature is to be added to the ST, 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 node's children we should move 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 in which child of the current node we should place the new 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.

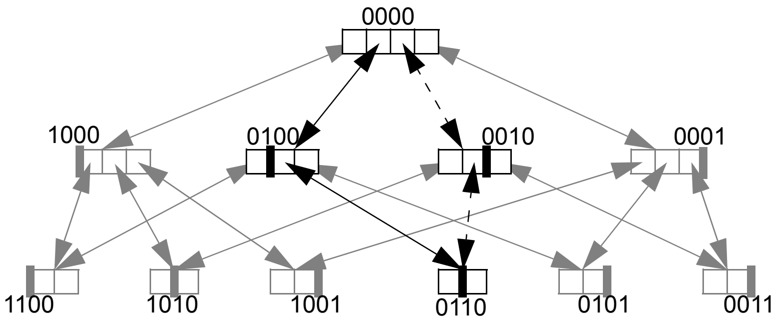


Figure 6.4: A signature tree after the signature "0110" has been inserted

In Figure 6.4 the two first levels of a 4-bit signature ST are shown after a segment with the signature "0110" has been inserted. The pointer to the segment has been stored in the node labelled "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 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 non-full signature (a signature that contains fewer than F_g 1's) 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 the nodes.

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 increases that nodes and arcs that have to be created already exist, 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 to populate the ST with as few nodes as possible, without sacrificing any functionality.

6.5 Searching the Signature Tree

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 simply 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 the ST, the query signature would have to be scanned, from right to left, and each time a bit that is set to one is encountered we simply move the current node pointer to the child node corresponding to the position in which the set bit was found. Once all bits in the query signature have been scanned, the result of the search will be all nodes stored below the node pointed to by the current node.

To search for simple queries (queries that only consist of one signature) it would be possible to perform very fast searches, see Figure 6.5. 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 an ST, see Equation 6.6.

```
STNode searchST(ST st, Signature Query){
  boolean aborted = false;
  STNode currentNode = st.getRoot;
  if( currentNode.NumberOfChildren == 0 ){
    aborted = true;
  }
  for(int i=0; i<Query.NumberOfBits && !aborted; i++){
    if( Query.getBitValue(i) == 1 ){
      currentNode = currentNode.getChild(i);
      if( currentNode == NoNode ||
          currentNode.NumberOfChildren == 0 ){
        aborted = true;
      }
    }
  }
  if( aborted ){
    return NoNode;
  } else {
    return currentNode;
  }
}
```

Figure 6.5: ST search algorithm

$$C = 19 \cdot 10^3 \approx 2 \cdot 10^4 \tag{6.6}$$

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 5.5.1, “Cost of Scanning the Data,” on page 71.

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 5.11, on page 68 and Equation 5.12, on page 68. As we calculate in Section 7.5, “Test System,” on page 100, 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 on 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 in which the search terminated.

The search algorithm will have to be expanded since the algorithm described here can only 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.

6.6 Test Results

We have tested a small ST based on 20-bit signatures with a data file consisting of $3 \cdot 10^5$ elements, and for simple searches the search time equals the depth of the structure, as predicted. Unfortunately, our tests also confirm that this structure is very large.

6.6.1 The Signature Tree Size

In this section we will take a closer look at the measurements we have obtained from our ST implementation.

As we can see in Figure 6.7, the ST will grow logarithmically. We can also see that the size ratio between the original data file and the ST index is very bad. If we assume that each signature encodes 4 windows, that corresponds to approximately 4 characters in the data file. So a file with 2,000 signatures corresponds to approximately 80,000 values. If each value is a 4-byte value, then the original data file would consist of 32,000 bytes. We can see from Figure 6.7 on page 93 that the ST will contain almost 35,000 nodes. Since each node is approximately 30 bytes, the ST will be almost 1,050,000 bytes, i.e. more than 30 times larger than the original data file.

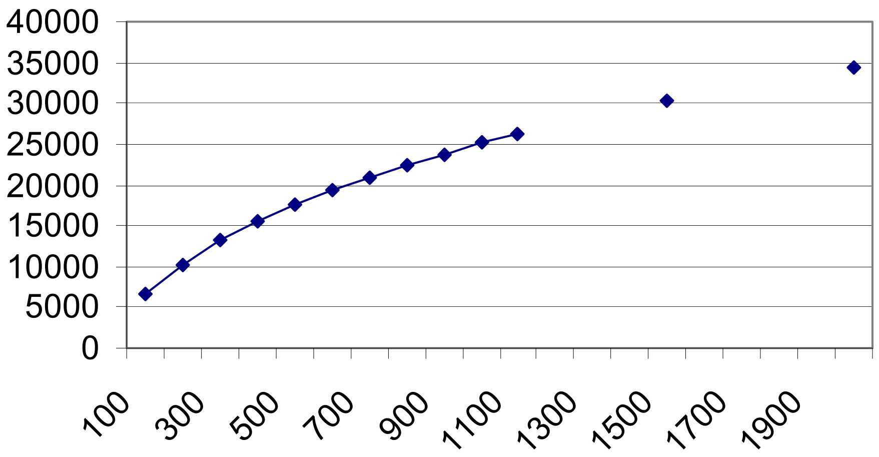


Figure 6.7: Size of the ST

Figure 6.7 shows how the ST grows when data is inserted into the structure. The vertical axis shows how many nodes the ST contains, and the horizontal axis shows how many signatures have been indexed. The distribution of nodes in the ST can be seen in Table 6.3. The values are obtained from tests done with our ST implementation.

Table 6.3: Size of different levels of the ST

Number of Signatures	**Level 1**	**Level 2**	**Level 3**	**Level 4**	**Level 5**	**Level 6**	**Level 7**
Max.	20	190	1140	4845			
100	20	190	1076	2521	1992	702	101
200	20	190	1138	3710	3665	1372	200
300	20	190	1140	4323	5190	2051	300
400	20	190	1140	4594	6503	2714	400
500	20	190	1140	4723	7626	3347	499
1000	20	190	1140	4843	11507	6420	997
1500	20	190	1140	4845	13474	9210	1486
2000	20	190	1140	4845	14489	11794	1981

6.7 Conclusions

A major problem with the signature file is that it has to be scanned sequentially. With the ST this is no longer necessary, and it will be possible to search the signature file efficiently in constant time. 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 the ST will not be a good solution because of the massive overhead.

It might be possible to make a smarter implementation of the ST that solves some of the problems we have encountered, especially with more complex queries. But if the results obtained with the ST are compared to the results obtained using a B-tree index structure, see Section 7, “The B-Tree Structure,” on page 95, we feel that unless we come up with a drastically new approach the ST can not compete with the B-tree structure, neither cost-wise nor size-wise. Our conclusion is that further work on this structure is not meaningful.

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 so that actually fetching the data, once it is located through the index structure, does not become as expensive, or even more expensive than searching the index structure in the first place.

Chapter 7

The B-Tree Structure

This chapter will describe how we have used the B-tree structure for indexing shapes of time series.

We will start by giving a short introduction to the index structure and then a theoretical description of the performance of the B-tree. We will then describe how we have used it to index time series, and finally we will present the results obtained when we used the B-tree to index time series. A short introduction to the B-tree can also be found in Section D.5.2, “B-Trees,” on page 198.

7.1 Inverted Files

Inverted files are a way of indexing large files. As with several other indexes, the idea is to reduce the number of records that have to be fetched from disc [FOX92]. For a description of inverted files see Appendix D.6, “Inverted Files,” on page 200.

7.2 Adapting Inverted Files for Time Series

Our system uses the following procedure to index the time series. First we use our chosen feature extraction, see Section 3.1, “Feature Extraction Using Time-Derivative,” on page 25 or Section 3.2, “Feature Extraction Using Curvature,” on page 29, to encode the time series into a text sequence. We then slide a window along the sequence. The first few characters of the window contents are used as a key value for the sequence – since we use very small window sizes the entire window contents have been used – and then a pointer back to the sequence is inserted into the inverted file.

Note that the sliding window is only moved one step each time.

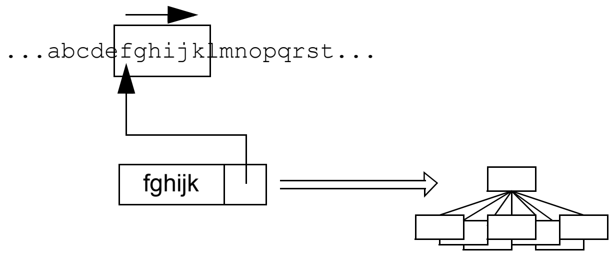


Figure 7.1: System using an inverted file index

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 would be to make several inverted file searches.

The pseudocode for the algorithm can be found in Figure 7.2.

```
ResultList searchBtree(BTreeNode root, String Query){
  ResultList resultPointers = EmptyResultList;
  boolean queryProcessed = false;
  boolean firstTime = true;
  boolean queryNotFound = false;
  int startPos = 0;
  while( !queryProcessed && !queryNotFound ){
    String subQuery = Query.substring(startPos, KEYLENGTH);
    if( subQuery.size < KEYLENGTH ){
      queryProcessed = true;
    } else {
      ResultList subResult = root.BTFind( subQuery );
      if( firstTime ) {
        resultPointers = subResult;
        firstTime = false;
      } else {
        for(int i=0; i<resultPointers.size; i++){
          int modPos = (resultPointers[i].blockPos)+startPos;
          if(!subResult.ListFind( modPos )){
            resultPointers[i].delete;
          }
        }
      }
      if( resultPointers == EmptyResultList ){
        queryNotFound = true;
      }
      startPos = StartPos+1;
    }
  }
  if( queryProcessed ){
    return resultPointers;
  } else {
    return EmptyResultList;
  }
}
```

Figure 7.2: B-tree intersection search algorithm

7.3 Introduction

The B-tree was first proposed in [BAYER72]. A B-tree is an index developed to be very efficient when used on data stored on secondary storage. Its usefulness 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 D.10 on page 199). 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 are 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 on. 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. There exists a variation of the B-tree, the AP-tree (append only B-tree), where inserts are done only at the end and therefore all nodes can be completely filled and no splits occur. For uniformly distributed data, extendible and linear hashing [FAGIN79, LITW80] will outperform the B-tree on the average for exact match queries, insertions, and deletions.

The biggest benefit of the B-tree is that it offers a very good pruning of the search, since for each step down, the B-tree discards a great number of elements in the indexed file.

7.4 Performance

This section will give a theoretical description of the B-tree performance for the basic operations: search, insert, and delete. The analysis of the B-tree performance are from [KNUT98].

7.4.1 Searching

An upper bound of the time it takes to search the B-tree can be said to be directly proportional to the number of disk pages that we have to fetch in order to find the element we are looking for.

Suppose that the B-tree is of order m, that there are N keys and that the $N+1$ leaves appear on level l. Then the number of nodes on levels 1, 2, 3, ... is at least 2, $2 \cdot \left\lceil \frac{m}{2} \right\rceil$, $2 \cdot \left\lceil \frac{m}{2} \right\rceil^2$, ... ; hence we can construct Equation 7.3.

$$N + 1 \geq 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{l-1} \tag{7.3}$$

Or if we reshuffle Equation 7.3:

$$l \leq 1 + LOG_{\left\lceil \frac{m}{2} \right\rceil}\left(\frac{N+1}{2}\right) \tag{7.4}$$

Since we need to access at most l nodes during a search, Equation 7.4 guarantees that the running time will be quite small even for large values of N as long as m does not become too small.

7.4.2 Insertion

When a new key is inserted into the B-tree we will first have to search the tree for the key, and we might have to split as many as l nodes. The average number of nodes that needs to be split is much less, since the total number of splittings that occur while the entire tree is being constructed is just the total number of internal nodes in the finished tree minus l. If there are p internal nodes, there are at least $1 + \left(\left\lceil \frac{m}{2} \right\rceil - 1\right) \cdot (p-1)$ keys; or if we move p to the left hand side of the equation, Equation 7.5.

$$p \leq 1 + \frac{N-1}{\left\lceil \frac{m}{2} \right\rceil - 1} \tag{7.5}$$

From this we see that the average number of splits we have to make while inserting N keys into the tree will be given by Equation 7.6.

$$\frac{splits}{insertion} = \frac{1}{\left\lceil \frac{m}{2} \right\rceil - 1} \tag{7.6}$$

7.4.3 Deletion

Since time series are append-only, we will never remove data from the index, therefore we will not look into the problems of deleting data from the B-tree. Neither will we discuss the performance of deletions in a B-tree. For a full discussion on this topic the reader is referred to e.g. [KNUT98] that contains a detailed description of B-trees.

7.5 Test System

We have used the method proposed in Section 7.2, “Adapting Inverted Files for Time Series,” on page 95 for adapting the B-tree structure for time series indexing.

We use the same data in this experiment as we did while we were discussing the ST (see Section 6 on page 85). Using Equation 5.11, on page 68 and Equation 5.12, on page 68, we can give the 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 the calculations, we can assume we have approximately 10^{16} elements in the original data file.

When the amount of data indexed becomes so great, it is not possible to use standard 4-byte pointers, so in the following equations we will use both 4- and 8-byte pointers, depending on how large the original data file is. For a certain region of data file sizes it is also possible to use a 4-byte pointers for the data and 8-byte pointers for the B-tree since the size of the B-tree grows faster than the size of the data file. Below is a discussion about the size of the B-tree, and for which data sizes we can use 4-byte pointers and when we have to use 8-byte pointers.

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 Equation 7.7. In Table 7.1 on page 100 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.

$$K_p = \left\lfloor \frac{PageSize}{KeyLength + 2 \cdot p} \right\rfloor \tag{7.7}$$

Table 7.1: Number of keys per secondary storage page for 4 different page sizes with a pointer size of 8 bytes

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

Table 7.1: Number of keys per secondary storage page for 4 different page sizes with a pointer size of 8 bytes

Key Length	**4kb**	**8kb**	**16kb**	**32kb**
8	170	341	682	1365
9	163	327	655	1310
10	157	315	630	1260

Table 7.2: Number of keys per secondary storage page for 4 different page sizes with a pointer size of 4 bytes

Key Length	**4kb**	**8kb**	**16kb**	**32kb**
4	341	682	1365	2730
5	315	630	1260	2520
6	292	585	1170	2340
7	273	546	1092	2184
8	256	512	1024	2048
9	240	481	963	1927
10	227	455	910	1820

Since the B-tree guarantees that no node will be less than half full, we can calculate a maximum B-tree size for a given data file size. If we assume that all nodes in the B-tree will be half full, we can, using Equation 7.8, calculate the upper bound of the number of disk pages we need to index a given file. Using the same equation we can also calculate the lower bound of the B-tree size by assuming that all nodes are full.

$$P_N = \left\lceil \frac{n - w_s + 1}{K_p \cdot \overline{P_u}} \right\rceil \approx \left\lceil \frac{n}{K_p \cdot \overline{P_u}} \right\rceil \tag{7.8}$$

In Equation 7.8, P_N is the number of disk pages needed to index the file, $\overline{P_u}$ is how full an average node in the B-tree is. The last simplification of the equation can be made since $n \gg w_s$. By multiplying P_N with the page size, the size of the B-tree will be obtained, Equation 7.9.

$$S \approx \left\lceil \frac{n}{K_p \cdot \overline{P_u}} \right\rceil \cdot P_s \tag{7.9}$$

From Equation 7.9, Equation 7.10 can be derived, and from this we can calculate at what data file size we have to move from 4-byte pointers to 8-byte pointers for the B-tree representation.

$$n \le \frac{S \cdot K_p \cdot \overline{P_u}}{P_s} \tag{7.10}$$

If we set $S = 4,3 \cdot 10^9$, $\overline{P_u} = 0,5$, $K_p = 1000$ and $P_s = 16 \cdot 10^3$, Equation 7.10 tells us that the number of elements in the data file has to be less than $13 \cdot 10^7$. So, if the data file is smaller than $13 \cdot 10^7$ elements, we can use 4-byte pointers for both the data file and the B-tree. If the data file contains between $13 \cdot 10^7$ elements and $43 \cdot 10^8$ elements, we have to use 8-byte pointers for the B-tree, but we can still use 4-byte pointers for the data. If we have more than $43 \cdot 10^8$ elements in the data file, we have to use 8-byte pointers for both the B-tree and the data file. Since the region where we can use mixed pointer sizes is small compared to the file size, we assume that both pointer sizes are the same.

In Figure 7.11, the upper and lower bounds of a B-tree have been plotted. Figure 7.12 shows the maximum B-tree size for a given data size. The plot is a log-log-plot, so both axes shows logaritmic values.

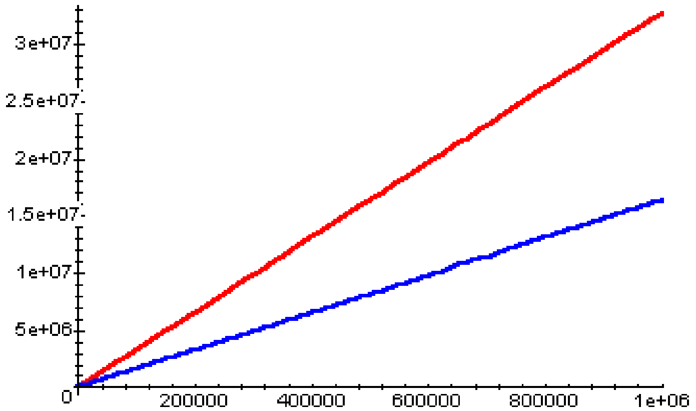


Figure 7.11: Upper and lower bound of the B-tree size

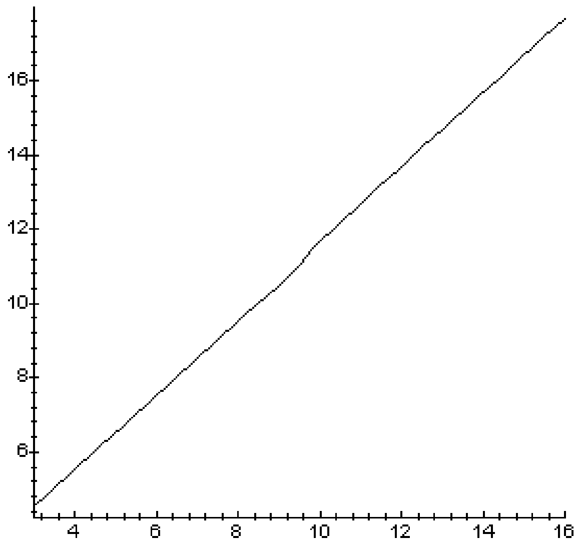


Figure 7.12: B-tree size relative to data file size, log-log-plot

The average number of page accesses, P_a, needed to search the B-tree can be calculated with Equation 7.13. n is 10^{16} elements.

$$P_a = \left\lfloor \frac{\ln n}{\ln K_p \cdot 0,63} \right\rfloor \tag{7.13}$$

The cost of searching the B-tree is approximately equal to the number of pages that need to be retrieved from secondary storage. The cost should only be thought of as a *very* approximate time estimate, and we have to add the cost of retrieving the data to the cost of scanning the index, see Section 5.5, “Searching the Signature File,” on page 71.

Table 7.3: Cost of searching the B-tree with a pointer size of 8 bytes

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

Table 7.3: Cost of searching the B-tree with a pointer size of 8 bytes

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

In Table 7.3 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 because 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 7.4: Cost of searching a smaller B-tree with a pointer size of 4 bytes

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

Using Equation 7.7, on page 100, and Equation 7.13, on page 103, we can calculate the cost of searching the B-tree index if the B-tree only contains 10^8 elements and if we use 4-byte pointers. In Table 7.4 we can see that the B-tree is very fast for this smaller sequence. We can also see, if we compare Table 7.3 and Table 7.4, that the B-tree scales very nicely.

If we compare these results with the cost of searching the signature file, the cost of searching the signature file is higher. The cost of searching the signature file is, with this cost model, approximately 10^{15}, so the B-tree search is much faster. The cost of an ST search would be approximately 10^4 (see Equation 6.6, on page 91), 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 only intended to provide the reader with some idea of what could be expected in future tests.

7.6 Search Algorithm

A simple way to search the B-tree is to take the first characters from the query so that we get a search key, and then use the B-tree to find all occurrences of that key. This search will result in a list of pointers to all occurrences of strings starting with these characters. This might be a bad solution if the query is longer than the key pattern, since then we will not take advantage of the extra information we have in the longer query. We might also end up with a very long list of occurrences that we have to scan, and since these can be evenly distributed in the file the cost of this search can become very high.

A better way of doing this is to take advantage of the longer query and do several searches in the B-tree. If we use the concept of a sliding window and several B-tree searches, we might get significantly better performance. The window length should be the same as the key length used in the B-tree, and we treat the contents of the window as our query pattern. We could also have used a partial key strategy like the one suggested in [BOHA01].

We run the first query on the B-tree and examine the resulting list of pointers. If the number of pointers is too great, i.e. the cost of accessing the actual data to find the places that match the entire query will be to great, we advance the window one position and run it through the B-tree. Since we know that the second query has to be found adjacent to the first query, we can now create an intersection of the two result lists, and the resulting list will only contain those pointers from the first whose successors could be found in the second list. This procedure will then be repeated until either we can not create any more queries from the original query pattern, or the number of remaining pointers in the result list is cheaper to scan than another (average) search through the B-tree.

The cost of an average search can be calculated with Equation 7.13, on page 103.

Using this search algorithm will be more costly than the simple B-tree search. However, since we do not have to fetch any data from the data file during the search, and the disjunction on the lists can be done directly with data found within the index, this method will be very fast.

7.7 Test Results

See Appendix D, "Indexing," on page 189 for the test results.

7.8 Conclusions

We have shown that it is possible to achieve very good search performance for time series using B-trees. The size of the B-tree is larger than the original data and the signature file, but the search cost is far lower compared to the search cost we can achieve using either the original, non-indexed data, or signature files (see Section 5.5, "Searching the Signature File," on page 71). We can also see that the B-tree is much faster than the ST and, by far, much smaller than the ST (see Section 6.3, "The Size of the ST," on page 87 and Section 6.5, "Searching the Signature Tree," on page 90).

Initially we believed that the B-tree structure would be unsuitable for very large time series, mainly because we could not find any references to related work using B-trees in this way. But as we have shown here, even very large time series can be efficiently indexed using B-trees.

Another fact that makes the B-tree a very good choice for indexing time series is that it is a very well known index that is available both as modules for program development, and as a component of virtually all database engines on the market today. This means that a solution using B-trees might easily be integrated into an existing database system without the need for adding additional index structures. There have also been several years of research on B-trees and how to efficiently search and implement them. All this together makes a strong case for using B-trees to index time series.

As a final reflection on our work with the B-tree structure we would like to point out that since we have not modified the B-tree in any way, and since, in our current system, we only search for exact matches in the index, we would probably get even better results if we used a linear hashing index instead.

Chapter 8

The PTrie Structure

In this chapter we will introduce a new index structure we have constructed, the paged trie, PTrie, structure.

We will start by introducing the problems we tried to address with this structure and the ideas underlying the structure. We will then describe how the structure is created and how the basic operations search, insert and delete are implemented. The chapter then concludes with a performance analysis and conclusions.

8.1 Introduction

In most indexes we have encountered, the time series are divided into smaller atomic entities and it is those entities that are stored in the index structure. When the index is scanned for a sequence that is larger than these atomic entities, the query has to be broken down into these entities and then each entity generates a search through the index. Each search generates a set of results. If any of the sets are empty we can immediately say that the sub-sequence we were looking for is not present. Otherwise we have to scan all the sets trying to combine them into a continuous result.

We have tried to find an index structure that would allow us to treat the time series as a continuous data set and would eliminate the last step of trying to combine a set of sub-query results into the result we queried for. This structure would also only require one search through the index.

Our solution is to combine Tries [FRED60] with a method of encoding the time series into a text sequence. The process of creating the text sequences is described in Section 3.1, “Feature Extraction Using Time-Derivative,” on page 25 and in

Section 3.2, “Feature Extraction Using Curvature,” on page 29. Feature extraction using curvature employs a concept of a sliding window we have used in previous work [ANDR97, ANDR99]. We have combined this with a page concept, to get an access structure specifically tailored for quick searches of sub-sequences in a larger time series or a collection of time series.

8.2 Idea

In our previous work [ANDR97, ANDR99] we have tried to simplify the problem of efficiently searching the time series into a more traditional text searching problem by transforming the time series into a text string. To transform the time series we use a pre-processor that calculates the curvature of the time series in each point, and maps it to a token from an alphabet. We use an alphabet with 24 tokens.

In previous and related work [FALO94, AGRA95, LI96] the time series are, after pre-processing, broken up into small atomic units using a small window that slides along the time series. When the system is queried, the query is broken down into the same atomic entities. In the index each unit is queried, resulting in a group of occurrences. If any of the groups are empty we can immediately say that the entire query pattern can not be found. Otherwise we have to scan all the groups to see if we can find any results that contain one segment from each group, in the correct order, in which segments are adjacent to each other.

Our approach removes the last step in the process since we do not break up the time series, but treat it as a continuous set of data.

We do this by combining the Trie structure with ideas from the B-tree, and at the same time we use the idea of a sliding window, now slightly modified, from our previous work.

First we introduce the sliding window. We create a sliding window with the same size as the entire time series string. As the window slides along the time series string, one position in each step, we decrement the window size by one each step. That way the last token in the window will always be the last token in the time series string.

Second, we introduce pages to a Trie. A Trie is a well known structure for storing words. If we treat the window contents as words and use a regular Trie to store them, we will get a very deep Trie since the maximum depth of a Trie equals the length of the longest word stored in it. In our case this will result in a Trie with the same height as we have tokens in the time series string. But if we introduce pages into the Trie and do not expand a node until the page is full, we will avoid that problem. We call this extended Trie a *Paged Trie* (PTrie).

We create the structure as follows:

- Each node contains a data page and an array of pointers to other nodes.
- The size of the array is the same as the number of symbols in the alphabet we use to encode the time series into a string.
- An entry into the data page is just a pointer into the time series string. The pointers in the array are arranged according to the alphabet used.
- When a page is full we scan all entries on the page, and by following the pointers back into the original time series string, we can find the next character from the "word" and use that to propagate the entry down into the children.

8.3 Inserting Data into the PTrie

In the following example we will assume that we have a small time series encoded as a string. In this example we have used a very small alphabet that consists of the four tokens 'a', 'b', 'c' and 'd'. We also assume that the page size is just eight bytes. In our example, each node has to have four one-byte pointers, since the alphabet contains 4 tokens. That leaves us with a data page size of four bytes.

When we create the empty PTrie it contains just a single node. The data page is empty, and all four pointers are blank, see Figure 8.1. If we want to index the string shown in Figure 8.2 we start by creating a window that contains all characters in the string. If we use the offset of each character in the string as a pointer, we only need to store the position of the first character in the window in the PTrie, since we know that the last character is the last character in the string.

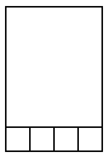


Figure 8.1: The empty PTrie

"aabdacadad"

Figure 8.2: An example string

We try to insert the first offset into the page, and since the data page is empty that is not a problem. We continue to slide the beginning of the window along the string, and for each position we only enter the offset of the window start position into the data page. Figure 8.3 shows the PTrie after the third element has been added. Note that it is only offsets, or pointers into the original data, that are entered into the PTrie. This process continues until we reach the fifth token of the string. When we try to insert it into the node, we discover that the data page is full. We have to split the node.

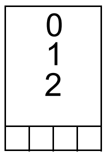


Figure 8.3: The PTrie with 3 elements

Now we scan all entries on the data page. The new pages we create will be placed on the first level of the PTrie (the root is at level 0). So now we follow each pointer back to the string and look at the first character of the substring. The first entry is 0 and the first character of that substring is ‘a’, so that substring should be moved to a new page linked to the first node pointer, since ‘a’ is the first symbol of the alphabet, in the root node. We create a new node, link it to the root node, and then store the offset pointer, 0, on the new data page (see Figure 8.4). The next entry in the root node will also be placed on the new page since the first character in that string is also ‘a’. The next entry will generate a new node, linked to the second node pointer in the root node, and finally the last entry on the page will generate a third new node linked to the fourth node pointer. Finally, we try to insert the string starting with the fifth token again, but since the top node is now split, and there are enough remaining tokens in the string, we directly follow the first pointer in the root node and insert 4 directly into the child node. In Figure 8.4 the PTrie is shown before and after the split is triggered

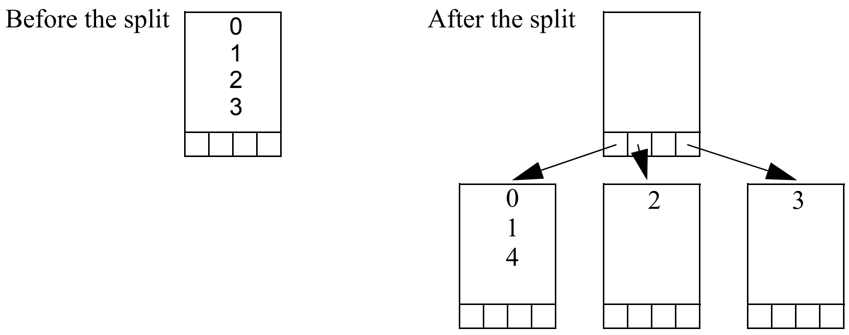


Figure 8.4: PTrie before and after the split

Figure 8.5 shows what the PTrie looks like when the entire test string is indexed.

If we now want to query the PTrie for all occurrences of “ad” in the string, we scan the query string and traverse the PTrie. We start in the root node, and since the query string starts with the token ‘a’, we move to the first child of the root node. The next token in the query string is ‘d’, so we move to the fourth child of the node. All pointers stored in this node and all pointers stored in nodes below this node are returned as the answer to our query. A search for “bd” would terminate in the second child node of the root instead, and all entries in that node would have to be scanned. In this case, the node only contains one element, so only one element would have to be checked.

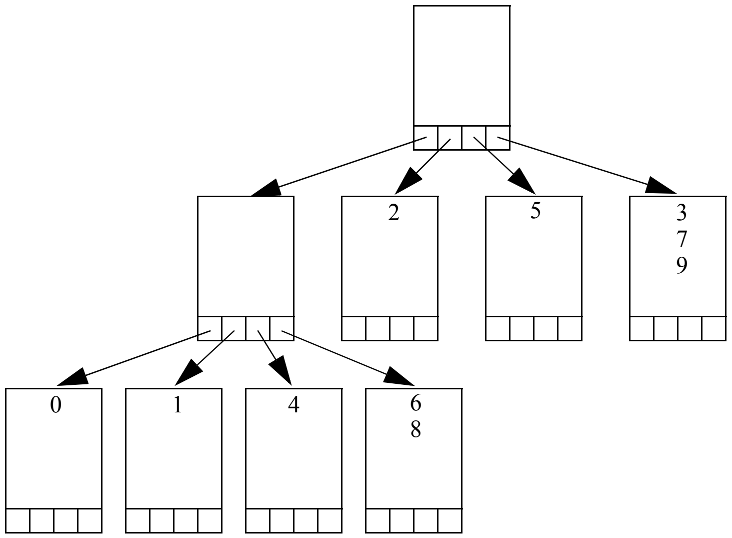


Figure 8.5: The PTrie when the entire string is indexed

The PTrie shown here can be used to index a collection of time series since each node contains a data page. If we intend to store only one sequence in the PTrie, all internal nodes only need to keep a data page that can store one pointer, since the only pointers that will not be stored in the leaf nodes are the pointers that point to the end of the sequence.

8.4 Performance

Since the PTrie is an unbalanced structure, the time for finding data will depend on what the data looks like. We have tested the PTrie on two different time series and with two different page sizes.

The time series we have tried indexing using a PTrie is an artificially generated time series consisting of 400,000 values, and a time series consisting of measured temperatures with approximately 47,000 values. The page sizes we have used are 4kb and 8kb.

With a page size of 4kb, the average depth of the PTrie was 4.5 and the maximum depth was 7. With a page size of 8kb, the average depth was 3.2 and the maximum depth was 5.

The nature of the PTrie makes the structure unsuitable for storage on secondary memory. The reason for this is that most nodes in the structure will be very small. If the PTrie is used to index one large sequence, all data except the end of the sequence will be stored in the leaf nodes, leaving only an array of pointers in the internal nodes. Since the size of the array is limited to the size of the alphabet, it will be a waste of space to save each internal node in one disk page. We can save several nodes in each disk page, but the structure of the PTrie will still, in the worst case, require as many disk accesses as the maximum height of the PTrie.

For main memory searches the PTrie might still be a good alternative, since the nodes can easily be compressed, the structure can be efficiently traversed as long as the entire structure fits in main memory, and the structure offers nice search possibilities.

8.4.1 Insertion

One problem with the PTrie is the performance of insertion. As the structure is built, the data initially stored at a higher level in the PTrie will be pushed down further in the structure. Since only references to the data are stored in the PTrie, we have to access the original data once for every entry on the PTrie page.

A worst case scenario is a sequence consisting of only one character type. When the first page is full, the next insertion would trigger a sequence of splits that would not stop until the depth of the PTrie was approximately the length of the sequence.

8.4.2 Search

The structure is very simple and fast to search, and it is quite simple to use a certain fuzziness in the queries, e.g. “the third character in the query might be an ‘a’, a ‘b’ or a ‘c’”.

For each step we go down into the structure, we reduce the number of possible answers by a factor directly proportional to the size of the alphabet. But we will almost never reach a single value in a search, we will always have to scan one or more pages before the result can be returned.

There are two cases possible: the query sequence is longer than or equal to the depth of the PTrie, or the query sequence is shorter than the depth of the PTrie.

In the first case, we traverse the PTrie until we reach a node that has not been split, and then we have to compare each entry on the page with the query sequence to see if any of them qualifies as a result to the query.

In the second case, the search will terminate at an internal node of the PTrie, and all entries on the current page and all pages below the current page contain valid results to the query.

A worst case search scenario would be to retrieve all sequences. If the PTrie was evenly populated (all branches of maximum length) the search time would be proportional to the height of the PTrie multiplied by the number of characters in the alphabet.

The best case scenario would be a search that terminated in a leaf node with only one entry. Since the structure is not balanced, that could mean that we only have to access one node other than the root node.

This leads us to the conclusion that the search time will be somewhere between 1 and the size of the original data file, minus the number of elements on a page, see EQ 8.6.

$$1 \le C \le n - P_s \tag{8.6}$$

In EQ 8.6, C is the cost, n is the number of elements in the data file and P_s is the maximum size of a page.

8.5 Conclusions

To our knowledge nobody has suggested this structure before. Even though we do not achieve the same performance using this structure as we could get using a BTree, the simplicity and the fact that we do not have to break up the time series and then try to stitch the result back together makes this approach very attractive.

Since the performance of this structure depends on the data, it is difficult to find a good general theoretical model of how it will behave. One big problem with this structure is that it is not possible to build it in such a way that it accesses disk blocks as efficiently as the B-tree. This makes the PTrie unsuitable for indexes on a secondary block based storage media like disks.

As we can see in the cost diagrams, see Appendix C, “PTrie Test Results,” on page 185, the cost of searching the index is good, but since we have to access the original data file for a potentially large number of occurrences, our conclusion has to be that this index structure is not very well suited for disk-based storage.

In a main memory application where the cost of accessing the original data is not so great, this index might still offer very good performance, especially since it is very easy to implement “fuzziness” into the search while traversing the PTrie.

Chapter 9

Comparison Between Tested Methods

The chapter begins with a comparison between inverted files and signature files while indexing text files. We will then continue with a comparison of the different methods we have presented in this thesis.

9.1 Introduction

Much of the discussion in this section is from [ZOBE98]. The comparison in this paper is very thorough and we only present a small part of their comparison that we find relevant for our work. In their paper they e.g. examine several different query types, but since we have only examined one type of queries in our system, the disjunctive queries, we only mention these here. For the full comparison the reader is referred to the referenced paper.

For textual data the reported index sizes are from 5% to 10% of the indexed data [BELL93]. These figures include storage of in-document frequencies used for processing ranked queries and indexing of all terms, including stop-words and numbers. They do not allow for the space wastage that a B^+-tree implementation would imply.

With a dynamic collection the upper limit of the index size is probably closer to 15% of the indexed data size. Dynamic indexes are, however, subject to the same problems as dynamic inverted file indexes, and slices must be over-allocated to allow for growth. Unless slices are stored as linked lists of blocks on disk there is extensive copying of the index, thus a 30% to 40% average space overhead must be accepted, taking the index size to around 30% to 55% of the text.

According to [ZOBE98] inverted files have often been judged based on two papers [HASK81] and [CARD75].

[ZOBE98] state that these papers no longer reflect the capabilities of inverted files and therefore no longer should be used as a basis of comparison, instead they have found that for text indexing inverted files are superior to signature files in almost every aspect, including speed, space and functionality.

According to [ZOBE98] the following widely-held beliefs are either fallacious, or incorrect once compression of index entries are taken into account:

- The assumption that sorting of inverted lists during query evaluation is an unacceptable cost (illusory, since the inverted lists should always be kept sorted).
- The assumption that a random disk access will be required for each record identifier for each term, as if inverted lists were stored as a linked list on disk (they should be stored contiguously or at least in a linked list of blocks).
- The assumption that if the vocabulary is stored on disk, log N accesses are required to fetch an inverted list where N is variously the number of documents in the collection or the number of distinct terms in the collection (only true if none of the nodes in the tree storing the vocabulary can be buffered in memory).
- The assertion that inverted files are more expensive to create than signature files (previous work in [MOFF92] and [MOFF95] has shown this to be fallacious).
- The assertion that inverted files are large, an oft-repeated claim being that they occupy between 50 and 300% of the space of the text they index [HASK81] (with current techniques inverted files are stored in around 10% of the space of the text they index [WITT94]).

Inverted file index has two main parts, the vocabulary, containing all distinct values being indexed, and for each distinct value, the inverted list storing the identifiers or the records containing the value.

Assume that we have a system that uses a B^+-tree for storing the vocabulary, and that the leaves in the B^+-tree contain pointers to inverted lists. If the vocabulary contains 1,000,000 distinct 12-byte terms and the disk uses 8-kb blocks and all pointers are 4 bytes we could fit all internal B^+-tree nodes in at most 64 kb. This is so little memory that all internal nodes can be assumed to fit in main memory. Then we would only need one disk access to retrieve the correct vocabulary entry, and then a second disk access to fetch the correct inverted list.

The reason that signature file systems fail to outperform inverted file systems is also that the result returned from the index search has to be checked for false matches or false drops, whereas in the case of inverted files, the result returned from the index search is the finished result.

Another reason inverted files can outperform signature files is that today's systems have more memory. And as [ZOBE98] states, it would be rare for a machine handling a 1 Gb database to be unable to hold 10 Mb in its memory, and in most cases less than 1 Mb is necessary to keep most of the search structures in memory so that one or two disk accesses are necessary to fetch a result.

9.2 Indexing Structures

In this section we compare the four different structures we have examined in this thesis. Two of the structures are well-known existing indexing structures that we have adapted for use with time series, the signature files and the B-trees, and two of them are structures we have developed during our work, the signature tree and the PTrie.

Note that these results are also valid for a main memory system due to modern computer hardware architecture. A modern computer has a small amount of very fast memory, currently between 128Kb and 256Kb, and then a larger amount of slower memory. The speed difference between the two different memory types is not as big as the speed difference between main memory and secondary storage, but can still be significant [RONS98].

9.2.1 Signature Files

We have looked at signature files and shown that it is possible to use them for shape queries on time series and to obtain a very low false drop rate, so low that it is not necessary to scan the results to retrieve the true results. Text ratio between text signature files and the original text data files is usually very low, under 10%; when we use signature files to index time series it is not possible to achieve such a low size ratio. Our tests indicate that the size ratio is around 50% (uncompressed signature files). Furthermore, the signature files have to be scanned sequentially, so the only speed advantage we get compared to the actual time series comes from the fact that the signature file is smaller and therefore can be scanned using fewer disk accesses, i.e. the signature file can be searched twice as fast as the original data if we only take the time it takes to scan the index structure into account. A big drawback with signature files is that they scale in linear time and this makes them unsuitable for very large time sequences. For smaller time sequences where the extra overhead and complexity of e.g. a B-tree cannot be justified, the signature files offer a good, simple solution that will always offer better performance than a straightforward scan of the original data.

9.2.2 The Signature Tree

To make the signature search time scale better, we tried to construct a tree-like structure that would allow us to search the signature file faster. We came up with the signature tree, ST. It stores the contents of the signature file in a lattice and makes the search time of the signature file dependent on the size of the signatures instead of the size of the data set. The ST allows us to perform very efficient searches of the signature file. However, the ST introduces enormous overhead, and even for very large data sets the B-tree performs significantly better. This has made us abandon the ST, as we do not think the ST is a viable solution.

9.2.3 The PTrie

Another structure we have investigated is the Paged Trie, PTrie. To our knowledge this structure has not been investigated before. The PTrie offers very simple and powerful searching capabilities. For very large time series, where the index is stored on disk, it suffers from the problem that it is difficult to construct the PTrie in such a way that we minimize the number of necessary disk accesses. For main memory applications the PTrie might be an interesting index structure, but when the time series become so large that the index structures have to be stored on disk, this index becomes unsuitable.

9.2.4 The B-Tree

Finally, we investigated the B-tree structure. The B-tree structure is well-researched and is known to offer exceptionally good performance for text files. We have described a method that allows us to use a B-tree to index time series. The B-tree scales very nicely, both from a search cost and from a size perspective, and offers superior search performance even for very large time series compared to the other methods we have examined.

9.2.5 Conclusion

The least cost-effective way of searching the data is, of course, to use no index structure at all. We can then use an efficient search strategy instead of an index structure like the Boyer-Moore algorithm [BOYE77]. Note that even if the data fit in main memory we still benefit from a paged-based index structure.

We can, however, improve the search speed dramatically by using an index structure. If the amount of data is small and we do not wish to introduce the added complexity of a more advanced index structure we suggest that a signature file be used. It is possible to construct a signature file system that produces very few false drops, the overhead is very small, and it will always be faster to search than the original data due to the fact that it is smaller than the original data file.

Since the search time for signature files, as for normal sequential scanning, grows linearly with the size of the indexed data set and the search time for tree structures grows logarithmically with the size of the indexed data there will always be a break-even point with regard to the data size. Once that data set size is passed, tree structures will always be faster to search.

For smaller data sets the simpler methods might still be advantageous to use since they are often less complex to implement.

When the time to scan the signature file becomes larger than the time to fetch two pages from secondary storage it will be faster to use a B-tree instead of a signature file. How large a signature file is that? That depends on the block size but we always need to search for the first block, then if the signature file is stored continuously we will be able to read about two more blocks in the same time it takes to load the appropriate block.

The break-even point between signature file performance and the B-tree is therefore about 3 disk blocks. To calculate the break even size expressed in number of values in the original time series we use Equation 5.13, on page 69. Table 9.1 shows break-even size of the time series between a 512 bit signature file and B-trees for different page sizes.

Table 9.1: Signature file and B-tree break-even size

	4kb	**8kb**	**16kb**	**32kb**
Number of values	8138	16275	32551	65101

If our system uses 32kb pages a time series consisting of more than 65101 elements will always be faster to search using a B-tree than a signature file.

If we use compressed signature files the break-even point will increase, but since the signature file search time grows linearly with the dataset and the B-tree search time grows logarithmically, at some point the B-tree will always become superior to the signature file.

It is somewhat interesting to see that B-trees are more efficient for such small amounts of data.

For larger time series our work shows that the B-tree is far superior to the signature files. Using the techniques we have presented in this thesis it would probably be possible to use other inverted file indices as well, but that has been outside the scope of this work to examine. We have also shown that B-trees continue to be a good index for time series even for very large time series.

Chapter 10

Applicability on a Control Problem

In this chapter we describe the preliminary investigation of a problem that could benefit from our work. The problem comes from the automation and control area.

We will give a short introduction to the problem, and then describe how the system might benefit from a database with a time series index. Finally we will present the problems we have identified from a database technology/data indexing point of view.

10.1 Introduction

During the course of our work with time series indexes, we have had discussions with the members of the Automation and Control group at Linköping University. Using a method called gain scheduling [GLAD91], a control engineer can model a non-linear control process using linear control techniques. This is done by dividing the non-linear control area into several small linear regions. The process shown in Figure 10.2 is an example of a non-linear process, since the output is not a linear function of the input. We can also see that if we divide the output into three different areas, each area can be described as a linear function of the input, and the value of the input can be used to determine which of the three functions we should use to calculate the output value.

This same technique is used to control a non-linear process. The control area is divided into smaller segments, and each segment can be described as a linear combination of the input signals, see Equation 10.1. In Equation 10.1 Y_j is output signal number j, U_i is input signal number i, and K_{ij}, and C_{ij} are constants.

$$Y_j = \sum_{i=1}^{n} K_{ij} \cdot U_i + C_{ij} \tag{10.1}$$

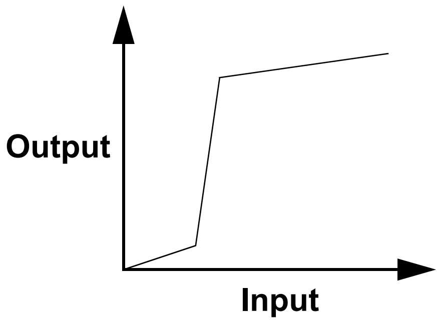


Figure 10.2: A non-linear process

The control model for each linear region is stored by the system, and by monitoring the input signals to the control process, it is possible to determine which control model to use.

The number of control regions is usually small, since it can be difficult to show that the resulting system is stable, but if this problem can be solved it is not impossible to imagine that the number of models that need to be stored could be large. A simple solution would be to store them in a database for easy retrieval. Whenever the control system needs to change control models it performs a search in the database and retrieves the proper control model.

A database solves this problem since it can easily store and manage a large number of models. A database system offers an efficient way of storing large amounts of data together with facilities to keep the data secure from unauthorized access and safe from system failures.

Unfortunately, it has not been possible to find the correct model based on the current input time series, since modern databases have been lacking in their ability to retrieve data based on time series[1]. A solution to the problem would be a database system equipped with time series indexing schemes, and capable of storing a large number of control models. When the control algorithm determines that the current control

1. Since then database systems with the capability to perform searches on time series have been released from, among others, Informix and Oracle. An evaluation of these is outside of the scope of this work. Future work will include an evaluation of existing time series support in database systems.

model is inappropriate, it can query the database using the input time series and then get a new control model from the database.

We identified two different approaches to performing an application study. Either we could identify an existing system that uses sub-sequence searching, or we could try to identify a new problem area and determine if it might benefit from our indexing methods. Using the first approach, the identified system could then be benchmarked and evaluated. Then the existing indexing method could be replaced by our methods and a new benchmark and evaluation of the system could be performed. The two versions could then be compared to each other, and we could see if the system using the indexing method had any benefits. Unfortunately, there is no existing system that works with the same kind of time series as we are aiming to support, i.e. very large time series, see Section 2.3, "Test Data," on page 18.

The second approach, as mentioned above, is to analyze a new problem that will benefit from our indexing methods, and see how it, from a database technology point of view, could be solved. Within the framework of our work with the Department of Automation and Control, we found a problem that included parts that could be solved using this approach, and in the subsequent sections we will describe the problem and then how the techniques we have presented in this thesis could be used to, with certain modifications, solve the indexing part of the problem.

10.2 Problem Description

When controlling large industrial processes, a large number of input signals are collected and used as feedback to the control algorithm. A typical control algorithm takes n input signals and outputs k control signals, where $n > k$, see Figure 10.3. The output of the control algorithm is a function of the current, and historical, input. Input data is not explicitly stored for future use, and it is discarded when it is no longer needed. Equation 10.4 shows how each output signal is calculated.

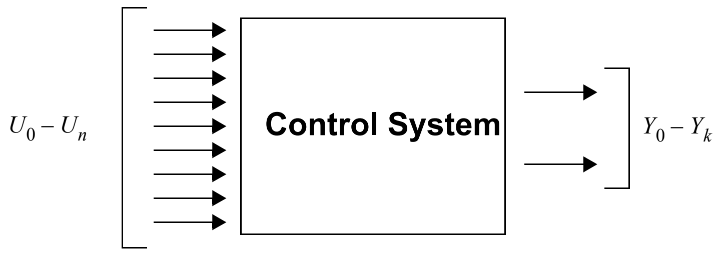


Figure 10.3: Control system

$$Y_j = \sum_{i=1}^{n} \sum_{k=0}^{l} (K_{ijk} \cdot U_{ik} + C_{ijk}) \tag{10.4}$$

In Equation 10.4 i iterates over all input signals, and k iterates over all stored historical values where $k = 0$ is the current value and $k = l$ is the value that was current l time steps ago. Y_j is output signal number j, U_{ik} is input signal number i that is k steps distant in time, and K_{ijk} and C_{ijk} are constants.

In some applications there is a need to save the input and output of the system. This need is not necessarily of a technical nature. It can be saved for security reasons, so that it is possible to go back in history and see if the system has been tampered with, or so that unexpected situations can be analyzed and the control model modified. It can also be stored so that engineers can perform data mining on the data. This is done in order to find new knowledge about the system from the behaviour of the signals, commonly known as system identification. For the problem of creating a gain schedule system that allows all control regions for all output signals to be broken up into large numbers of linear areas, we have a need to store a number of values, i.e. K and C values in Equation 10.4, each associated with a certain sequence of input values.

In discussing the gain schedule problem we identified three problem areas:

- The problem of making a sufficiently good linear approximation of a non-linear problem.
- The problem of verifying stability.
- The problem of storing a large number of control models so that they can be found based on the values of the input signals.

A few years ago the cost of storage media would probably have been identified as a problem but that is not true any more. The cost of computer memory and secondary storage has been greatly reduced over the last decade.

10.2.1 Linear Approximation of a Non-Linear Problem

Control theory has traditionally focused on linear processes since they are easier to model, and today there exists a vast amount of knowledge on how a linear process can be controlled. Most of the real control processes are not truly linear, but for a subset of these it is possible to create linear approximations of the process. Unfortunately this is not possible for all processes. It is possible, however, instead of approximating the entire process with a linear process, to divide the control area into a large number of small control areas. This makes it possible to approximate each control area with a linear model. This allows us to use linear control theory on a larger subset of all possible processes [GLAD91].

10.2.2 Stability

It is important to note that as the number of small control areas increases, the problem of proving that the resulting, combined, control process is stable becomes increasingly difficult. For that reason it is often not realistic to divide the control area into many smaller segments. No general solution to this problem exists yet.

10.2.3 Finding Data Based on Time Series

For large and complex processes the number of linear segments can theoretically become very large.This is where a database with the ability to index time series can be beneficial.

If the different control models were stored in a database with the ability to perform searches on time series, it would be possible to search the incoming signals and time series. Based on the result, a search could then be conducted for the correct control model that should be used.

10.2.4 Summary

The first two problems listed in Section 10.2.3 are control problems and will not be dealt with further in this thesis. The final problem can be solved by using a database system capable of indexing time series.

If we concentrate on the database problem we can identify the following sub-problems:

- Finding the control model associated with a set of input time series.
- Storing the control models in a suitable format.

We do not need to have any knowledge about the control model for solving the indexing problem, it can be treated as a BLOB, Binary Large OBject, that the system returns to the control application. Since the question of storing the models in a suitable format falls outside the scope of our current work, we have concentrated on the first point. The second point is left for future work.

10.3 Proposed Solution

If we consider what kind of searches we would like to be able to do in such an application, we can see that they differ from traditional sub-sequence searching in an important way. In traditional sub-sequence searching we have one, or several, time series. Given a search pattern, we then wish to find all sub-sequences (or their positions) within these time series that match a single query time series. This is what we call a one-dimensional query. In this case, we have several time series as input, and based on what they all look like, we wish to find the proper control model. This is what we call a two-dimensional query.

There is one more important difference between our current system and the system we propose as a solution to this problem. In our current work we state a query to the system and retrieve pointers to sub-sequences within the indexed time series that match the query. In the system we propose for solving this problem we do not associate the positions of the sub-sequences with the sub-sequences, but possible control models instead. So when we state a query to the proposed system, we do not receive a set of pointers to occurrences within the indexed time series, but a set of pointers to possible control models.

We have considered two approaches to transform this two-dimensional query into a one-dimensional query.

- Treat each input time series as separate time series.
- Create a modified indexing mechanism.

In the first case it would be necessary to perform several one-dimensional queries, each one resulting in a number of possible control models, and then perform an intersection of the results. E.g. if there are seven input time series to the process, we perform seven different queries on the seven different time series. Each query results in a number of possible control models, so to get the final result, the intersection between all queries has to be created. All time series might be indexed in the same index structure, but we would still have to search the index structure repeatedly.

The problem with this approach is that each query will result in several searches through the index structures. For one time series it will perform well, for two time series the performance will be reduced to half (each traversal through the index structure has a certain cost, see Section 2.1, "Measuring the Search Cost," on page 16), and if two searches have to be made, the cost will double. For a general system with n time series, the cost of performing a search will be n-times as high as for the one time series system.

The second approach is to modify the index structure so that these searches can be performed without having to traverse the index structures repeatedly. Instead of indexing each time series independently, we create an intersection of all time series for each time position. Figure 10.5 shows an example with four incoming time series. For each time position we create the intersection time series.

Since we have four time series the intersection time series will have a length of four. The intersection, at time t, is constructed by taking the value at time t of the first time series as the value in position one, the value at time t of the second time series as the second value, etc.

The result is that each time position will result in a small time series, each with the same length as the number of input time series.

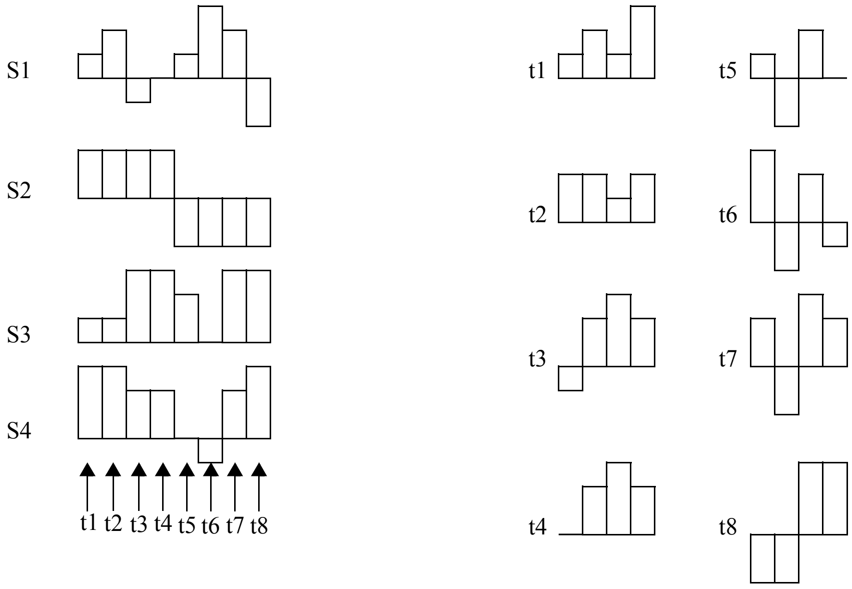


Figure 10.5: A new feature extraction method

This new time series can not be treated as a regular time series, since now it is not interesting to find sequences that are positioned between these intersection time series. Each cross section can be used as input to the indexing algorithms we have described. Instead of using the method with a sliding window described in Section 5, “The Signature File,” on page 59, we use each cross-section as a window.

With this change we believe that it would be possible to use our indexing methods to index this kind of data. However, these modifications open up several new areas that have to be investigated. The two most important questions we have considered are:

- The time fuzziness problem - is it possible to introduce a fuzziness in time?
- The scaling problem - how do we handle the scaling of the different time series?

These two problems will be described in detail below in Section 10.3.1 on page 130 and Section 10.3.2 on page 130.

10.3.1 The Time Fuzziness Problem

This problem is related to the fact that the time series might be slightly out of sync since, in most cases, they are generated by sampling an analog signal. Since we are "freezing" the signal at a certain time, we might not only introduce a certain fuzziness in the amplitude, but also along the time axis. This fact is not a real issue when we work with natural occurrences of time series.

If we only work with one time series, the values might be shifted along the time axis in the time series but nothing more. The result might be that we find the pattern we are searching for at offset 356 instead of 355.

When we have several time series this has the potential to become a problem, as one of the sequences might become slightly out of sync with the others. This might lead to a situation were we fail to detect a pattern even though it is present in the input time series. In Figure 10.6 we can see an example of this. In case one we are looking for the sequence "abagce" and we find it (marked with the box). In the second case the fourth time series from the top has been offset by one. The sequence we are looking for is marked by the boxes. The normal intersection technique outlined above will not allow us to find the sequence in the second case. The preferred solution would be to have a more flexible matching algorithm that can match the query pattern to a two-dimensional area instead. How this could be implemented remains to be investigated.

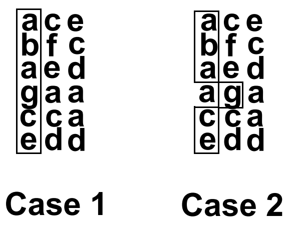


Figure 10.6: Time fuzziness

10.3.2 The Scaling Problem

The second issue we have considered is scaling of the different time series. When the feature extraction processes described in Section 3.1, "Feature Extraction Using

Time-Derivative," on page 25 or Section 3.2, "Feature Extraction Using Curvature," on page 29 are used on a single time series, it is possible to find a suitable scaling. For example, the threshold values for the time derivative can be modified, so that relevant information is retrieved from the time series.

If intersection time series are created and then subjected to one of the feature extraction processes (Section 3.1, "Feature Extraction Using Time-Derivative," on page 25 or Section 3.2, "Feature Extraction Using Curvature," on page 29) it is very likely that some of the incoming time series will have to be scaled, since in a general application we must assume that the different time series can have different data ranges. To exemplify this, assume that the domain of two of the time series is $[0, 10]$ and that the domain of the third is $[100, 1000]$. Unless some kind of scaling is introduced, the system will always map the difference between the time series with the smaller domain, and the one with the larger domain as a very rapid change, no matter how the larger time series varies with time.

Figure 10.7 exemplifies this problem. To the left, three different time series can be seen, S1, S2 and S3. To the right, the first five intersection time series are shown. As can be clearly seen in this figure, the difference between the middle value (originating in time series S2) and the right value (originating from time series S3) will always be very large. Any behaviour in S3 will be lost when these intersection time series are translated into symbol strings.

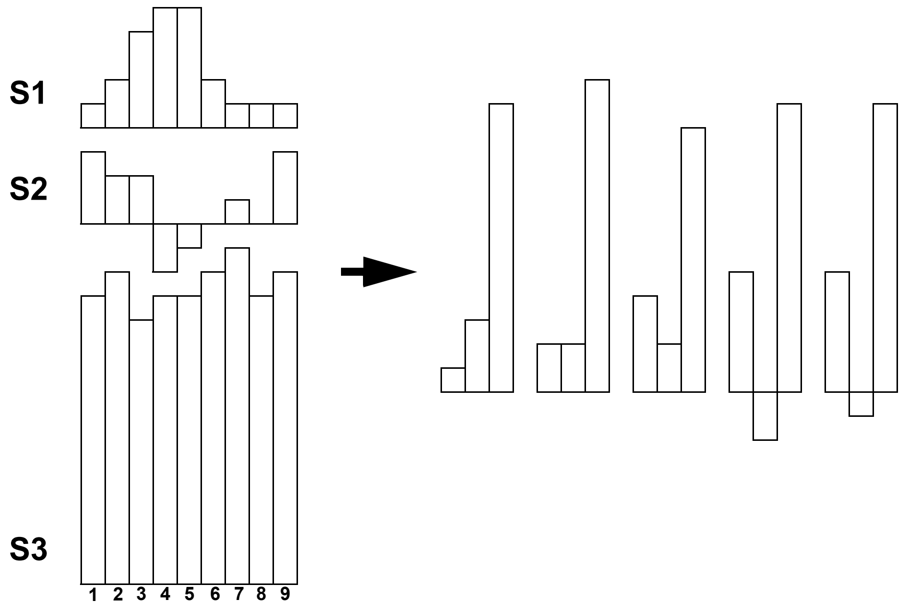


Figure 10.7: Scaling problem

We believe that one of our proposed indexing methods can offer a solution, giving the system the functionality it needs. We have not implemented these changes yet since we feel that it is important to get a firm understanding of the requirements before doing any modifications. The necessary examination has been outside the scope of our current project, and will be addressed in future work.

10.4 Conclusions

In many control applications we are not interested in querying for the history of each input signal. We might more often wish to make queries on a group of input signals (time series once they have been sampled). Stating individual queries on each input time series and then creating an intersection of all answers will result in a very high search cost if there are several input time series. A better solution would be to create a combined index structure for all time series, and then perform the searches on that instead. A possible solution we are considering is to create intersection time series from the incoming time series and use these smaller time series as input to our indexing methods. This approach introduces some additional problems that have to be addressed before the approach can be tested in a real application.

These issues will be examined in our future work with the Automation and Control group at Linköpings Universitet.

Chapter 11

Conclusions

In this chapter we conclude our work presented in this thesis.

In this thesis we have investigated the possibilities of using text indexing methods to index very large time series. We have investigated several methods for extracting information from the time series and then transforming the extracted data into a text string. Using this approach it is also possible to achieve a very simple and intuitive concept of similarity.

11.1 Feature Extraction

Several ways of extracting information from a sequence exist. The most simple techniques extract the shape of the sequence by calculating the time derivative or curvature in each point and use these values as a description of the sequence. More advanced features are feature extraction processes that use DFT, or other mathematical transform processes, to extract the most dominant features or to use polynomial approximations.

We have done a survey of the most popular methods in this thesis, but decided to use a simple feature extraction, using the time derivative or curvature, a process that allows us to easily map the time series to a text string. There is nothing in our approach that prohibits other feature extraction methods.

11.2 Indexing Structures

We have examined four different methods for indexing time series. Our conclusion is that for small time series where we do not wish to introduce the added overhead of a more complex method, signature files might be a good index structure. For larger time series, and if the extra overhead does not deter us, B-trees offer much better performance. Since the search time grows linearly with the size of the data set when we use signature files and it grows logarithmically when we use B-trees we will always reach a break-even point when B-trees will be much faster to search than signature files.

The other structures we have looked at have been shown to lack some of the properties we wish to have in a disk based system, e.g. minimizing the number of disk blocks that have to be loaded while performing an index search.

11.3 Summary

Our conclusion is that it is quite possible to use text indexing methods to index time series. For small time series where we want a small overhead, signature files might be a good index structure, but for larger time series the B-tree is the prime candidate. We have developed a technique that allows us to use B-trees to index time series.

For main memory applications where we want a very fast and flexible index structure that allows us to easily modify the questions, the PTrie might also be a very good solution.

Chapter 12

Future Work

Feature extraction is the part of our work that has the largest potential for growth. Our goal has been to see if it was possible to use text indexing methods to index large time series. We have found two feature extraction methods that allow us to create a sequence of symbols from a time series.

We have focused on finding a fast scalable index structure for large time series. While pursuing this goal we have made certain trade-offs, e.g. search flexibility. One problem with our proposed solution is that the level of fuzziness is set when the index is created and it can not be controlled at query time. It would be desirable to find a feature extraction method or modification to the index structure that allowed the user to control the level of desired fuzziness at query time. Future work in this area should include a more thorough examination of how different extraction techniques can be used with our approach to see if this functionality can be supported.

It would also be interesting to examine what these features cost in search performance and size. One way of doing this would be to extend our survey to other time series indexing methods and evaluate them against our system.

We also plan to benchmark and test our approach to time series indexing against other approaches. We plan to compare our approach to the now-available implementations of time series support in modern commercial database systems such as Oracle, Informix, etc.

Another interesting issue to examine is whether the methods we have looked at in this thesis can be generalized and used for data sets with more than one dimension. For example, it would be interesting to see whether these methods would allow us to index two-dimensional data, e.g. images. An ongoing research effort at the University of Colorado, Colorado Springs, is to index fingerprints. One possible solution would be to transform the two-dimensional fingerprint image into a sequence of one-dimensional sequences and then index these sequences with an extended version of our process. However, even if image data is indexed as a sequence of one-dimensional sequences, it will have other properties than regular time series, and because of that it will have to be treated differently.

Another issue that should be investigated is how the index should be integrated into an existing database system. A query language has to be proposed. An investigation of the type of query language suitable for time series queries has to be made. Together with the query language, further issues have to be resolved: how should the queries be optimized, and can it be incorporated into an existing query language like SQL. Can the query language be incorporated into an existing query language in such a way that it would be possible to state queries that contain both regular database queries expressed in SQL and queries over time series?

We also feel that an investigation of how these tools we have developed could be used to solve existing problems is a natural extension of our work. In cooperation with the control and automation department, we will investigate several problems. These investigations will determine whether our indexing mechanisms are useful for groups that need to search time series. The first limited investigation we have conducted has shown us that we face additional problems when we try to use these techniques in real applications. Within this cooperation we will also examine whether it is possible to use the same indexing techniques on things other than time series. One area we have discussed is whether it is possible to treat high-dimensional vectors as time series, and then use our techniques to quickly find stored vectors that are within a certain distance from a given query vector. This would allow the use of new control algorithms that benefit from the ability to perform searches on time series in applications where it has not been feasible before due to time constraints.

Since our goal has been to index very large time series, an area closely related to this is the matter of how very large time series should be stored on disk. An investigation should be performed on how very large, (terabyte sized) time series data should be stored.

Chapter 13

References

[AGRA95] Fast Similarity Search in the Presence of Noise, Scaling and Translations in Time-Series Databases, *In Proc. of the 21st International Conference on Very Large Databases (VLDB95), pp. 490-501*, R. Agrawal, K.-I. Lin, H.S. Sawhney, K. Shim.

[AGR295] Querying Shapes of Histories, *In Proc. of the 21st International Conference on Very Large Databases (VLDB95), pp. 502-514,* R. Agrawal, G. Psaila, E.L. Wimmers, M. Zaït.

[ANDR97] Retrieval of One-Dimensional Data, *In Proc. of BIWIT97 International Workshop on Information Technology, pp. 133-139,* H. André-Jönsson, D.Z. Badal.

[ANDR99] Indexing Time Series Data using Text Indexing Methods, *Lic. Thesis No. 723 at the Department of Computer and Information Science, Linköpings Universitet,* H. André-Jönsson.

[ANDR00] Indexing Time-Series Using Signatures, *to be published in the Journal of Inteligent Data Analysis, Elsevier Science,* H. André-Jönsson, D.Z. Badal.

[BADA95] Investigation of Unstructured Text Indexing, *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 the ACM SIGMOD International Conference on Management of Data (SIGMOD90), pp. 322-331*, N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger.

[BELL93] Data Compression in Full-Text Retrieval Systems, *Journal for American Society for Information Sciences 44(9), pp. 508-531*, T. Bell, A. Moffat, C. Nevill-Manning, I. Witten, J. Zobel.

[BENT75] Multidimensional binary search trees used for associative searching, *Communications of the ACM(18) 9 1975, pp. 509-517*, J.L. Bentley.

[BERC96] The X-tree: An Index structure for High-Dimensional Data, *In Proc. of the 22nd International Conference on Very Large Databases (VLDB96), pp. 28-39*, S. Berchtold, D.A. Keim, H.-P. Kriegel.

[BOHA01] Main-Memory Index Structures with Fixed-Size Partial Keys, *In Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD01), pp. 163-174*, P. Bohannon, P. McIlroy, R. Rastogi.

[BOYE77] A Fast String Searching Algorithm, *In Communications of ACM 20, 10(October), pp. 762-772,* R.S. Boyer, J.S. Moore.

[CARD75] Analysis and Performance of Inverted Data Base Structures, *In Communications of ACM 18 (9), pp. 253-263*, A. Cárdenas.

[DATE00] An Introduction to Database Systems, 7th edition, *Addison-Wesley*, C.J. Date.

[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.

[ELMA00] Fundamentals of Database Systems, 3rd edition. *Addison-Wesley*, R. Elmasri, S. Navathe.

[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 Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD94) 5/94, pp. 419-429*, C. Faloutsos, M. Ranganathan, Y. Manolopoulos.

[FOX92] Inverted files, *In Information Retrieval: Data Structures and Algorithm, pp. 28-43*, E. Fox, D. Harman, R. Baeza-Yates, W. Lee.

[FRED60] Trie memory, *In Communications of the ACM 3, pp. 490-500*, E. Fredkin.

[GLAD91] Reglerteori, Flervariabla och olinjära metoder. *ISBN 91-4400-472-9*,T.Glad, L.Ljung.

[GUTT84] R-trees: A dynamic index structure for spatial searching, *In Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD84), pp. 47-54*. A. Guttman.

[GÜNT97] Oversize shelves: A storage management technique for large spatial data objects, *In the International Journal of Geographical Information Systems 11(1) pp. 5-32*. O. Günther, V. Gaede

[HASK81] Special Purpose Processors for Text Retrieval, *In Database Engineering 4(1), pp. 16-29*, R. Haskin.

[HOEL92] A qualitative comparison study of data structures for large segment databases, *In Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD92), pp. 205-214*. E.G. Hoel, H. Samet.

[INFO97] Informix Software, “Informix Time Series Data-Blade Module”.

[JAGA00] On Effective Multi-Dimensional Indexing for Strings, *In Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD00), pp.403-414*, H.V. Jagadish, N. Koudas, D. Srivastava.

[KAHV01] Efficient Index Structure for String Databases, *In Proc. of the 27th International Conference on Very Large Databases (VLDB01), pp.351-360*, T. Kahveci, A.K. Singh.

[KNUT98] The art of Computer Programming, vol3: Sorting and Searching, 2nd edition, D.E. Knuth.

[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 Proc. of the 24th International Conference on Very Large Databases (VLDB98), pp. 170-181,* L. Lin, T. Risch.

[LIN99] Management of 1-D Sequence Data - From Discrete to Continous, *Dissertation No. 561, Linköping University, ISBN 91-7219-402-2*, L. Lin.

[LITW80] Linear hashing: A new tool for file and table addressing, *In Proc. of the 6th International Conference on Very Large Databases (VLDB80), pp. 212-223*, W. Litwin.

[MOFF92] Economical inversion of large text files, *In Computational Systems 5(2), pp. 125-139*, A. Moffat.

[MOFF95] In-situ generation of compressed inverted files, *In Journal of American Society for Information Sciences 46(7), pp. 537-550*, A. Moffat, T. Bell.

[MOFF96] Self-indexing Inverted Files for Fast Text Retrieval, *In ACM Transactions on Information Systems 14(4), pp. 349-379*, A. Moffat, J. Zobel.

[ORAC97] Oracle Corporation, “Oracle Time Series Cartridge User’s Guide”.

[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.

[PGP00] PGP is an encryption system available from “Network Associates Inc.”, 3965 Freedom Circle, Santa Clara, CA 95054, USA, *http://www.nai.com/.*

[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.

[RONS98] Design and Modelling of a Parallel Data Server for Telecom Applications, *Dissertation No. 520, Linköping University, ISBN 91-7219-169-4*, M. Ronström.

[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.

[SAME84] The quadtree and related hierarchical data structure, *In ACM Computing Surveys 16(2), pp.187-260*, H. Samet.

[SAME89] The design and analysis of Spatial Datastructures, *Addison-Wesley*, H. Samet.

[SAME90] Applications of Spatial Data Structures, *Addison-Wesley*, H. Samet.

[SEGE93] A Temporal Data Model Based on Time Sequences, *In "Temporal Databases, Theory, Design and Implementation", The Benjamin/Cummings Publishing Company Inc, ISBN 0-8053-2413-5, 1993, pp. 248-269*, A. Segev, A. Shoshani.

[SELL87] The R+-tree: A Dynamic Index for Multi-Dimensional Objects", *In Proc. of the 13th International Conference on Very Large Databases (VLDB87), pp.507-518*, T. Sellis, N. Roussopoulos, C. Faloutsos.

[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.

[SHOS86] Temporal Data Management, *In Proc. of the 12th International Conference on Very Large Databases (VLDB86), pp. 79-88,* A. Shoshani, K. Kawagoe.

[TANS93] Temporal Databases, Theory, Design and Implementation, *The Benjamin/Cummings Publ. comp*, A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass.

[WANG01] Indexing very high-dimensional sparse and quasi-sparse vectors for similarity searches, *In Proc. of the 27th International Conference on Very Large Databases (VLDB01), pp.344-361*, C. Wang, X.S. Wang.

[WITT94] Managing Gigabytes: Compressing and Indexing Documents and Images, *Van Nostrand Reinhold, New York*, I. Witten, A. Moffat, T. Bell.

[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.

[ZOBE98] Inverted Files Versus Signature Files for Text Indexing, *In ACM Transactions on Database Systems, Vol. 23, No. 4, pp. 453-490*, J. Zobel, A. Moffat, K. Ramamohanarao.

Appendix A

Signature File Precision Diagrams

In this appendix we present the precision diagrams we have obtained from testing our signature file implementation.

This appendix presents the different signature file precision diagrams obtained from our experiments. Each diagram represents one test configuration; the signature size, the window size and the number of bits set by each window is constant and only the query length varies.

Figure A.1 shows an example precision diagram. Each line in the diagrams is one test run. A test run is created by a simple procedure. The test system fetches a sub-sequence from the stored sequence that is “20+window size” characters long. Then it starts testing the signature file. The first “window size” characters from the sub-sequence 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 sub-sequence is added to the query pattern, and the procedure is repeated. This is continued until the query pattern consists of the entire sub-sequence.

We have tested each configuration with 7 test runs.

In the first 24 diagrams (Figure A.2 to Figure A.26) each window sets only one bit in the signature, and in the next 24 diagrams (Figure A.27 to Figure A.51) each window sets 4 bits in each signature.

In the first 5 diagrams (Figure A.2, Figure A.3, Figure A.4, Figure A.5 and Figure A.6) a 16-bit signature was used with varying window sizes. As can be seen, the precision of the system does not start to improve until we use a 9-character window, Figure A.5, or a 20-character window, Figure A.6, but it is still not good. A 16-bit signature is not useful with this setup unless we search for very large sequences.

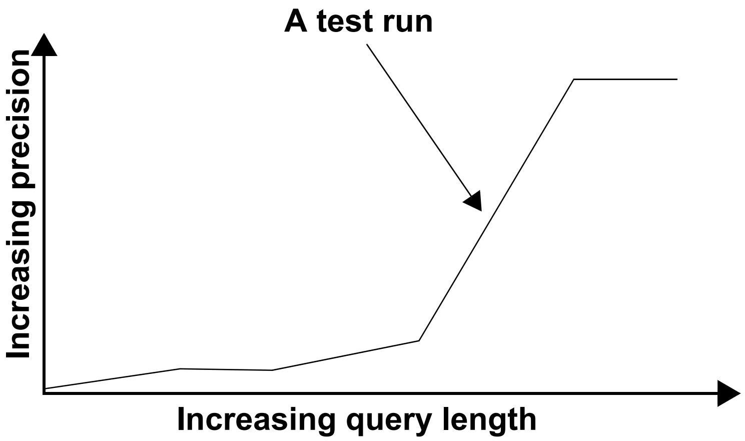


Figure A.1: A precision diagram

In the next 5 diagrams (Figure A.7, Figure A.8, Figure A.9, Figure A.10 and Figure A.11) 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 A.17, Figure A.18, Figure A.19, Figure A.20 and Figure A.21 a 251-bit signature is used. With a very small window size, Figure A.17, 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 A.19, the precision starts to come up to tolerable levels for mid-sized sequences, and for larger window sizes, Figure A.20 and Figure A.21, we get a precision of 1 when we search for sequences that consist of 13 to 14 windows (window size+13 characters).

In Figure A.22, Figure A.23, Figure A.24, Figure A.25 and Figure A.26 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 A.23, we already 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 A.27, Figure A.28, Figure A.29, Figure A.30 and Figure A.31, we use the same configuration as in Figure A.2, Figure A.3, Figure A.4, Figure A.5 and Figure A.6, 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.

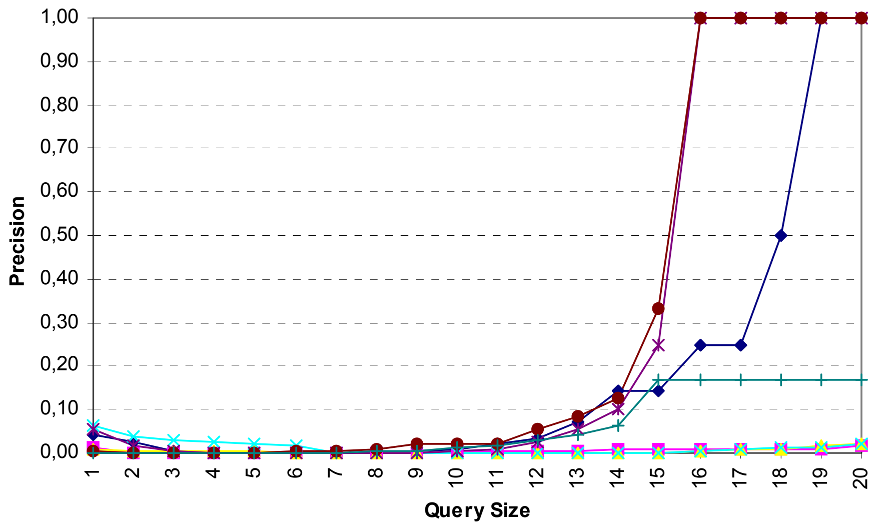


Figure A.2: 16-bit signature, 3-character word, 1 bit per word

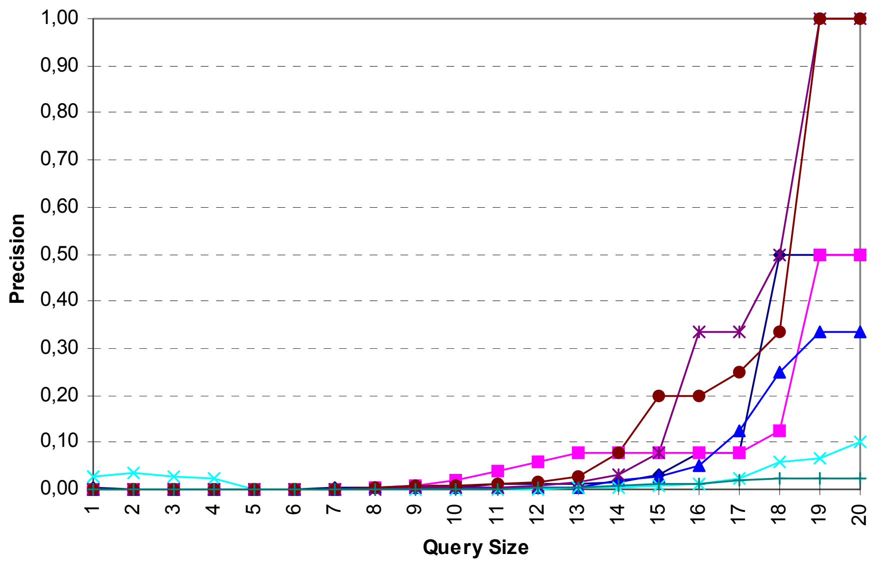


Figure A.3: 16-bit signature, 5-character word, 1 bit per word

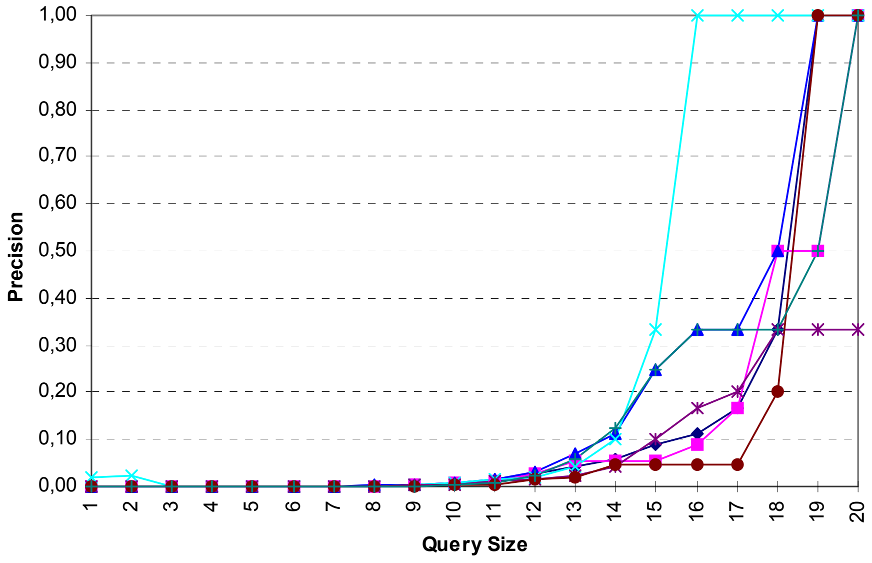


Figure A.4: 16-bit signature, 7-character word, 1 bit per word

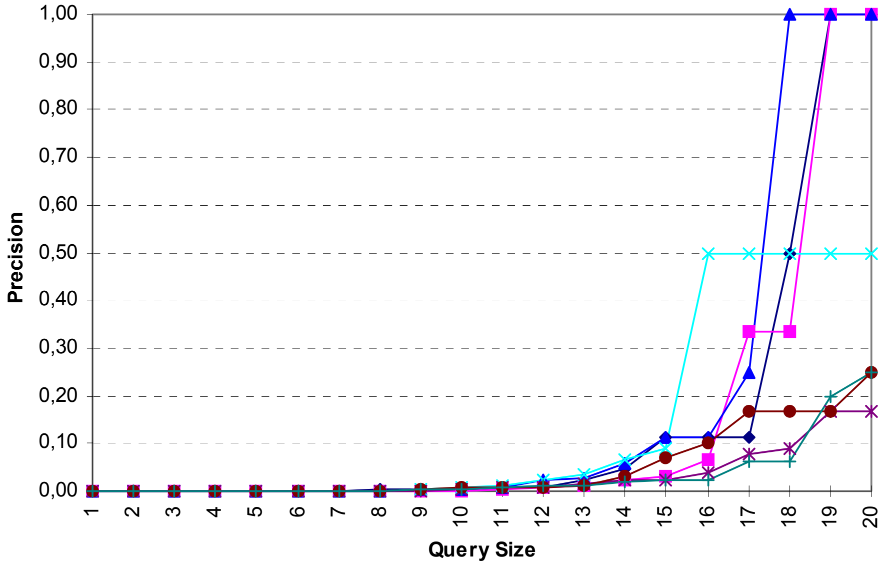


Figure A.5: 16-bit signature, 9-character word, 1 bit per word

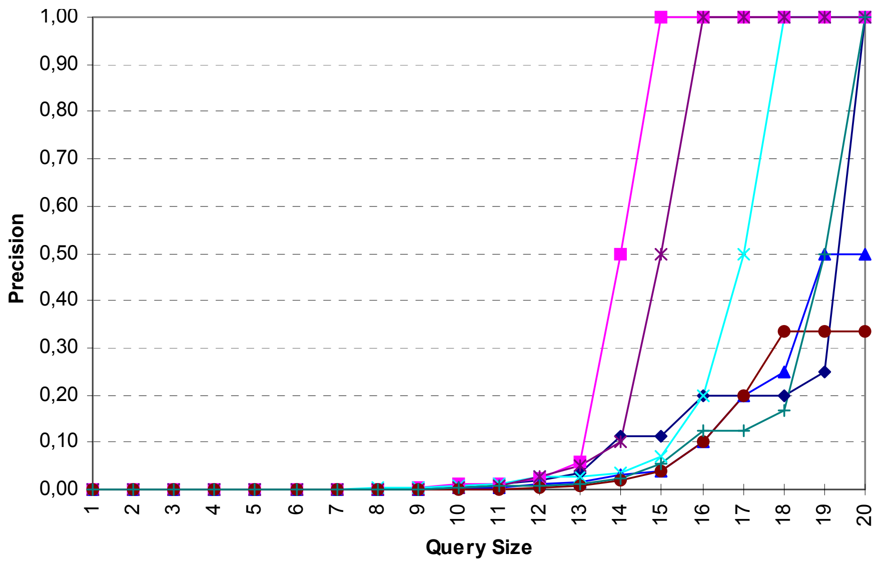


Figure A.6: 16-bit signature, 20-character word, 1 bit per word

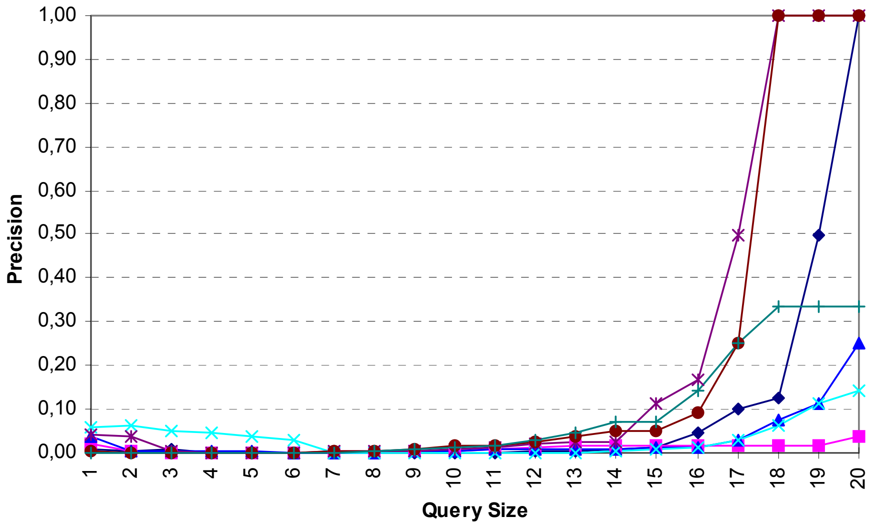


Figure A.7: 24-bit signature, 3-character word, 1 bit per word

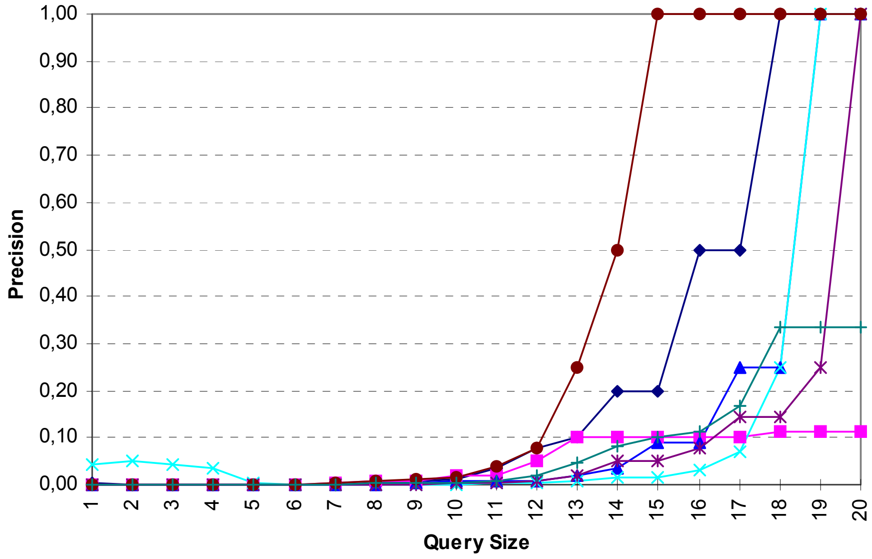


Figure A.8: 24-bit signature, 5-character word, 1 bit per word

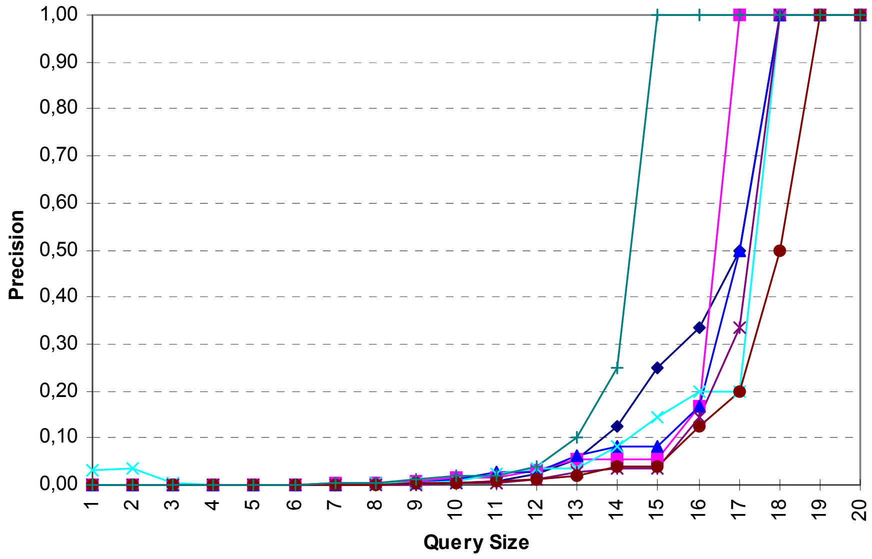


Figure A.9: 24-bit signature, 7-character word, 1 bit per word

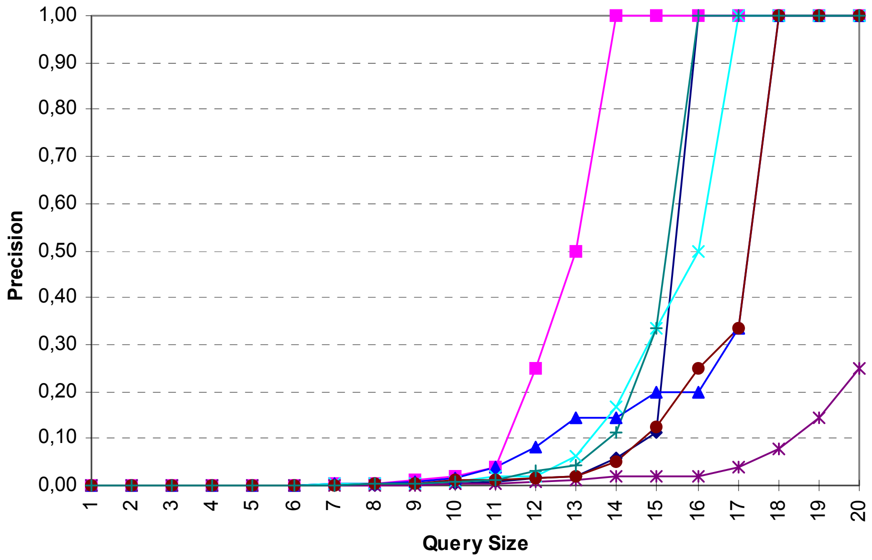


Figure A.10: 24-bit signature, 9-character word, 1 bit per word

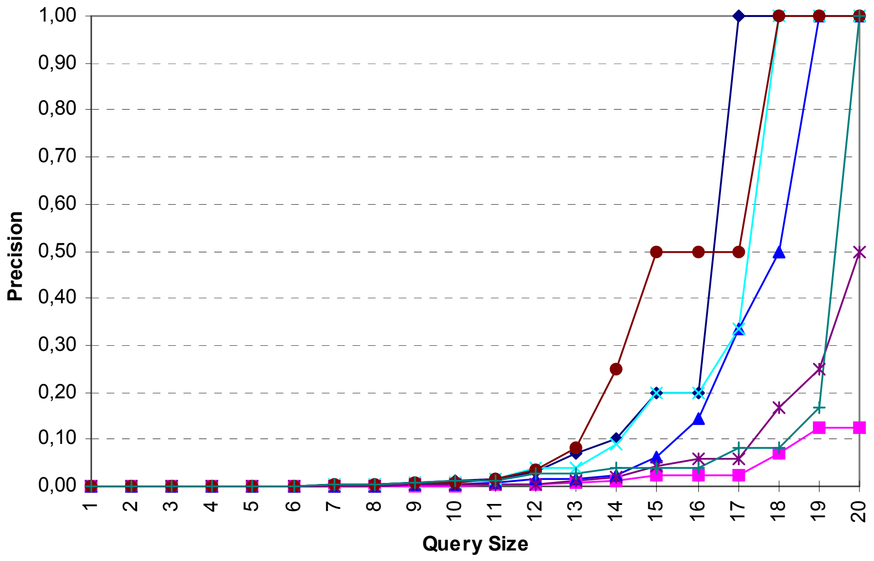


Figure A.11: 24-bit signature, 20-character word, 1 bit per word

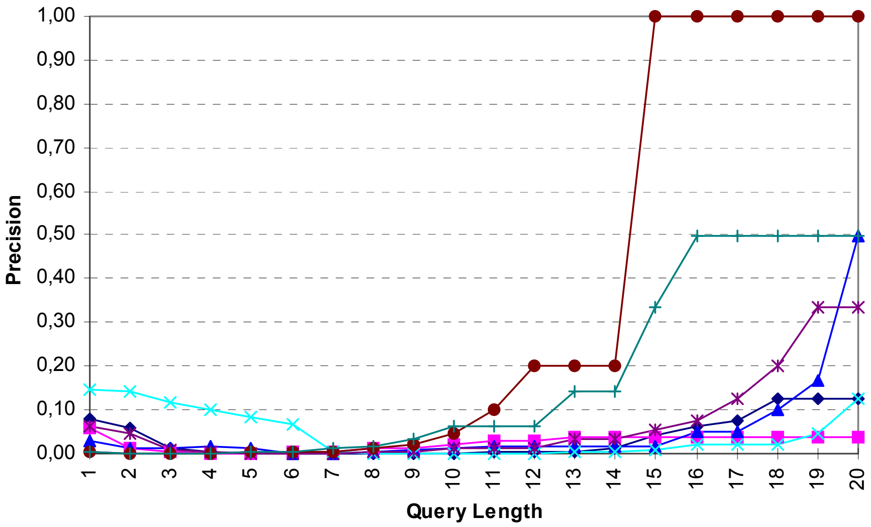


Figure A.12: 61-bit signature, 3-character word, 1 bit per word

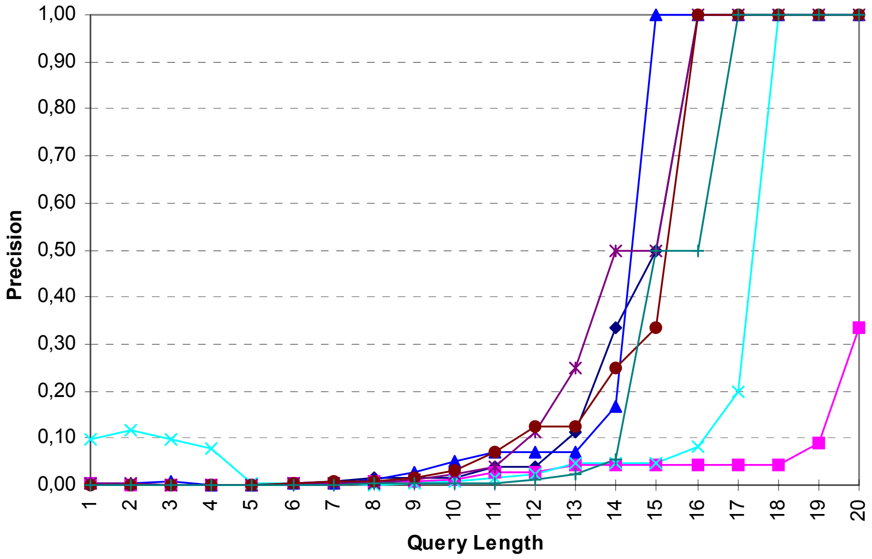


Figure A.13: 61-bit signature, 5-character word, 1 bit per word

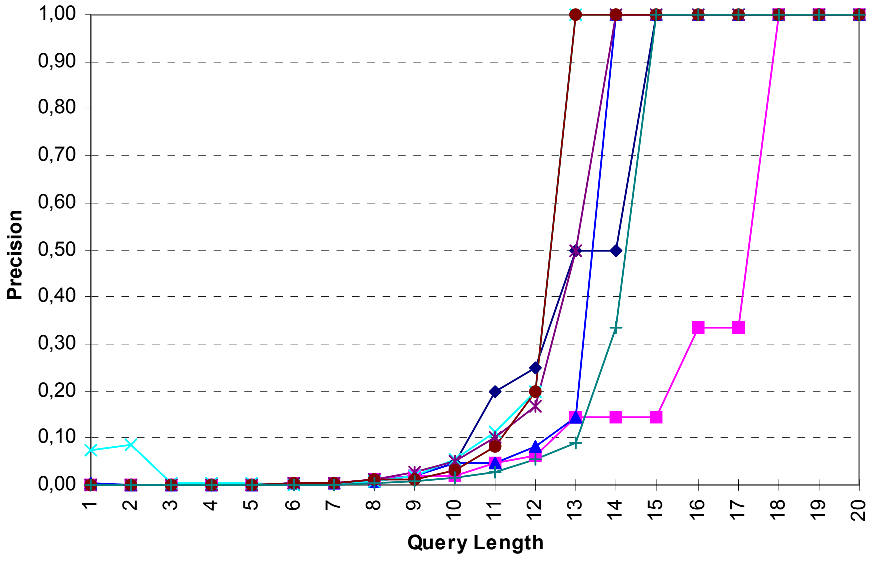


Figure A.14: 61-bit signature, 7-character word, 1 bit per word

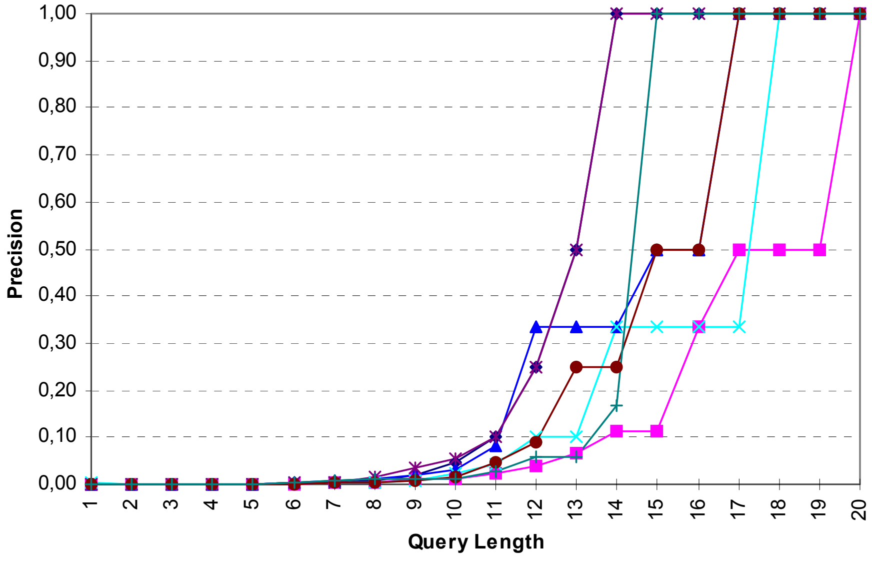


Figure A.15: 61-bit signature, 9-character word, 1 bit per word

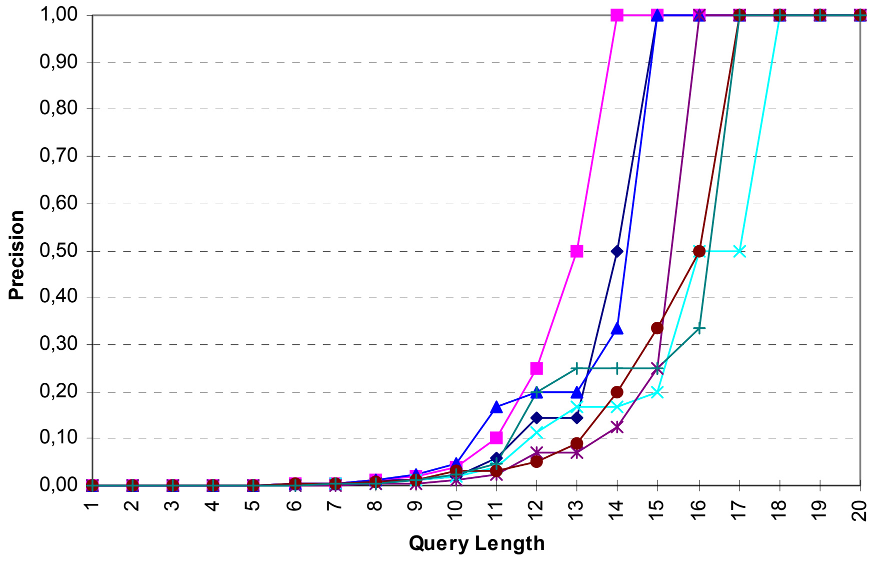


Figure A.16: 61-bit signature, 20-character word, 1 bit per word

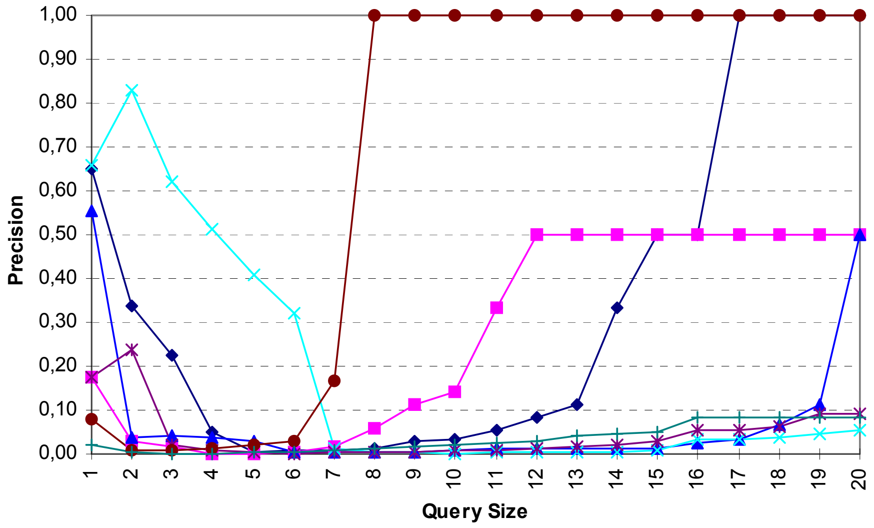


Figure A.17: 251-bit signature, 3-character word, 1 bit per word

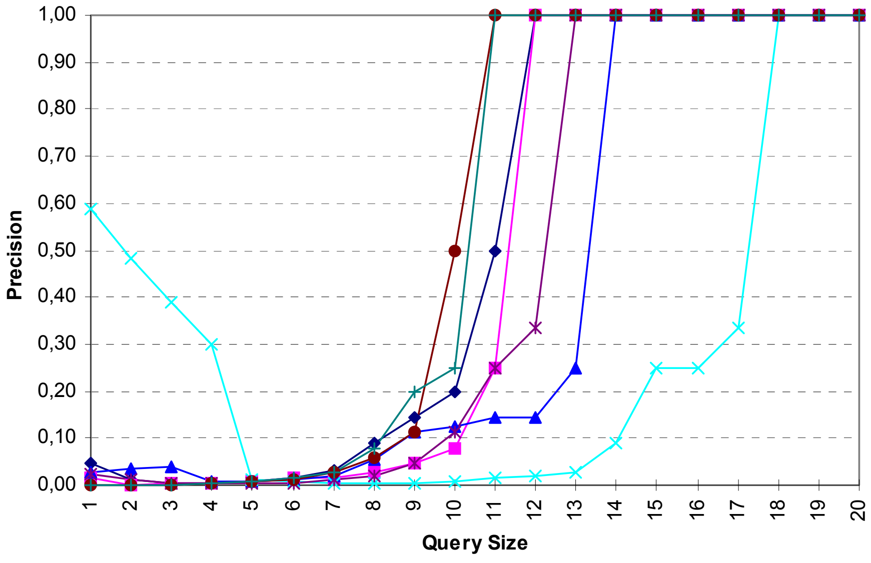


Figure A.18: 251-bit signature, 5-character word, 1 bit per word

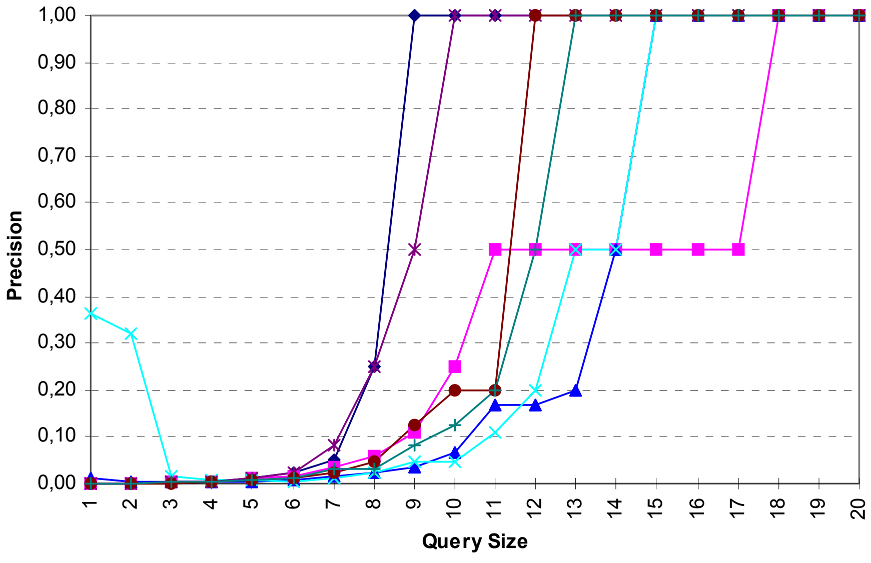


Figure A.19: 251-bit signature, 7-character word, 1 bit per word

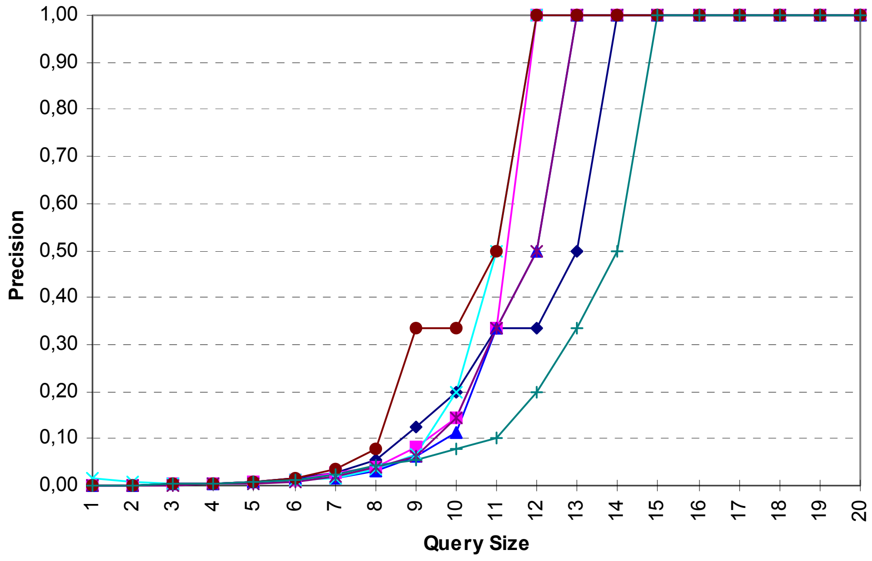


Figure A.20: 251-bit signature, 9-character word, 1 bit per word

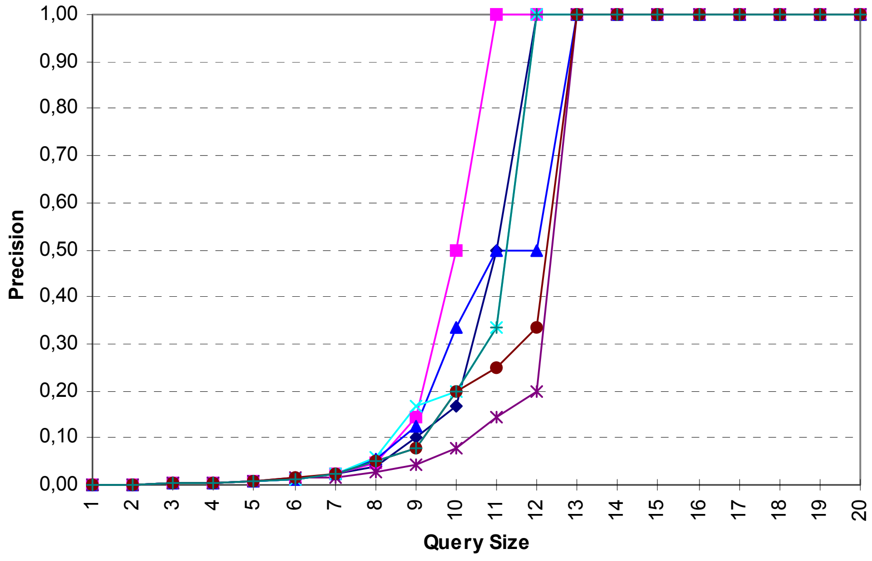


Figure A.21: 251-bit signature, 20-character word, 1 bit per word

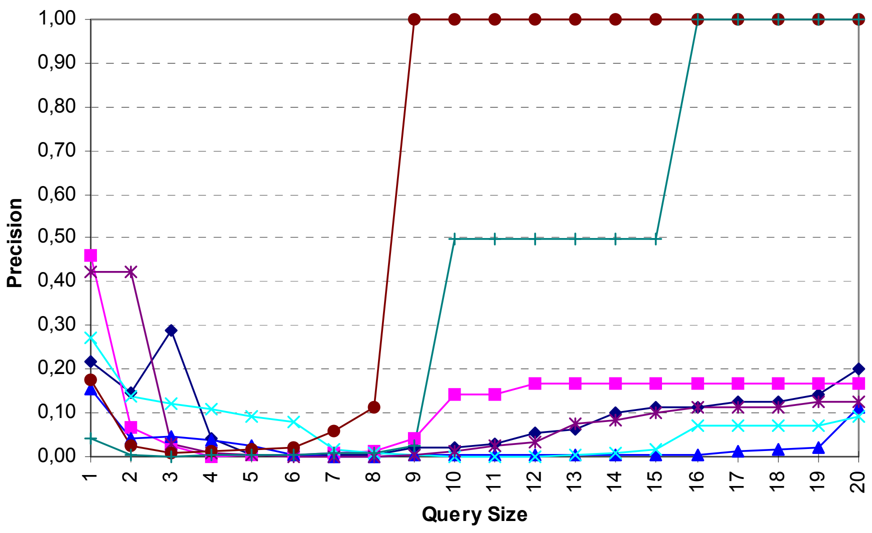


Figure A.22: 509-bit signature, 3-character word, 1 bit per word

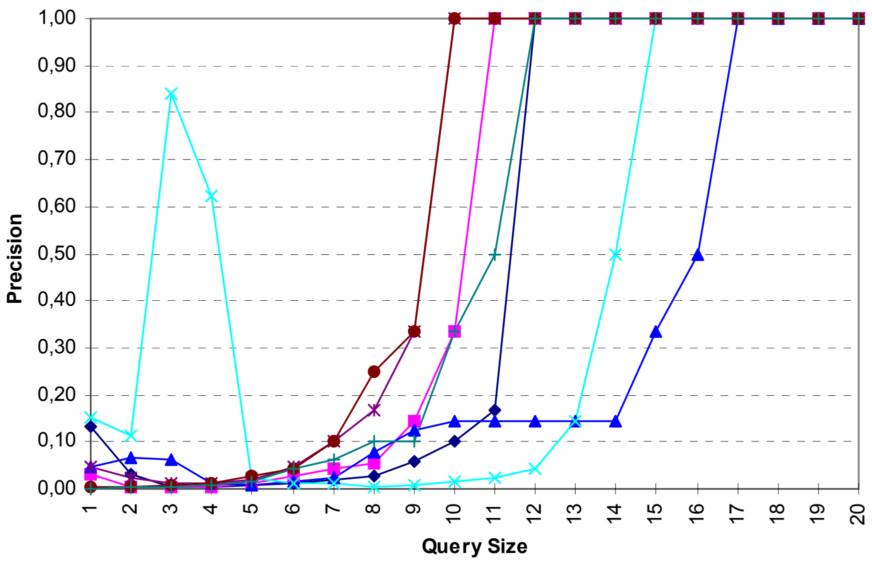


Figure A.23: 509-bit signature, 5-character word, 1 bit per word

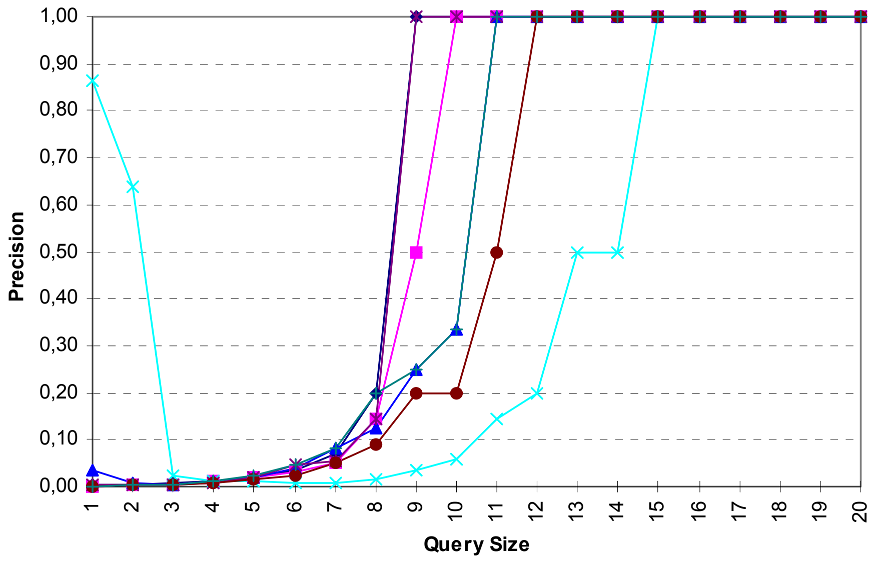


Figure A.24: 509-bit signature, 7-character word, 1 bit per word

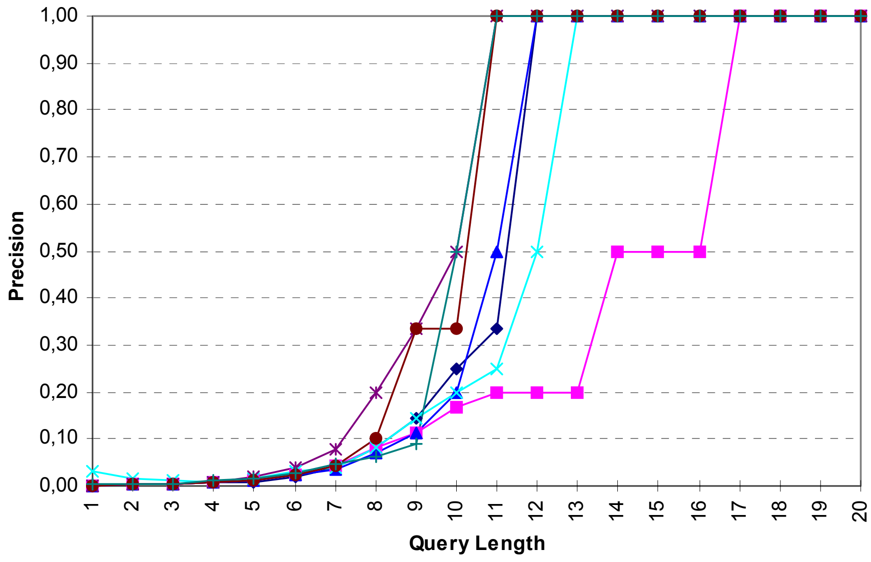


Figure A.25: 509-bit signature, 9-character word, 1 bit per word

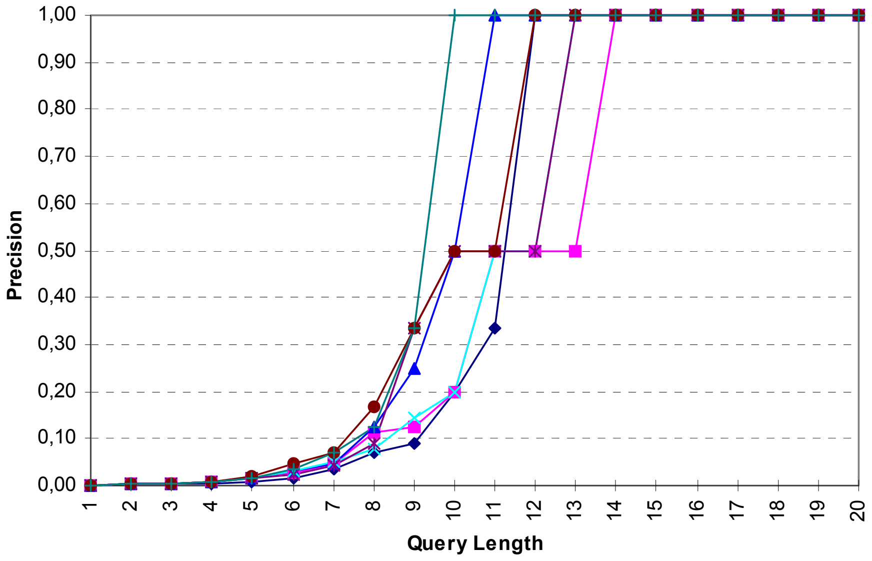


Figure A.26: 509-bit signature, 20-character word, 1 bit per word

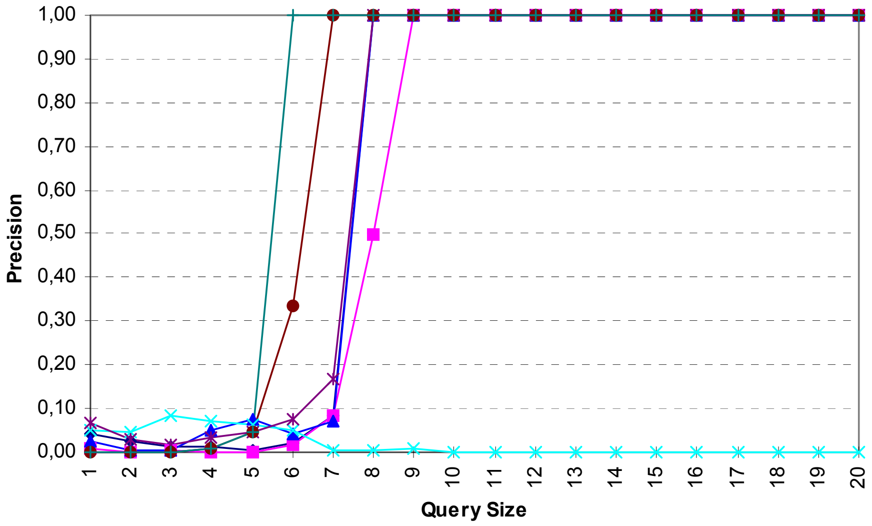


Figure A.27: 16-bit signature, 3-character word, 4 bits per word

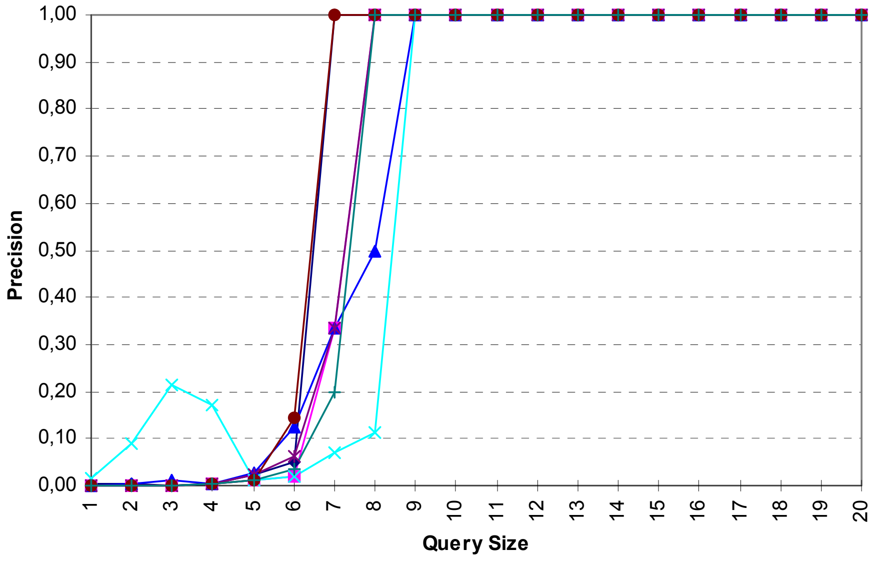


Figure A.28: 16-bit signature, 5-character word, 4 bits per word

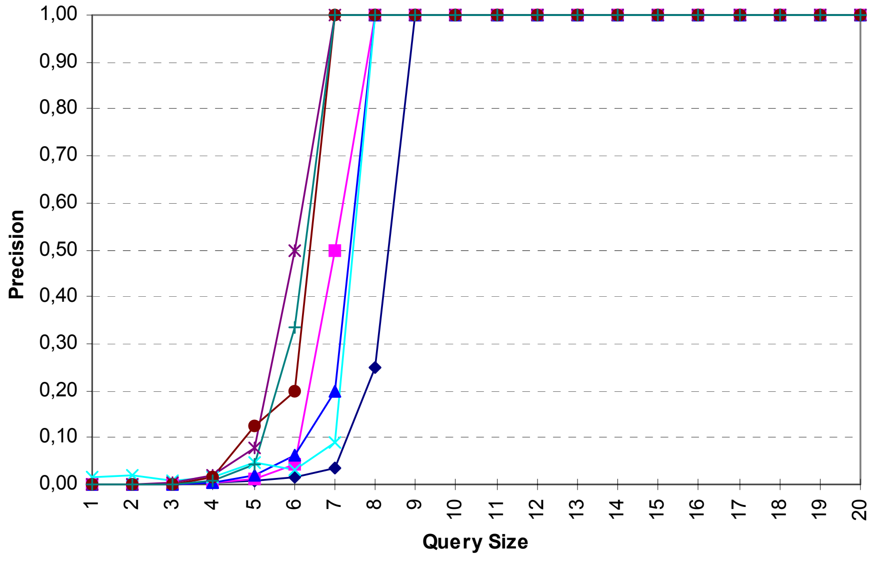


Figure A.29: 16-bit signature, 7-character word, 4 bits per word

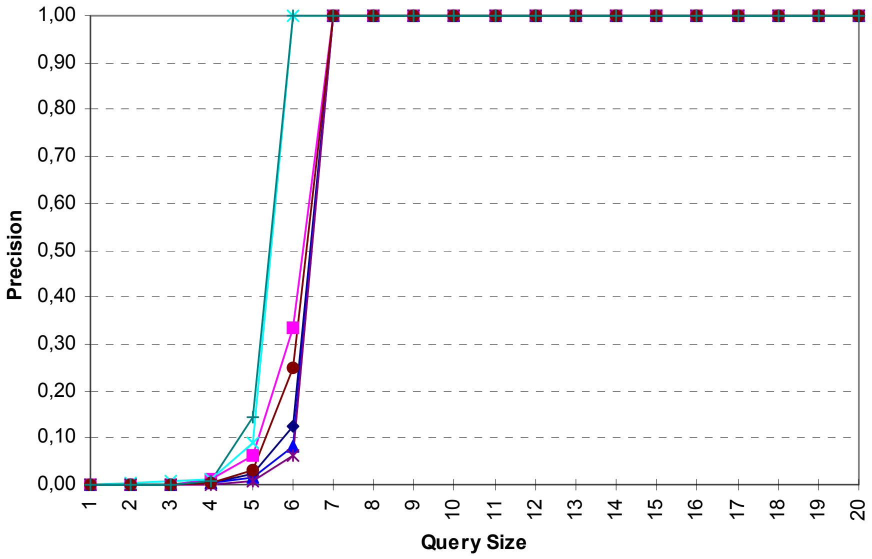


Figure A.30: 16-bit signature, 9-character word, 4 bits per word

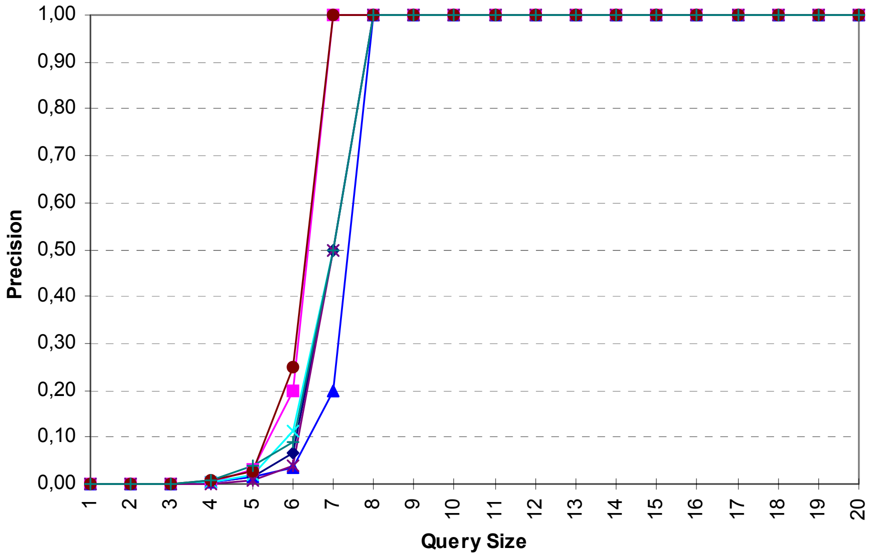


Figure A.31: 16-bit signature, 20-character word, 4 bits per word

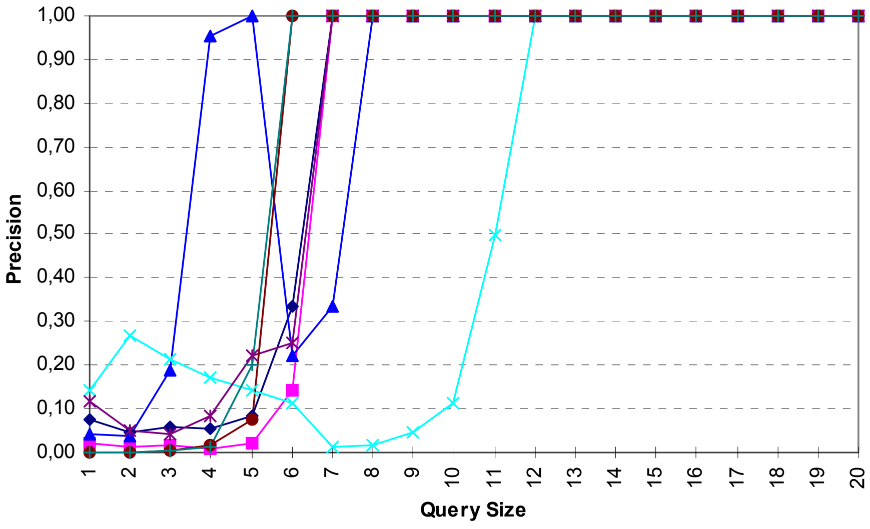


Figure A.32: 24-bit signature, 3-character word, 4 bits per word

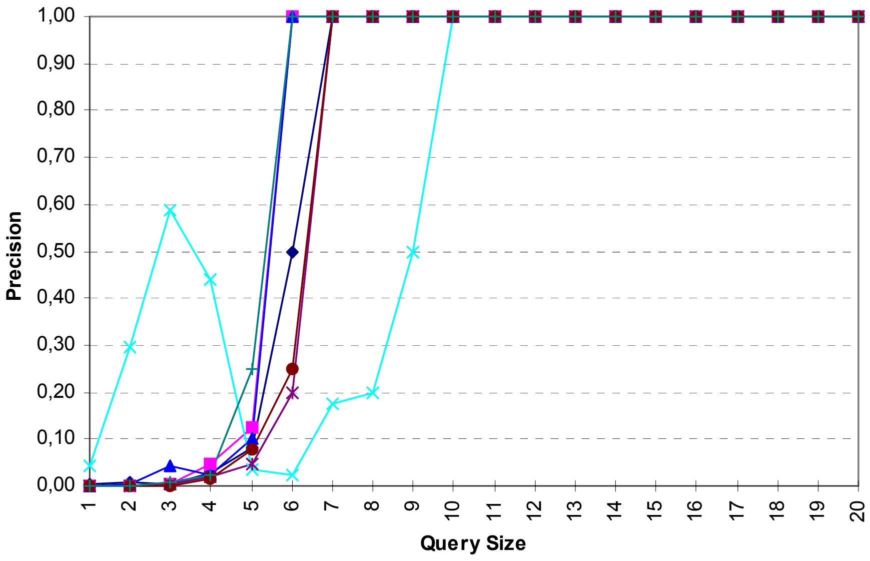


Figure A.33: 24-bit signature, 5-character word, 4 bits per word

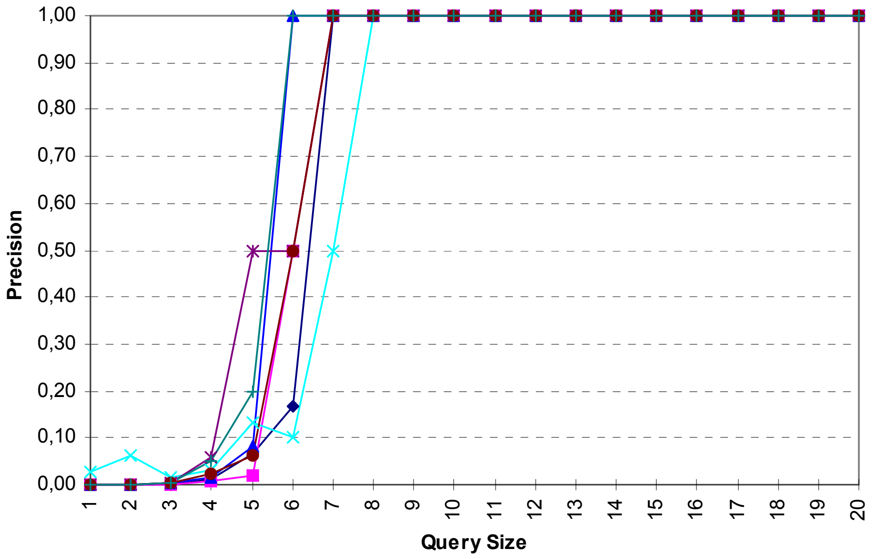


Figure A.34: 24-bit signature, 7-character word, 4 bits per word

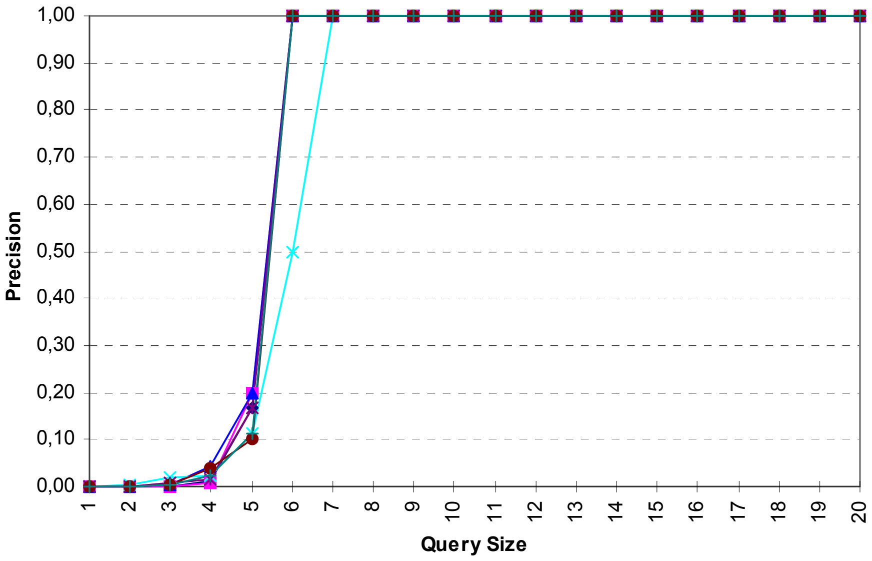


Figure A.35: 24-bit signature, 9-character word, 4 bits per word

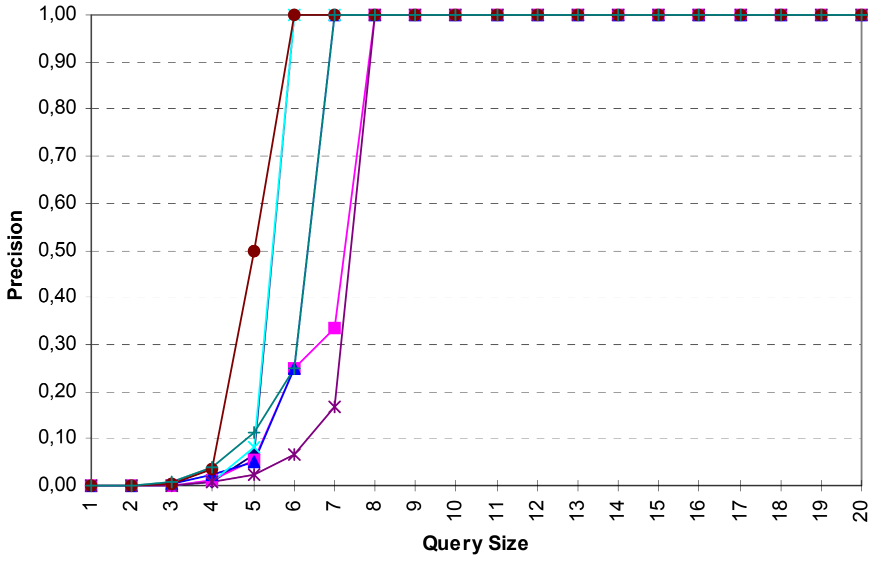


Figure A.36: 24-bit signature, 20-character word, 4 bits per word

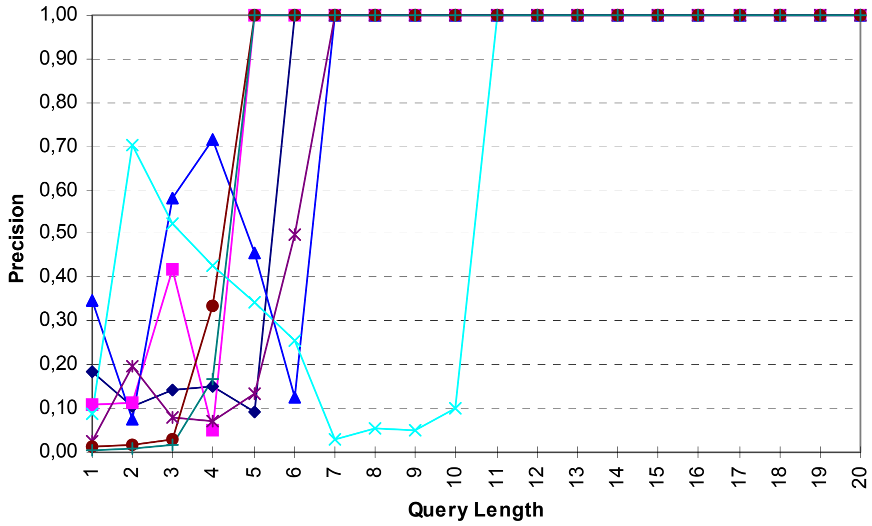


Figure A.37: 61-bit signature, 3-character word, 4 bits per word

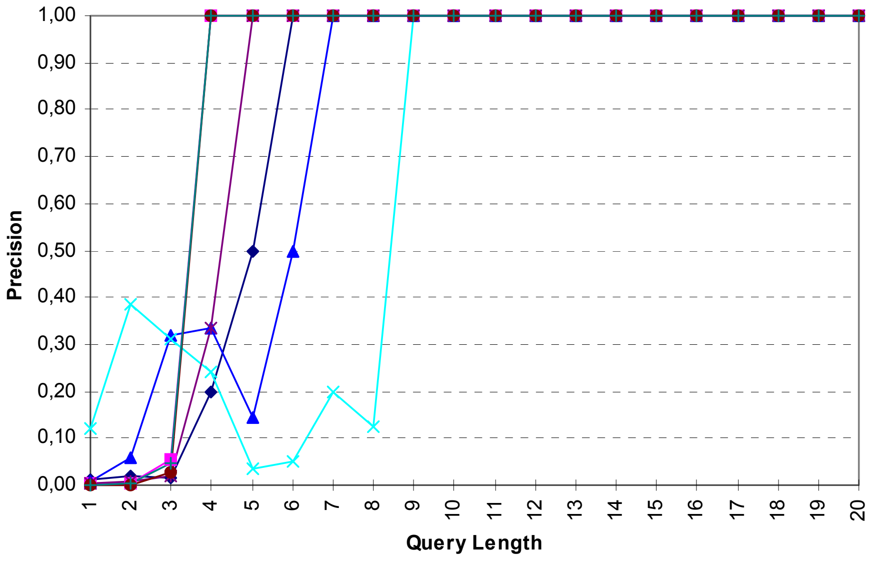


Figure A.38: 61-bit signature, 5-character word, 4 bits per word

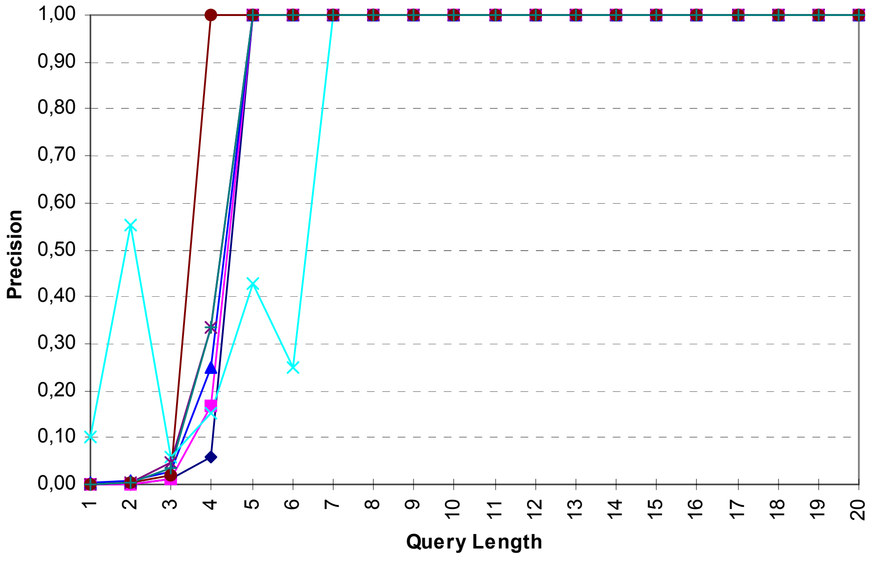


Figure A.39: 61-bit signature, 7-character word, 4 bits per word

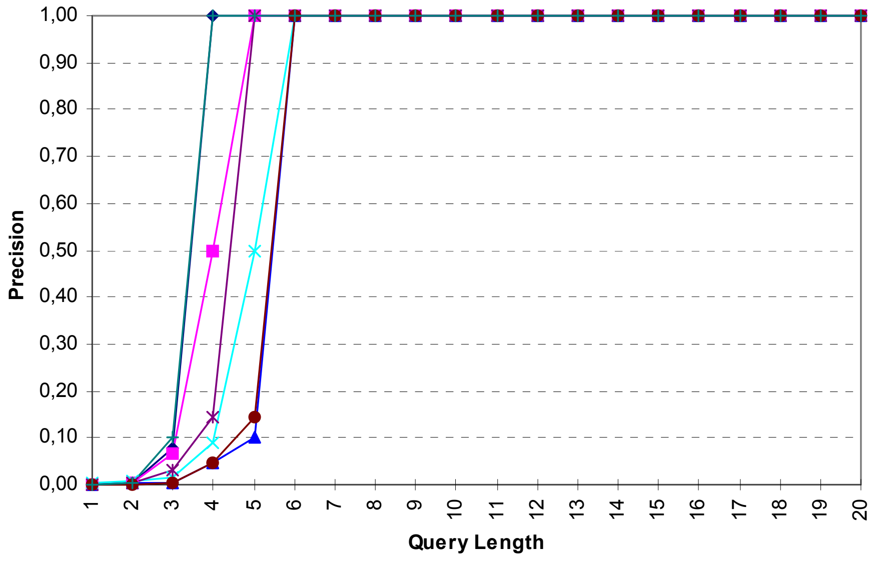


Figure A.40: 61-bit signature, 9-character word, 4 bits per word

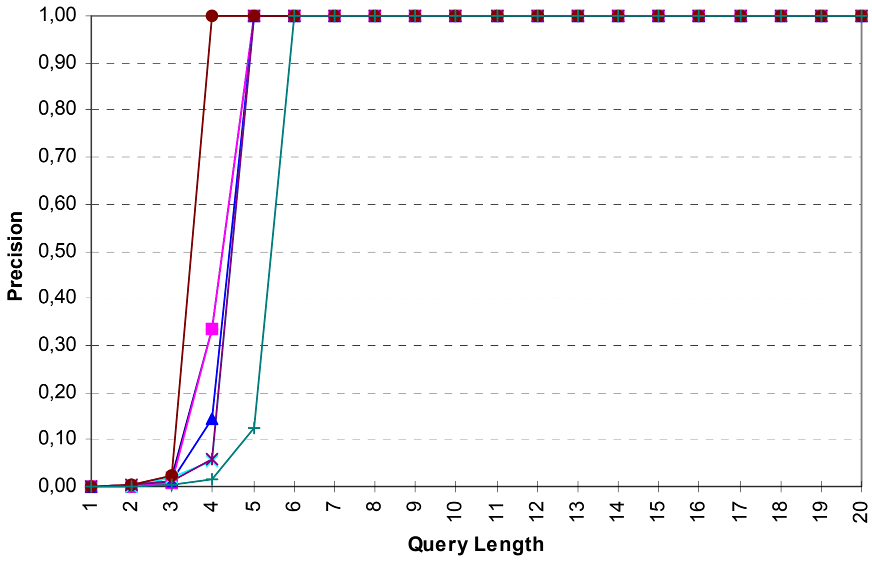


Figure A.41: 61-bit signature, 20-character word, 4 bits per word

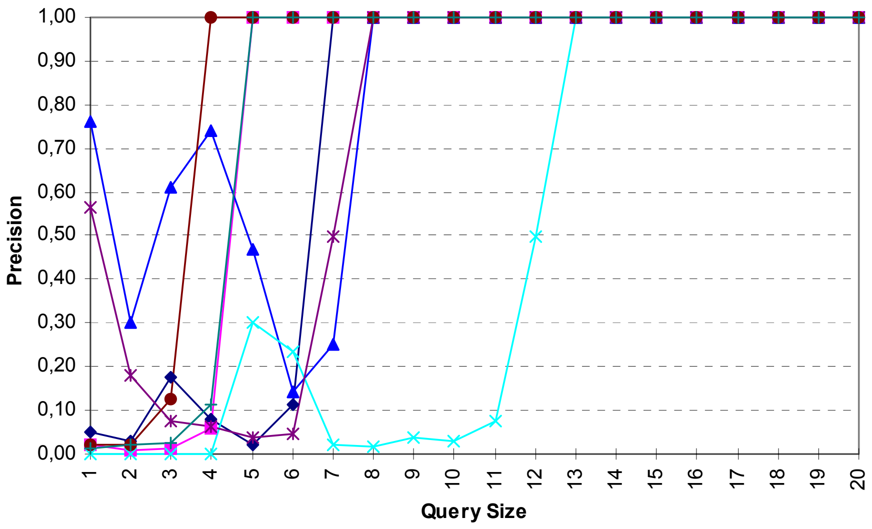


Figure A.42: 251-bit signature, 3-character word, 4 bits per word

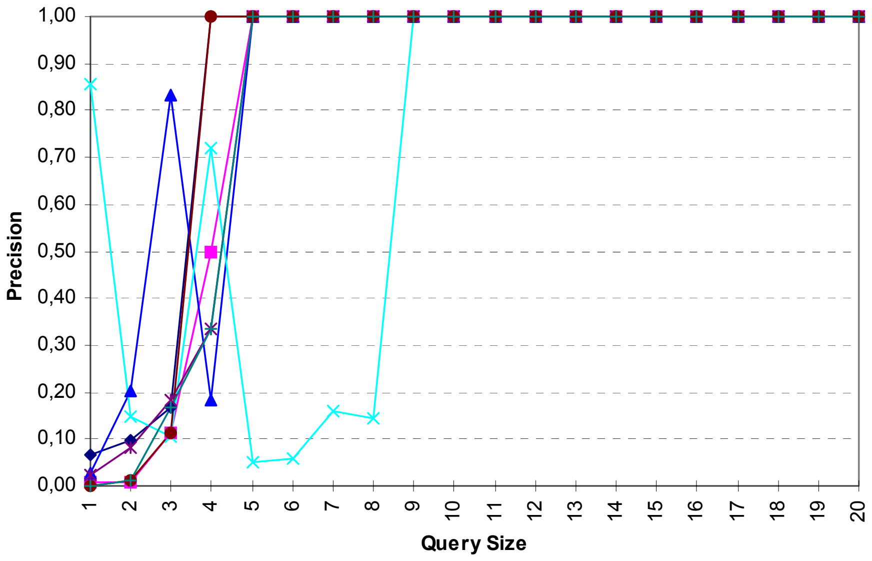


Figure A.43: 251-bit signature, 5-character word, 4 bits per word

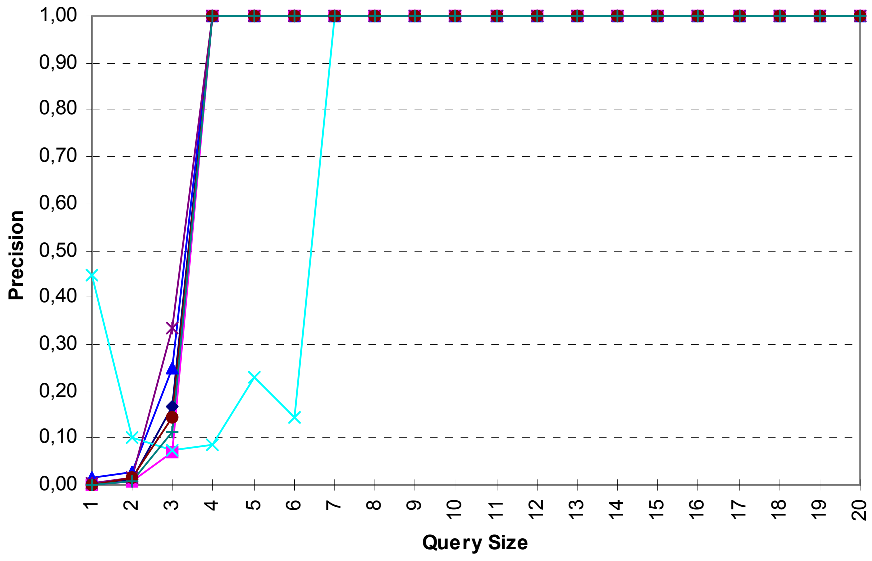


Figure A.44: 251-bit signature, 7-character word, 4 bits per word

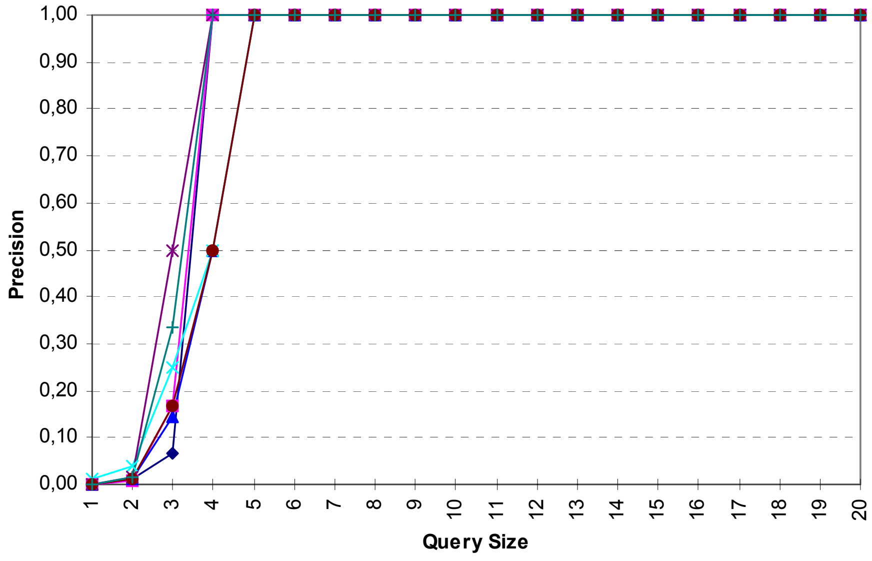


Figure A.45: 251-bit signature, 9-character word, 4 bits per word

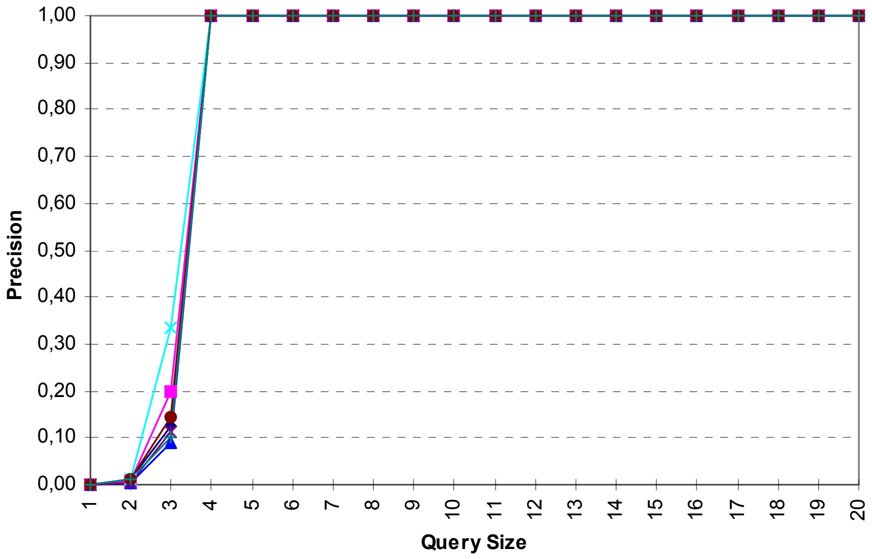


Figure A.46: 251-bit signature, 20-character word, 4 bits per word

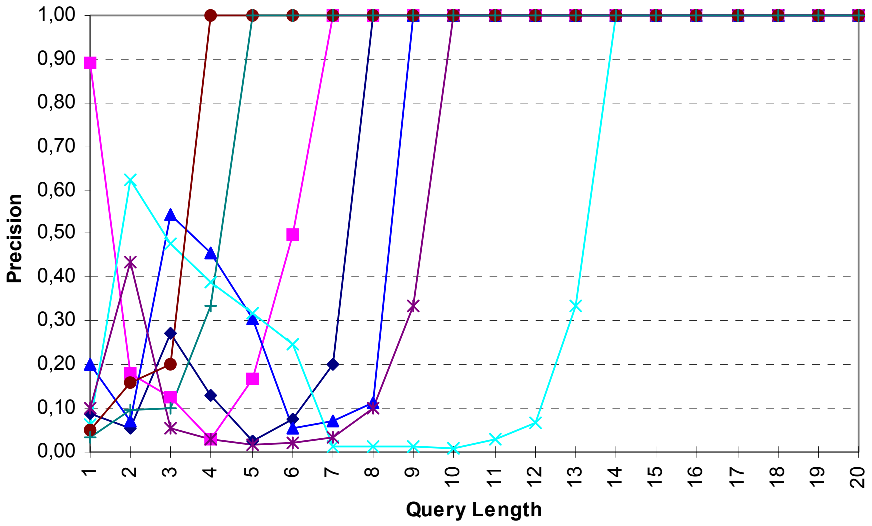


Figure A.47: 509-bit signature, 3-character word, 4 bits per word

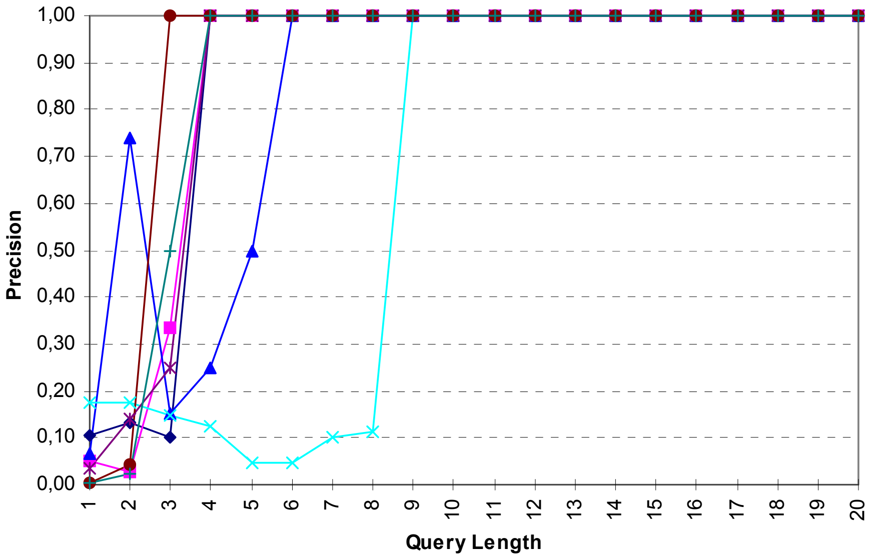


Figure A.48: 509-bit signature, 5-character word, 4 bits per word

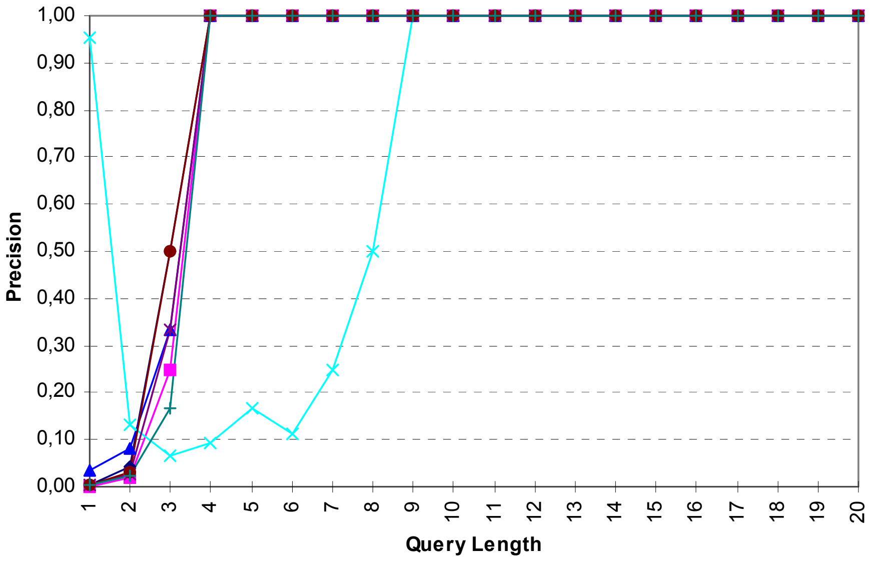


Figure A.49: 509-bit signature, 7-character word, 4 bits per word

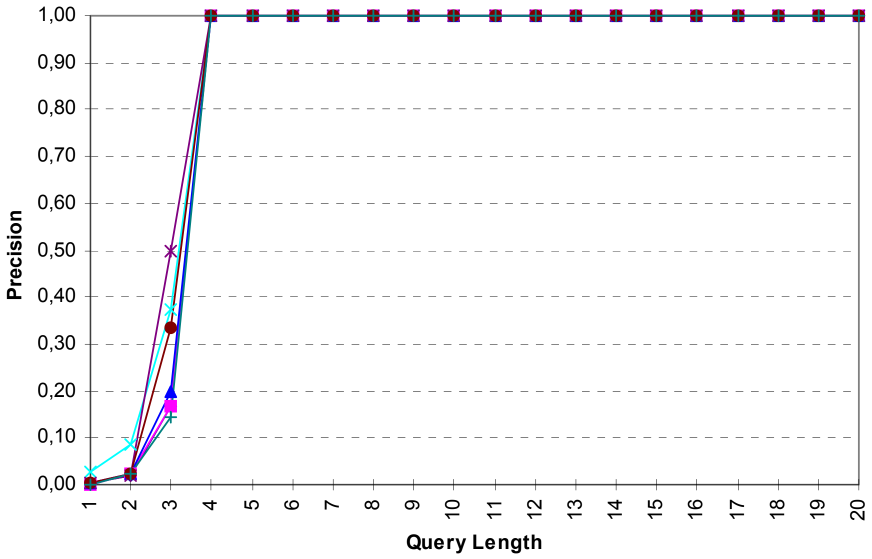


Figure A.50: 509-bit signature, 9-character word, 4 bits per word

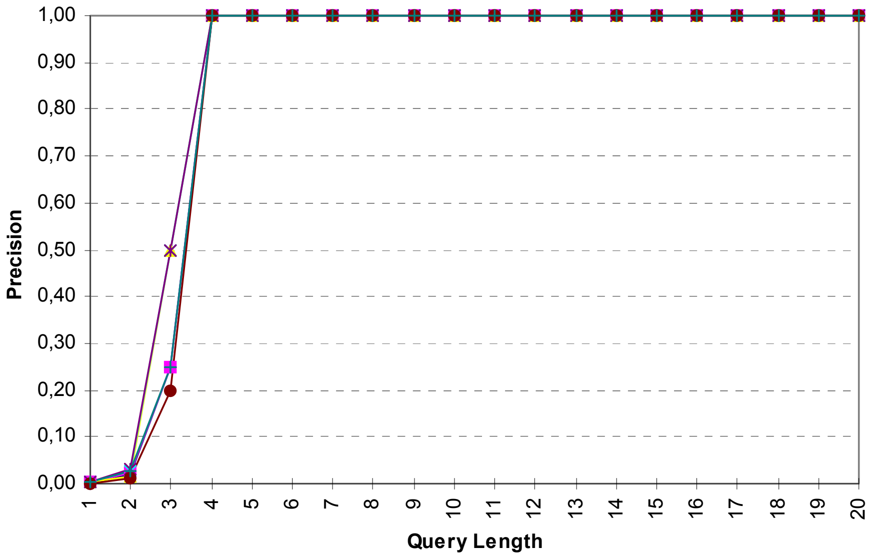


Figure A.51: 509-bit signature, 20-character word, 4 bits per word

Appendix B

The B-Tree Test Results

This appendix presents the different test results obtained while testing B-tree based implemented time series indexing systems.

In this appendix we present the different test results obtained from our tests with B-tree based indexing methods.

B.1 Test Results

We have tested two B-tree search algorithms. First we performed a plain B-tree search for the node that contains all sequences that might match the query. These elements would then have to be scanned if we needed to find an exact match to the query we stated. Second, we tested the intersection search algorithm.

As expected, the first approach leads to lower costs of searching the index, but the resulting number of occurrences is very large. If these results are compared to the results from the second approach, we can clearly see that the search cost is slightly higher, but the number of results is significantly lower.

B.1.1 The Test Setup

The tests consist of a large number of queries on the index. We have tested the index with queries from 7 characters up to 30 characters. For each query length the system has generated 100 random queries from segments found in the data file.

The tests have been conducted on four different data files, one with approximately 10^4 elements, one with $5 \cdot 10^4$ elements, one with 10^5 elements, and finally, one with $3 \cdot 10^5$ elements. The number of keys per page has been set to 200, a very low value, and the word size has been set to 7.

B.1.2 Plain Search Results

Figure B.1, Figure B.4, Figure B.7, and Figure B.10 show the cost of searching the index using a plain search algorithm. As can be seen in these figures, the average cost is given by the size of the data file being indexed, and matches the expected values given by Equation 7.13, on page 103.

This low search cost is, however, not the real search cost, since this simple search algorithm only searches for the first, in this case, 7 characters of the query. So if the query is longer than 7 characters, the result might contain references to occurrences that only match the first 7 characters of the query. If the query is longer than 7 characters we have to scan the resulting occurrences to find the true results of the query.

Figure B.2, Figure B.5, Figure B.8, and Figure B.11 show the number of results for a given query. As can be seen in the diagrams, the number of results for a query is independent of the query length. Since each result will probably not be close to another result in the data file, each result will require a disk access to verify that it is a real match.

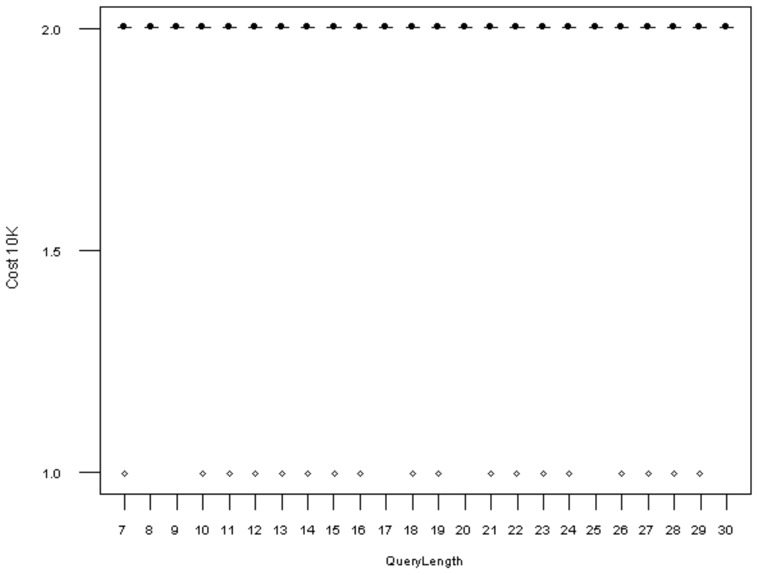


Figure B.1: Plain search cost for a 10K data file

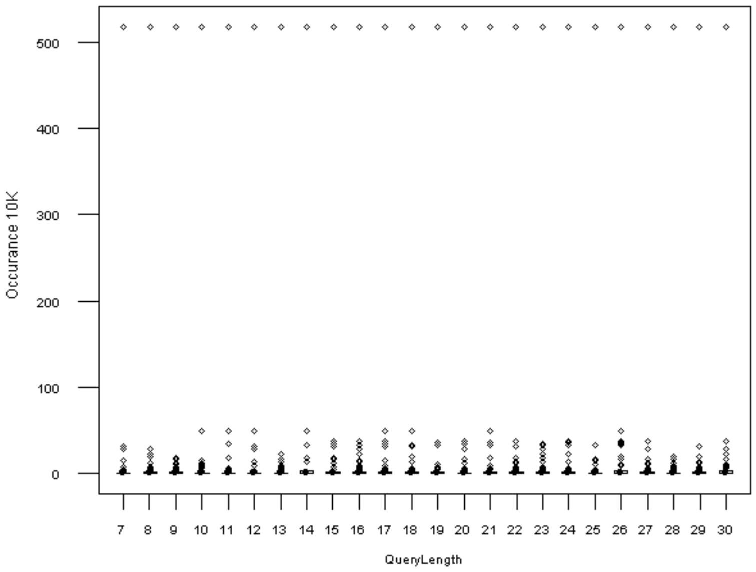


Figure B.2: Plain search occurrences for a 10K data file

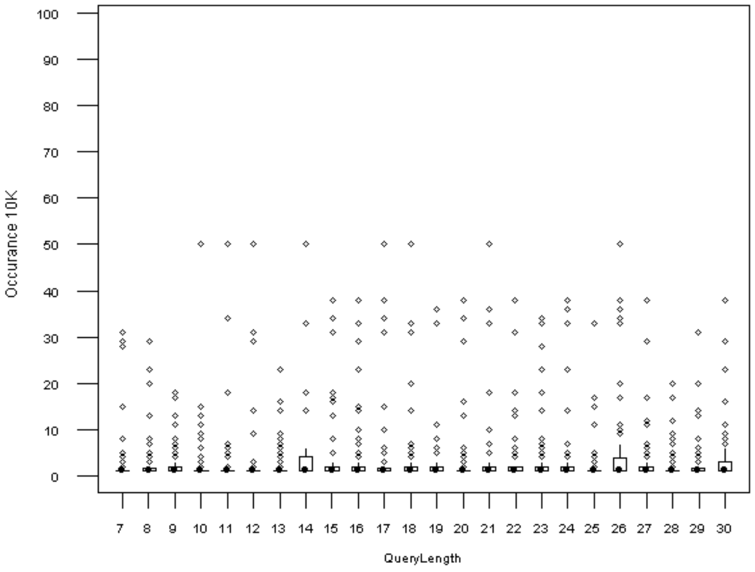


Figure B.3: Plain search occurrences for a 10K data file, zoomed.

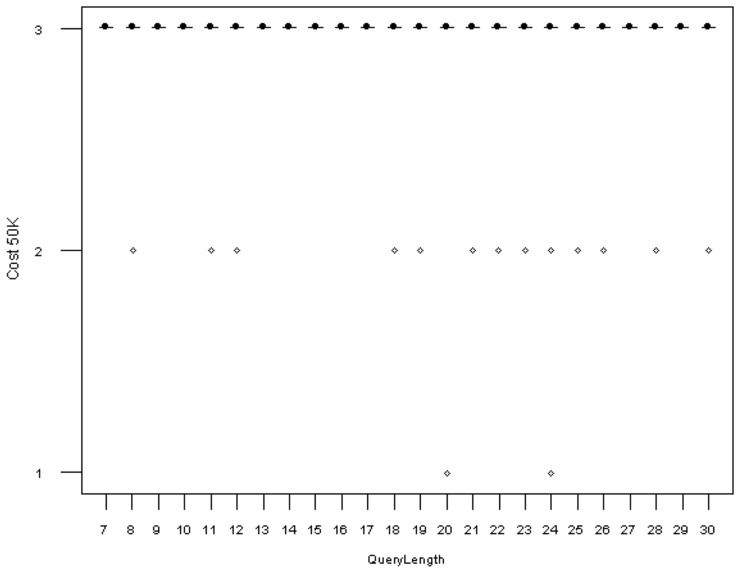


Figure B.4: Plain search cost for a 50K data file

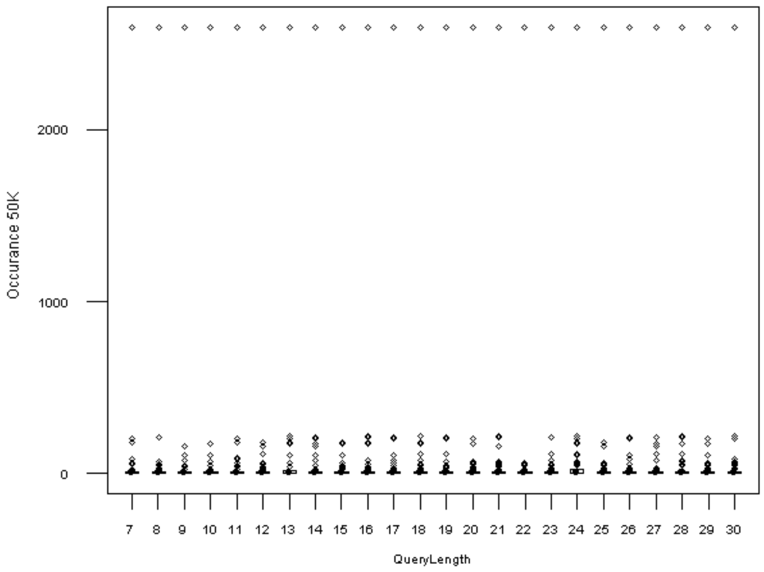


Figure B.5: Plain search occurrences for a 50K data file

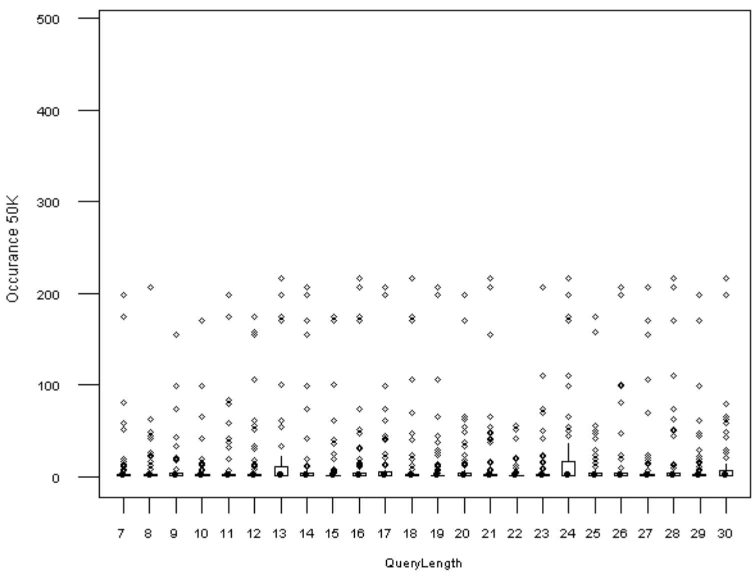


Figure B.6: Plain search occurrences for a 50K data file, zoomed

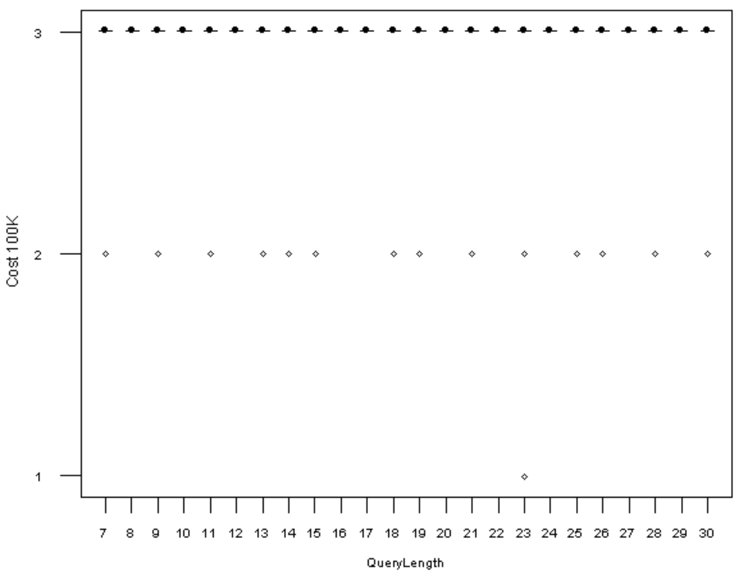


Figure B.7: Plain search cost for a 100K data file

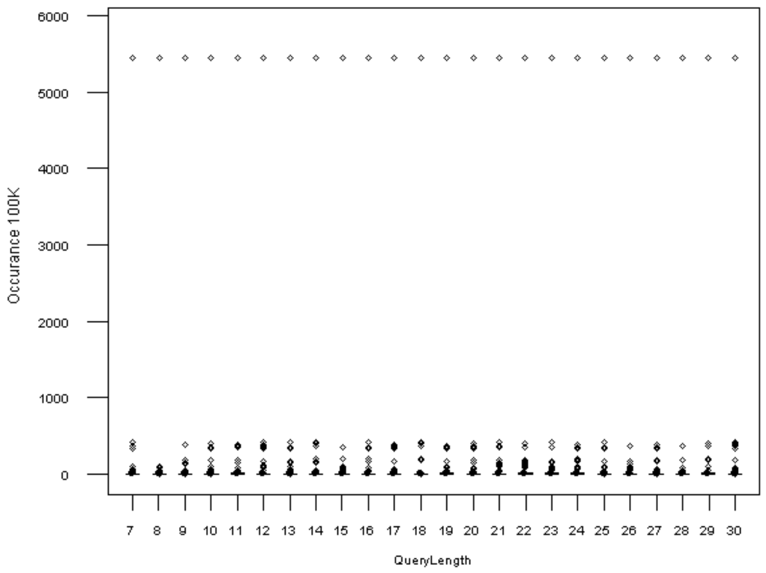


Figure B.8: Plain search occurrences for a 100K data file

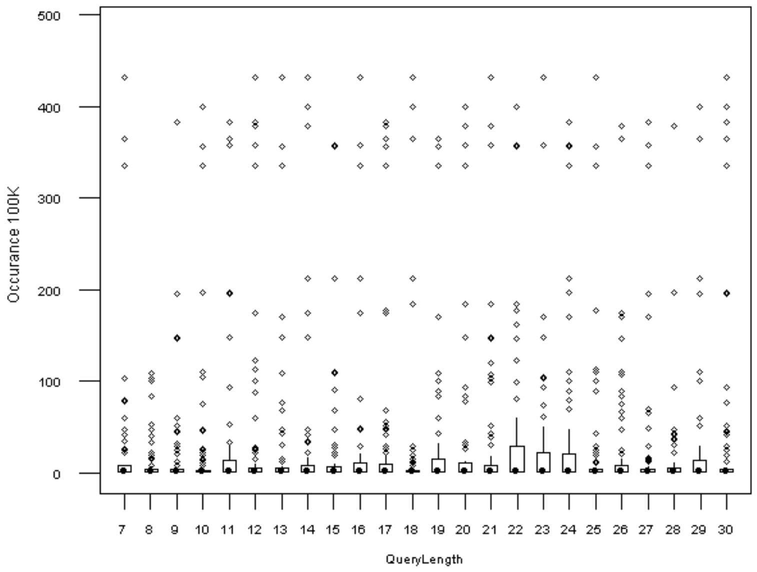


Figure B.9: Plain search occurrences for a 100K data file, zoomed

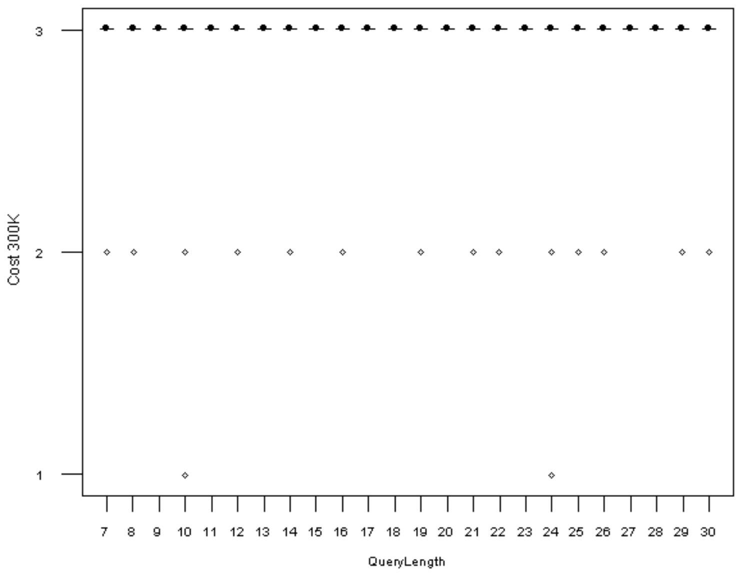


Figure B.10: Plain search cost for a 300K data file

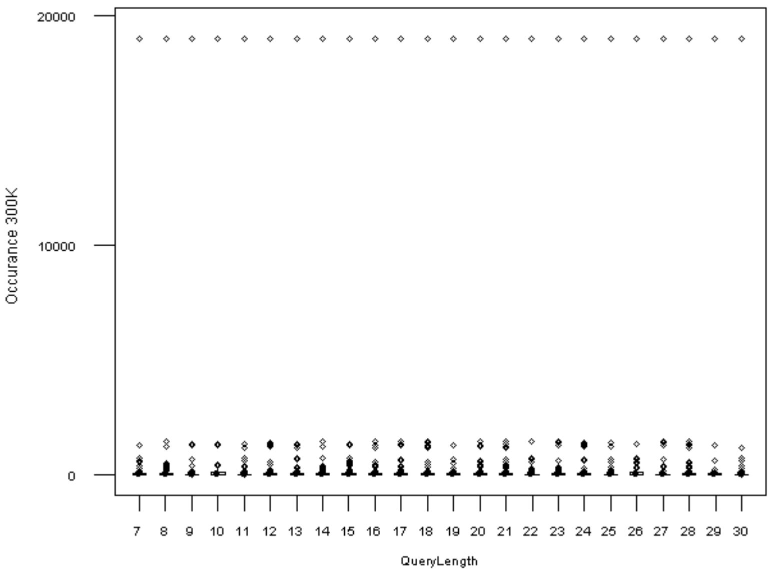


Figure B.11: Plain search occurrences for a 300K data file

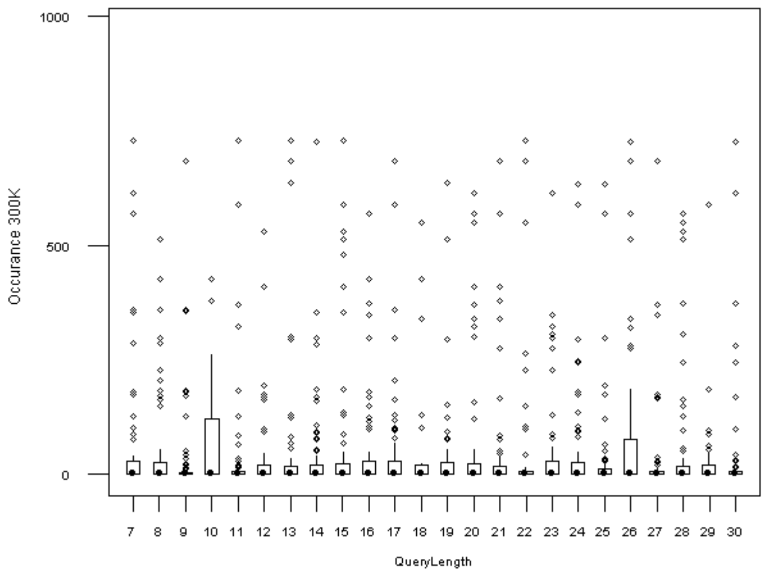


Figure B.12: Plain search occurrences for a 300K data file, zoomed

B.1.3 Intersection Search Results

Figure B.13, Figure B.16, Figure B.19, and Figure B.22 show that the cost of a more complex intersection search is higher than a simple search. The median cost is still the same as before, but for queries that are longer than 7 characters, some searches cost a few additional disk accesses since they might result in additional searches through the index.

The biggest differences can be seen in Figure B.14, Figure B.17, Figure B.20, and Figure B.23. For longer queries the number of results to a query are now always "true" matches to the query. As a result, the number of returned references are far fewer than the number of returned references with the plain search method, and they do not have to be scanned in order to verify that they are true hits.

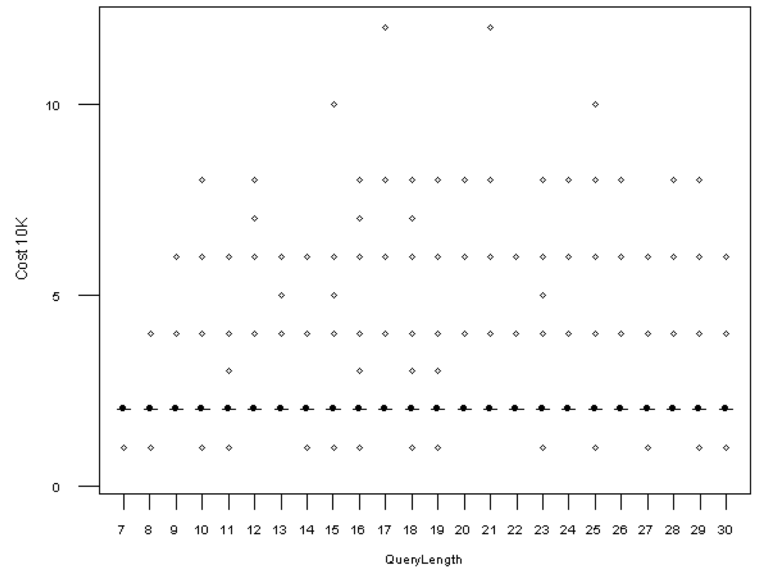


Figure B.13: Intersection search cost for a 10K data file

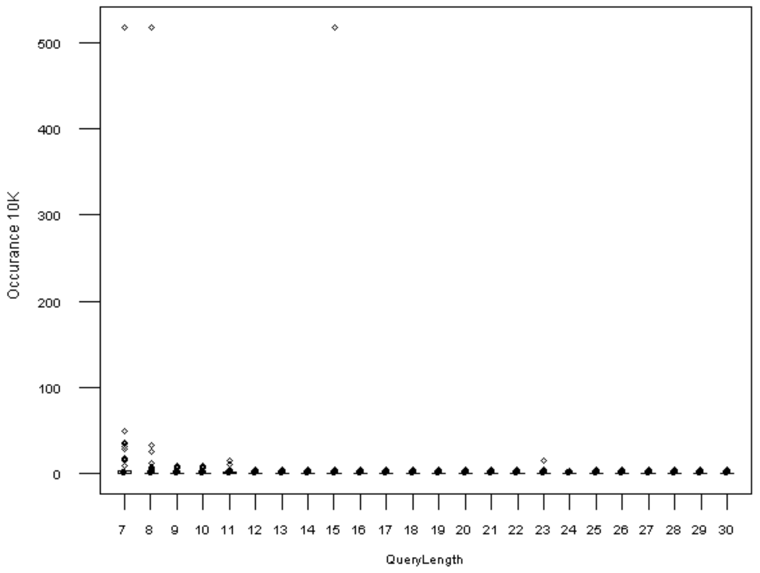


Figure B.14: Intersection search occurrences for a 10K data file

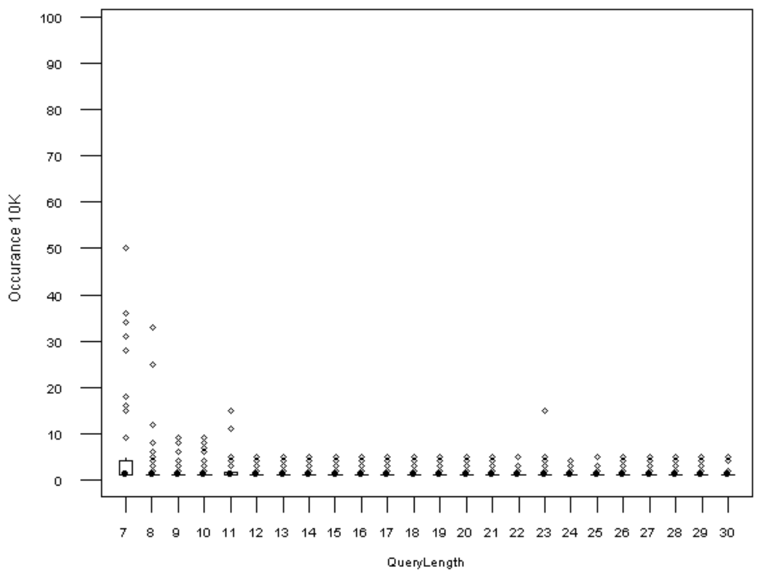


Figure B.15: Intersection search occurrences for a 10K data file, zoomed

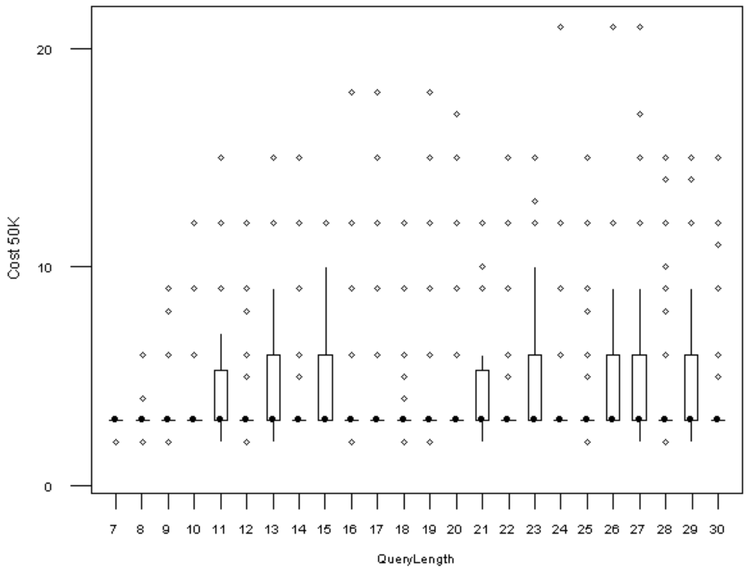


Figure B.16: Intersection search cost for a 50K data file

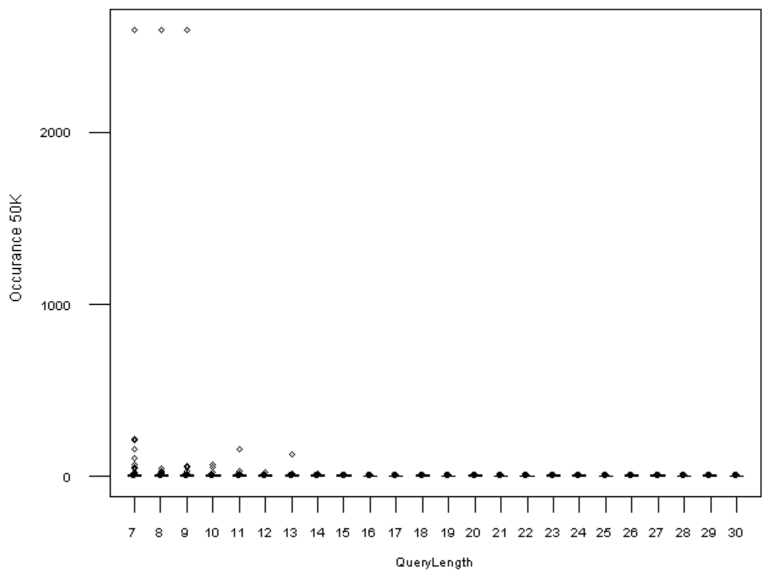


Figure B.17: Intersection search occurrences for a 50K data file

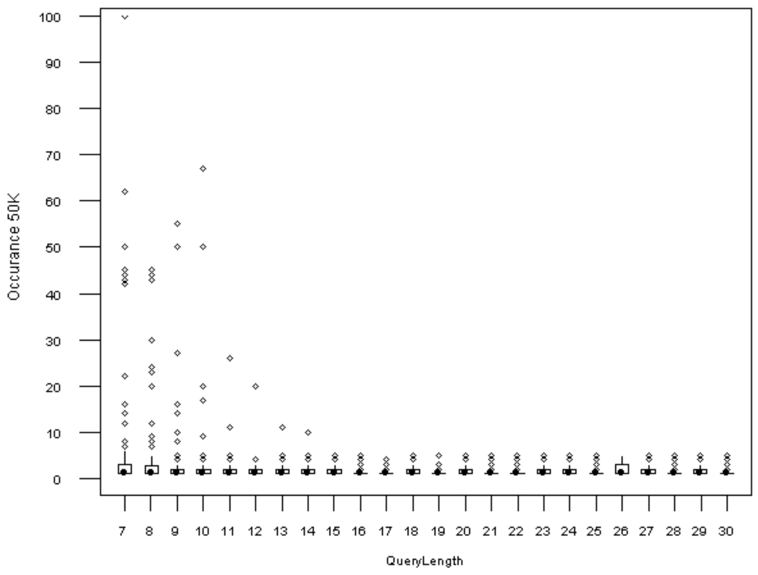


Figure B.18: Intersection search occurrences for a 50K data file, zoomed

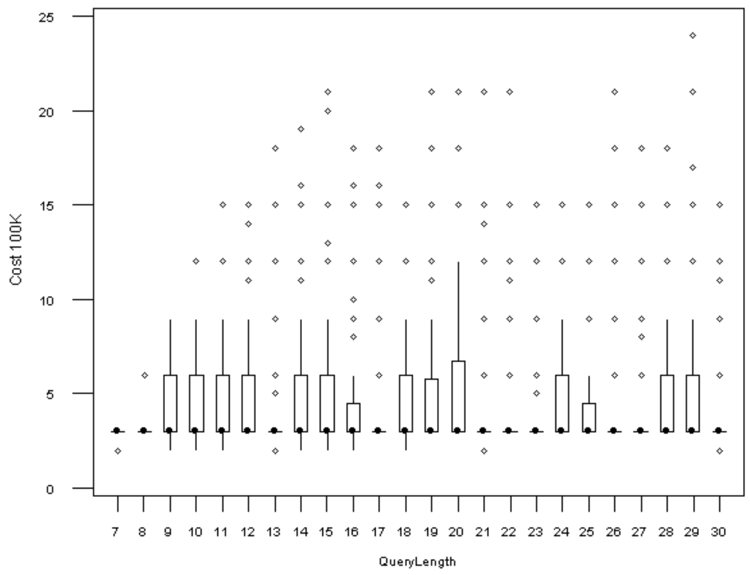


Figure B.19: Intersection search cost for a 100K data file

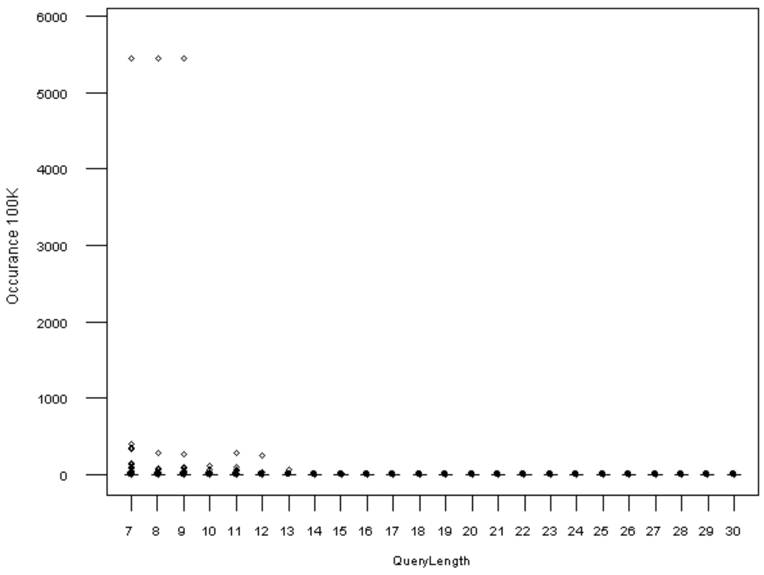


Figure B.20: Intersection search occurrences for a 100K data file

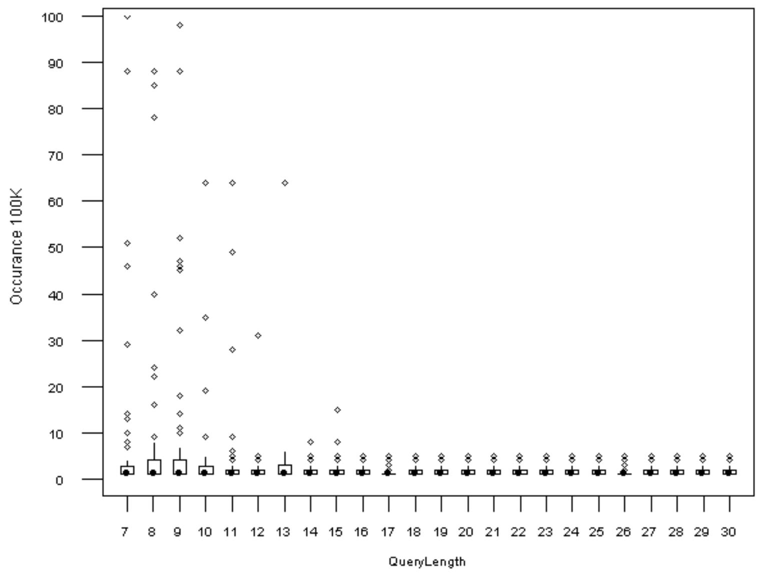


Figure B.21: Intersection search occurrences for a 100K data file, zoomed

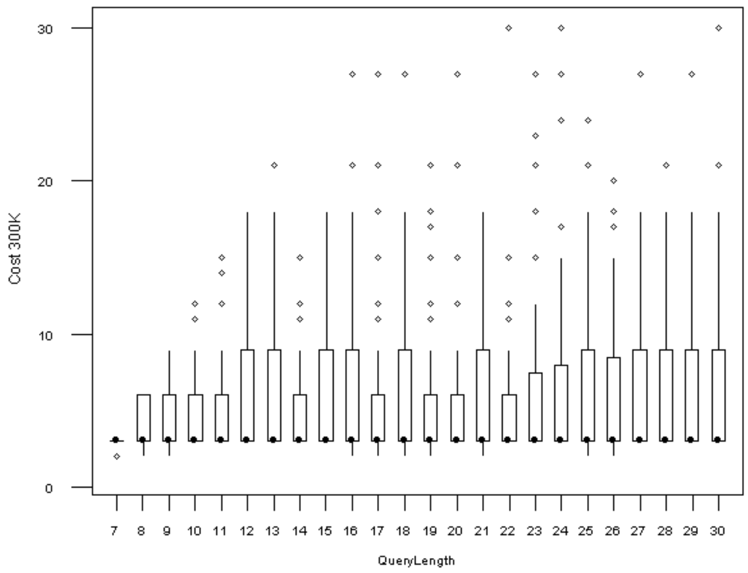


Figure B.22: Intersection search cost for a 300K data file

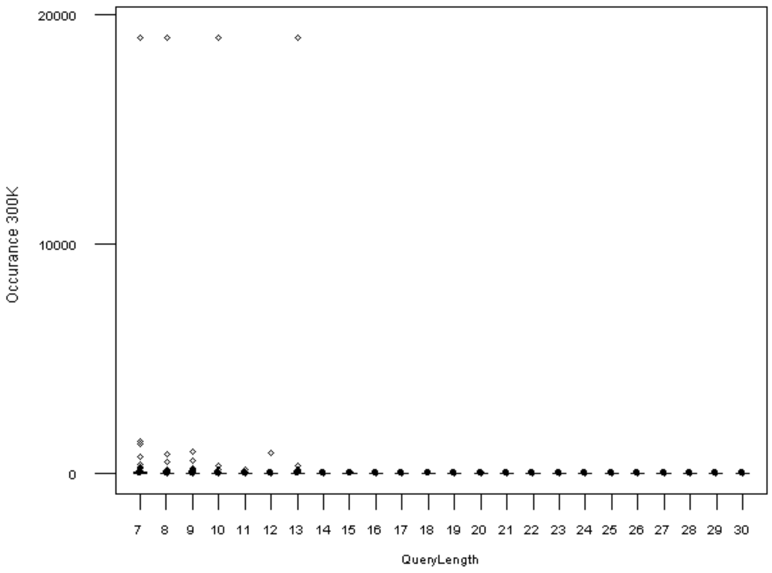


Figure B.23: Intersection search occurrences for a 300K data file

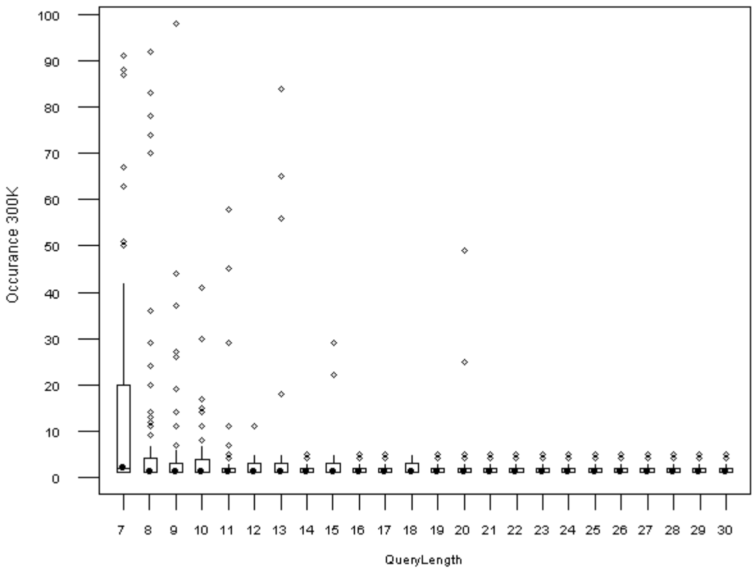


Figure B.24: Intersection search occurrences for a 300K data file, zoomed

Appendix C

PTrie Test Results

This section will describe the tests we have done using the PTrie and will present the results obtained from these tests.

In this appendix we present the results obtained from our tests with our PTrie-based index schema.

C.1 The Test Setup

The tests consist of a large number of queries on the index. We have tested the index with queries from 7 characters up to 30 characters. For each query length, the system has generated 100 random queries from segments found in the data file.

The tests have been conducted on four different data files, one with approximately 10^4 elements, one with $5 \cdot 10^4$ elements and finally one with 10^5 elements.

Figure C.1, Figure C.3, and Figure C.5 show the cost of scanning the PTrie for a query sequence. But since the index is a paged structure, and the only information contained in the index is pointers back to the data, we will always have to scan the page we end up in, unless we finished the search on a non-leaf page. Then all pointers contained in pages below that node will be “real” hits, otherwise the pages will have to be scanned for hits.

Figure C.2, Figure C.4, and Figure C.6 shows the number of occurrences returned by the index. Since the index pages only contain references to the original data file, we will always have to use the pointers to access the original data. The pointers on the page the search was terminated on are used to verify that the occurrences really match the query.

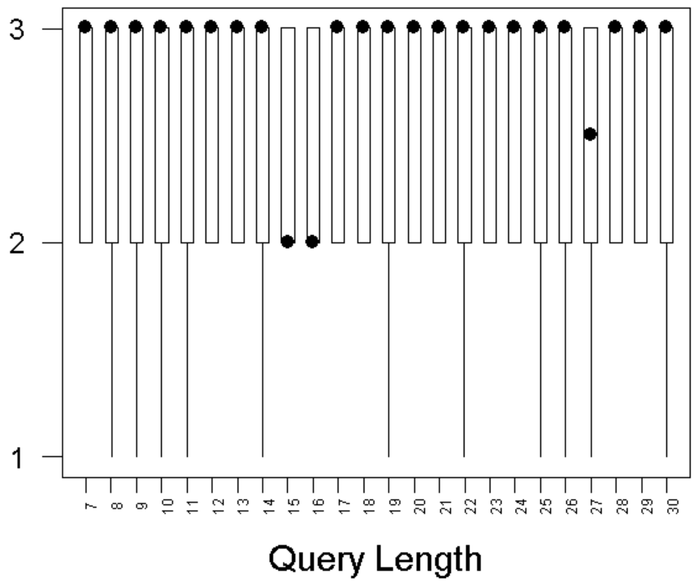


Figure C.1: Search cost for a 10K data file

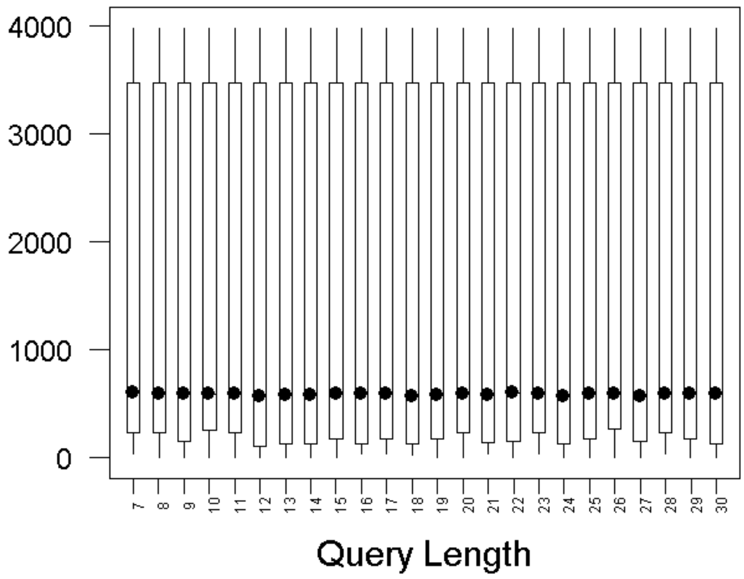


Figure C.2: Search occurrences for a 10K data file

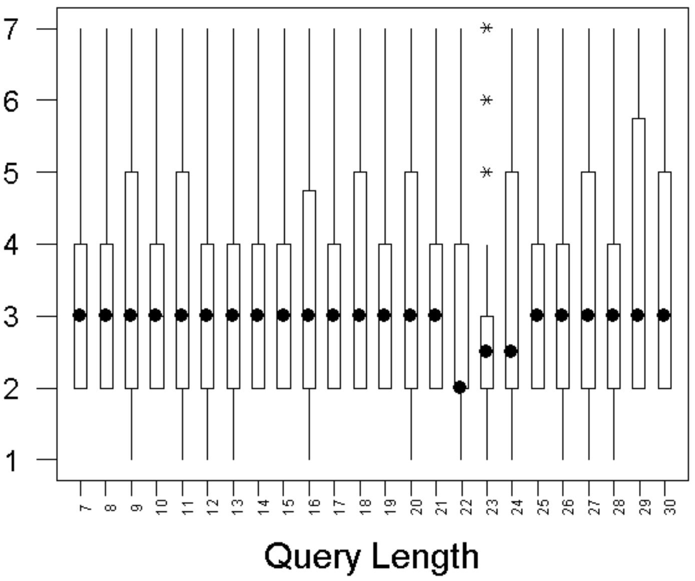


Figure C.3: Search cost for a 50K data file

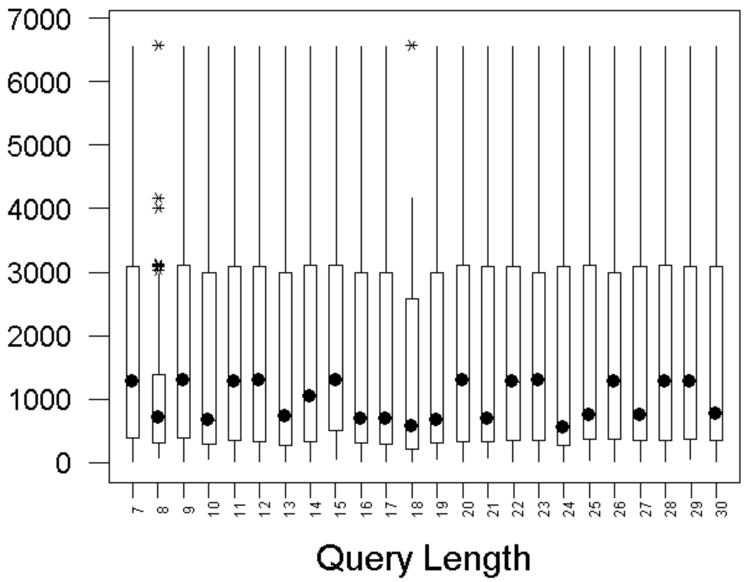


Figure C.4: Search occurrences for a 50K data file

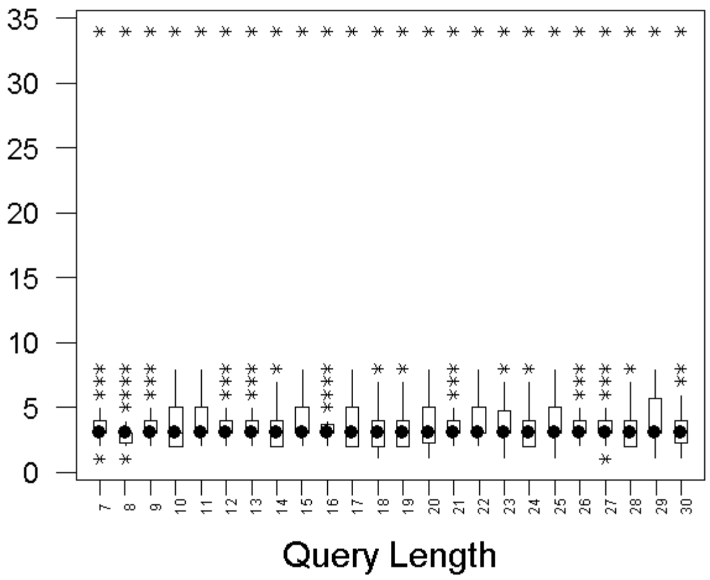


Figure C.5: Search cost for a 100K data file

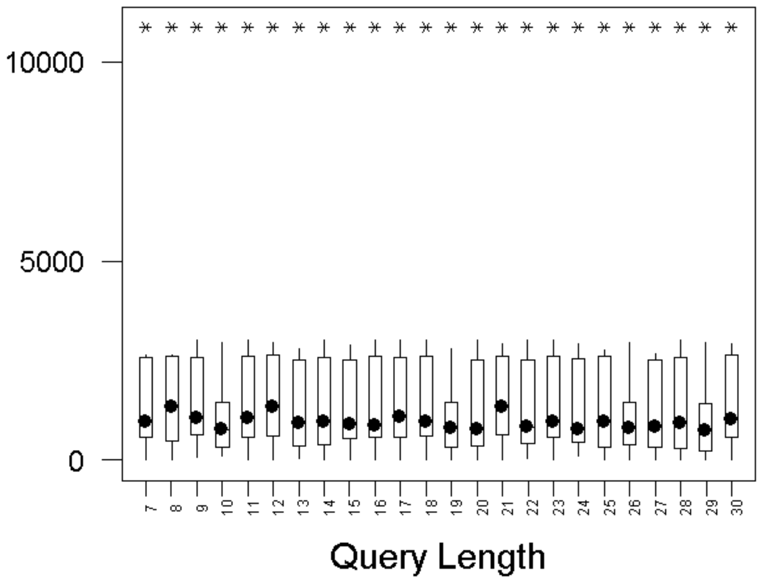


Figure C.6: Search occurrences for a 100K data file

Appendix D

Indexing

This appendix is intended for the readers that are not familiar with basic indexing schemes and contains a very brief introduction to indexing and various index structures commonly used today.

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. A brief introduction to the most common techniques will be presented here.

D.1 Ordering

The simplest approach to indexing 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" e.g. "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.

D.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 the lists in the open hashing grow too large or we have many collisions in closed hashing a new, larger, bucket array has to be created and the data has 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].

D.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 D.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.

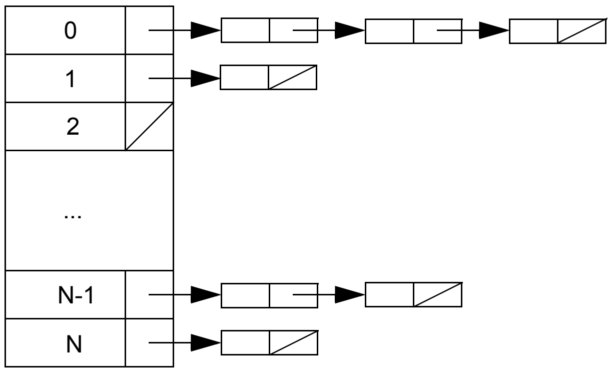


Figure D.1: Open hashing

D.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.

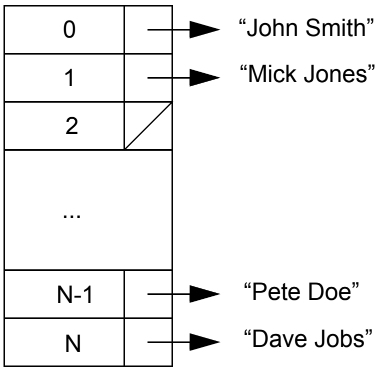


Figure D.2: Closed hashing

D.2.3 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 in 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 start-up 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.

D.3 Trie Structures

The trie structure was first proposed by [FRED60]. It is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level l represents the set of all keys that begin with a certain sequence of l characters called its prefix. The node specify an M-way branch, depending on the $(l+1)$ st character.

Figure D.3 shows a trie structure that indexes the five words HIS, HER, HE, HAVE, HAD.

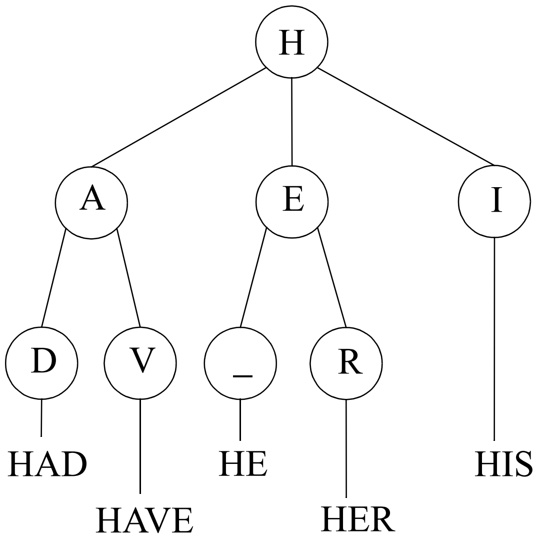


Figure D.3: Example of a Trie structure

D.4 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 the signature file, the index, is searched for signatures that match the query signature.

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). These different methods will be described in Section on page 193, Section D.4.2 on page 193, Section D.4.3 on page 194 and Section D.4.4 on page 194.

D.4.1 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 favoured than relevance-based retrieval methods because of the added search expertise needed to use Boolean queries. Signature access paths are not well suited to relevance ranking methods because of the document properties lost in the signature file processing.

D.4.2 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 D.4 on page 194 for an example).

Searching is done by hashing the query terms into a query signature and then matching it with the document signature.

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

Figure D.4: Example of word signatures

D.4.3 Superimposed Coding

In 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 D.5 on page 194 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 13.1, on page 194 is satisfied.

$$S_i \wedge S_q = S_q . \tag{13.1}$$

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

Figure D.5: Example of superimposed coding

D.4.4 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 D.6.

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

Figure D.6: Example of bit block compression

The signatures are then stored in the signature file in one of two concatenated ways shown in Figure D.7 and Figure D.8. In the first method, (Figure D.7), each group in the sparse vector is stored together, first part 1, then part 2 and finally part 3. In the second method, (Figure D.8), all part 1 bits are first stored, then part 2 bits and finally the part 3 bits.

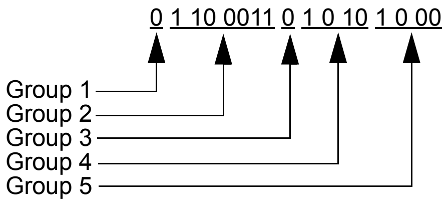


Figure D.7: Method 1 for storing BC signatures

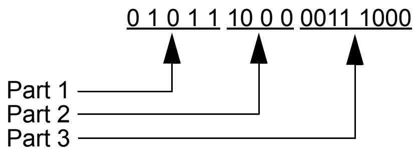


Figure D.8: Method 2 for storing BC signatures

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.

D.4.5 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], and 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 to 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 sub string 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.

D.5 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 run-time. Figure D.9 shows an example of a basic search tree.

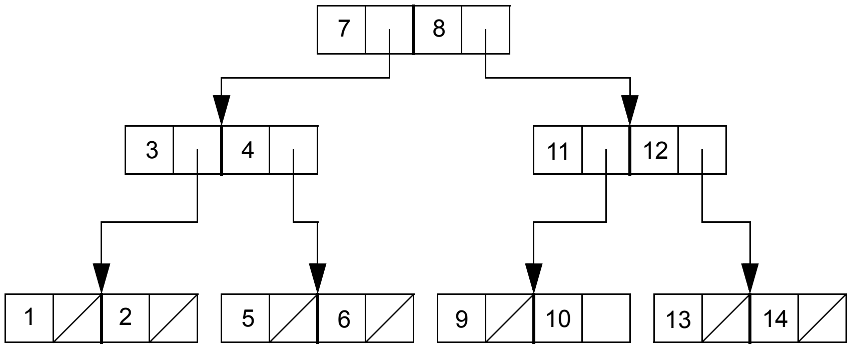


Figure D.9: A search tree

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.

D.5.1 Binary Search Trees

The most simple search trees are binary search trees. The tree in Figure D.9 is an example of a binary search tree. In a binary search tree each internal node has at most two children.

To keep the height of the tree from getting too high it is common to also include tree balancing functions.

Binary trees are simple to implement but have one problem. Since every node has at most two children the height of the tree grows to fast as more data is added to the tree.

D.5.2 B-Trees

A more advanced search tree that addresses these problems with binary search trees are B-trees [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. So it takes the same time for the computer to fetch one byte from secondary storage as an entire page, 4 kb to more than 64 kb depending on the system.

Each node in the tree 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 D.10). 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 leftmost 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.

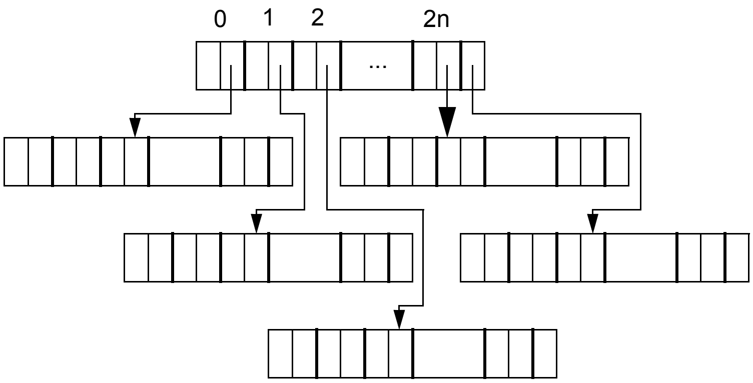


Figure D.10: A B-tree

Figure D.11 shows a small B-tree with two elements per page. Into this tree the sequence “8, 5, 1, 7, 3, 12, 9, 6” has been inserted. Each element has, in addition to the key value, a pointer to the actual data structure.

B-trees are discussed in more detail in Section 7, “The B-Tree Structure,” on page 95.

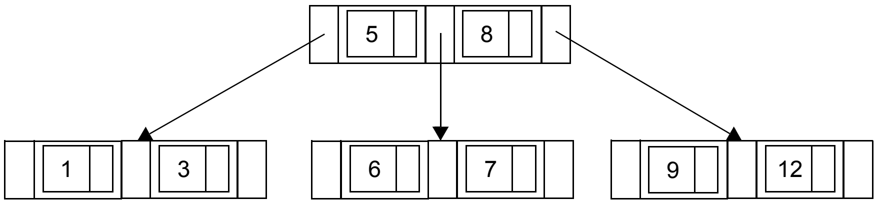


Figure D.11: A B-tree with the sequence “8, 5, 1, 7, 3, 12, 9, 6”

D.5.3 B^+-Trees

A small modification of the B-tree is the B^+-tree. In the B-tree each search key appears once at some level in the tree. In a B^+-tree datapointers are stored only at the leaf nodes. Furthermore, the leaf nodes are usually linked together so that ordered access can be provided.

Some search keys from the leaves are repeated in the internal nodes of the B$^+$-tree to guide the search.

Figure D.12 shows a small B$^+$-tree with two key values per page. In this tree the sequence “8, 5, 1, 7, 3, 12, 9, 6”, in that order, has been inserted, the same sequence used in the B-tree example (see Figure D.11). As can be seen the actual elements are all stored on the leaf level and some key values are repeated in the internal nodes to guide the search. As can be seen if these two figures are compared, the B-tree contains fewer nodes and if we e.g. search for the element with key value 5 only one node has to be fetched, compared to three if a B+-tree is used. What is the advantage of a B$^+$-tree? Since the leaf nodes are linked together it will be possible to perform very effective ranged searchedes, i.e. find all elements with a key value between 5 and 9. This would be very difficult to perform in a B-tree, but in a B$^+$-tree we only need to find the first element and then we can just scan the leaf nodes until we find the first element that does not match our criterion.

Furthermore, each internal node can contain more key values since only the key value is stored in the internal nodes, unlike the B-tree where each key value has to be accompanied by a data pointer.

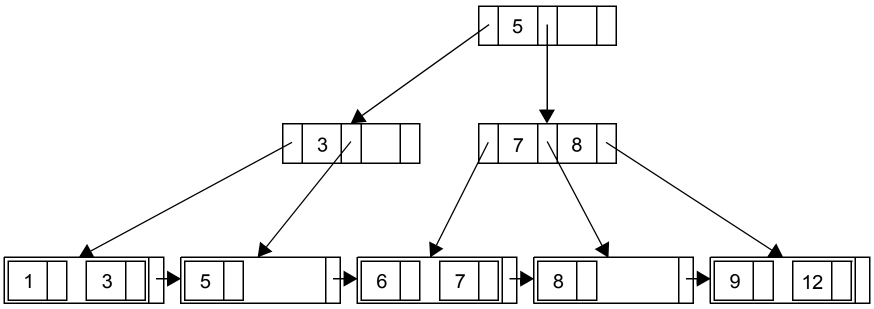


Figure D.12: A B$^+$-tree with the sequence “8, 5, 1, 7, 3, 12, 9, 6”

D.6 Inverted Files

Inverted files are a way of indexing large files. As with several other indexes the idea is to reduce the number of records that have to be fetched from disc [FOX92].

The inverted file index has two main parts, a search structure that contains all distinct keys of indexed elements and for each such distinct key, an inverted list containing references to the record containing the value.

An effective index structure for inverted files is a B^+-tree. The leaves in the B^+-tree will then contain pointers to inverted lists. Other index structures, e.g. Hash tables, can also be used for inverted files.

When indexing text using inverted files it is possible to compress the index by using variable length codes for the element keys. By using compressed inverted files the size of the index can be compressed by a factor of around six to around five to ten percent of the data size [BELL93]. By inserting a small amount of additional indexing information in each inverted list a large part of the decompression can be avoided, with the result that the limiting factor on modern hardware will be transfer times, not decompression time [MOFF96].

An interesting feature of compressed inverted lists is that the best compression ratio is achieved for the longest lists, i.e. the most frequent terms. Because of that there is no need to remove common terms from the index [ZOBE98].

Queries to the inverted file structure are evaluated by fetching the inverted lists for the query items and then intersecting them. The inverted lists should be fetched in order of increasing length to minimize buffer space. For a conjunctive query we start with the smallest inverted list and then remove elements from it as we go through all retrieved inverted lists.

D.7 Multidimensional Access Structures

These traditional access structures were created for indexing textual or numerical data. To index other kinds of data it is often necessary to have an index that can index multidimensional data. This section gives a short description of some of the most well known, but is far from complete. I have found references to more than 30 different multidimensional access structures.

D.7.1 K-D-Trees

The k-d-tree [BENT75] is a very early multidimensional index structure. It was not taken into account that the data being indexed would be placed on a paged secondary memory, and this structure is therefore less well suited for large spatial databases.

The k-d-tree is a binary search tree that represents a recursive subdivision of the d dimensional universe into subspaces by means of $d-1$ dimensional hyperplanes.

The hyperplanes are iso-oriented and their direction alternates between the d possibilities.

For example, for $d = 3$ the splitting planes are alternately perpendicular to the x-, y- and z-axis. Each splitting hyperplane has to contain at least one data point. See Figure D.13 to Figure D.15 for an example of how a number of points in a 2-dimensional space are entered into a k-d-tree.

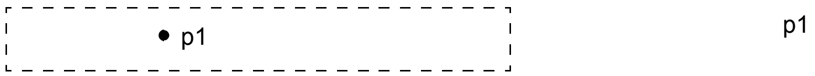


Figure D.13: K-D-tree first point

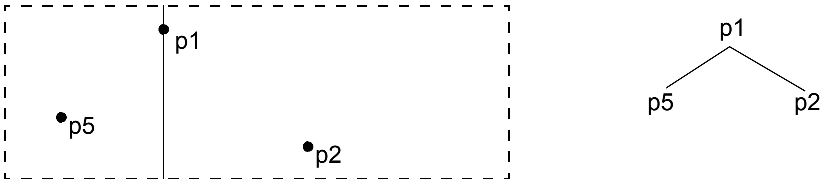


Figure D.14: K-D-tree third point

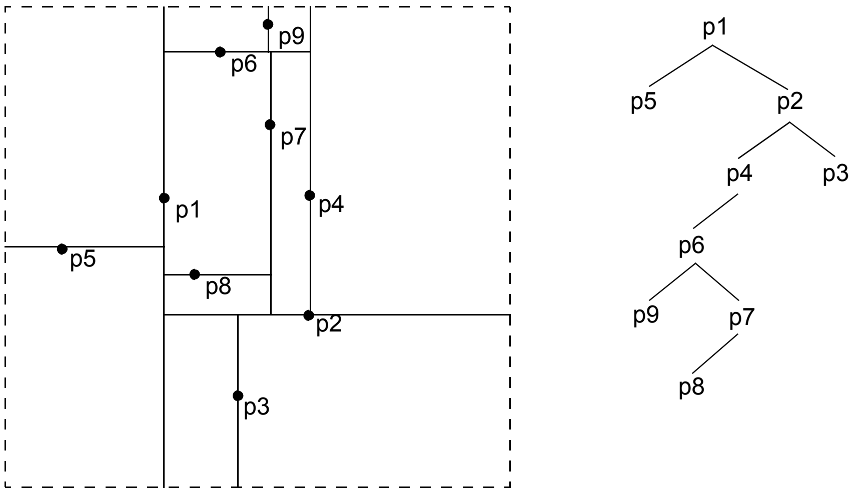


Figure D.15: K-D-tree third to ninth points

D.7.2 Quad Trees

The quadtree [SAME84, SAME89, SAME90] is a close relative to the k-d-tree. While the term "quadtree" usually refers to the two dimensional variant, the basic idea applies to a d-dimensional variant. Like the k-d-tree the quadtree decomposes the universe by means of iso-oriented hyperplanes. An important difference is that quadtrees are not binary trees. In a d dimensional quadtree each interior node have 2^d children. For a two-dimensional quadtree each interior node has 4 children, each corresponding to a rectangle. These rectangles are typically referred to as the NW, NE, SW, and SE quadrants.

D.7.3 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.

When the tree is searched each entry is checked to see if the stored MBR has a non empty intersection with the search MBR and all such entries have to be checked. In a worst case all index pages will have to be examined.

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

Figure D.16 shows a 2D region containing a number of MBR's (m1 to m8) and a number of points (p1 to p6). It also shows how the area have been divided into several recursive regions, R1 to R8. Figure D.17 shows the resulting R-tree.

There exist several improvements to the R-tree, e.g. the R*-tree, see Section D.7.4, “R*-Trees,” on page 204, or R+-trees [SELL87].

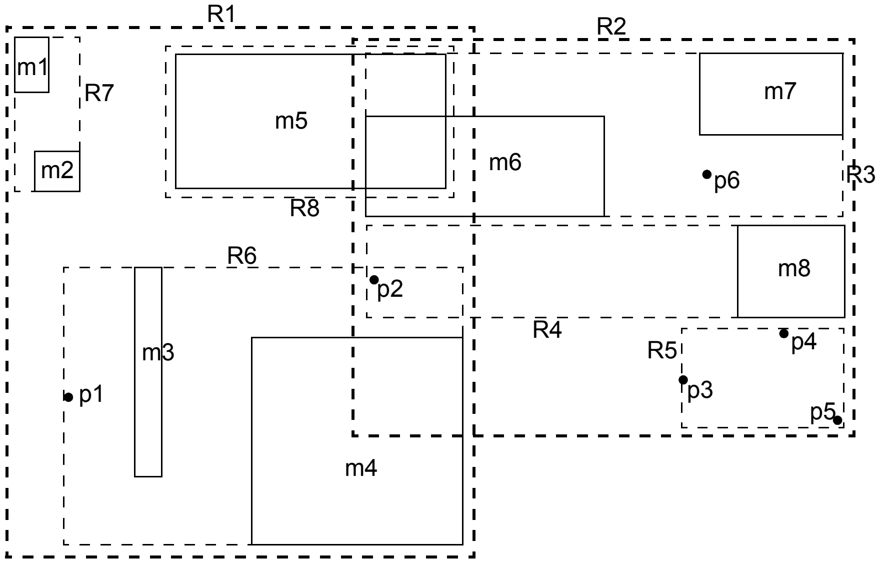


Figure D.16: Figure showing a 2D region

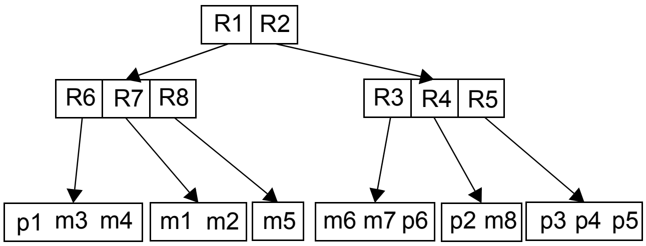


Figure D.17: Figure showing the R-tree

D.7.4 R*-Trees

Based on a study of the weaknesses of R-trees, an improvement of the R-tree was suggested in [BECK90]. Some of the problems addressed were that the search performance was very dependent on the insertion phase, the bucket overlap on each level in the tree should be minimized, and region parameters should be minimized.

The main difference between the R-tree and R*-tree algorithms lie in the insertion algorithm, deletion, and searching are virtually unchanged. With these changes the performance of the R*-tree is approximately 50% higher, but the higher search performance comes at a high CPU time overhead. [HOEL92, GÜNT97].

List of Figures

List of Tables