UNIVERSITY
of
GLASGOW

Department of
Computing Science

A Disk-Resident Suffix Tree Index and Generic Framework for Managing Tunable Indexes

Robert Philip Japp

Submitted for the degree of
Doctor of Philosophy
at the University of Glasgow

November 2004

Abstract

This thesis introduces two related technologies. The first is a disk-resident index for biological sequence data, and the second is a framework and toolkit for the management of operational parameters for applications of which this index is typical.

The *Top-Compressed Suffix Tree* is a novel data structure that can be used to provide a scalable, disk-resident index for large sequences. This data structure is based on the suffix tree, but has been designed to overcome the problems associated with using such structures on secondary memory. Top-Compressed Suffix Trees can be constructed incrementally, allowing indexes to be created that are larger than the amount of available main memory. Correspondingly, querying such an index only requires part of the data structure to be resident in main memory, thus allowing support for on-demand faulting and eviction of index sections during search. Such an index may be of great benefit to scientists requiring efficient access to vast repositories of genomic data.

The *Generic Index Development and Operation Framework (GIDOF)* is a framework and toolkit that supports various tasks relating to the management of operational parameters. The performance of an index's implementation is typically influenced by several operational parameters—parameters that must be tuned carefully if optimum performance is to be obtained. Indexes implemented using GIDOF can be structured in such a way that values of selected operational parameters can be adjusted; resulting in an index implementation that can be tuned to suit a given workload or system environment.

This thesis presents a detailed description of the design of both the Top-Compressed Suffix Tree and the algorithms that operate over it. Extensive performance measurements are then presented and discussed, covering such aspects of index performance as construction time, average query performance and the size of the completed index. An overview of the GIDOF parameter model and toolkit is then given together with examples of how this framework can be used to manage tunable indexes, such as the Top-Compressed Suffix Tree.

Thesis Statement

We propose that the *Top-Compressed Suffix Tree*, a carefully constructed disk-resident index, is a viable data structure for providing an index over large volumes of infrequently updated sequence data and that the performance of this index is competitive with, and in many cases better than, that of the suffix tree. Additionally, we demonstrate that successful deployment of such an index is dependent on carefully managing the set of operational parameters associated with its implementation. We propose that a general framework can be created to aid the processes of parameter management at all stages of the development and deployment of novel indexing technology. This framework will consist of generic components for use by the developer, together with tools for use by both developers and clients of the index technology.

Acknowledgements

I thank the following people for their support and encouragement throughout the preparation of this thesis.

- My supervision team, *Dr Richard Cooper*, *Prof. Malcolm Atkinson* and *Dr Rob Irving*, for helping to shape my ideas and introducing me to many new challenges.
- *Dr Ela Hunt* and *Dr Nigel Harding*, for sharing their knowledge of bioinformatics and indexing technology.
- The *Engineering and Physical Sciences Research Council (EPSRC)*, for funding my PhD studentship (award number 00305209).
- *The Royal Society*, for funding the equipment used in Chapter 4 (grant to Ela Hunt, *Indexes for Biological Sequences*).
- *Torsten Will* and *Jochen Scheel*, for inviting me to undertake an enjoyable internship at Exelixis Deutschland, Tübingen, during the summer of 2002.
- Those who I have shared an office with over the past four years, in particular *Gordon Cooke*, for many useful, and sometimes distracting, discussions.
- Finally, I thank my parents, *Robert & Marlene Japp*, for their support, and for providing an alternative source of funding during the latter stages of this research. *Thanks!*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Science is becoming increasingly data driven. Collections of digital data are now fundamental to almost all branches of science and play a significant role in the process of scientific research and discovery [9]. Although the analysis of empirical data gathered via observation has been a cornerstone of science for over a thousand years, the nature and sheer volume of data that can be generated by modern automated instruments has presented a significant challenge to those charged with the management and interpretation of the data. When *Computational Science* first emerged to complement the traditional empirical and theoretical approaches it was primarily concerned with simulation; however, this explosion in the volume of available data has seen information management take on an increasingly important role [46].

Projects such as the Sloan Digital Sky Survey [104] and the numerous genome sequencing efforts (the most famous example being the Human Genome Project[1]) are archetypal of this new direction in science. The primary outcome of such projects is to make the vast amounts of data generated available to the scientific community. For such projects to be successful, it is vital that data management issues are addressed and accounted for as an integral part of the project [44]. This raises questions not only about how best to publish and archive such data sets [25, 47, 43], but also on how to support efficient queries over the data set.

So why is querying scientific data challenging? Firstly, there is the volume of the data that must be processed. The volume of data produced from such projects ranges

[1]National Human Genome Research Institute, `http://www.nhgri.nih.gov/10001772`, as accessed August 2004.

from being multi-gigabyte[2] through to multi-terabyte[3] and beyond. Therefore, it is necessary to choose querying techniques that can scale accordingly—ad hoc solutions based on each user taking a local copy and using standard utilities such as 'grep'[4] will quickly become impractical. Secondly, the nature of both data and queries differ from that targeted by traditional database technology. The data is infrequently, possibly never, updated (thus support for efficient update operations is not necessary) and a greater emphasis is placed on inexact matching.

The nature of scientific data does, however, present an opportunity. With data that is rarely updated (we refer to such data as *reference data*) it is possible to invest computational time in the creation of specialised indexes that will be available for the lifetime of the data. The time taken to create the index can then be amortised over the useful lifespan of the data, as can any time spent tuning or optimising such structures. The use of such indexes may lead to the faster provision of results, more complete results and may also lead to better integration with database management systems. Providing a suitable index implementation, either as a component of the database software, or through using an appropriate technique for extending the functionality of the database, would allow the data to be both stored and queried within the framework of a database management system. This is preferable to situations where the data must be extracted from the database prior to being queried using a third party tool.

1.1 Sequence Indexing

Genomics[5] is one area of biological research that is becoming increasingly data driven. Vast repositories of genomic data are available to researchers, with one such repository, GenBank [16], currently containing over 28 GB of sequence data and growing exponentially.[6] Additionally, the volume of accesses to such databases has also been shown to be growing exponentially.[7] The challenge of providing efficient support for pattern matching over this data has prompted much research into the applicability of various

[2]Sequence repositories such as GenBank (`http://www.ncbi.nlm.nih.gov/Genbank/`) and the EMBL Nucleotide Sequence Database (`http://www.ebi.ac.uk/embl/`) typically contain many gigabytes of genomic data (both accessed August 2004).

[3]The volume of data used by the Sloan Digital Sky Survey in 2002 was 2.4 TB [43].

[4]The utility grep, typically available on most UNIX based operating systems, allows text files to be searched for arbitrary complex patterns.

[5]Genomics is the study of how an organism's genes relate to its biological function.

[6]GenBank Statistics, `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html`, as accessed January 2004.

[7]The Finished Human Genome—Wellcome to the Genomic Age, Sanger Institute Press Release, `http://www.sanger.ac.uk/Info/Press/2003/030414.shtml`, as accessed May 2003.

string searching techniques to biological data. The most popular techniques currently available for solving this problem are based on BLAST [4], a heuristic solution that relies heavily on parallelisation to achieve suitable levels of performance. However, such heuristic based approaches are not guaranteed to return all possible matches for a given query. Therefore, an efficient index based technique that is guaranteed to return all potential matches could be of great benefit to those working with sequence data.

A common form of biological sequence data is that of a one-dimensional listing of the nucleotides that comprise the DNA of the organism in question. By using a single letter to represent each of the constituent nucleotides, it is possible to view sequence data simply as a collection of one-dimensional character strings. Given the similarity between sequence data in this form and English text, it may seem that the provision of suitable indexes is a solved problem. After all, the indexing of English text is highly advanced [103], largely due to the popularity of storing such text in databases. However, this similarity is only superficial and the development of scalable indexes over sequence data is still in its infancy. This disparity is largely due to the nature of the queries that such an index must support and the fact that the data cannot readily be broken into separate words. Databases are designed for the management of large numbers of small data items and not small numbers of large data items, as is the case with genomic data. Here, techniques are required for querying the sequence data itself and not merely the meta-data associated with the sequence. In particular, researchers working with sequence data are interested in performing efficient *approximate matching* (see Section 2.3.1), whereas such queries are much less common over English text.

Sequence data is an example of reference data. It is derived experimentally, using various 'wet lab' techniques, and is not modified once published (although data sets can be superseded if more accurate or complete results are obtained). This property of the data suggests that bespoke indexing technology may be of use as we have an opportunity to invest computational time in creating specialised indexes.

One form of index that has often been proposed as a suitable choice for indexing sequence data is the *suffix tree* (see Section 2.4 and Gusfield [48] for a description of the suffix tree and its applications). The suffix tree is an attractive option as it can be used to solve efficiently a wide variety of bioinformatics related problems [49], and it can be constructed in linear time. Until recently, it has been assumed that such structures were too large to be stored in main memory and that they were unsuitable for construction on secondary memory [87]. Recent work has shown that by sacrificing the use of the linear time construction algorithm it is possible to construct suffix trees using secondary storage [58]. Although this technique has been successful in allowing the construction

of suffix trees of a size that exceeds the available main memory, it does have some limitations (see Section 2.4.4). In particular, the proposed construction algorithm is inefficient in some situations, and the chosen method of persistence is platform specific and incurs a substantial on-disk space overhead.

1.2 Thesis Idea

We introduce the *Top-Compressed Suffix Tree*, a novel data structure (based on the suffix tree) that indexes the suffixes of a given text in order to support efficient matching of target patterns. Unlike classical suffix-based data structures, namely the suffix tree [102, 81, 100] and the *suffix array* [80], the design of the Top-Compressed Suffix Tree is such that it can provide efficient support for both construction and querying when only part of the data structure is resident in main memory. This allows indexes to be created and used where the total size of the index exceeds that of the available main memory—an important property if we are to index large biological sequences.

The Top-Compressed Suffix Tree differs from the suffix tree in that the uppermost section of the index is represented using a collection of arrays rather than tree nodes. This allows for more efficient matching of the first few characters of each query and can provide a more compact index than is possible with traditional suffix tree representations. With the Top-Compressed Suffix Tree, it is intended to improve upon previous techniques for providing indexes over large volumes of sequence data. In particular, we extend the work of Hunt et al. [59, 58] by improving upon the measured performance of the prefix-partitioned suffix tree construction algorithm and by providing a compact platform-independent persistence solution for our structure, allowing current performance and scalability limitations to be overcome.

In addition to implementing and evaluating the Top-Compressed Suffix Tree, we aim to identify the techniques required to successfully manage the parameters associated with a bespoke persistent indexing solution. In order to explore, in some detail, the performance of this data structure it will be necessary to try numerous parameter combinations. Such experimentation will reveal how certain parameters affect index performance as data set and workload vary. This information can then be used to tune a given index for use in a specific environment. However, it would be unacceptable to expect a client of this index technology to have the same detailed knowledge of the implementation as the developer, potentially preventing the results of such experimentation from being fully exploited.

We introduce the *Generic Index Development and Operation Framework (GIDOF)*,

a framework and toolkit that can be used by both developers and clients of novel indexing technology to manage operational parameters. The aim of this framework is to allow the developer's detailed knowledge of how the operational parameters affect the index's performance to be more fully exploited. The developer can iteratively create an annotated XML model of all the operational parameters, with an in-memory representation being created when the application is invoked. As implementation (and subsequent experimentation) progresses, the model can be refined, with some parameters becoming fixed at a suitable value and others being determined by a set of rules to be evaluated at run time. Additionally, selected parameters can then be made available to the client in such a manner that, if desired, they can make an informed choice when tuning the index for a given workload.

1.2.1 Thesis Statement

We propose that the Top-Compressed Suffix Tree, a carefully constructed disk-resident index, is a viable data structure for providing an index over large volumes of infrequently updated sequence data and that the performance of this index is competitive with, and in many cases better than, that of the suffix tree. Additionally, we demonstrate that successful deployment of such an index is dependent on carefully managing the set of operational parameters associated with its implementation. We propose that a general framework can be created to aid the processes of parameter management at all stages of the development and deployment of novel indexing technology. This framework will consist of generic components for use by the developer, together with tools for use by both developers and clients of the index technology.

1.3 Research Method

In order to realise the ideas described in the previous section, and to confirm our hypothesis, it is necessary to provide implementations of each of the main concepts. Such implementations, together with ancillary programs, are required for a number of reasons. Firstly, they demonstrate that the concepts embodied within them are viable and that they can be implemented using the technologies specified. Secondly, they allow experiments to be undertaken, providing valuable empirical evidence, and they allow comparisons between technologies to be made. Finally, the availability of fully functional implementations is a key step towards the greater adoption of the concepts introduced in this thesis. The research methods employed here, and in particular the role of implementation and empirical evidence, are now discussed.

1.3.1 Top-Compressed Suffix Tree

The process of designing, implementing and evaluating the Top-Compressed Suffix Tree was preceded by a critical review of the available literature on suffix-based indexes. In particular, previous accounts of the use of such indexes in a persistent environment were examined in detail. Reflecting upon the limitations of previously reported techniques for providing such indexes the design of the Top-Compressed Suffix Tree and related algorithms was formulated.

Engineering

The Top-Compressed Suffix Tree was implemented using Java (see Section 1.4.1), as were all the programs discussed here. In addition to implementing the data structures and algorithms required for the Top-Compressed Suffix Tree, it was also necessary to provide implementations of more traditional suffix tree representations. This allowed various aspects of the performance of the different data structures to be compared. In addition, by comparing the results produced by executing various sets of queries over each of the differing implementations it was possible to confirm, within reason, the correctness of each implementation. Wherever possible, code was shared between different index implementations. This was to ensure that all observed differences in performance were due solely to the differences in data structure and algorithms. The implementations used in this work are listed below.

- *Top-Compressed Suffix Tree (persistent)* A Top-Compressed Suffix Tree implementation that can operate when only part of the data structure is resident in main memory.
- *Top-Compressed Suffix Tree (transient)* A main-memory only implementation of the Top-Compressed Suffix Tree. As this version is solely for use when the entire structure is resident in main memory, it will not have the overheads associated with the persistent implementation.
- *Suffix Tree (linear)* A main-memory implementation of a suffix tree constructed using Ukkonen's construction algorithm [100]. This implementation was derived from that of Ela Hunt [94], although it has been substantially re-engineered and optimised.
- *Suffix Tree (naive)* A main-memory implementation of a suffix tree constructed using the naïve suffix tree construction algorithm.

Gathering Empirical Evidence

Given the software described in the previous section, it is possible to address the first three parts of our hypothesis. By comparing various aspects of the performance of the three main-memory index implementations, it is possible to demonstrate that the performance of the Top-Compressed Suffix Tree can better that of the suffix tree. By using the persistent version of the Top-Compressed Suffix Tree, it is possible to demonstrate that this form of index can scale to accommodate large sequence files. As part of the process of carrying out these experiments, it will be required to tune our index implementations, thus the need for careful parameter management shall be demonstrated as a fundamental part of index development and tuning.

The empirical evidence gathered whilst exploring these aspects of the thesis will take the form of measurements of selected properties of the indexes when used on our system. Principally, these measurements will relate to the time taken to execute certain tasks and the amount of disk space required to store a particular version of the index. Clearly, such measurements are likely to vary with system characteristics; however, the analysis of experimentally derived evidence does have advantages over theoretical algorithm analysis (as is often used in the analysis of suffix-based indexes).

Algorithm analysis provides a convenient way of predicting the behaviour of an algorithm without implementing it on a specific platform [79]. The very purpose of such analysis is that in many situations it may be too time consuming to test every possible parameter combination, and that general indicators of performance can be used to decide upon the best algorithm for a given task. However, the assumptions made in such an analysis tend to be too simplistic to guarantee that the predicted results will correspond to real-world performance. For example, it is common in such analysis to assume that all accesses to Random Access Memory will have the same cost—an assumption that does not reflect the complexity of the interplay between memory accesses and caching on a typical modern computer. When using a computer with a memory hierarchy consisting of several layers of caching in addition to main memory, the time taken to access data stored in the CPU cache and data stored in main memory can differ by a factor of fifty [45]. This problem is exacerbated if the algorithm must operate in a persistent environment (where disk can be thought of as an extra layer of memory that is significantly slower than main memory).

Although algorithm analysis plays an important role in the development of indexing technology, careful engineering and experimentation will still be needed to explore how the algorithm performs on any given system. In cases where the index is likely to be long lived, it is worth spending time optimising the index for the system on which it

will run; with experimentation being the only technique currently available that can adequately address this issue.

1.3.2 Generic Index Development and Operation Framework

Having already demonstrated the importance of parameter exploration during the evaluation of the Top-Compressed Suffix Tree, the final aspect of our hypothesis is addressed through the design, implementation and use of the Generic Index Development and Operation Framework (GIDOF). Even though the tasks supported by GIDOF appear to be common, and are arguably a fundamental part of performance engineering, there is little available literature describing directly comparable systems. Therefore, the functionality and design of this framework will largely be guided by comparing the experience of implementing and tuning the Top-Compressed Suffix Tree to similar accounts of implementing novel index technology.

Engineering

In order to demonstrate that a framework such as GIDOF can be used for the tasks identified, implementations of the key components are provided. These components are: the parameter model, which is used throughout the framework to represent various forms of parameter; the XML representation of the parameter model, which is used to provide persistence for the parameter definitions and values; the developer's parameter tool, which allows the specification of the parameters for a specific technology; and finally, the client's parameter tool, which allows the client to alter operational parameters within the bounds set by the developer.

Given the implementation of the framework and toolkit described above, it is possible to demonstrate their functionality by providing examples of their use during the implementation and evaluation of the Top-Compressed Suffix Tree. By examining different stages of the index's implementation and operation, and showing how the relevant parts of the framework are used, we can illustrate the utility of such a framework.

1.4 Technologies Used

The implementation of the concepts and techniques described here made extensive use of two technologies, namely the *Java*[8] programming language and the *XML* mark-up language. These technologies were chosen for their wide range of applicable features,

[8] Java™ is a trademark of Sun Microsystems, Inc.

mutual compatibility and for the wealth of freely available tools that have contributed to their popularity.

1.4.1 Java

Java is a general-purpose object-oriented programming language [76], developed by Sun Microsystems, Inc.[9] Java programs are compiled in two stages: the first creates platform-independent byte-code from the source code at compile time; the second translates this byte-code into platform specific instructions at run time (typically using a *Just In Time* compiler).

Although it was originally conceived as a compact language for use on consumer devices with limited computing power, the range of features provided by Java proved to be attractive to developers working on many varied projects [92]. The language has since been augmented with a large set of standard classes, covering a range of facilities including remote method invocation (Java RMI—see below), relational database access (JDBC) and lightweight windowing (*Swing*, as used to implement the configuration tools discussed in Chapter 5). Such augmentations have led to the increased use of Java as a platform for the development of large-scale programs and server applications.

RMI *Java RMI (Remote Method Invocation)* is a technique that abstracts the communication between distributed systems to the level of method invocation [98]. This is similar to the *Remote Procedure Call (RPC)*, but with semantics akin to that of object method invocation. This package is used to implement the distributed version of the Top-Compressed Suffix Tree (see Sections 3.5.3 and 3.9.2).

1.4.2 XML

Extensible Markup Language[10] (XML) is a flexible document format (derived from the language *SGML*) that allows structured data to be stored using plain text [18]. XML has become the dominant means by which structured data is stored and transported when using Java [82]. This is largely due to the overlapping attributes of the two languages, for example, platform-independence and extensibility, but is also due to the extensive support for XML now provided with most Java installations. XML, together with XML Schema (see below), were used to provide a structured persistent representation of the parameter model introduced in Chapter 5.

[9] *Java Technology*, `http://java.sun.com/`, as accessed July 2004.
[10] *Extensible Markup Language (XML)*, `http://www.w3.org/XML/`, as accessed July 2004.

XML Schema *XML Schema*[11] is a schema specification used to describe the structure of XML documents. XML schemas, which are themselves written in XML, contain type definitions and element declarations that can be used to constrain and validate the XML documents to which they refer.

1.5 Contributions of This Work

This thesis presents two primary contributions. The first is a novel data structure, the Top-Compressed Suffix Tree—a structure that can be used to address the important problem of providing an efficient disk-based index for biological sequence data. In addition to the data structure, improvements upon previous partitioned suffix tree construction techniques are presented. This includes a partitioned construction algorithm with a superior average case running time to that of Hunt's algorithm [58] and a demonstration of parallel index construction.

The second contribution is the Generic Index Development and Operation Framework (GIDOF). GIDOF is a toolkit that supports the systematic management of operational parameters that affect the performance of bespoke indexing technology. This framework includes a comprehensive parameter model and toolkit. The reasoning behind using such a framework when implementing novel indexing technology is also of interest, and is arguably the third contribution of this thesis.

1.6 Related Publications

Four publications containing results presented here were produced during the course of this research. A summary of the main concepts contained in this thesis was presented at the 20^{th} British National Conference on Databases [64] and at the PhD Forum of the same conference [63]. The main results from Chapters 3 and 4 were presented at the Bioinformatics Workshop of the 21^{st} British National Conference on Databases [65], with a more detailed account presented as a technical report [66].

1.7 Thesis Outline

The remainder of this thesis is organised as follows. Chapter 2 presents a review of related indexing and database systems. The role of indexes within different persistence paradigms is explored together with a discussion of how each paradigm impacts upon

[11] *W3C XML Schema*, `http://www.w3.org/XML/Schema`, as accessed July 2004.

the process of implementing novel index structures. A detailed critique of suffix-based indexing is then given, highlighting limitations of current techniques and identifying opportunities for improving upon existing ideas.

Chapter 3 introduces the Top-Compressed Suffix Tree. Details of both disk-resident and memory-resident representations of the data structure are presented, as is the rationale for their design. Suitably modified versions of standard suffix tree construction and search algorithms are given together with a novel construction algorithm that can significantly improve performance in some cases. Furthermore, it is shown how this novel data structure can be used to provide a scalable index that supports incremental construction and operation when the size of the structure exceeds that of the available main-memory.

Chapter 4 explores various aspects of the performance of a Java implementation of the Top-Compressed Suffix Tree. Using biological sequence data as the test case, the performance of the Top-Compressed Suffix Tree is compared to that of two different suffix tree implementations. Index construction time, query performance and index size are explored in detail, showing that the Top-Compressed Suffix Tree is competitive with more conventional suffix tree implementations.

Chapter 5 introduces the Generic Index Development and Operation Framework (GIDOF). Reflecting upon the experiences of the design, implementation and evaluation work discussed in Chapters 3 and 4, the rationale for such a framework is presented. Details of the design and implementation of the framework are given together with a discussion of the parameter model that is central to the methodology encapsulated in GIDOF.

Chapter 6 discusses the parameter list associated with the configuration of the TCST implementation. The nature of the identified operational parameters is discussed, illustrating how the GIDOF parameter model can be used when implementing a tunable index.

Chapter 7 concludes the thesis. The preceding chapters are summarised, and the main contributions listed. Potential directions in which this research could be extended are given, together with a discussion of how such extensions may be achieved.

Chapter 2

Related Work

Index technology has played an important role in the continued development and success of database management systems. Indexes are required in order to allow efficient access to the vast quantities of data that can be stored in persistent systems such as relational databases, persistent object stores or even bespoke persistence solutions. In this chapter, the role of indexes in various persistent environments is reviewed together with a discussion of how such techniques can be adapted to support the indexing of new data types (such as genomic data). An introduction to the challenge of sequence indexing is then given together with a critique of existing suffix-based indexing techniques.

2.1 Persistence and the Role of Indexes

The term *persistence*, at its simplest, means the provision of storage and retrieval of data for as long as it is needed. In particular, this means the support for data values that have a longer life span than the computations that operate over them. Persistence technology can vary in complexity from simply storing data in text files, which will have to be read from or written to at the start and end of program execution, through to relational databases and persistent programming languages, both of which provide an abstraction to hide the complexities of the persistence mechanism. Managing data in a persistent environment is a significant engineering challenge, particularly when efficient support for complex queries is a priority.

An *index* is a data structure that can be used to locate efficiently items within a data set that match a given condition. Indexes are a vital part of most database management systems, and have been used to accelerate application performance in

many varied data processing environments. The main role of an index, which itself may reside on secondary storage, is to allow desired items in the database to be identified and accessed without the need to traverse the entire data set (which would be a costly operation when the data is located on secondary storage). This becomes increasingly important as the volume of data available continues to grow at a rate greater than that of the improvements in disk access times [45].

2.2 Persistence Models

Various models of persistence are available to developers. While they vary greatly in complexity, they all share a common purpose: that is, to simplify the management of disk-resident data. Common examples of such models are now discussed.

2.2.1 Relational Databases

Database technology is dominated by the *relational model*, first proposed by Codd [29] in 1970. According to this model, data is organised in tables, or *relations*, with all entries in a given column being of the same type. Querying this data consists of extracting one or more rows from the set of available tables: this is typically achieved by using a special-purpose query language, such as *SQL* [83, 34]. The indexes used to accelerate such queries consist of ordered pairs of keys and record addresses that, together with an appropriate data structure, can be used to identify the required records (rows) [34]. This abstract view of the data is used to hide the complexities of managing large quantities of disk-resident data from the application programmer, who can simply describe and manipulate their data using a proven set of mathematical relations.

Systems based upon the relational model, known as *relational databases*, first gained popularity during the 1970s and are now by far the most popular and widely used means of storing and manipulating large amounts of data [34]. The continued success of such systems suggests that this seemingly simple model of storing and manipulating data has met the data processing needs of many different projects. Consequently, the majority of research into effective indexes over large data sets has concentrated on relational databases (as discussed in the following section).

However, there is growing demand for applications that require the storage of large quantities of complex data: data that cannot be adequately represented using records and is often better served by object-oriented technology. Consequently, support for the queries that operate over this data, and mechanisms to accelerate such queries, will also be required. Examples of such applications include the highly connected spatial data

and indexes used in Geographic Information Systems [1] and the varied data processing tasks that come under the heading of bioinformatics. This has resulted in a greater interest in alternative persistence mechanisms as well as techniques that can extend the relational model (both of which are discussed later).

The 'Ubiquitous' B-Tree

The *B-Tree* [31] is the standard method for indexing within a relational database. This data structure, and its numerous variants, has long been associated with database indexing and has played an important role in the success of traditional database technology. The B-Tree is an example of a *search tree*[1] and is similar to the *binary search tree*, except that more than one key value is stored at each node in the tree. It can be used to provide an index over the key values of a collection of records, thus allowing efficient random access to individual records as well as supporting other operations, such as range queries. Typically, both the collection of records and the index will reside on secondary storage, with the on-disk layout of the index tuned to give the best possible performance on the host system.[2] When accessing a particular record, or range or records, the index is queried, causing the required nodes of the index to be brought into main memory. Once identified, the required records are simply read sequentially from secondary storage.

There are many variants of the B-Tree; each designed to optimise the index for certain given conditions. One of the most notable examples is the *B^+-Tree*[70]. In this variant, key values are only stored in the leaf nodes of the tree—the upper levels of the tree, which are organised as a B-Tree, serve as a 'road-map' to the actual values. The leaf nodes are then connected in an ordered linked list, giving the principal advantage of the B^+-Tree—it can be used to efficiently perform both random and sequential access. Another approach to improving the performance of the B-Tree is to alter the constraints on node capacity. An example of this is the *B^*-Tree* [70], where all nodes are required to be at least two-thirds full (as opposed to the B-Tree, where nodes are only required to be half full). This has the effect of making the tree more compact, allowing for faster queries. Variants of the B-Tree continue to be proposed, with concurrency being an

[1] A survey of search trees is given by Knuth [70], with a more detailed account of the B-Tree being given by Comer [31]. More recently, Hellerstein et al. introduce the *generalized search tree (GiST)* [54], an extensible search tree implementation that allows previously disparate data structures such as B^+-Trees[70] and R-Trees [50] to be implemented using a common data structure and set of operations.

[2] One of the most common optimisations associated with the use of search trees on secondary storage is using a node representation of a size that corresponds favourably to the page size of the host system. This allows the movement of nodes to and from main memory to be implemented using the native paging facilities of the operating system, thus minimising disk access overheads.

area of particular interest. (Recent examples include the work of Lim et al. [75] and that of Lomet [77].)

Although a hugely popular form of persistent index, the B-Tree is limited to indexing data that can be arranged in the form of key-value pairs, making it unsuitable for indexing many of the data types that may need to be stored in persistent systems. For example, while it is possible to create a B-Tree over textual data (the prefix B^+-Tree [14] being the most common example), the resultant index can only be used to solve a restricted set of problems. Such indexes are provided as a means of searching for key values that happen to be character strings and are not well suited for searching within a longer text. Providing support for other forms of index will continue to be an important part of exploiting database technology for the management of domain specific data.

Extending the Relational Model

Several techniques have been put forward for extending the range of data types supported by relational databases, thus allowing domain, or application, specific data types to be stored and retrieved using the database. One simple extension is allowing arbitrary data to be stored as a *Binary Large Object (BLOB)*—a field that can be used to store any form of binary data. This allows application specific types to be stored in the database, but it does not, however, allow these fields to be queried or indexed—tasks that are deferred to the application. Although BLOBs allow a wide variety of data to be stored in a relational database, they are primarily intended to support streams of digitised information, such as audio or video data, and do not provide support for more complex structured data types.

A more comprehensive approach to extending the range of data types supported by relational database technology can be achieved through using systems that allow some use of object-oriented data types. Such systems have become known as *object-relational databases* [34] and aim to allow complex data to be stored in databases while retaining the benefits of the robust and highly tuned implementations of the relational model. Complementary to object-relational databases are frameworks to allow the implementation of client-defined data types, queries and indexes. An example of such a framework is the support for *DataBlade*[3] modules within the Informix[4] product range.

[3]Informix DataBlade Modules, `http://www-306.ibm.com/software/data/informix/blades/`, as accessed August 2004.

[4]Informix Product Family, `http://www-306.ibm.com/software/data/informix/index.html`, as accessed August 2004.

Such frameworks are vital if novel indexing techniques are to be exploited within relational database environments (see below for an example of using this framework to implement a novel index).

Example Systems

Two examples of how the DataBlade framework can be used to implement novel indexing technology are: the work of Bliujute et al., who implement an index for bi-temporal data [19] (the index is based on the *R-Tree* [50], and is named the *GR-Tree*); and the work of Kornacker [71], who describes an implementation of the *generalized search tree (GiST)* [54]. Both indexes are examples of search trees, with the latter being particularly suited for use with such a framework as both index and framework provide support for an extensible set of data types and queries. In both cases, the authors report that index performance is comparable to, or better than, that of the built in mechanisms for achieving the same results.

Although the projects discussed above were successful in allowing novel indexes to be exploited within a commercial database, the framework used may not be suitable for all forms of index. In order to use an index within the DataBlade environment, several access methods must be implemented (see Bliujute et al. [19] for details). However, these methods are tailored towards the use of search trees and may not be suited to implementing indexes that do not fit this pattern. Additionally, Bliujute et al. conclude that such frameworks are only a 'first step' towards allowing novel indexes to be exploited within commercial databases and that further integrated support may be required.

Another example of the implementation of a novel indexing technology within a relational database is given by Cooper et al., who describe a path-expression index for semi-structured data [32]. This index combines features from both the B-Tree [31] and the *PATRICIA* tree [85] (see Section 2.4), to give a layered index over key strings that represent path expressions. Unlike the two examples given above, the implementation of this index did not make use of a framework such as the DataBlade environment. Instead, index blocks (each layer of the index consisted of one or more blocks) were stored as normal data within the relational database environment. This had the advantage of being able to exploit the maturity of the relational database management system to achieve performance and concurrency. The resulting index was found to be more efficient than the default indexes provided by the database (when used with a collection of documents totalling 72 MB in size). The techniques used to implement this index could be used to complement the framework-based approach described above.

2.2.2 Object-Oriented Persistence

Object-oriented databases are an alternative to the relational database [13, 34]. In such systems, the object is the basic unit of data and objects can be combined in arbitrary ways to create complex data models. The most common technique for indexing a collection of such objects is to use a variant of the B-Tree to index groups of objects associated with a particular path[5] in the class hierarchy [17] (although it is possible to implement other forms of index). Frequently, such systems are tightly coupled with a specific programming language[6], resulting in a system that has a minimal development overhead: little additional effort is required from application developers in order to make their application persistent as the in-memory representation of the data is translated directly to the persistent representation by the database software. Although such systems have attracted a lot of interest, particularly amongst database researchers, commercial success has been limited to certain domains (for example, Computer Aided Design and telecommunications) [34]. This can be attributed to the continued popularity of relational databases (especially object-relational databases) and to the lack of universally adopted standards for interoperability and query languages.

Object-oriented databases present both an opportunity and a challenge to those working with novel data types and indexes. The versatility of such systems allows both data type and index to be implemented within the framework of the system, with persistence being achieved through the mechanisms provided. However, implementing an index using such general purpose mechanisms will necessarily incur some overhead: the default on-disk object representation may result in wasted space (potentially giving slower transfer of objects from disk) and the mechanisms used to transfer objects to and from disk may not be tuned for accessing on-disk indexes (see below). Although such systems allow for rapid development of novel indexing technology, the developer may ultimately require more control over how their index is managed if optimum performance is to be obtained.

Example Systems

Two indexing projects illustrate some of the benefits and challenges of using various forms of object-oriented databases. Firstly, Garratt et al. [40] compare the implementation of an inverted file index over two object-oriented persistence platforms. The two platforms chosen were an orthogonally persistent programming language (PJama [10])

[5] The sequence of objects corresponding to a class in the hierarchy is known as a *path instantiation*.

[6] Orthogonally persistent object-oriented programming languages [8, 11] are examples of object-oriented databases where the language is tightly coupled to the persistence mechanism.

and an unnamed commercial object-oriented database. Garratt et al. conclude that index implementation was straightforward in both cases, but that the two platforms gave significantly different performance characteristics—index creation was three times quicker when using PJama, but the size of the resultant index was 25% smaller when using the object-oriented database. The second example is the use of suffix trees with PJama by Hunt et al. [58]. (A fuller discussion of this work is given in Section 2.4.4.) Initial implementation is described as being straightforward, however, in this case it was necessary to drastically change the index construction algorithm to achieve adequate performance. Again, the size of the completed on-disk index is reported as being excessively large. Such examples show that although object-oriented persistence mechanisms allow for simple index development, aspects of the chosen platform's implementation may impact heavily upon performance.

2.2.3 Bespoke Persistence

The final category of persistent system of interest here is that of systems that employ 'bespoke' data management techniques, i.e. those that do not use recognised database technology in order to make data or indexes persistent. Such techniques are inherently varied, but do have some defining characteristics. Persistence is achieved through using standard language and operating system concepts, with files being the underlying technology. The use of simple files can then be complemented with language specific technologies, for example, the object serialisation methods provided as part of Java [97].[7]

Using a bespoke persistence mechanism to achieve persistence for an index will incur a significant development overhead. The developer will have to implement methods to support a number of tasks that would ordinarily be provided by the database management system. Such tasks include: methods for transferring sections of the index to main memory from disk (and vice versa); a method of monitoring main-memory usage so that cached data can be released when more space is required; finally, the developer will also be responsible for designing the on-disk layout of the index. However, the developer also has the opportunity to tailor the persistence mechanism to suit the application, giving the potential for a more compact, tuned on-disk representation.

[7] Although technologies such as *Java Object Serialization* provide a convenient means of serialising object graphs, they do not provide a complete persistence solution. No support is given for incremental reading or writing of the object streams, resulting in the *big inhale / big exhale* problem, where applications are limited by the need to transfer such structures as a whole to and from main-memory [35].

2.3 Sequence Indexing

As discussed in Section 1.1, there has been an increased demand for techniques that can accelerate various forms of query over biological sequence data and suffix trees have often been put forward as a candidate for providing a suitable index. We now go on to define the terminology related to sequence indexing, with a discussion of the use of suffix trees in this context given in the following section.

2.3.1 Preliminary Definitions

String Terminology

Definition 1 *A* string *S of length n consists of a sequence of n characters drawn from a given alphabet. Elements of the string are implicitly numbered from 1 to n.*

Definition 2 *The i^{th}* prefix *of a string S consists of the first i characters of S (i.e. $S[1..i]$).*

Definition 3 *The i^{th}* suffix *of a string S consists of the last $n-i+1$ characters of S (i.e. $S[i..n]$).*

Definition 4 *A* substring *of a string S is a sequence of consecutive characters drawn from S.*

Definition 5 *A* subsequence *of a string S is any string that can be obtained by deleting up to n characters (which need not be consecutive) from S.*

Sequence Data

Biological sequence data can be treated as a collection of one-dimensional strings of varying length (we refer to such a string simply as a *sequence*). The characters of these strings are drawn from specific alphabets that correspond to the constituent chemicals[8] of the sequence. DNA and RNA sequences are both comprised of elements drawn from four letter alphabets ({**A,G,C,T**} and {**A,G,C,U**} respectively) and have lengths expressed in base pairs. Protein sequences have elements drawn from a twenty letter alphabet and have lengths expressed in bases. Additional characters are sometimes used to mark unknown sections in sequences. Most sequence data will be accompanied by textual annotations, with such annotations covering many varied aspects of the

[8]DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) sequences are comprised of nucleotides, whereas protein sequences are comprised of amino acids.

sequence's origin and projected function. Only the indexing of the sequence data (and not the accompanying annotations) is addressed in this work, with particular focus on the challenge of providing an index over large sequences. The lengths of sequences that are of interest to biologists can vary from a few tens of characters up to multi-gigabyte sequences corresponding to complete genomes.

Pattern Matching

Pattern matching problems can be split into two categories: *exact matching* and *approximate, or inexact, matching.* Exact matching (see Definition 6), involves searching a given string for substrings that precisely match the supplied target string. Efficient solutions to this problem are of some interest to biologists as they can be directly extended to solve problems that are more complex.

Definition 6 *Given a string* P *called the pattern and a longer string* T *called the text, the* ***exact matching*** *problem is to find all occurrences, if any, of pattern* P *in* T *[48].*

Approximate matching is of great interest to biologists: it is a key tool in locating sections of sequences that are common to two or more species. Approximate matching, or *matching allowing errors*, is the process of finding 'the positions in a text where a given pattern occurs, allowing a limited number of "errors" in the matches' [89]. Such errors may include the deletion, insertion or replacement of one or more characters. In order to be able to extract meaningful matches, application specific costs are allocated to each of the named operations and a threshold is chosen to determine how close matches must be to the original in order to be counted as a significant match.

Both exact and approximate matching can be performed efficiently using suffix trees [48, 101, 88, 58]. We focus on providing support for exact matching, with the performance of approximate matching over the Top-Compressed Suffix Tree still to be explored.

2.4 Suffix-Based Indexes

The need to perform complex queries over ever growing volumes of textual data has seen naïve string processing algorithms abandoned in favour of index based solutions. In particular, data structures that index the suffixes of a given text have grown in popularity due to their support for efficient construction and queries.

Indexes over strings have been in use for some time, with data structures such as *tries* (originally proposed by Fredkin [39]) and *digital search trees* proving to be

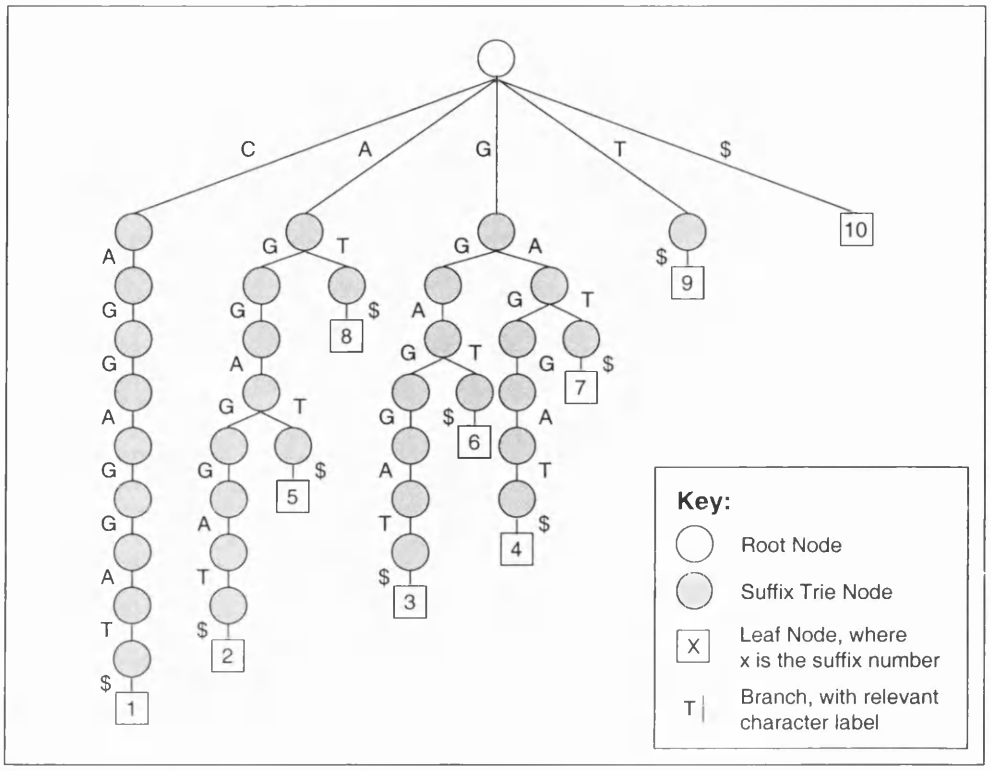

Figure 2.1: A Suffix Trie for the string **CAGGAGGAT$**.

effective at allowing simple problems to be solved efficiently. Tries index a given set of strings via a tree structure consisting of one node per prefix letter common to two or more strings (with paths consisting entirely of unary nodes representing the remainder of each string). Tries can be used to index the suffixes of a given string, resulting in a structure known as the *suffix trie* (illustrated in Figure 2.1). The suffix trie can be defined as *a Trie that contains precisely the suffixes of the string formed by appending to S a single character that does not appear in S.* This additional character, known as the *termination character* is added to the string to ensure that no suffix of the string can be a prefix of any other suffix.[9] This requirement is necessary to ensure that there is a one-to-one correspondence between the suffixes of the sequence and the leaf nodes of the tree (as is assumed by the algorithms that operate over suffix trees).

A data structure related to the trie is the PATRICIA tree [85]. A PATRICIA tree for a given set of strings is equivalent to a trie where each path consisting only of unary branches is collapsed into a single node. Each branch node, or *internal node*, will have

[9] We use **$** as the termination character.

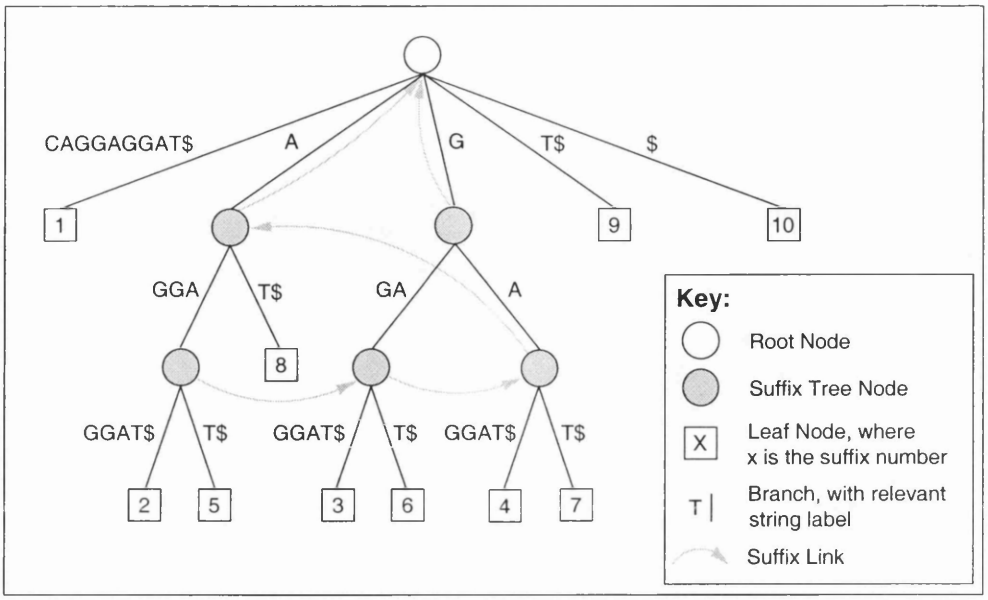

Figure 2.2: A Suffix Tree for the string **CAGGAGGAT$**.

at least two children and no two edges from an internal node can have labels beginning with the same letter. Note that the labels corresponding to each branch will not be stored explicitly; instead, pointers into the original string will be used. This provides a more compact representation of the information stored in a trie.

2.4.1 Suffix Trees

A suffix tree (illustrated in Figure 2.2) for a given string is equivalent to a PATRICIA tree containing precisely the suffixes of the string, S, formed by appending to S a single character that does not appear in S. However, typical suffix tree implementations will differ from the PATRICIA tree as additional information is stored within each node (see below). A suffix tree for a string of length n will have exactly n leaf nodes, numbered from 1 to n, and the concatenation of the labels on the path from the root node to leaf node i will give the i^{th} suffix of the string S. The number of branch nodes will be less than n and typically between $0.7n$ and $0.8n$. A thorough introduction to suffix trees and their uses is given by Gusfield [48].

Node Definition

A typical suffix tree implementation will require node implementations containing some, or all, of the fields listed in Table 2.1. The right label and suffix number can be calculated on demand as the tree is traversed, and therefore can be omitted in order to save space. To determine the value of a node's right label, the node's children are examined: for internal nodes, the right label will be exactly one less than the lowest left label of the node's children (this is always the left label of the first child in the suffix tree representations used throughout this work)[10] and for leaf nodes the right label is simply the length of the indexed string. The value of the suffix number for a given leaf node is $n - q + 1$, where n is the length of the indexed text and q is the length of the path to that node from the root node (this is calculated by summing the lengths of the labels of each branch as they are traversed). The suffix link, discussed below, plays an important role in some algorithms. However, it can be omitted if a given implementation does not use any of the algorithms that require it. This gives rise to a form of suffix tree that we refer to as a *minimal suffix tree* (see Figure 2.3). Note that we have assumed a *sibling-chain* representation of the tree.

Field Name	Type	Field Purpose
Left Label	Integer	Left index into the string for this node.
Right Label	Integer	Right index into the string for this node.
Suffix Number	Integer	The number of the suffix to which this node corresponds (leaf nodes only).
Child	Reference	The first child of this node.
Sibling	Reference	The next sibling of this node.
Suffix Link	Reference	A reference to the node corresponding to the suffix link rule for this node.

Table 2.1: The fields of a Suffix Tree node.

Construction

Several algorithms exist for the construction of suffix trees. Most notable are the linear-time (with respect to the length of the string) construction algorithms of Weiner [102], McCreight [81] and Ukkonen [100]. A simpler algorithm, which we refer to as naïve construction, is also used as it is more straightforward to implement and its behaviour is

[10]Note that the left and right labels of a suffix tree node may refer to any substring of the original text that matches the branch's label; however all practical construction methods allocate the labels in a predictable manner.

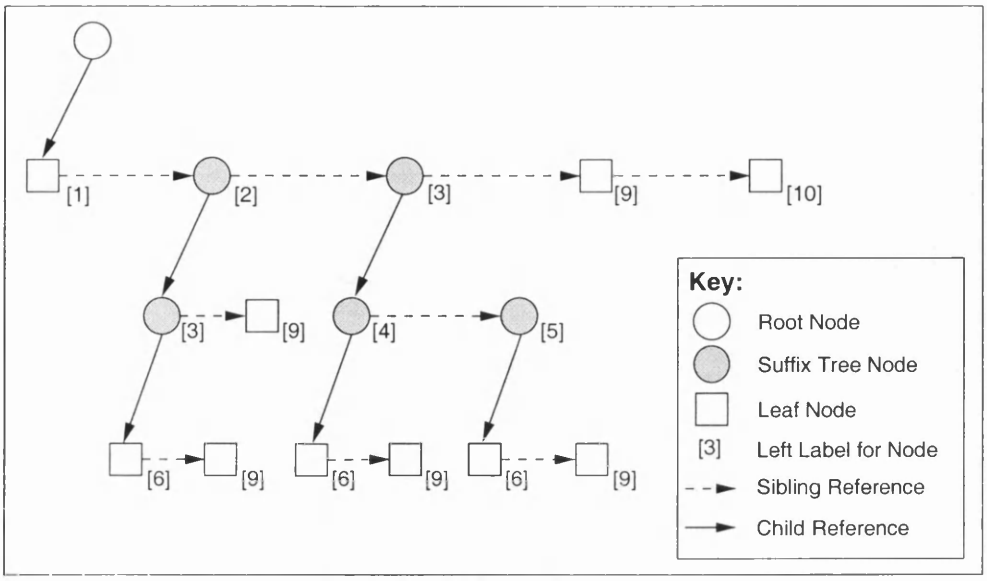

Figure 2.3: A Minimal Suffix Tree for the string **CAGGAGGAT$**.

preferable in some situations. We now summarise both naïve and Ukkonen's algorithms.

Naïve Construction To achieve naïve construction of a suffix tree, simply add each suffix in turn to the growing tree. This is equivalent to adding a set of strings into a PATRICIA tree. To insert a suffix into the tree, it is necessary to first work down a unique path matching characters from the suffix. This continues until no further matches are possible and since no suffix can be a prefix of any other suffix, we will not yet have matched the entire suffix. The remainder of the string is then added to the tree by either adding a new child to the current node (if the first mismatch occurs at the end of a node's label) or by splitting the branch leading to the current node. The worst case running time of the naïve construction algorithm is $O(n^2)$, for a string of length n. However, the average case running time has been shown to be $O(n \log n)$ for suitably random data (biological sequences are commonly believed to fall within this category) [7].

Linear-Time Construction Linear-time construction of suffix trees was first presented by Weiner [102], however the algorithms of McCreight [81] and Ukkonen [100] (which can be shown to be a variant of McCreight's algorithm) are generally preferred due to their more prudent use of main memory. In Ukkonen's algorithm, the tree is constructed by processing each letter of the string in turn, working from character 1

through to character n. In order to achieve linear-time construction, it is necessary to avoid traversing the tree from the root node upon the insertion of each new character. This is achieved by maintaining the suffix link (see Definition 7) at each node so that a direct path can be followed to each node that requires updating after adding the character at position j (these are the nodes whose path from the root node is labelled $S(i..j-1), \forall i.i < j$). This path is known as the *boundary path*. Ukkonen defines two points on the boundary path that determine how the nodes are updated. Each node before the *active point* is a leaf node and is simply extended to incorporate the new character. For each node from the active point to the *endpoint*, a new branch is created (the endpoint is excluded). The nodes on or after the endpoint do not need updated.

Definition 7 *Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring. For an internal node v with path-label $x\alpha$, there is another node $s(v)$ with path-label α and a pointer from v to $s(v)$ is called a* suffix link.

Search

Given a completed index over a text, suffix trees can be used to solve the exact matching problem in $O(l+m)$ time, where l is the length of the target pattern and m is the number of matches (provided that alphabet size can be regarded as a constant). This is achieved by matching the target pattern against a unique path through the tree until either of the following is found: a path matching the complete target pattern (in which case the target pattern is present in the original text); or a mismatch (in which case the target pattern is not present in the original text). This step requires at most $O(l)$ comparisons between the pattern and the tree. If the target pattern is found to be in the text, each occurrence of it in the text can be located by visiting each leaf node below the point at which the complete target pattern was found in the tree. This can be done in $O(m)$ time. This technique can be directly extended to support more complex queries, such as longest common substring. Additionally, it has been shown that suffix trees can be employed to accelerate approximate matching algorithms [89, 88].

2.4.2 Suffix Tree Variations

Given the view that suffix trees are frequently too large for practical use in main memory and that they are unsuitable for construction on secondary memory [87], many alternative suffix based indexes have been proposed. Several variations of the suffix tree exist, many of which incur a performance penalty in order to save space. Notable

examples include the *sparse suffix tree* [67], *word suffix tree* [6] and the table based implementations proposed by Kurtz [72]. Both the word suffix tree and the sparse suffix tree save space by only indexing a subset of the suffixes of a given text, with the word suffix tree indexing only the suffixes that start with a word and the sparse suffix tree indexing every k^{th} suffix. Clearly, the word suffix tree is only applicable to situations where the text can be broken into words, making the sparse suffix tree a more suitable solution for DNA and protein sequences. However, minimising the size of the index must be carefully balanced against required levels of performance.

Kurtz proposes several table based suffix tree implementations [72]; each designed to exploit redundancies within suffix trees in order to provide a compact representation. The *Improved Linked List Implementation* was shown to be the most compact index for DNA sequences, occupying an average of 12.55 bytes per input character. This betters previous tree-based implementations by more than eight bytes per input character. Recent work combines this representation with *lazy top-down construction*, giving promising results [41].

2.4.3 Other Suffix-Based Indexes

Several other data structures exploit the suffixes of a text in order to provide an efficient index. The suffix array [80], is a simple data structure consisting of an array of the suffix numbers of the text sorted in lexical order together with an array of the *longest common prefixes (lcps)* of adjacent elements in the suffix number array. Searching this structure is achieved by means of a binary search over the sorted suffixes ($O(l + \log n)$ time, for a pattern of length l and a text of length n). A related structure is the *augmented suffix array* [30], which minimises the time spent searching the array by using a suffix tree in order to locate the relevant section. This gives performance closer to that of the suffix tree (searching using this structure can be achieved in $O(l + \log \log n)$ time), while still achieving a space saving. Recent work discusses the use of suffix arrays on external memory [33], concluding that the development of such indexes is an under-developed area.

The *suffix cactus* [68] can be viewed as a compact representation of a suffix tree and, like the augmented suffix array, gives performance between that of the suffix tree and the suffix array. The defining characteristic of the suffix cactus is the joining of every internal node with one of its children. This reduces the overall size of the tree, but at the cost of a performance penalty during search: the other children must be accessed via two tables, the first being an alphabetical ordering of the children, and the second a list of the positions at which the children branch from the parent node.

The *suffix binary search tree* [78, 59], indexes the suffixes of a text via a binary search tree, where each node is augmented with additional information in order to improve search performance. An empirical exploration of the suffix binary search tree has shown that it can perform comparably with the suffix tree and suffix array [61].

2.4.4 Suffix Trees on Secondary Memory

It is frequently stated that use of suffix trees on secondary memory is not feasible [42, 87, 37, 68], as the associated construction algorithms perform unacceptably. This has prompted the development of structures, such as the *String B-Tree* [37], specifically designed to use secondary memory in order to index large texts. However, such structures do not have the versatility of the suffix tree (in particular, the performance of approximate matching has yet to be adequately explored). Overcoming the 'memory bottleneck' associated with suffix trees is, therefore, an important problem.

Two techniques have been proposed to improve the performance of suffix trees on secondary memory, however the applicability of these techniques to the construction of large indexes has not been explored. Clark and Munro [28] discuss a suffix tree representation that minimises the number of disk access to the structure in order to perform a given search. However, they do not discuss how to efficiently construct the on-disk index. Farach et al. [36] discuss the reduction of suffix tree creation to that of sorting, and subsequently optimise their algorithm in order to minimise the number of disk accesses. As with the work of Clark and Munro, no empirical evidence is given to demonstrate that this approach would allow the construction of indexes that greatly exceed the size of the available main memory.

Prefix-Partitioned Construction

Hunt et al. demonstrate how to create large persistent suffix trees using an orthogonally persistent programming language [59, 57, 58]. This approach introduces the *Prefix-Partitioned Construction Algorithm* (which they refer to as *phased tree construction*), a variation of the naïve suffix tree construction algorithm that allows incremental construction of the index. This algorithm performs multiple passes of the sequence to be indexed, with only the suffixes whose prefixes lie within a certain range being inserted during each pass. Thus, it is possible to choose ranges in such a manner that completed sections of the index can be transferred to secondary memory, allowing the main-memory space to be re-used. This technique has been successfully used to provide an index for up to 286 Mbp of DNA sequence data, bettering previously reported

volumes of indexed data by a factor of thirteen. The persistence platform used during this work was PJama, an orthogonally persistent variant of Java™ [10].

However, there are some limitations in the construction algorithm presented by Hunt et al. [58]. In particular, the method given for predicting the number of partitions required to index a given sequence for a fixed amount of available main memory does not accurately reflect the number of partitions required in practice. Hunt et al. give a linear relation between the number of partitions and the predicted size of the main-memory representation of the completed index. However, this fails to address the amount of main memory required to store the sequence being indexed (all suffix tree implementations require the indexed text to be present in main memory). The number of partitions required (given a fixed amount of available main memory) will grow super-linearly,[11] as the size of the sequence increases (as the sequence grows, so does the total index size, thus we are trying to create a larger index in a smaller space). Hence, the relationship between the number of partitions required and sequence length cannot be linear.

In practice, it is not possible to accurately predict the number of partitions required simply from the sequence length, as the number required is largely dependent on the nature of the sequence being indexed (also, it cannot be assumed, as Hunt does, that suffixes are evenly distributed across all prefix partitions). Furthermore, sequences containing highly repetitive substrings[12] cannot be indexed using this technique and prefix-partitioned construction will fail in such circumstances. Overall, it can be seen that the rate of growth in the number of partitions required is, at the very least, worse than linear; however, it is not possible to give an upper bound on this value as prefix partitioning will fail in some circumstances.[13] Given that the relationship between the number of partitions required and sequence length is not linear, it then follows that the phased tree construction algorithm has an average case running time of $O(mn+n\log n)$ and not $O(n\log n)$ as claimed [58] (where n is the sequence length and m is the number of partitions). This is significant when the size of the sequence being indexed is of the same order of magnitude as the amount of available main memory. The distribution of suffixes over prefix partitions is explored in Section 4.5.1.

Another limitation of the construction technique presented by Hunt is in the method

[11] Here, super-linear is used to refer to growth that is at least worse than linear.

[12] Such repetitive substrings are unlikely to occur in genomic data.

[13] An example where prefix-partitioned construction could fail is when indexing a sequence with a large substring consisting solely of one letter repeated. The section of a suffix tree corresponding to such a substring must lie in exactly one prefix partition, thus if this section of index is too large to fit into main memory then index creation will fail. (The number of partitions required will be infinite in such circumstances.)

for determining whether or not a particular suffix lies within the current prefix range. The given method requires the evaluation of an l degree polynomial for each of $m \times n$ comparisons (for a prefix length of l), whereas it is only necessary to evaluate two terms of the polynomial for each comparison. This avoids $m \times n \times (l-2)$ calculations during index construction. A revised version of this construction algorithm is presented in Section 3.5.2.

Recent work has shown how McCreight's construction algorithm [81] can be adapted to allow prefix-partitioned construction [24]. This is achieved by maintaining the suffix links of the tree in such a way that it is possible to follow the chain of suffix links during tree construction, even though not all nodes will be present (only those within the current sub-tree will be present). However, this algorithm does not give partitioned $O(n)$ suffix tree construction (c.f. Brown [24]). As was the case with Hunt's algorithm, multiple passes of the sequence are used in order to identify the suffixes belonging to the current partition and, as discussed previously, the number of partitions required will grow super-linearly as the size of the sequence increases (given a fixed amount of available main memory). Therefore, the algorithm presented by Brown [24] will have an average case running time of $O(mn)$ (where n is the sequence length and m is the number of partitions). Applying the techniques presented in Section 3.5.2 to this algorithm would yield a truly $O(n)$ partitioned construction algorithm.[14] Other limitations of this work are that no persistence mechanism is described and no method is given for using the suffix links over the completed tree (in order for query algorithms to use the suffix links it would be necessary to either modify the query algorithm for use with partial suffix link chains, or to impose the complete suffix link chains on the tree once construction is complete).

Buffering Strategies for Suffix Tree Construction

Two recent projects have explored the use of carefully tuned buffering strategies to allow the construction of suffix trees of a size that exceeds that of the available main memory. The first such technique is given by Bedathur and Haritsa [15], who aim to provide a paging mechanism that can be used with an unmodified in-memory suffix tree construction algorithm to achieve practical construction of large suffix trees. The second approach is given by Tata et al. [99] who, like Hunt, modify an in-memory suffix tree construction algorithm to allow construction of on-disk suffix trees. However, this work differs from that of Hunt as persistence is achieved using bespoke buffering policies

[14] This assumes that the other claims made by Brown regarding the performance of his algorithms are correct (no such proof was provided).

rather than general-purpose persistence mechanisms.

Two novel paging policies, *TOP* and *TOP-Q*, that can be used in conjunction with Ukkonen's construction algorithm [100] to achieve on-disk suffix tree construction, are presented by Bedathur and Haritsa [15]. The first policy, *TOP*, is derived from the observation that nodes nearer the top of the tree are accessed more frequently during construction than those lower in the tree, thus nodes higher in the tree should be preferred for retention in main memory. Within TOP, nodes are added to fixed size pages in the order that they are created. Associated with each page is a score giving the average depth of the nodes contained in the page, and this score is then used to rank pages in order of preference for retention in main memory. However, it was observed that this strategy is rather inefficient when used with Ukkonen's algorithm: when inserting a suffix, it is often necessary to update the parent node of the node where the first mismatch is found, and as such a node may be deep in the tree it will be on a page that is not likely to have been retained in main memory. The second policy, *TOP-Q*, aims to address this limitation by introducing an additional buffer, operating on a first-in first-out basis, to buffer recently evicted pages. The page containing a given node's parent will have been accessed as the algorithm traverses the tree, so even if the page has been subsequently marked for removal from main-memory it should still be retained in the second-chance buffer.

Bedathur and Haritsa show that the use of TOP-Q can improve the performance of Ukkonen's construction algorithm when used on disk [15]. However, it is not possible to conclude from the results presented whether or not TOP-Q is a suitable mechanism for constructing large suffix trees. Firstly, the volume of data used in the experiment was significantly smaller than that used previously to demonstrate the use of the Prefix-Partitioned Construction Algorithm (70 Mbp compared to 286 Mbp [58]). Secondly, no comparison is given between TOP-Q and other mechanisms for constructing disk-resident suffix trees. Without such information, it cannot be concluded that this approach will give adequate performance or that it will scale to accommodate the creation of large indexes.

Tata et al. [99] propose the *Top-Down Disk-based (TDD)* suffix tree construction technique, which is a combination of the *wotdeager* construction algorithm of Giegerich et al. [41] (a write-only top-down suffix tree construction algorithm with average and worst case performance equivalent to that of the naïve algorithm) with a carefully tuned buffering policy. Four buffers are used in this system. The first two correspond to the string being indexed and the suffix tree respectively. The second pair of buffers are used to manage a list of suffixes in the order that they are to inserted in the

tree (with one buffer used only during sorting). The first step of this algorithm is to allocate suffixes to partitions. Each partition is then taken in turn; the suffixes within the partition are sorted and added to the tree using the wotdeager algorithm. The key to this technique is in deciding what size each buffer should be, with those that are accessed randomly being allocated the greatest percentage of the available main memory. When constructing large indexes, the majority of the available main memory is allocated to the string being indexed, with small sections of the tree being constructed in main memory before being transferred to disk. In essence, this approach is equivalent to the Prefix-Partitioned Construction Algorithm: by partitioning and sorting the suffixes they allow the chosen construction algorithm (wotdeager) to be implemented in such a way that the locality of reference allows completed sections of the index to be transfered to secondary storage, and this is exactly what is achieved by the use of partitions in the Prefix-Partitioned Construction Algorithm. The main difference between the two techniques is in the mechanism used to transfer completed sections of index to disk. With TDD, nodes are automatically transferred to disk by the buffering system, whereas with Prefix-Partitioned Construction the transfer of nodes to disk is explicitly invoked by the algorithm.

Tata et al. [99] claim that suffix tree construction is significantly faster when using Top-Down Disk-based construction as opposed to Hunt's Prefix-Partitioned Construction Algorithm, despite both algorithms having the same predicted average and worst case running times. However, the implementation of Hunt's algorithm used for the evaluation appears to be fundamentally flawed. Tata et al. state that the reason Top-Down Disk-based construction is faster than Hunt's algorithm is because *'Hunt's algorithm traverses the on-disk tree during construction, while TDD does not'* [99]. This contradicts the stated purpose of prefix-partitioned construction (*'data structures for the complete partitions can be evicted from main memory and will not be faulted back in during the rest of the tree's construction'* [58]), and suggests that the implementation used was unsuitable. Additionally, the partitioning scheme used by Tata et al. is somewhat inflexible and is based upon a potentially inappropriate assumption. When determining buffer sizes, it is assumed that suffixes are evenly distributed across all partitions, whereas this is unlikely to be true for most forms of data that the index may be used for (see Section 4.5.1). In addition, the number of partitions used is restricted to being a power of the alphabet size, which may cause significantly more partitions to be used than is necessary (increasing the number of partitions incurs a performance overhead with this algorithm).

Although the two techniques discussed above have been successful in allowing the

construction of suffix trees that are greater in size than the available main memory, there are limitations to such approaches. In particular, a persistence mechanism that is tuned to support efficient index construction may not be suitable for use when the index is to be used to support efficient queries. Different types of query that operate over the index can have different access patterns, and such access patterns may differ significantly from that of index construction. The predictability of node accesses exhibited by the construction algorithms is unlikely to be repeated when the index is being used to answer queries (especially if several users are sharing the index) and it may not be possible to provide an equivalent buffering mechanism to support efficient queries using an on-disk suffix tree.

2.4.5 Summary of Suffix Tree Construction Techniques

Here, we are interested in providing a suffix tree implementation that allows the index to be both constructed and queried when resident on secondary memory—necessary properties when large volumes of data are to be indexed. This precludes the use of the classical construction techniques of Weiner [102], McCreight [81] and Ukkonen [100], as these techniques do not have a locality of reference that allows for incremental construction using secondary memory.

Suitable locality of reference can be obtained when using the prefix-partitioned construction algorithm, as demonstrated by Hunt et al. [57]. Here, multiple passes of the sequence are required in order to allow incremental construction. Hunt claims that this algorithm has an $O(n \log n)$ average case running time. More recently, Brown [24] applies the prefix-partitioned technique to McCreight's algorithm, and claims that the resultant algorithm has $O(n)$ average case construction time. However, the algorithmic analyses presented by both Hunt and Brown is flawed. Both assume that the number of partitions required in order to create the index is directly proportional to the sequence length n. However, this assumption has been shown to be false, and the average case running times of the two algorithms are in fact $O(mn + n \log n)$ and $O(mn)$. Use of a suitable $O(n)$ pre-processing step would allow the repeated scans of the sequence to be avoided, and would give algorithms that match the claimed average case running times of both Hunt et al. [57] and Brown [24]. Additionally, both techniques rely upon general purpose persistence mechanisms to manage the on-disk index—such mechanisms give an on-disk index that is several times larger than its main-memory representation.

Tata et al. [99] and Bedathur [15] both present suffix tree construction techniques that make use of buffering mechanisms that have been carefully tuned to accommodate suffix tree construction. However, this approach is limited in that a buffering mechanism

tuned solely to accommodate suffix tree construction may be entirely unsuitable for supporting index querying (especially so if a variety of different query algorithms are supported). Thus, the utility of this approach is likely to be limited.

Reflecting upon the limitations of the techniques presented here, a disk-resident suffix tree implementation should have the following properties. Firstly, it should make use of persistence mechanism that supports both construction and subsequent querying of the index when only part of the index is main memory. Secondly, the persistence mechanism should allow for a more compact index than that obtained using general purpose mechanisms. Lastly, the construction algorithm should avoid the repeated scans of the sequence associated with the prefix-partitioned approach given by Hunt et al. Overall, the index should be implemented in such a way that the operations related to persistence do not limit the utility of the index.

2.5 Summary

This chapter has given an overview of the use of indexes in persistent environments, exploring both traditional use of indexes and why it is necessary to allow the indexing of domain specific data within persistent systems. A critique of the use of suffix-based indexes with biological sequences was then given, highlighting the need for ongoing research within this domain. The following chapter presents the design of the Top-Compressed Suffix Tree, a data structure specifically designed to overcome some of the limitations of current sequence indexing techniques that have been identified in this chapter.

Chapter 3

The Top-Compressed Suffix Tree: Design and Implementation

This chapter presents the Top-Compressed Suffix Tree (TCST), a data structure that has been designed to address some of the limitations identified in previous accounts of the use of suffix trees on secondary memory. In particular, we improve upon the work of Hunt, by addressing the deficiencies identified in the prefix-partitioned construction algorithm (see Section 2.4.4) and by providing a platform-independent persistence mechanism. Additionally, the design of the TCST exploits trends identified in the density of suffix trees (see Section 3.8[1]) to allow a more efficient and compact index.

3.1 Design Criteria

Given the limitations in the use of suffix trees discussed in Section 2.4, the principal criteria for the design of the TCST were identified as being support for the following:

- *Incremental Construction* In order to allow the construction of indexes that exceed the size of the available main memory, it is required that a TCST can be constructed in such a way that completed sections of the index can be transferred to secondary memory allowing the corresponding main-memory space to be reclaimed and reused.
- *Scalable Construction* The TCST must support construction of indexes over sequences comparable in size to the available main memory, thus allowing large sequences to be indexed using modest computing resources.

[1] These trends were identified during an exploration of the use of suffix trees with a general purpose persistence mechanism, a full account of which is given in Japp [62].

- *On-demand Faulting* When a required section of a TCST is not present in main memory during a search operation, the requested section must be able to be efficiently identified and transferred from secondary memory to main memory.
- *Eviction Management* In order to support a long-running query server that exploits a TCST index, it will be necessary to remove unneeded sections of the index from main memory to create space to store the sections of the index required for the current query.

Additionally, we aim to provide a compact on-disk index representation that is not dependent on a platform specific technology. By employing a bespoke persistence mechanism for the TCST, we can avoid the storage overhead associated with the use of general purpose persistence mechanisms. For example, the PJama based suffix tree implementation of Hunt [57] required some 66 bytes per character of the indexed sequence which does not compare favourably with in-memory suffix tree representations (such as those presented by Kurtz [72]). Clearly, there is scope for providing a more compact on-disk index.

3.2 Overview of the Data Structure

The key criteria for the design of the TCST all require the index to be structured in such a way that it can be partitioned into non-overlapping regions. This is achieved by replacing the nodes nearest the root of a suffix tree with a more compact array-based representation, which is then used together with a collection of relatively small sub-trees that can be manipulated independently of each other. This also gives rise to the name of the data structure: the Top-Compressed Suffix Tree is so named as the array-based representation of the top of the tree allows for a more compact index than is possible when using simple nodes (see also Section 3.8). This two-tiered structure is split at a chosen character depth c, with the first c characters of each suffix indexed via the array-based representation and the remainder of each suffix indexed via the sub-trees. By arranging the structure in this way, we avoid having to read in large data structures in order to perform simple queries, as only the required array entry and corresponding sub-tree are necessary.

Each sub-tree will correspond to a unique c-character prefix p, and is equivalent to the sub-tree that would be located by matching p in a suffix tree. The collection of sub-trees can be viewed as the set of sub-trees that would be formed by cutting a suffix tree at a character depth c. Each sub-tree is associated with exactly one entry

in the array based representation of the top section of the data structure, and each entry in the array will either be null or be associated with exactly one sub-tree. This relationship is achieved by encoding each c-character prefix as a unique integer, which then gives the index of the array element. Additionally, the prefix codes are used to delimit partitions during prefix-partitioned construction. A detailed description of the relationship between the two-tiers of the structure is given in Section 3.3.

3.3 In-Memory Representation

Figure 3.1 illustrates a complete in-memory TCST for the string **CAGGAGGAT$**, and is the equivalent TCST to the suffix tree shown in Figure 2.3. This TCST has been split at a character depth of two, a value that will increase as the size of the sequence being indexed increases. The illustration shows the implicit labelling of the array based representation of the top of the data structure. Furthermore, it can be seen that the array is represented using a two-level structure—the reasons for this, and the purpose of each level, are discussed in Section 3.3.2. It should be noted that TCSTs do not index the final $c-1$ characters of the sequence (in this example, the final character of the sequence, **T**, is not present in the index): this issue is addressed in Section 3.6.2. Each aspect of the TCST is discussed in greater depth in the remainder of this section, providing a complete description of the data structure.

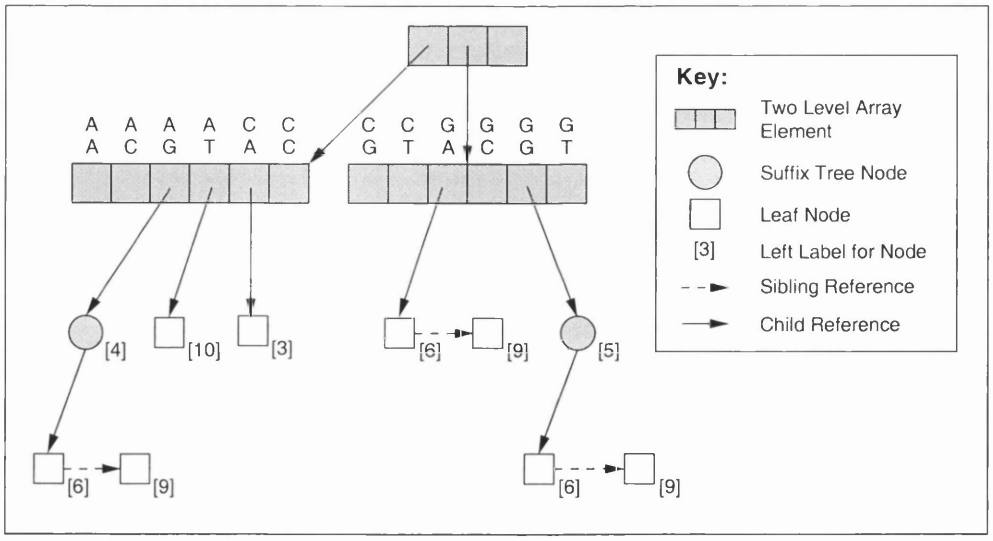

Figure 3.1: A main-memory resident Top-Compressed Suffix Tree for the string **CAGGAGGAT$**.

3.3.1 Densely Coding the Alphabet

In order to provide a unique integer for a c-character prefix, we are first required to use a *dense coding* for our alphabet. A dense coding for an alphabet of size a will range from 0 through to $a - 1$. So, for our four letter DNA alphabet we would have the following: $\mathbf{A} = 0$, $\mathbf{C} = 1$, $\mathbf{G} = 2$ and $\mathbf{T} = 3$. Now the letters of a c-character prefix can be used as the coefficients in a c-degree polynomial, which gives a unique integer when evaluated. For example, the four character prefix of a string S, using an alphabet of size a, can be encoded as: $(((S[1] \times a + S[2]) \times a + S[3]) \times a + S[4])$. This method of evaluating a polynomial is known as *Horner's Method* [22]. This single integer can then be used to locate a required sub-tree in the top section of a TCST.

When calculating the codes for successive prefixes (which would happen during index construction) it is possible to transform one code into the next using fewer calculations than would be required to compute the value of the complete function. For example, if we currently held the code for the twelve character prefix starting at position ten in the sequence, we could obtain the equivalent code for the prefix starting at position eleven as follows: $(((p - S[10] \times a^{c-1}) \times a) + S[22])$, where p is the current prefix code, S is the string, a is the alphabet length, and c is the prefix length. Note that the value of a^{c-1} is constant, and is only calculated once. This avoids $n \times (c - 2)$ calculations during each pass of the sequence being indexed (for a sequence of length n and a prefix length of c), providing a significant performance boost over implementations that evaluate the complete polynomial for each prefix (such as that described by Hunt [57]).

3.3.2 Two-Level Arrays

Logically, all sub-trees will be stored in a linear array with their position determined by the polynomial technique discussed in Section 3.3.1. However, the array will be relatively sparsely populated and its size (given by a^c, where a is the alphabet size and c is our chosen prefix length) is likely to be large (as sequence length increases it is expected that c will increase correspondingly). Hence, a two-level array is chosen in preference.

A two-level array of capacity x is addressed, from 0 to $x - 1$, as though it were a linear array. However, it is actually split into two distinct parts: a *backbone* of size $d1$ (where $d1 \leq \frac{1}{2}x$) and up to $d1$ *ribs* of size $d2 = \lceil x/d1 \rceil$.[2] Each rib is allocated on

[2]The terms *backbone* and *ribs* are chosen as a two-level array resembles a rib cage if the second-level elements are illustrated as being perpendicular to the first-level element.

demand, therefore giving a more compact solution than a one-dimensional array (when sparsely populated). When used within a TCST, typical two-level array capacities ranged from 390,625 when using a compressed depth of eight through to 244,140,625 when using a compressed depth of twelve (5^8 and 5^{12} respectively, assuming an alphabet size of five). Note that the value chosen for c (the compressed depth) is expected to increase as sequence length increases.

The i^{th} element in a two-level array can be accessed using the following two steps. Pick out the appropriate rib: this will be element $\lfloor i/d2 \rfloor$ in the backbone. Then select the appropriate entry in the located rib (if present), which will be element i mod $d2$.

Figure 3.2 shows how a set of data values can be stored in both a linear array and a two-level array. Both arrays shown have twenty potentially usable 'locations' available, of which only six are used. The two-level structure is comprised of a backbone of size five and three ribs of capacity four. In this small example we make a space saving of three units, but with much larger structures the space saving will be more substantial (given an appropriate choice of backbone and rib size, which can be determined through experimentation).

The two-level array of the TCST shown in Figure 3.1 indexes the first two characters of each of the suffixes, thus giving an supported range of zero to fifteen. A backbone size of three and rib size of six has been used in this example. The (implicit) prefix labels on the diagram show how prefixes relate to entries in the two-level array.

3.3.3 Suffix Sub-Trees

The sub-tree representation used in the TCST is based on the minimal suffix tree (see Figure 2.3). Furthermore, two specialised node types are provided, corresponding to those with children (internal nodes) and those without (leaf nodes). This avoids fields being present in nodes where they are not needed. Note that it is possible to introduce further node types, in particular node types corresponding to the two already given, but without sibling references. However, for reasons of simplicity it has been chosen not to do this here. The node definitions used within this work are given in Table 3.1.

Node Type	Fields (with corresponding types)
Internal Node	Left Label (Integer), Sibling (Reference), Child (Reference)
Leaf Node	Left Label (Integer), Sibling (Reference)

Table 3.1: Suffix Sub-Tree node definitions.

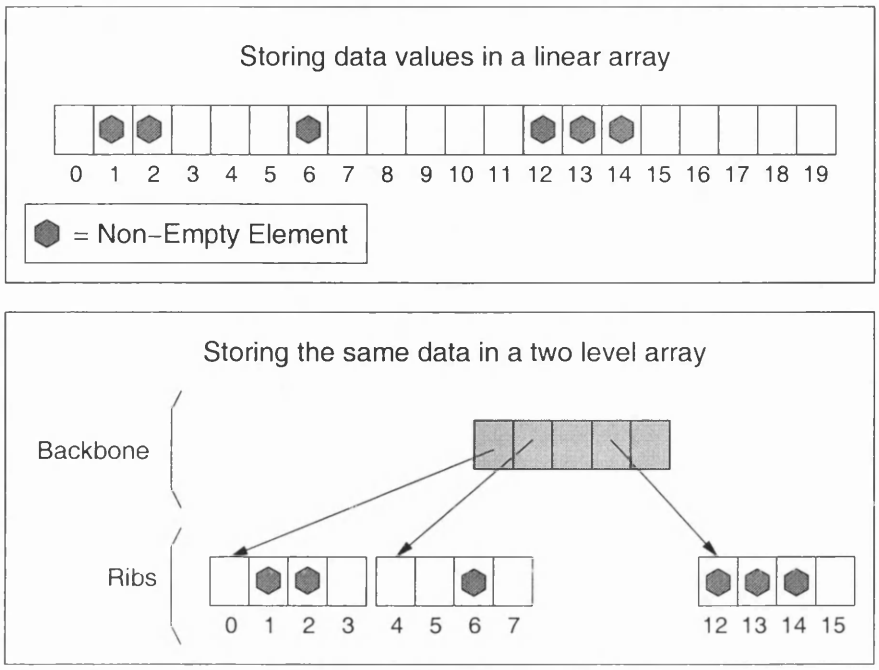

Figure 3.2: Contrasting storing a sparse set of values in a linear and a two-level array.

3.4 Representation on Secondary Storage

The Top-Compressed Suffix Tree was designed to be used as a disk-based index. The combination of sub-trees and the two-level array make mapping this structure to and from disk quite simple. The disk-based index consists of a number of files, each with the same basic structure. Each file has a name of the form `x_y` where x and y are the start (inclusive) and end (exclusive) of the range of prefixes indexed in that file. An in-memory index to the available files is be created upon starting a new server instance. Additionally, there may be a descriptor file which can store index meta-data (including which sequence has been indexed and values for operational parameters). Note that it is assumed that only one index of this form is stored in any given directory.

3.4.1 Backbone and Ribs

The elements of the two-level array are stored on disk as a series of file offsets, thus allowing random access to the required sections of the index. The first $y - x$ entries in each file represent the corresponding section of backbone, with each entry giving an offset into the file for the location of the corresponding rib (null ribs are marked with a

negative offset). Following the backbone are all the ribs and sub-trees reachable from that section of the backbone.

Ribs are written to disk in a similar way to backbone sections, with rib entries consisting of offsets (from the start of the rib) giving the location in the file of the marshalled sub-tree. Again, null entries are marked with a negative offset. The sub-trees reachable from a given rib immediately follow the series of offsets for that rib. Figure 3.3 illustrates how the TCST of Figure 3.1, is represented on disk. The file offsets representing the two-level array are shown, assuming that 32-bit integers are used. Note that we have only shown one partition here, whereas for large indexes the index would be spread over several files.

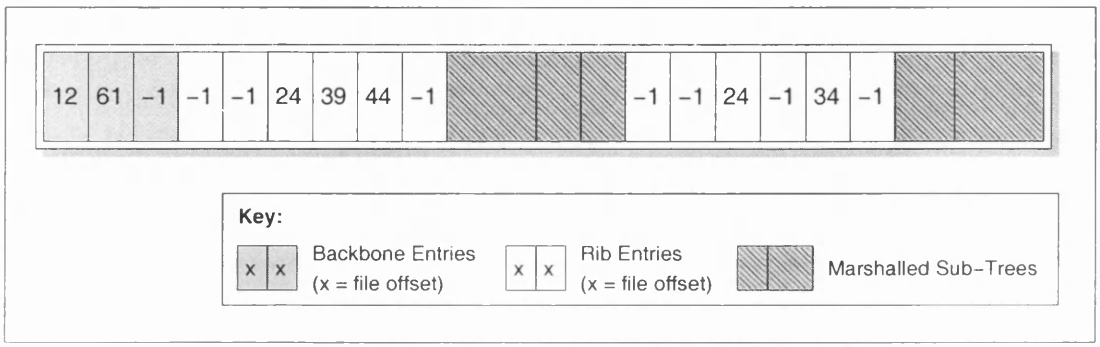

Figure 3.3: Disk representation for a region of a TCST.

3.4.2 Marshalling Sub-Trees

The serialised representation for each of the index's sub-trees is achieved by using a simple recursive marshalling algorithm. On disk, each sub-tree node requires one byte to identify the node's type and one integer to store the node's left label. Where present, the child node and sibling of a given node are written out recursively (depth first). In order to avoid storing null nodes (and thus save space) we introduce two further node types which correspond to the two node types give in Table 3.1, but without sibling references. The arbitrarily chosen identifiers for the four node types are given in Table 3.2.

Node Type	Identifier
Leaf Node	10
Leaf Node (without Sibling)	11
Internal Node	12
Internal Node (without Sibling)	13

Table 3.2: On-disk node type identifiers.

3.5 Construction Algorithms

The main requirement of an algorithm used to build a TCST is that it must have good *locality*—when a section of the index is complete we must be able to transfer it to secondary storage without requiring access to this section later during index construction. This requirement precludes the use of the original $O(n)$ construction algorithms of Weiner [102], McCreight [81] and Ukkonen [100].

Four TCST construction algorithms, each based on the naïve suffix tree construction algorithm [48], are given here. The first is a variation on the phased tree construction algorithm of Hunt et al. [57]. The next algorithm given is the *Improved Prefix-Partitioned Construction Algorithm*. This algorithm eliminates the redundant multiple passes of prefix-partitioned construction, giving greatly improved performance whenever a large number of partitions are required in order to create the index. Finally, both of the previous algorithms are extended to allow multi-threaded and distributed TCST construction.

3.5.1 Prefix-Partitioned Construction

To achieve incremental construction we must be able to split the complete task into jobs that can be processed independently. This can be accomplished by allocating non-overlapping ranges of prefixes, for which the corresponding part of the index can be constructed without reliance on parts of the structure that do not lie within the current range. For instance, if we were to partition the construction of the data structure shown in Figure 3.1 we could choose the following ranges: `[0..5]`, `[6..11]` and `[12..15]` (which are equivalent to `[AA..CC]`, `[CG..GT]` and `[TA..TT]`), and build each of these sections of the index independently.

The Prefix-Partitioned TCST Construction Algorithm is similar to the phased tree construction algorithm of Hunt et al. [57], but with two notable differences. The first is that the *insert* operation used differs from the standard suffix tree insert as it must make allowance for the presence of the two-level array. The second difference is that in order

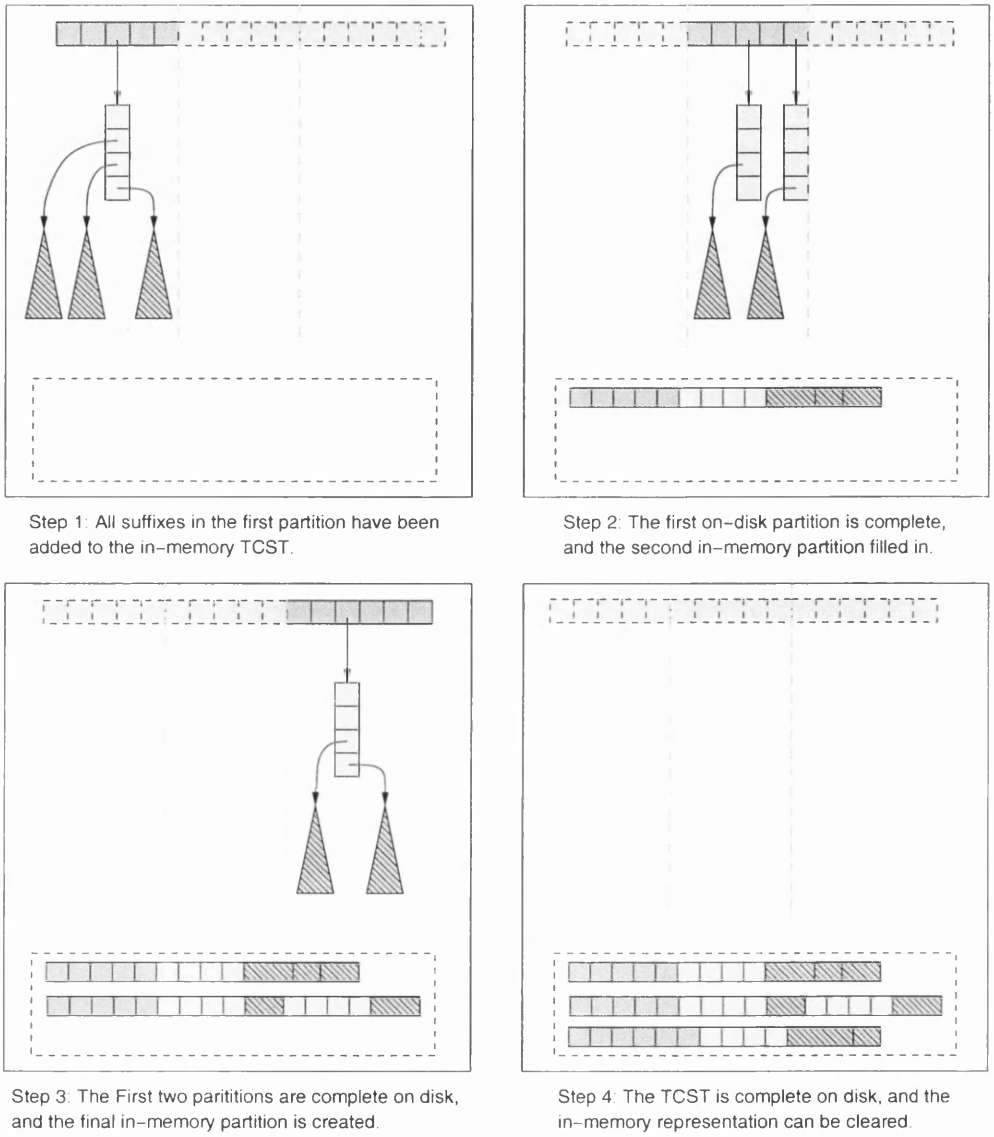

Figure 3.4: Incremental TCST construction with three partitions.

to avoid redundant calculations the current prefix code is advanced to reflect the next prefix using the technique given in Section 3.3.1. Figure 3.4 illustrates prefix-partitioned construction of a TCST using three partitions and shows the relationship between the in-memory structure and the disk-based partitions. The complete algorithm is given in Figure 3.5.

Inserting a given suffix into a growing TCST first requires the current contents of the two-level array to be examined. If the entry at the position for the suffix's prefix

```
set c to be the compressed depth of the TCST
create list of non-overlapping prefix ranges
for each prefix range loop
   calculate code for the first prefix
   for i in 1 to SequenceLength loop
      if prefix code in current prefix range then
         { We now insert the suffix }
         if two-level array entry for code is null then
            create new node with left label i+c
            add created node to two-level array
         else
            get sub-tree for current prefix code
            insert suffix i+c to sub-tree
         end if
      end if
      advance prefix code to next position
   end loop
   transfer index section to disk
   reclaim main memory
end loop
```

Figure 3.5: The Prefix-Partitioned TCST Construction Algorithm.

code is null, then we simply create a new leaf node with a left label equal to $i+c$ (where i is the suffix's position and c is the compressed depth) and add this to the two-level array. Note that this may result in a new rib being allocated in the two-level array. Alternatively, if a sub-tree is found at the relevant position in the two-level array then suffix $i + c$ is added to the sub-tree.

Choosing the Number of Partitions

In order to use this algorithm, it must be possible to establish the number of partitions required to index a given sequence. If it can be assumed that the c-character prefixes of the suffixes of the sequence being indexed are uniformly distributed, we can derive the number of partitions required to construct the index. For a given amount of available main memory A_{mm}, the number of partitions m will be as follows:

$$m = \lceil \frac{Z_{mm}}{A_{mm} - N_{mm}} \rceil$$

where Z_{mm} is an estimate of the overall size of a main-memory resident instance of the index and N_{mm} is the size of the sequence representation in main memory. Note that as the sequence length (and therefore N_{mm}) grows, the amount of available main-memory that can be used to construct the index sections (which is given by $A_{mm} - N_{mm}$) will decrease, thus the number of partitions required will grow super-linearly (c.f. Hunt et al. [58]). However, our assumption regarding the distribution of suffixes over the partitions does not always hold true for genomic data,[3] so the required number of partitions can exceed the predicted value (see Section 4.5).

3.5.2 Improved Prefix-Partitioned Construction

The Prefix-Partitioned TCST Construction Algorithm has an average case running time of $O(mn + n \log n)$ (see Section 2.4.4). When m (the number of partitions) is small this is equivalent to the average case running time of the naïve suffix tree construction algorithm. However, if indexing a given sequence for a particular amount of main memory requires numerous partitions the $O(mn)$ component of the running time can dominate. The algorithm given in the previous section requires one complete scan of the sequence for each partition. This can be avoided by pre-processing the sequence in such a way that the suffix numbers of the sequence can be grouped according to which partition they correspond to. This pre-processing stage can be performed in $O(n)$ time, thus the overall running time of the Improved Prefix-Partitioned TCST Construction Algorithm is $O(n \log n)$. Figure 3.6 gives an overview of the improved algorithm.

Pre-Processing Stage

The overall aim of the pre-processing stage is to create, in linear time, a list of the suffix numbers of a sequence grouped according to which prefix partition they lie within. For larger sequences it may not be possible to hold this ordered list in main memory, thus a technique for constructing this list on secondary memory is required. The simplest method for achieving this is to maintain one file for each partition, and periodically append each of the partial lists held in main memory to its corresponding file (and subsequently clearing the in-memory lists). This has the advantages of being straightforward to implement and only requiring one pass of the sequence. However, when dealing with a large number of partitions this method would necessitate the manipulation of many files, which may result in unwanted performance overheads. Hence, an

[3] Although DNA data is pseudo-random in nature, a given sequence may have features which lead to the number of suffixes falling within certain partitions to exceed the amount estimated.

```
create list of non-overlapping prefix ranges

{ Begin pre-processing stage }
calculate code for the first prefix
for i in 1 to SequenceLength loop
   allocate suffix i to its prefix partition
   advance prefix code to next position
end loop

{ Begin TCST construction }
for each partition loop
   for each suffix in partition loop
      insert suffix
   end loop
   transfer index section to disk
   reclaim main memory
end loop
```

Figure 3.6: The Improved Prefix-Partitioned TCST Construction Algorithm.

alternative algorithm which builds the complete list incrementally using one file is now given.

In order to construct the grouped list of suffixes incrementally using one file, we must first know how many suffixes lie within each partition. This can be achieved by performing one complete scan of the sequence, and tallying the number of suffixes within each partition. Once complete, the size of each partition list is retained in main memory. This sizing pass is necessary so that appropriately sized gaps can be left when writing out the partial lists to disk. A second scan of the sequence is then used to allocate suffix numbers to partitions. Allocating a suffix number to its partition simply consists of appending the number to the end of the linked list corresponding to that partition. A reference to each of the linked lists is held in an array, and the required linked will be entry $\lfloor \frac{prefixCode}{partSize} \rfloor$ (where *prefixCode* is the prefix code of the current suffix and *partSize* is the size of each partition).

Periodically, the current contents of each list will be transferred to disk, leaving suitably sized gaps in the file so that the as yet unallocated suffixes can be inserted at the appropriate locations. Note that writes to disk will be contiguous with the occasional (strictly forward) jump. Additional information regarding the start and end

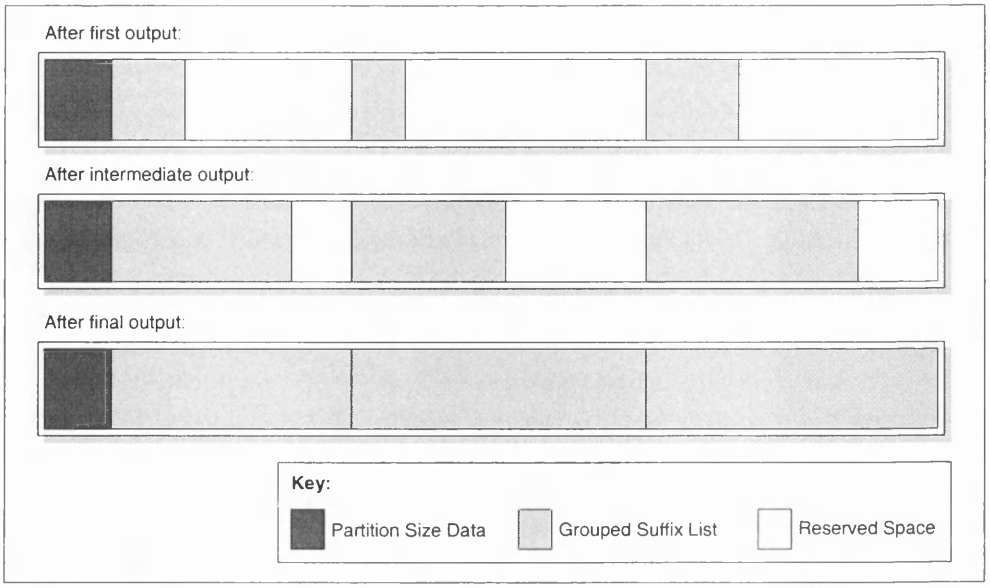

Figure 3.7: Incrementally grouping suffixes on secondary memory.

point of each partitioned list will be present at the start of the file. Note that during both passes of the sequence only a small section of the sequence need be present in main memory, thus the majority of the available main-memory can be used to create the suffix lists. Additionally, this means that the need to write to disk does not become more frequent as sequence length grows. Figure 3.7 illustrates how the contents of the file change to reflect the growing lists during the second pass of the sequence and Figure 3.8 contains the complete two-pass suffix grouping algorithm.

3.5.3 Parallel Construction

Both the Prefix-Partitioned and the Improved Prefix-Partitioned TCST Construction Algorithms can be extended to allow each partition of the index to be constructed in parallel. For the original prefix-partitioned algorithm this allows each sequence scan to be undertaken in parallel, whereas with the improved prefix-partitioned algorithm it is the creation of the TCST partitions from the grouped suffix lists that is computed in parallel. The techniques given here apply to both algorithms, although in the case of the Improved Prefix-Partitioned TCST Construction Algorithm we assume that the grouped list of suffixes has already been created.

As the prefix ranges used during construction are non-overlapping, there will be no contention for access to sub-trees or access to a given entry in the two-level array. The

```
for each partition loop
   create empty list
   set list size to 0
end loop

{ Begin sizing pass }
calculate code for the first prefix
for i in 1 to SequenceLength loop
   increment size of list to which code belongs
   advance prefix code to next position
end loop
write out list size information

{ Begin allocation pass }
calculate code for the first prefix
for i in 1 to SequenceLength loop
   add i to the list to which this code belongs
   advance prefix code to next position

   { Check available memory }
   if free memory below threshold then
      for each partition list loop
         transfer partial list to disk
         reclaim memory
      end loop
   end if
end loop
```

Figure 3.8: The Two-Pass Suffix Grouping Algorithm.

only data that is required by each process is the sequence being indexed, and as this is read-only data it can be shared safely. It would be possible for two threads to interfere if they both required access to a particular rib in the two-level array (this could happen if a range boundary is in the middle of a rib) and both threads attempt to create the rib at the same time. This problem can be avoided by choosing range boundaries that correspond exactly to the start and end of ribs.

Multi-Threaded Construction

Given the observations above, all that is required to achieve multi-threaded TCST construction is to provide a suitable method of co-ordinating the various threads. The required number of build threads will be created, each having access to a job queue that has been populated with the partitions that have to be processed. Each thread will then take one job at a time from the queue, process that job (i.e. build the corresponding section of the TCST) and finally report back to the queue that the job has been completed and attempt to dequeue another job. The main loop (i.e. the one that populates the job queue) will wait until the queue reports that all jobs have been completed.

Distributed Construction

Multi-threaded construction can easily be extended to allow simple distributed construction, which can vastly improve the total time required to create a large index. The job queue described in the previous section is replaced with a central job server that provides build clients with details of the current sequence to be indexed and where to store completed sections. The job server will also allocate partitions to build clients and report when all jobs have been completed. Note that each distributed build client can be multi-threaded if desired. This assumes that all machines share a common file system (to provide access to the sequence to be indexed and for storing the completed sections of index).

To construct an index using this technique requires three steps. Firstly, details of the current index to be constructed are supplied to the job server (this consists of the location of the sequence file, the location of the output directory, how many partitions are used, the location of grouped list of suffixes—if using the improved prefix-partitioned algorithm—and other implementation specific information). Next, the job server will add details of the partitions to be indexed to the job queue. Finally, the build clients are invoked: they will query the job server for details of the work to be done, and report back after each partition has been processed.

3.6 Exact Matching over the TCST

Exact matching using a TCST is performed using a similar algorithm to that which is used for traditional suffix trees. However, the presence of the two-level array (which implicitly represents the first c characters of every suffix) must be dealt with. This

results in search targets that are longer than c characters being treated differently to search targets that have c characters or fewer. The following descriptions assume that the relevant parts of the structure are already resident in main memory—a description of how to transfer the structure from secondary storage to main memory is given in Section 3.7.

3.6.1 Target Patterns Longer than the Compressed Depth

When searching for a target pattern of more than c characters, we are first required to calculate the prefix code for the first c characters of the pattern using the technique given in Section 3.3.1. The location in the two-level array corresponding to this prefix code is then examined. If there is a sub-tree present at this location, then the first c characters of the search target have been matched. The remainder of the search target is then matched (or not) using the algorithm given in Section 2.4.1. If no sub-tree is present then there are no matches in the sequence for the given search target. This algorithm is given in Figure 3.9. Note that if it is only required to test for the presence of the string, and not to report each occurrence, then it is not necessary to traverse the tree—we can simply return true or false as soon as either the first mismatch is found or the complete target pattern has been matched.

3.6.2 Target Patterns Shorter than the Compressed Depth

Target patterns that are equal in length, or shorter than, the compressed depth c are matched (or not) by examining the presence of sub-trees in the corresponding entries in the two-level array. If one or more sub-trees are located, we have matched the target pattern and the locations of the matches are found by traversing each sub-tree. If no sub-trees are found, then the target pattern has not been matched.

For a given target pattern, we must check every location in the two-level array that is 'prefixed' by the target string. This is accomplished by calculating both the starting position and the size of the range of entries in the two-level array to be scanned. The size of this range is given by a^{c-l}, where a is the alphabet length, c is the depth of the compressed region and l is the length of the search target. The start point of the range is the product of the range size and the polynomial code for the target string. Now we can scan the range of entries to locate the relevant sub-trees. Note that wherever a null rib is encountered, we can skip $d2$ entries in the range being searched. For a target pattern of exactly c characters, the size of the range to be explored will be exactly one entry (if $c = l$ then $a^{c-l} = 1$).

```
initialise result set

calculate code for c-character prefix of query
if two-level array entry for code is null then
   return empty result set
else
   get sub-tree for code
   match remainder of query using the sub-tree
   if miss-match found then
      return empty result set
   else
      traverse sub-tree adding locations to result set
   end if
end if

return result set
```

Figure 3.9: Finding all occurrences of query string longer than the compressed depth using a TCST.

It is worth noting that a TCST implemented as described would not provide a means for locating matches in the final $c-1$ characters of the sequence.[4] Thus, when searching for target patters of less than c characters we must use another technique for locating matches at the tail end of the sequence. One solution is to use another search technique, such as the Boyer-Moore algorithm [48], or by building a suffix tree for the final few characters. Alternatively, we could add c dummy characters to the end of the sequence (a sequence of c separator characters[5] would not be searched for, hence would not interfere with results) allowing all meaningful suffixes to be entered into the index. The complete algorithm for finding all the occurrences of a target pattern of length less than or equal to the compressed depth is given in Figure 3.10. Note that if we are solely testing for the presence of the string, and not locating each occurrence of the query string, then we can simply return true as soon as a sub-tree is found, thus avoiding scanning the complete range of possible locations in the two-level array.

[4]The final $c-1$ suffixes are not entered into the TCST as we cannot calculate a c-character prefix for such suffixes and hence cannot allocate them a location in the two-level array.

[5]A *separator character* is used to separate sections of the sequence, such as those corresponding to individual chromosomes.

```
initialise result set

set length to be query length
calculate code for complete query
if length = c then
   if two-level array entry for code is null then
      return result set
   else
      get sub-tree for code
      traverse sub-tree adding locations to result set
   end if
else
   { Match against the TCST }
   set size to be power(alphabetLength,c - length)
   set start to be code * size
   set end to be start + size - 1
   for i in start to end loop
      if two-level array entry i is not null then
         get sub-tree for i
         traverse sub-tree adding locations to result set
      end if
   end loop

   check for match in last c characters of sequence
end if

return result set
```

Figure 3.10: Finding all occurrences of query string of length less than or equal to that of the compressed depth using a TCST.

For particularly short target patterns, it may be found that the performance of this technique will be marginally worse than that of a suffix tree. However, if such short target patterns are of particular interest (which is rarely the case) the size of the compressed depth can be chosen accordingly (a shorter compressed depth will result in smaller ranges to be explored). Additionally, for particularly short search targets it may be preferable to serially scan the sequence rather than making use of an index. Section 4.6.3 explores the performance of finding short target patterns.

3.7 Incremental Faulting

Given that one of the aims of this data structure is to support the use of indexes that are larger than the available main memory, it is clearly not possible to always expect a desired section of the index to be present in main memory. This prompts the need for incremental faulting—bringing in sections of the index on demand.

All faulting actions take place upon access to the two-level array. The contents of the two-level array are essentially a cache of the disk-based index, with some sub-set of the index's sub-trees being present in main memory at any given time. Thus, with a TCST, faulting consists of identifying a required sub-tree that is not currently in main memory and transferring that sub-tree (if it exists) from disk to main memory. Note that there are many alternative schemes that can be used to manage faulting. The scheme given here gives the basic faulting operation for the TCST, which may be extended to suit particular applications.

The absence in main memory of a required sub-tree is indicated by attempting to access a null rib in the two-level array, or by locating a null entry in a rib of the two-level array. In both cases faulting begins by accessing the relevant backbone entry on disk (after selecting the correct partition for the entry). In the first faulting case it is possible that this entry may lead to a null rib (as indicated by a negative offset), meaning that the requested sub-tree is not present in the index. If the entry indicates that a rib is present (as is guaranteed in the second faulting example), we then access the on-disk representation of the rib. If the relevant entry in the on-disk rib indicates a null sub-tree (negative offset) the sub-tree is not present in the index and no further action is required. If a sub-tree offset is found, we then seek to that position and transfer the on-disk sub-tree to main memory and enter the sub-tree at the correct location in the two-level array. Note that in the first faulting case this will also require the creation of a new (empty) rib. Figure 3.11 illustrates the first faulting case, where a new rib is created prior to the faulted in sub-tree being added to the in-memory index.

Known Null Entries

Here we have used null entries in both the backbone and ribs of the in-memory two-level array in order to indicate sections of the index that are not in main memory. We also need to distinguish between entries that are known to be null (i.e. are not part of the index) and entries that are not yet known (i.e. those that require the on-disk index to be examined). This can be achieved by having all 'known null' ribs point to one object that represents a null rib. This incurs no space overhead as all such references point to

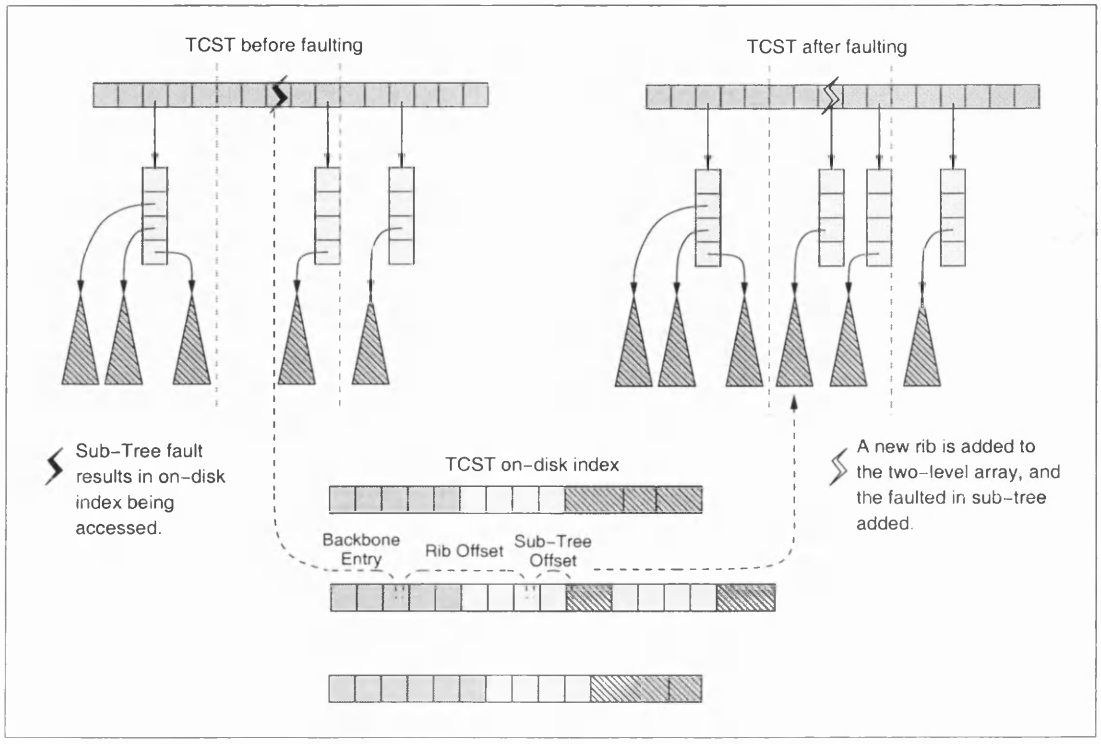

Figure 3.11: Sub-Tree faulting in a TCST.

one trivial object and will help avoid unnecessary disk access. A simple equality test can then be used to distinguish between known null entries and valid entries. A similar technique can be used for sub-trees.

3.7.1 Eviction Management

When supporting a long-lived query server we would aim to use nearly all of the available main memory to cache the disk-based index (as this would improve performance). However, we must also provide a means for releasing memory to allow continuous operation. This can be achieved by monitoring the current free memory, and when this goes below a threshold value we choose sections of the two-level array to evict. Eviction is achieved by setting entries in the two-level array to 'null' and allowing them to be garbage collected. Regions will continue to be evicted until the amount of free memory is above the set threshold. Currently active sub-trees (i.e. those being used to answer a current query) will not be garbage collected until the query is complete. Any appropriate strategy can be used to determine which range of entries to set to null,

including random, round-robin and second chance.

3.8 Justification of Design

The design of the TCST reflects some assumptions about how such an index is to be used. In particular, this data structure has been designed to be a write-once read-many index. If the indexed data changes we will have to dispose of the current index and create a new index over the revised data. Most suffix based indexes follow this trend: it can be shown that updating a suffix tree for a dynamically changing text has a lower bound equal to that of suffix tree construction [12]. However, when indexing texts that either do not change or infrequently change this is not a problem, so long as index creation is not longer than the period of text revision. Genomic data can be classified as reference data (finished sequences are rarely altered, although errors may be corrected, and draft sequences are typically revised only a few times each year). Thus, this form of index is suitable for genomic sequence data.

As commented on in Section 3.6.2, searching for a particularly short target pattern (say less than four characters long) may not be efficient using this index, and it may be preferable to serially scan the sequence. However, such short queries are rarely of interest to those working with genomic data as the number of results produced by such a query would be very large (up to hundreds of millions of results for a 3 Gbp sequence) and beyond useful analysis. Additionally, the index can be tuned at the point of construction to improve the performance of finding short target patterns (lowering the value of the compressed depth is expected to give improved performance for short queries) and a serial algorithm could be provided as a complement to searching using the index for this case. Initially, this structure has been designed to solve matching problems, and may not be directly suited to pattern discovery.[6]

The decision to represent the first c characters of each prefix via a two-level array rather than using tree nodes was based upon two observations. The first is that in order to allow sub-trees to be treated independently we need to be able to represent the top part of our index in such a way that there is no contention when indexing each partition. The second is that the density of suffix trees is at its greatest nearest the top of the tree.

When using Hunt's partitioned tree for an index requiring several partitions, it can be seen that the nodes near the top of the tree may be shared between more than one

[6] *Pattern Discovery* aims to find new *patterns* in a given set of sequences with the hope that such patterns may be important in determining the biological function of the sequences [23].

partition. If we consider the case when eight partitions (`[AA..AC]`, `[AG..AT]` through to `[TA..TC]`, `[TG..TT]`) are used, it is clear that each node at a depth of one will be shared between two partitions and that the number of shared nodes will increase as the number of partitions increases. General purpose persistence mechanisms, such as PJama, can resolve such issues when creating the index on disk, although contention for access to nodes would still require consideration if multi-threaded construction is desired. Additionally, when using bespoke persistence technology, such sharing of nodes between partitions would require a more complex disk-based index. Thus, by choosing an array based representation for the top of the index such contention is avoided as no two partitions will share an entry in the array.

Through examination of two different measures of tree density, it can be seen that the top of a suffix tree is considerably more dense than the deeper sections of the tree. Figure 3.12 shows that for a range of suffix trees[7] the number of nodes at each depth grows rapidly until a depth of between eleven and thirteen is reached (a node is said to be at depth x if the length of the path label to that node is x). Thus, a sizeable amount of memory will be used to represent nodes that are near the top of the tree—in fact, it was found that the top twelve layers typically accounted for one third of the total space occupied by the tree. This suggests that a more compact representation may be possible and that the performance of finding large target patterns could be improved by allowing more direct access to deeper parts of the index. Figure 3.13 shows the number of prefix combinations found within each sequence for prefix lengths in the range one to thirty when using an alphabet of five characters (this is expressed as a percentage of the theoretical maximum). From this it can be seen that the percentage of prefix combinations present within each sequence drops rapidly, reaching less than 50% at a depth of three and less than 1% at a depth of thirteen. An array-based prefix representation will be sparsely populated: thus, the two-level array is chosen as it can be tailored to provide a compact representation of a sparsely populated array. If, for a given sequence, it is possible to use an alphabet size of four (i.e. the sequence contains no 'unknown' characters) then the percentage of prefix combinations used at each level would be higher, significantly so for short depths. However, the density would still drop rapidly as prefix length increases.

[7]The sequences used in this section range from 3 Mbp to 27 Mbp in length, and will be discussed in Section 4.2.

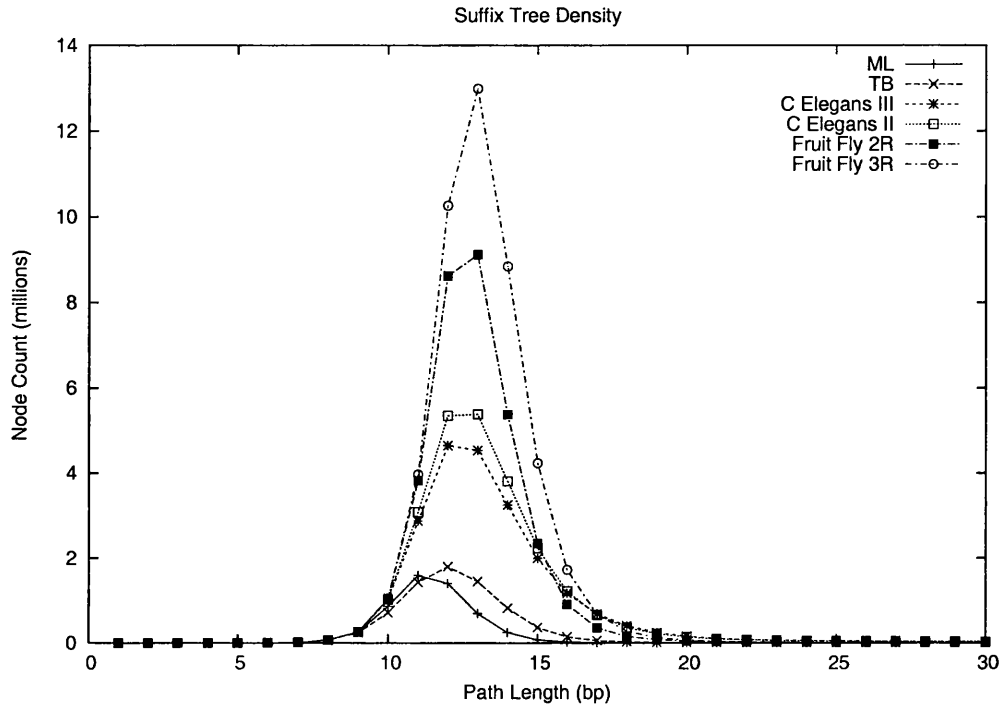

Figure 3.12: Density of suffix trees for path lengths 1–30 as number of nodes at each depth.

3.9 Implementation Notes

The previous sections introduced the Top-Compressed Suffix Tree, giving details of both data structure and algorithms. We now go on to discuss notable aspects of the TCST implementation used to provide the performance analysis given in Chapter 4. The primary language used for this implementation was Java^{TM} (version 1.4.1), a platform independent object-oriented language (see Section 1.4.1).

The various components that constitute this TCST implementation can be split into five main categories: sequence representation, which deals with encoding and decoding of both sequence and alphabet; the distributed server, which co-ordinates parallel index creation; a TCST construction algorithm implementation; the persistence layer, which manages all file related activity; and finally, the query engine which supports exact matching over a disk-based TCST.

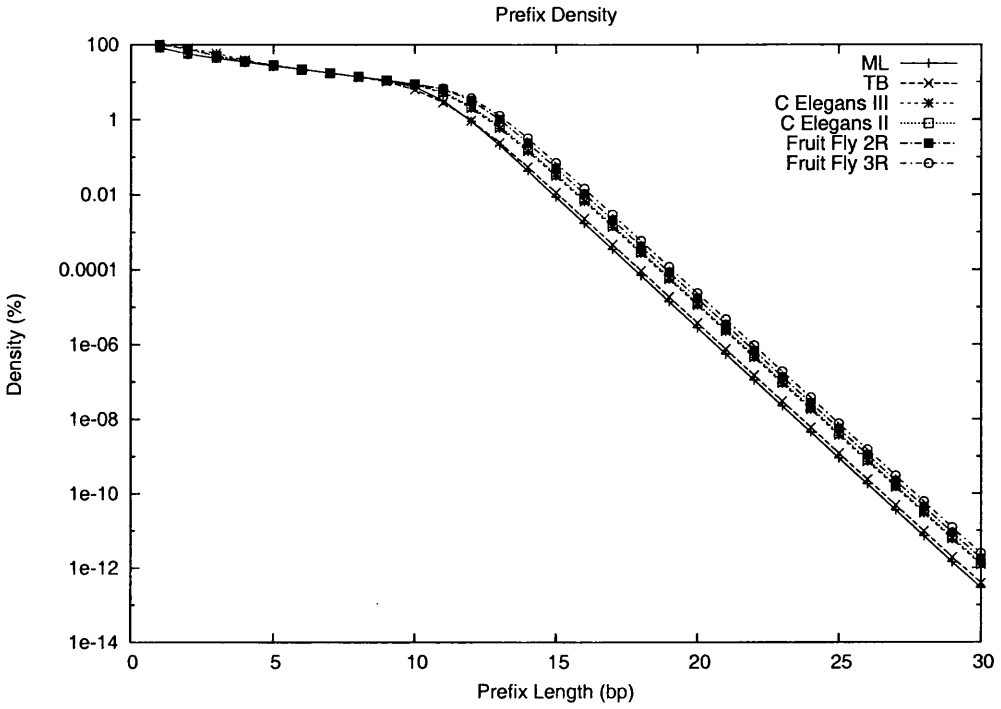

Figure 3.13: Prefix density for prefix lengths 1–30 as a percentage of the theoretical maximum.

3.9.1 Sequence Representation

In order to maximise the amount of memory available for index creation, a compact sequence representation was used. This representation packed several base pairs into one 32-bit integer. Individual characters can be extracted from a given integer by using appropriate shifts and masks (the masks are computed once and subsequently stored). The minimum number of bits required to store one character from an alphabet of length a is given by $\lceil \log_2 a \rceil$. For example, if we have an alphabet of length five (a DNA, or RNA, alphabet with an additional 'unknown' character) the minimum number of bits to represent one character is three, allowing ten characters to be stored in one 32-bit integer (whereas allocating one byte per character would give four characters in one 32-bit integer). Adopting such a representation necessarily incurs a performance penalty when accessing the sequence (in comparison to using the languages native representation). However, this can be minimised by providing efficient access methods that allow substrings to be extracted (thus avoiding repeated shifts that would be necessary if accessing each character independently). Using a compact sequence representation allows larger sequences to be indexed for a given amount of available main memory,

and reduces the number of partitions required. 32-bit integers were used for all text indices, including the left labels of the suffix sub-trees, thus the largest sequence that can be indexed using this implementation is 2 Gbp.

3.9.2 Construction

The pre-processing stage of the Improved Prefix-Partitioned TCST Construction Algorithm has been implemented using the Two-Pass Suffix Grouping Algorithm. However, it was found that construction performance can be improved by storing the prefix codes (which are calculated when grouping the suffixes) in addition to the suffix numbers. Therefore, this implementation can optionally store the prefix codes in the grouped suffix file. Distributed TCST construction, as described in Section 3.5.3, has been implemented using Java™ Remote Method Invocation (RMI), which provides a means for the necessary components to communicate.

Distributed Garbage Collection within the JVM

The Java Virtual Machine provides support for distributed garbage collection. In order to allow accurate identification of garbage objects where remote references are possible (as is the case when RMI is used), the JVM initiates frequent local garbage collection cycles. The default maximum time between each local garbage collection is one minute.[8] This proves to be problematic when the local heap contains a large number of objects (as is the case with the construction large trees): if each garbage collection takes in excess of one minute, then more processing time will be dedicated to garbage collection than to useful work. In such cases it will be necessary to reduce the frequency of local garbage collection. This can be achieved by setting the system properties described in Table 3.3.

Inheritance from the class `java.rmi.server.UnicastRemoteObject` is the primary criterion for the distributed garbage collection system to be used. Within the TCST implementation, only the class implementing the Job Server inherits from this class. Therefore, there will only be two remote references per construction client, and the amount of garbage created that is only detectable via distributed garbage collection will be insignificant. For the TCST implementation, the maximum time between local garbage collection cycles was reduced to once every fifteen minutes (although the JVM will initiate other garbage collection cycles as dictated by other criteria, such as the

[8] Tuning Garbage Collection with the 1.4.2 Java™ Virtual Machine, Sun Microsystems Inc., `http://java.sun.com/docs/hotspot/gc1.4.2/`, as accessed April 2004.

JVM System Property	Purpose
`sun.rmi.dgc.server.gcInterval`	Sets the frequency of garbage collection on the remote objects.
`sun.rmi.dgc.client.gcInterval`	Sets the frequency of garbage collection on the local heap.

Table 3.3: JVM System Properties.

amount of available free memory).

3.9.3 Persistence Layer

The implementation of the TCST is such that algorithms that query the index can be implemented without having to make allowances for the fact that a necessary part of the index may or may not be resident in main memory. This is achieved by implementing the persistence layer wholly within the class corresponding to the two-level array.[9] By having all operations which require disk access contained within one class and restricting access to this class to a few specific methods, it is possible to make the faulting of required index sections transparent to the query code that makes use of them. In particular, the value returned by any of the supported accessor methods will be the same regardless of whether or not the on-disk index was accessed.

Four methods are provided by the class representing the two-level array to support different types of query that may operate over the TCST (see Table 3.4). The first two, `get` and `processRange`, provide a means of accessing required sub-trees (reading them from the on-disk index if necessary) and can support any algorithm that queries the TCST by traversing one or more sub-trees. This includes the exact matching algorithm and testing for the presence of a target string of a length that exceeds that of the compressed depth. The second two methods, `test` and `entryInRange`, are provided to support algorithms that need only know if a given sub-tree is present in the index or not (for example, if we are testing for the presence of a target string that is equal in length, or shorter than, the compressed depth). Although such algorithms could be implemented using the first two methods, this would incur an unnecessary performance overhead—in some cases a complete sub-tree would be brought in from disk and not subsequently accessed. Hence, methods are provided that simply test for the presence

[9]Note that, although all index operations that may invoke disk access are contained within one class, it is not necessary to have the methods that actually read and write the on-disk index in this same class. Such methods can be implemented using additional classes, although these classes should not be exposed to the areas of the code implementing query algorithms.

Method Name	Parameters	Description
`get`	`int i`	Returns the sub-tree stored at entry `i` in the array, `null` if no such sub-tree.
`processRange`	`start s` `end e`	Processes each non-null sub-tree in the specified range using the supplied node processor.
`test`	`int i`	Returns `true` if entry `i` corresponds to a non-null sub-tree, `false` otherwise.
`entryInRange`	`start s` `end e`	Returns `true` if a non-null entry is found in the specified range of entries, `false` otherwise.

Table 3.4: Two-level array accessor methods.

of the sub-tree in the index and would not cause the sub-tree to be faulted (although the backbone and rib entries of the on-disk two-level array may be accessed).

The implementation of the faulting operation that may be invoked when calling either `get` or `processRange` is based on the technique described in Section 3.7. Firstly, the amount of available main memory is checked and if it is found to be below a specified threshold eviction will take place. The previous two steps are repeated until the amount of available main memory exceeds the chosen threshold value. After this, the file corresponding to the required partition is retrieved from the list of available partitions. The backbone and rib entry (assuming the rib is present) are then accessed in turn by locating the corresponding locations in the file and reading the stored offset values. The sub-tree (if present) is then accessed by seeking to the start of its on-disk image and then reading in the marshalled sub-tree (see Section 3.4.2). Once complete, the sub-tree is added to the correct location in the two-level array, where it will remain until chosen for eviction. If either a null rib or null sub-tree is accessed on disk then a reference to the corresponding 'known null' marker is added to that location in the two-level array.

If, when calling either `test` or `entryInRange`, it is necessary to access the on-disk index, the process will begin in an identical manner to that of the method described above. However, once the sub-tree offset has been read from the appropriate rib, these methods do not then need to read in the corresponding sub-tree (since at this point they have enough information to determine if the sub-tree is present or not). As described above, any null ribs or null sub-trees encountered are recorded. One limitation of this technique is that information relating to a found sub-tree is not recorded in the main-memory index, so, if the query is repeated at some point in the future, the same

information will need to be read from disk. This can be avoided by introducing a special node that corresponds to a 'known non-null node' and adding a reference to this node at the appropriate location in the two-level array, thus eliminating the need to re-read this information from disk. When this technique is adopted, it is necessary to introduce additional checks to ensure that this marker node is not treated as a valid sub-tree. (This also has to be done with the 'known null' markers described previously.)

3.9.4 Query Server

Given a completed on-disk TCST, a query server can be initialised using the following three steps. Firstly, the specified sequence and parameter values are read into main memory. Next, an index is created to provide efficient access to the collection of files that constitute the on-disk TCST. Given that in this implementation partitions are assumed to be of equal size (with the possible exception of the final partition), the index to the files is simply an array of the file handles. Therefore, the file corresponding to a specific prefix is given by entry $\lfloor \frac{PrefixCode}{PartitionSize} \rfloor$ in the array. The final step in creating a query server instance is to initialise an empty two-level array to represent the top of the index. One simple optimisation can be carried out at this stage: the backbone sections of the on-disk index can be scanned and all 'null rib' entries can be entered into the two-level array. This incurs no space overhead (all 'known null' entries point to the same object), and avoids disk access when answering a query where a null backbone entry would have been accessed on disk.

3.10 Summary

This chapter has introduced the Top-Compressed Suffix Tree, a disk-resident suffix-based index that can be used to index large biological sequences. Details of the design of both in-memory and disk-resident versions of the structure were described together with techniques for converting between the two representations. The necessary algorithms for constructing and querying the structure were given, each allowing for the use of the structure when its total size exceeds that of the available main memory. Of particular note are the Improved Prefix-Partitioned Construction algorithm and the distributed variants of the construction algorithms as they have the potential to dramatically improve the performance of index construction. A justification of the index design was then given, as was a summary of noteworthy implementation details. The performance of this index is explored in the following chapter.

Chapter 4

The Top-Compressed Suffix Tree: Performance Evaluation

This chapter presents the results of a number of experiments that explored various aspects of the performance of a Java implementation of the TCST (as described in the previous chapter). Top-Compressed Suffix Trees were constructed for a variety of DNA sequences. These sequences ranged in length from 3 Mbp for a complete bacterium genome, through to 1.5 Gbp for chromosomes 1 to 8 of the human genome. The size of completed indexes, the behaviour of different construction techniques and the performance of sample queries are all explored in detail. Additionally, we explore parameter sensitivity to establish which operational parameters impact upon which aspects of index performance. Such information can enable clients of the index technology to select the values for operational parameters that best suit their needs.

4.1 Experimental Process

One of the main reasons for performing the experiments discussed in this chapter was to explore how the values of various operational parameters effect performance of the TCST. However, an exploration of all possible combinations of parameters would be infeasible as the parameter space would be too large. Hence, the values of operational parameters explored here are those that were identified during the design stage as being likely to have a significant impact on the performance of the TCST. Parameters, such as buffer sizes, that have a predictable impact upon performance were not explored in detail and are omitted from the results presented here. Furthermore, it was observed during the design phase that values of the rib size and compressed depth both affect the

nature of the two-level array, and thus different combinations of these parameters were explored when gathering results. Note that, unless stated otherwise, all comparisons of algorithms and index representations make use of the same test data.

Several of the experiments presented in this chapter make use of measurements of total execution time as the primary means of comparing the efficiency of different indexes. As previously discussed in Section 1.3.1, this method has been selected in preference to techniques such as algorithmic analysis or counting the number of operations as these techniques do not reflect the complexity of the memory hierarchy in a typical modern computer. In particular, the objects comprising a given object graph can be arranged in main memory in numerous different ways, meaning that the pattern of memory accesses can differ widely for a particular sequence of operations over equivalent object graphs. Such variations can lead to significant differences in the time taken to perform a given task—differences that cannot be predicted or measured through techniques such as counting the number of operations. All of the experiments that make use of execution time were performed on a lightly loaded computer, thus contention for resources was minimal.[1] Furthermore, these experiments were repeated at least twice to ensure that the trends observed were consistent and that they could be replicated. Overall, it was found that the variance in the measured execution time was small.

4.1.1 Test Environment

All experiments described in this chapter were undertaken using a lightly loaded IBM xSeries 235 server, with two 2.8 GHz Intel Xeon™ processors and 6 GB of RAM. The operating system was based on GNU/Linux version 2.4.20-20.9 and all code was developed using the Java™ 2 Runtime Environment, Standard Edition, version 1.4.1 (the server variant of the Java virtual machine was employed for all performance measurements). Note that only 2 GB of main memory is available to the Java virtual machine. The processors used in this system make use of Hyper-Threading technology to improve the performance of multi-threaded applications [60]. One consequence of this technology is that the operating system will treat each physical processor as two distinct logical processors, with process scheduling influenced accordingly.

[1] The only processes running on the computer were those of the operating system and those of the TCST implementation (no external services were provided by the computer).

4.2 Sample Data

The DNA sequences used for the evaluation of the TCST were obtained from various publicly accessible repositories of genomic data. Each sequence was stripped of all formatting, leaving a file containing only the characters that constitute the sequence. Additionally, some sequences contained large sections where sequencing has not yet been completed, resulting in long substrings consisting solely of the character **N**. Such sections were removed from the sequences, as they are of little interest to those working with sequence data.[2] For the purposes of the experiments described here, all substrings consisting of the character **N** repeated one hundred or more times were replaced by a single character **N**. Where appropriate, individual chromosomes have been concatenated to create a single input file. Table 4.1 lists all of the sequences used during the evaluation. Sequences 1 and 2 are complete genomes of the bacteria Mycobacterium Leprae[3] and Mycobacterium Tuberculosis[4], both obtained from the Wellcome Trust Sanger Institute. Sequences 3, 4 and 5 are selected chromosomes of Caenorhabditis Elegans (a small worm), obtained from WormBase[5]. Sequences 6 to 10 where obtained from UCSC Genome Browser[6], and consist of selected chromosomes from the following species: Drosophila Melanogaster (a species of fruit fly), Fugu Rubripes (puffer fish), Rat and Human.

	Organism Name	Version	File Length
1	Mycobacterium Leprae	Final	3 Mbp
2	Mycobacterium Tuberculosis	Final	4 Mbp
3	C. Elegans, chromosome II	May 2003	15 Mbp
4	C. Elegans, chromosome III	May 2003	13 Mbp
5	C. Elegans, all chromosomes merged	May 2003	100 Mbp
6	Drosophila, chromosome arm 2R	June 2003	20 Mbp
7	Drosophila, chromosome arm 3R	June 2003	27 Mbp
8	Fugu Rubripes	April 2002	316 Mbp
9	Rat (chromosomes 1–5)	June 2003	965 Mbp
10	Human (chromosomes 1–8)	July 2003	1482 Mbp

Table 4.1: Genomic data used for performance evaluation.

[2] If substrings are removed from a given sequence, for example, when omitting sections that have not yet been sequenced, the locations of such omissions must be recorded in order to allow accurate results to be reported.

[3] Obtained from `http://www.sanger.ac.uk/Projects/M_leprae/`, August 2003.

[4] Obtained from `http://www.sanger.ac.uk/Projects/M_tuberculosis/`, August 2003.

[5] Obtained from `ftp://ftp.sanger.ac.uk/pub/wormbase/WS100/CHROMOSOMES/`, August 2003.

[6] Obtained from `ftp://genome.cse.ucsc.edu/goldenPath/`, August 2003.

4.3 Storage Requirements

The disk-based TCST was designed to allow on-demand access to relevant parts of the index while avoiding having to read in large sections of the index in order to perform a simple query (incremental faulting). A secondary aim was to take advantage of the bespoke nature of the persistence solution in order to provide a compact disk-resident index representation. In particular, it was intended that the on-disk space requirements of the TCST would be significantly less than the disk space occupied by suffix tree implementations deployed using generic persistence platforms. Reducing the storage requirements allows a greater number of such indexes (and corresponding sequences) to be stored in a given database and using a compact representation for the suffix sub-trees reduces the amount of data that must be read from secondary storage in order to transfer a sub-tree from disk to main memory.

When using the on-disk representation discussed in Section 3.4, two operational parameters affect the size of the completed TCST. They are the compressed depth, c, and the rib size, $d2$. The compressed depth determines the depth at which the data structure is split and, correspondingly, the capacity of the two-level array. Given the capacity of the two-level array, the ratio between backbone size, $d1$, and rib size, $d2$ is determined from the chosen rib size. This section discusses how the size of on-disk TCSTs grows with sequence length and how size is affected by the operational parameters.

4.3.1 On-Disk Index Size

TCSTs were created over a variety of DNA sequences using compressed depths of 8, 9, 10 and 12 along with both a rib size of 32 and a rib size of 64. Figure 4.1 shows how the size[7] of the completed on-disk indexes grows with sequence length. Only the results from using indexes with a rib size of 32 are shown as equivalent trends are obtained using a rib size of 64. As expected of a data structure based on the suffix tree, the on-disk index size is directly proportional to the sequence length. For each sequence indexed, the most compact on-disk representation was obtained using a compressed depth of 8. However, as sequence length increases, the difference in the size of index obtained using different compressed depths decreases. For example, the size of the indexes created over the longest sequence (Human Chromosomes 1–8, 1.5 Gbp) differed by less than 0.6%, whereas for the shortest sequence (Mycobacterium Leprae, 3.2 Mbp) the difference in

[7]The total space required for a given on-disk TCST was measured using the disk usage command, du, on the relevant directory.

size between the smallest and largest TCST was over 600%.

Figure 4.2 shows how the relative index size (in terms of bytes per input character) changes with sequence length. For the shortest sequences indexed, there is a notable difference in relative index size for each of the compressed depths used. However, as the length of sequence increases, the relative index sizes converge. This trend is due to the amount of free space present in the two-level array of the TCST: if the capacity of the two-level array greatly exceeds that which is required, the amount of free space in the array may contribute a significant amount to the overall space required to store the index. On the other hand, with larger sequences a greater number of entries will be used in the two-level array, resulting in significantly less wasted space.

Each entry in the two-level array corresponds to exactly one c-letter prefix, therefore the maximum number of two-level array entries that can be required when indexing a given sequence is $n - c$ (where n is the sequence length). However, the two-level array must be able to accommodate every possible combination of c letters drawn from the given alphabet. In practice, the number of entries used will be significantly smaller than this (the prefixes of the suffixes of a DNA sequences are not evenly distributed, see Section 4.5.1). It then follows that providing a two-level array with a minimum size that exceeds $n - c$ will lead to an inefficient representation (the minimum size of the two-level array being determined by the size of the backbone). The minimum fixed cost of storing the two-level array is one 32-bit integer for each location in the backbone. If this exceeds $n - c$ then a relatively large amount of the total space requirement of the on-disk index will be used to store null values. For example, if using a compressed depth of 12, a rib size of 32 and a DNA sequence of length 3.2 Mbp, the backbone of the two-level array will occupy over 30 MB, which is almost ten times the size of the original sequence, whereas the backbone would only occupy 49 KB on disk if a compressed depth of 8 is used. Clearly, for smaller sequences there are advantages to using a lower value for the compressed depth (see Section 4.3.2). On the other hand, for larger sequences the difference in index size is small, and therefore the size of the compressed depth will be determined by other aspects of index performance.

The choice of rib size was found to have minimal effect on the overall size of the index. In particular, for compressed depths of 8, 9 and 10 the difference in size between corresponding indexes was found to be less than one percent. In most cases, using a rib size of 32 gave a smaller index than using a rib size of 64. However the optimum choice of rib size will depend on the exact nature of the sequence being indexed. Larger differences were observed when using a compressed depth of 12. With the two shortest sequences indexed (the two bacteria genomes), a difference of seven percent was found

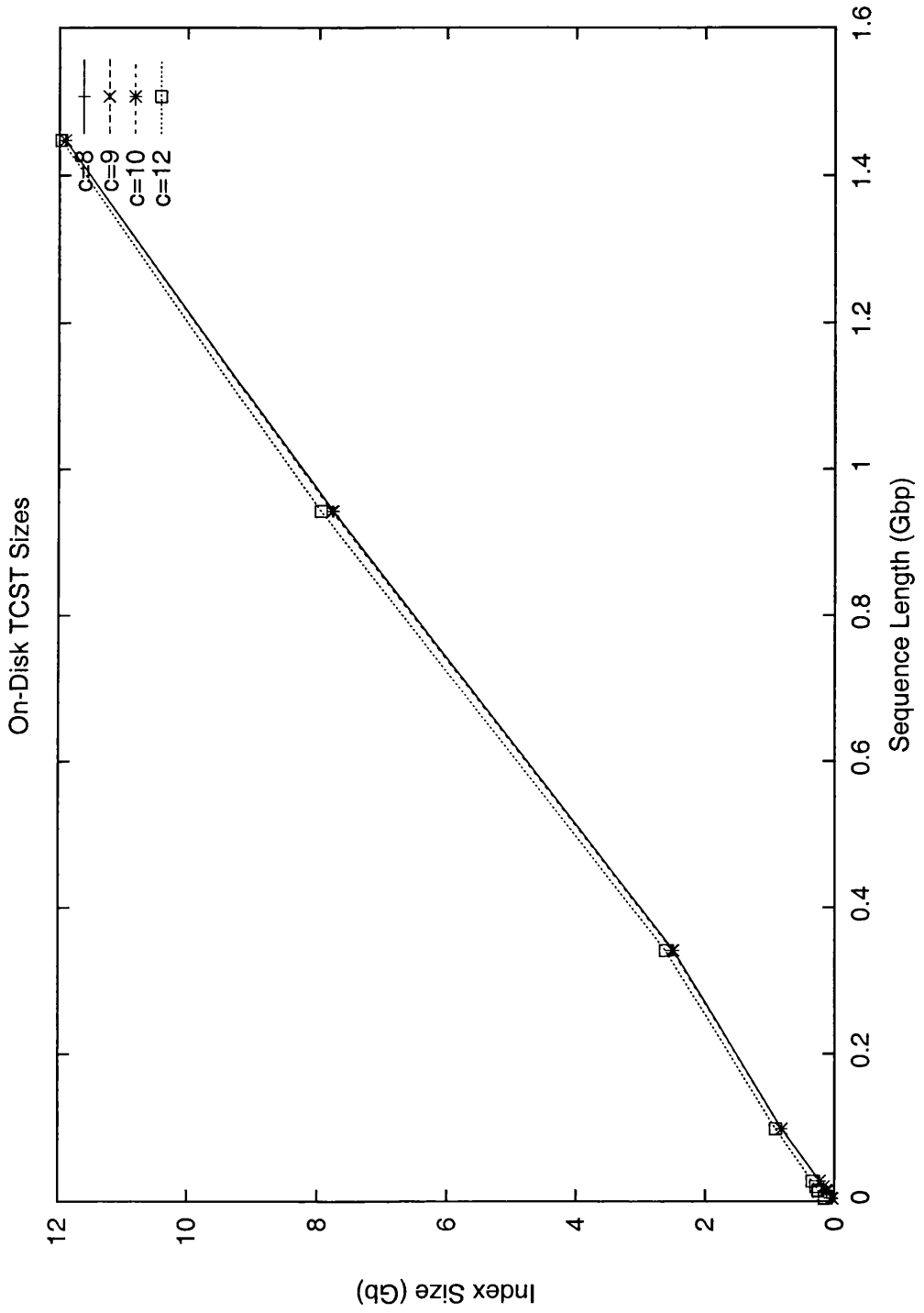

Figure 4.1: On-disk Top-Compressed Suffix Tree space requirements.

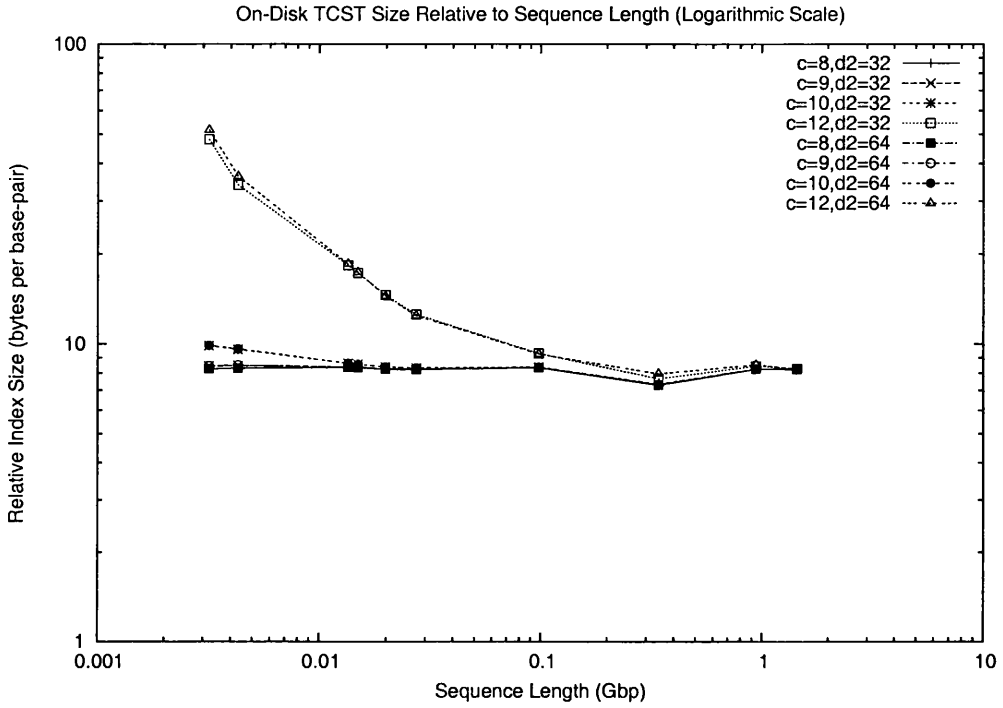

Figure 4.2: Relative sizes of on-disk TCSTs (logarithmic scale).

between corresponding indexes. Additionally, one of the larger sequence files (Fugu genome, 350 Mbp) produced a four percent difference. From Figure 4.2 it can be seen that altering the rib size does not affect the overall trend of index size growth and that only a small difference in index size is observed between corresponding indexes of different rib sizes.

Taking the smallest TCST created for each sequence, the average on-disk index size was found to be 8.17 bytes per base pair. The sizes for individual indexes ranged from 7.28 bytes per base pair to 8.83 bytes per base pair, with the majority of indexes requiring between 8.22 and 8.35 bytes per base pair. The same average (to four decimal places) was obtained over the TCSTs created using a compressed depth of 8 and a rib size of 32, suggesting that this combination of parameters is a suitable choice when attempting to minimise the on-disk size of the completed index. Additionally, an average size of 8.20 bytes per base pair was obtained when using an increased compressed depth as sequence length increased. Table 4.2 shows the sequence lengths at which the compressed depth was increased—these lengths are derived from the points on Figure 4.2 at which the difference in relative index size between corresponding indexes of different compressed depths converge to within 0.5%. This result shows that using an

Sequence Length	Compressed Depth
<27 Mbp	8
27 Mbp	9
316 Mbp	10
1482 Mbp	12

Table 4.2: Sequence lengths at which the compressed depth is increased.

increased compressed depth for larger sequences, as may be beneficial to other aspects of performance, has only minimal impact on index size.

4.3.2 Parameter Sensitivity for Short Sequences

In Section 4.3.1, it was observed that altering the values of the operational parameters produced greatest variation in on-disk TCST sizes when indexing shorter sequences. We now explore how the size of on-disk TCSTs changes over a wider range of values for both compressed depth and rib size when used with relatively short sequences. Figures 4.3 and 4.4 show how these parameters affect the size of TCSTs over sequences of lengths 3.2 Mbp and 28 Mbp respectively. In both cases, using compressed depths between 2 and 8 produced negligible differences in the size of the completed index. Additionally, within this range of compressed depths, the choice of rib size also had negligible impact.[8] However, once the compressed depth is increased beyond 8 both parameters affect the size of the index.

As discussed in Section 4.3.1, the amount of space used to represent the backbone of the two-level array greatly contributes to the size of the of the completed index (when indexing relatively short sequences). This is confirmed in Figures 4.3 and 4.4, where the largest TCSTs were obtained when using a rib size of 4 (lowering the rib size increases the backbone size, and vice versa). This suggests that using a larger rib size may be beneficial when indexing shorter sequences. However, in both cases using a rib size of 128 produced indexes larger than those obtained using rib sizes of 16, 32 and 64. As with the longer sequences, the most compact indexes were obtained using a rib size of either 32 or 64. Although altering the values for the compressed depth and rib size produced greatest variation in index size when indexing shorter sequences, the overall trends identified in Section 4.3.1 remain, with the additional caveat that reducing the compressed depth below 8 does not achieve any further reduction in the

[8]Note that it is not possible to use a rib size greater than 32 when using a compressed depth of 2 and an alphabet size of 5, as this would cause the two-level array to degenerate into a linear array.

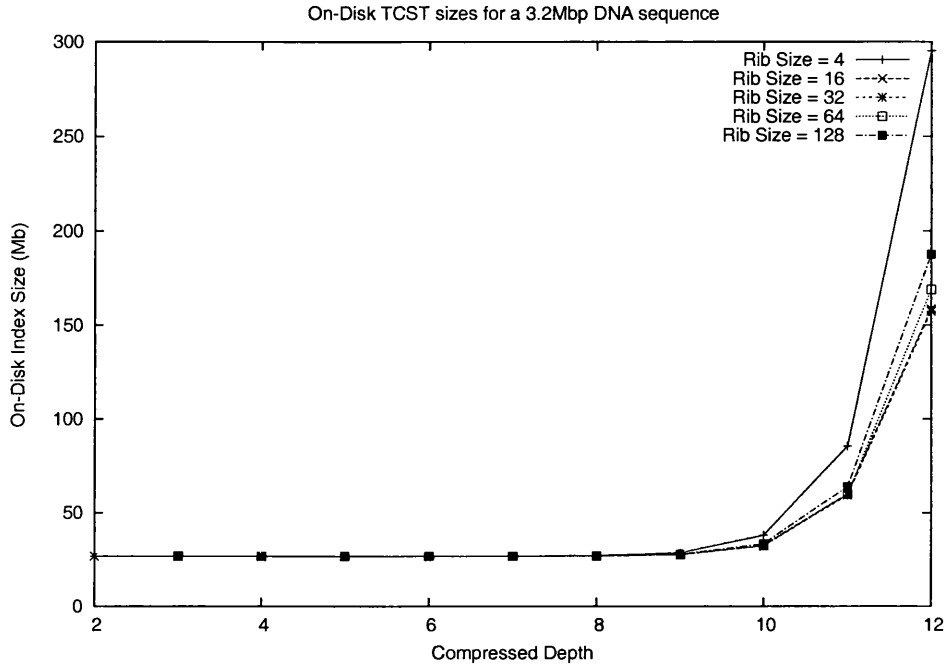

Figure 4.3: Sizes of on-disk Top-Compressed Suffix Trees over a 3.2 Mbp sequence.

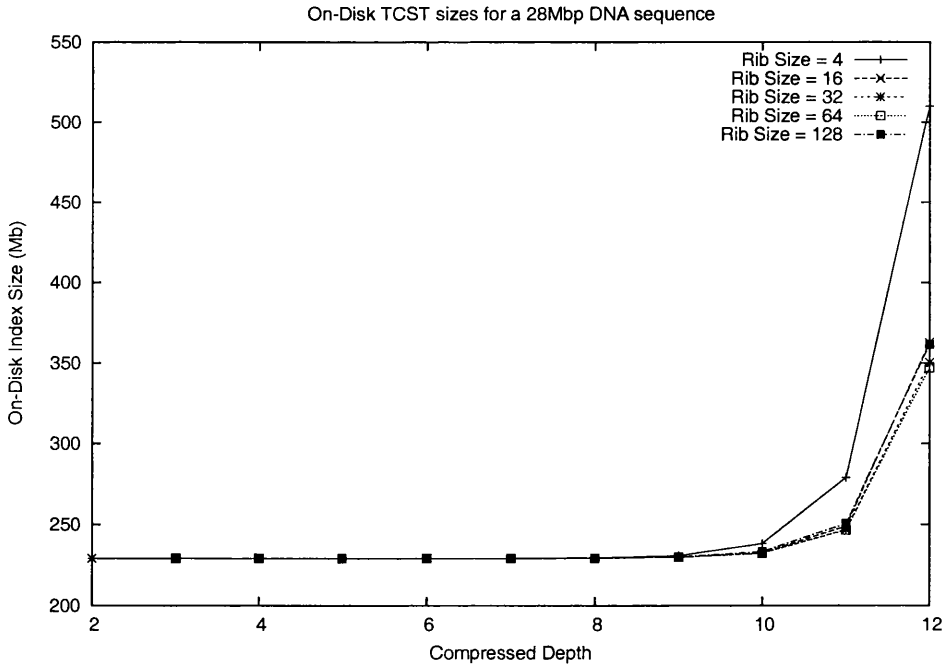

Figure 4.4: Sizes of on-disk Top-Compressed Suffix Trees over a 28 Mbp sequence.

amount of space required by the index.

4.3.3 Comparisons With Other Representations

We now compare the space requirements of the on-disk TCST with other published implementations of the suffix tree and related structures. The space requirements given in this section are those that were reported by the original authors when evaluating their data structures against DNA data. This means that the comparisons being made between the TCST and other forms of index do not use the same test data. In each case, 32-bit values have been used for both references and string indices.

Persistent Suffix Trees The persistent suffix tree of Hunt et al. [57] uses a representation equivalent to the minimal suffix tree described in Section 2.4.1 (as used to represent the sub-trees of the in-memory TCST). Persistence was achieved by using PJama [10], an orthogonally persistent implementation of Java. The longest sequence indexed was 263 Mbp in length, and required 18 GB of storage together with a 2 GB log file (which is used by PJama to provide recovery management [51]). This gives a relative index size of 68.44 bytes per base pair (excluding the space required for the log). More recently, Bedathur and Haritsa [15] propose a persistent suffix tree based upon a bespoke paging system that has been tuned to match the requirements of suffix tree construction. Using a suffix tree representation similar to that of McCreight [81], this technique produced a typical index size of 27.7 bytes per base pair. The buffering system of Tata et al. [99] uses a linear array to represent the suffix tree, giving an average size of 8.5 bytes per character (although, additional space is required during construction). Finally, Harding and Atkinson [52] describe a persistent suffix tree that is comprised of two arrays that together represent the nodes of the suffix tree (without suffix links) in depth-first traversal order. This technique results in an index that occupies an average of 10.3 bytes per base.[9]

Main-Memory Suffix Trees Of the various suffix tree representations given by Kurtz [72], the Improved Linked List Implementation gives the most compact index when used with DNA sequence data. This implementation made use of two tables (one storing leaf nodes, the other storing branches) and occupied an average of 12.55 bytes per base pair. This is the most compact main-memory suffix tree representation published to date that includes suffix links. The Lazy Suffix Tree of Giegerich et

[9]Note that the values provided by Harding and Atkinson [52] relate to protein sequences, however the technique described would give similar values when used with DNA sequence data.

al. [41], which does not make use of suffix links, is reported to require 9.38 bytes per input character when used over a variety of texts. However, the authors acknowledge that the average storage requirement will be higher when used over DNA sequences. The compact vector based suffix tree representation of Brown [24] also omits suffix links, and has an average space requirement of 9.3 bytes per base pair.

Related Structures The suffix array [80], which is often cited as an alternative to the suffix tree, is implemented using two integer arrays, one of length n and the other of length $\frac{n}{4}$ (see Section 2.4). This gives an overall space requirement of 5 bytes per character indexed. The recently proposed SPINE data structure of Neelapala et al. [90] (which is a essentially a horizontal compaction of a trie) is reported to require '*less than 12 bytes per input character*' when used with DNA. Kurtz [72] gives average space requirements for a number of automaton based data structures. They are: the *Directed Acyclic Word Graph (DAWG)* [20], the *Compact DAWG (CDAWG)* [21] and the *Position End-Set Tree* [74] which respectively require 34.03, 22.54 and 26.52 bytes per input character (when used with DNA).

Summary The on-disk Top-Compressed Suffix Tree provides a more compact index than all previously reported suffix tree implementations, bettering the best previously reported result by over 0.3 bytes per base pair. Although, it should be noted that as the same test data was not used in both cases, the significance of the measured difference in average number of bytes occupied per base pair requires further exploration. Additionally, the space requirement of the TCST was smaller than the persistent tree of Hunt [57], which used general purpose persistence mechanisms to support prefix-partitioned tree construction, by over 60 bytes per base pair. Of the various data structures proposed as alternatives to the suffix tree, only the suffix array had lower space requirements than the on-disk TCST. However, the performance of this structure does not match that of the suffix tree for many applications.

4.3.4 Grouped Suffix-Number Lists

When using the Improved Prefix-Partitioned TCST Construction Algorithm, disk space will be required to represent the grouped suffix-number lists. These lists, which can be deleted as soon as index creation is complete, consist of the suffix numbers of the sequence grouped according to which prefix partition they fall within. Optionally, the corresponding prefix codes may also be stored on disk (as this was found to improve performance in some cases). The amount of space required to represent these lists is

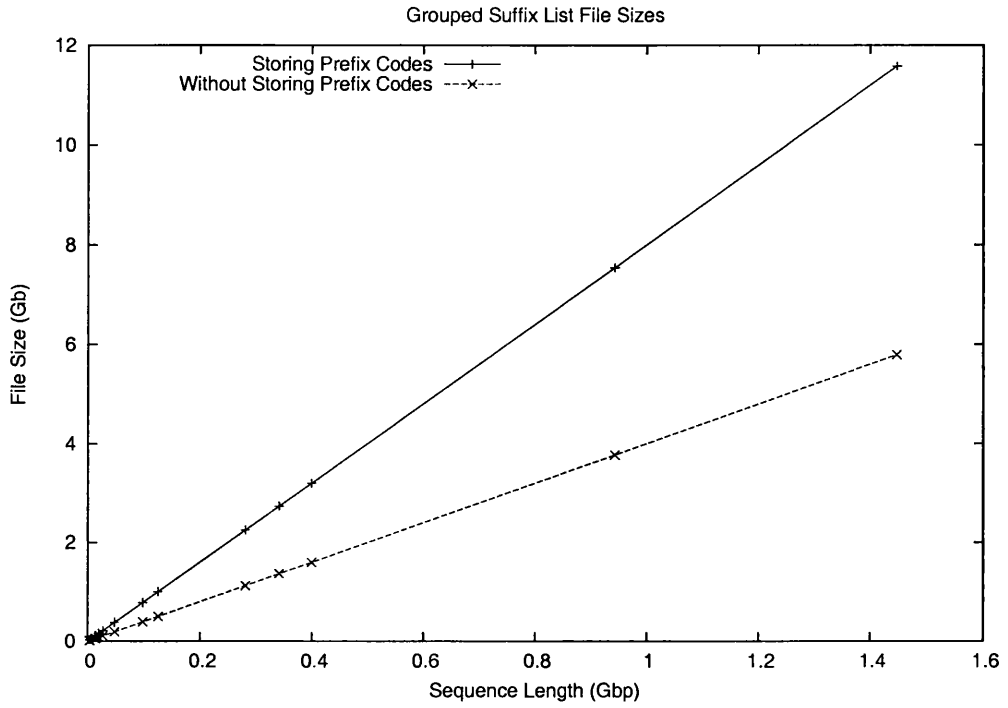

Figure 4.5: Sizes of on-disk grouped suffix-number lists.

directly proportional to the sequence length (see Figure 4.5). When using 32-bit integers (as is the case with the implementation discussed here) the total storage requirement of the relevant suffix-number lists will be approximately $4n$ or $8n$ bytes for a sequence of length n (depending on whether prefix codes are stored).

4.4 Index Construction

The cost of constructing an index can be amortised over the lifetime of the index (which will primarily be determined by the lifetime of the data). With reference data, such as genomic data, this lifetime could range from several months to several years. For example, DNA sequences from completed projects (such as the Mycobacterium Leprae Genome Project[10]) have an indefinite lifetime, whereas the draft sequences from ongoing projects (such as the C. Elegans sequences available form WormBase[11]) have a typical lifetime of several months, with some minor revisions occurring more frequently. Efficient index construction is therefore important as it is essential that the

[10]The Sanger Institute: Mycobacterium Leprae genome project, The Wellcome Trust Sanger Institute, `http://www.sanger.ac.uk/Projects/M_leprae/`, as accessed April 2004.

[11]WormBase Web Pages, `http://www.wormbase.org/`, as accessed April 2004.

time taken to construct a TCST over a draft sequence is significantly less than the time taken for that draft to be superseded. Additionally, the 'turnaround' time may be of interest to those working with the data. That is, the length of the delay between the data becoming available and indexed access to the data becoming available. Finally, TCST construction must be able to scale to accommodate large sequences and index construction must be competitive with alternative suffix tree representations.

This section discusses several aspects of index construction time. For sequences that can be indexed entirely in main memory, we compare the time taken to create a TCST against both naïve and Ukkonen's suffix tree construction algorithms. For larger sequences, the performance of both the Prefix-Partitioned and the Improved Prefix-Partitioned TCST Construction Algorithm are explored. Finally, the use of parallel TCST construction is discussed.

4.4.1 In-Memory Index Construction

Figure 4.6 shows how the performance of in-memory TCST construction compares to both Ukkonen's algorithm [100] and naïve suffix tree construction. Each of the sequences used during this experiment could be indexed using only one partition, i.e. the completed structure can fit entirely in main memory. Values of 8, 10 and 12 were used for the compressed depth, c, of the TCST. As c increases, the amount of time taken to construct a TCST over a given sequence decreases. Correspondingly, if c is reduced to 1, TCST construction will degenerate to that of naïve suffix tree construction and will give identical performance (see also Section 4.4.2). For values of c equal to (or greater than) 10, it was found that TCST construction will out-perform our implementation of Ukkonen's construction algorithm. When using a compressed depth of 12, TCST construction was almost twice as quick as Ukkonen's method for the size of sequence that can be indexed entirely in main memory. The times observed over different runs of this experiment showed a variation in time taken of less than 1%. From these results, it can be seen that little, if anything, is lost in practice by sacrificing the linear-time construction requirement.

The observed improvement in performance over the naïve algorithm can be attributed to the presence of the two-level array in the TCST. Matching the first c characters of a suffix in the TCST requires two steps: the calculation of the prefix code (evaluation of a polynomial, the coefficients of which are the first c characters of the suffix being inserted[12]) and the accessing of the correct location in the two-level array

[12]Note that, when accessing sequential prefixes we can use the optimisation discussed in Section 3.3.1.

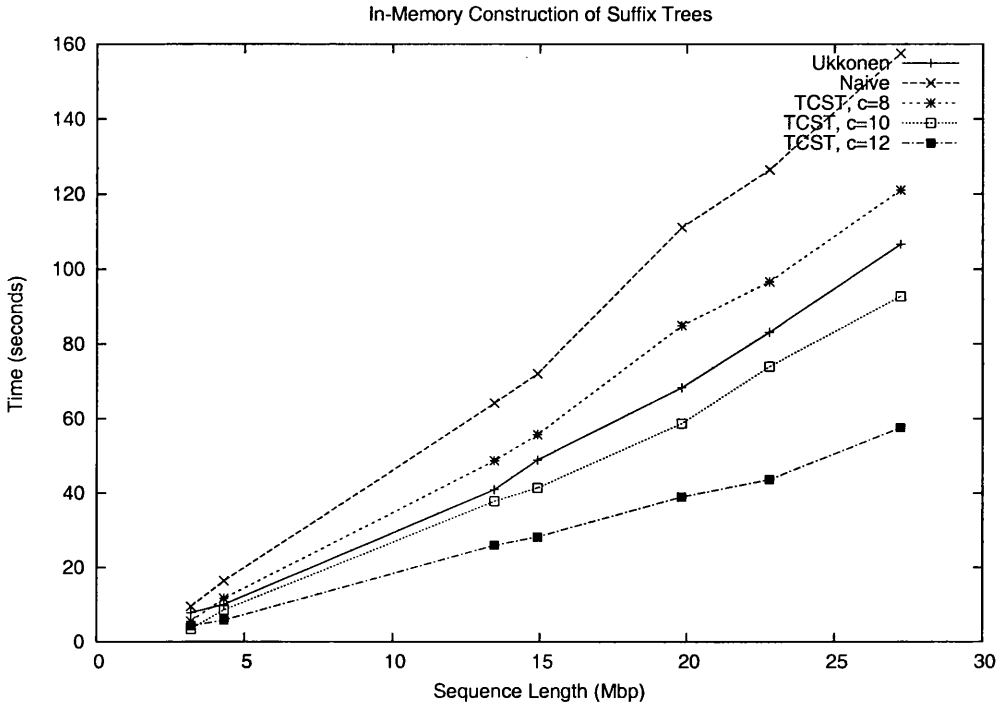

Figure 4.6: In-memory performance of suffix tree construction.

(which consists of accessing a single location in two arrays). In contrast, the same operation in a suffix tree requires up to $a \times c$ nodes to be accessed (where a is the length of the alphabet), together with up to $a \times c$ character comparisons between the suffix being inserted and the complete sequence.[13] It can be deduced that a typical insertion into a TCST will require fewer operations than the equivalent insertion into a suffix tree. Additionally, when using object oriented implementations (as was done here), fewer objects will be created when using the TCST than with the suffix tree—the numerous nodes (each being an object) of the top few layers of the suffix tree are replaced with the two-level array (which should consist of fewer objects).

A further contributing factor to the improved performance is that the pattern of memory accesses is also likely to be better with the TCST. With the TCST the first c characters of the suffix being inserted will be accessed contiguously during polynomial evaluation, and this will then be followed by the access of the one array location in

[13]The value $a \times c$ derives from that fact that at each node in the path being followed in the suffix tree there could be one child node for each element of the specified alphabet. Each of these nodes may need to be examined to find a potential match. Some suffix tree implementations may support direct access to each possible child, reducing the number of possible node accesses (and character comparisons) to c, however this is at the expense of additional storage costs at each node.

each of the backbone and appropriate rib of the two-level array. In contrast, with the suffix tree the pattern of memory accesses is less predictable. Each character comparison between the suffix being inserted and the complete sequence requires two separate locations in the sequence to be accessed, with the patterns of access likely to vary considerably as tree construction continues. Additionally, this will be intertwined with the access to the nodes, giving potentially inefficient memory access patterns—a problem which is lessened when using the TCST.

4.4.2 Parameter Sensitivity for Short Sequences

Creating a disk-resident TCST is done in two phases: the first is the construction of the relevant part of the in-memory index and the second is the writing of the completed index section to disk (we refer to this phase as *checkpointing*).[14] The operational parameters will influence the performance of both phases, thus it is necessary to understand how each phase is affected in order to select suitable values. As was the case with index size, it is expected that construction time is most influenced by choice of operational parameters when indexing short sequences, thus construction time is explored using the same set of parameters used in Section 4.3.2.

Figures 4.7 and 4.8 show how the in-memory phase of TCST construction is affected by compressed depth and rib size. The performance of the in-memory phase confirms the assertions given in the previous section, with performance approaching that of the naïve algorithm as compressed depth is decreased. Additionally, it can be seen that choice of rib-size has negligible affect upon the performance of the in-memory phase of construction. Given that no further reduction in on-disk index size was observed for compressed depths lower than 8, and that construction performance is adversely affected, it can be concluded that the use of compressed depths smaller than 8 is not beneficial.

Figures 4.9 and 4.10 show how overall construction time varies with compressed depth and rib size. For the shortest sequence (Figure 4.9), the checkpointing phase begins to dominate for compressed depths larger than 9. This is indicated by an increase in total construction time, which is contrary to the trend observed for the in-memory phase (see Figure 4.7). This increase in construction time corresponds directly to the increase in on-disk size shown in Figure 4.3. For the larger sequence (Figure 4.10), this effect is lessened but still present (as indicated by no further improvement in performance for compressed depths greater than 10). Rib size only affects the overall

[14] When using prefix-partitioned construction, the construction of each independent index section will involve both phases (in-memory construction and checkpointing).

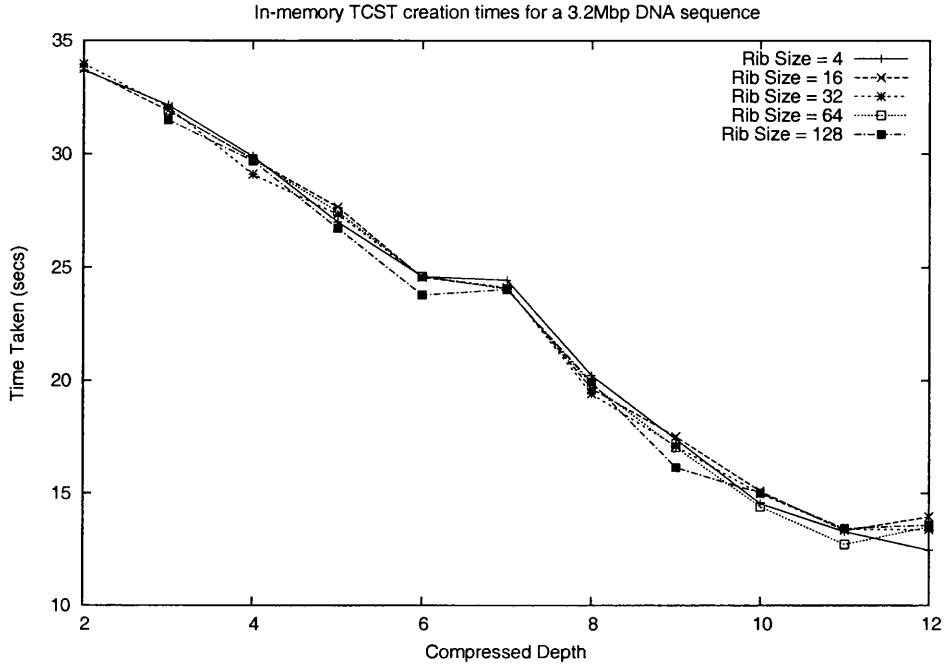

Figure 4.7: In-memory construction times for Top-Compressed Suffix Trees over a 3.2 Mbp sequence.

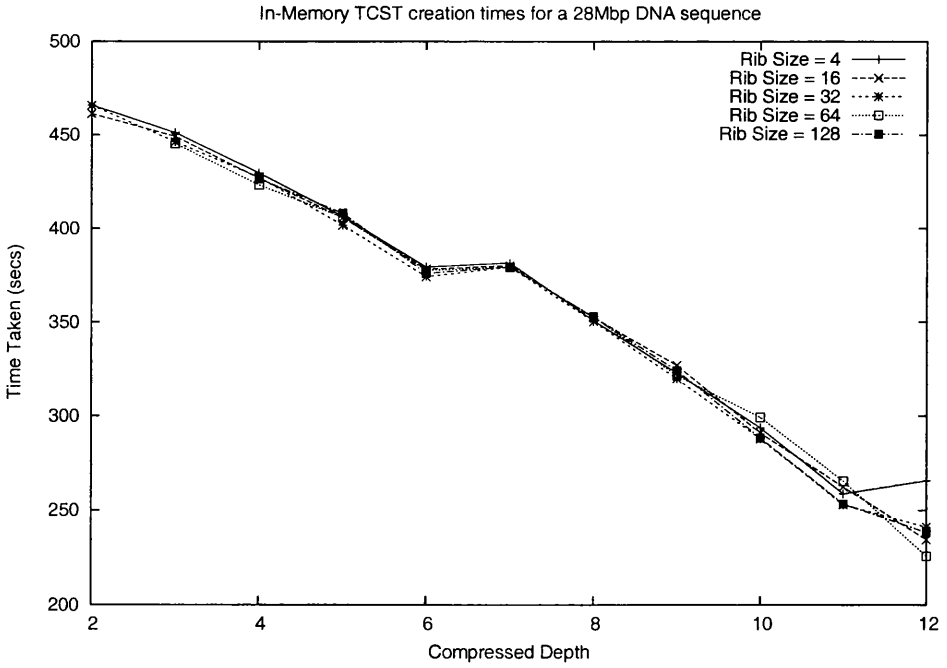

Figure 4.8: In-memory construction times for Top-Compressed Suffix Trees over a 28 Mbp sequence.

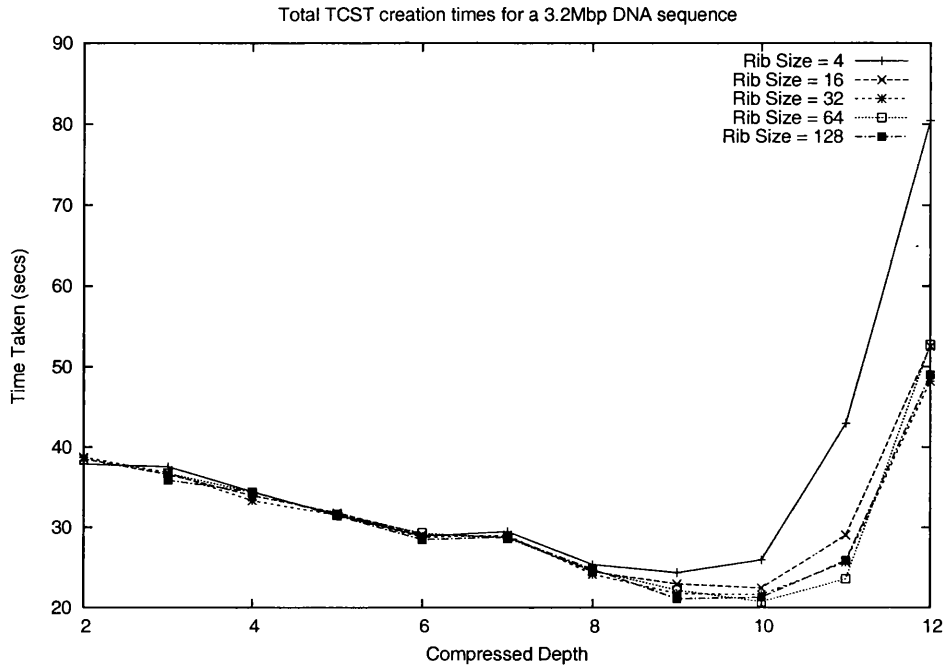

Figure 4.9: Overall construction times for TCSTs over a 3.2 Mbp sequence.

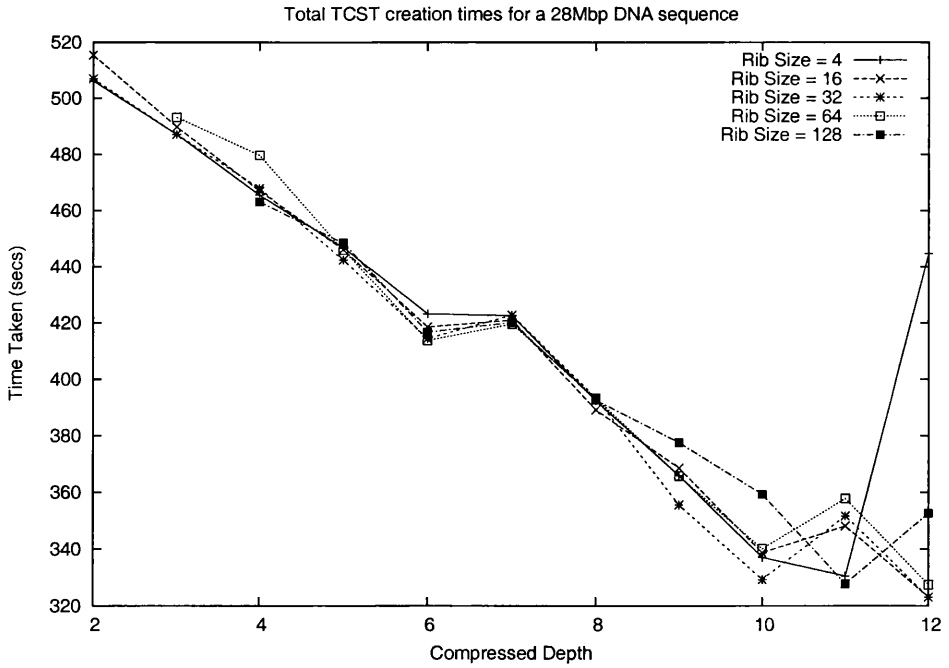

Figure 4.10: Overall construction times for TCSTs over a 28 Mbp sequence.

construction time indirectly—if increasing rib size results in a larger on-disk index size, it will increase the time taken to checkpoint the index. Repeating the experiments described in this section produced differences of less than 1%.

4.4.3 Prefix-Partitioned TCST Construction

So far, we have explored the performance of index construction when only one partition is required. We now go on to explore the performance of partitioned TCST construction. Figure 4.11 shows how the overall time taken to create a disk-resident TCST varies as sequence length increases. The indexes were created using compressed depths of 8, 9, 10 and 12 together with a fixed rib size of 32 (the effect of altering the rib size was explored in the previous section). In each case, the number of partitions used was the minimum number that would allow successful indexing (see Section 4.5 for details). This ranged from a single partition when indexing the shortest sequences, through to 260 partitions for the longest sequence indexed. It can be seen that the prefix-partitioned approach to the construction of suffix trees can scale to accommodate large sequences. In this example, it was possible to index a 1.5 Gbp sequence using a heap size of just under 2 GB (note that the compressed in-memory representation of the sequence occupies 600 MB). This demonstrates that prefix-partitioned construction is still feasible even when a substantial amount of the main-memory is used to accommodate the sequence being indexed. The difference in times measured for repeated runs of this experiment was, on average, 1%.

As with the in-memory suffix tree examples given in Section 4.4.1, it can be seen that increasing the compressed depth decreases the overall time taken to create a prefix-partitioned TCST (see Figure 4.11). Over the complete range of sequences indexed, increasing the compressed depth from 8 to 12 resulted in an average decrease of 30% in the time taken to create the index. It can be concluded that prefix-partitioned TCST construction will take less time than prefix-partitioned suffix tree construction based upon the naïve construction algorithm (as described by Hunt [57]), as this is equivalent to prefix-partitioned TCST construction with a compressed depth of 1.[15]

Construction Phases Figure 4.12 shows the total amount of time spent in each phase (in-memory construction and checkpointing) for the indexes discussed above. It can clearly be seen that index construction is dominated by the creation of the in-memory sections of the index, with checkpointing typically accounting for less than

[15]Note that if a compressed depth of 1 is used then the maximum number of partitions would be equal to the alphabet length, thus limiting the size of sequence that could be indexed.

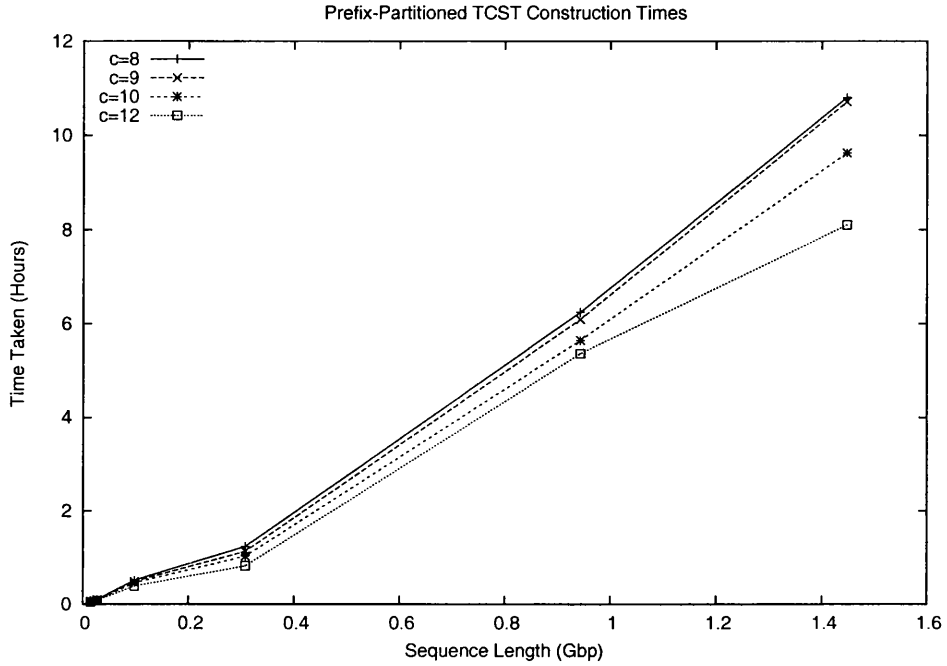

Figure 4.11: Top-Compressed Suffix Tree construction performance using the Prefix-Partitioned algorithm.

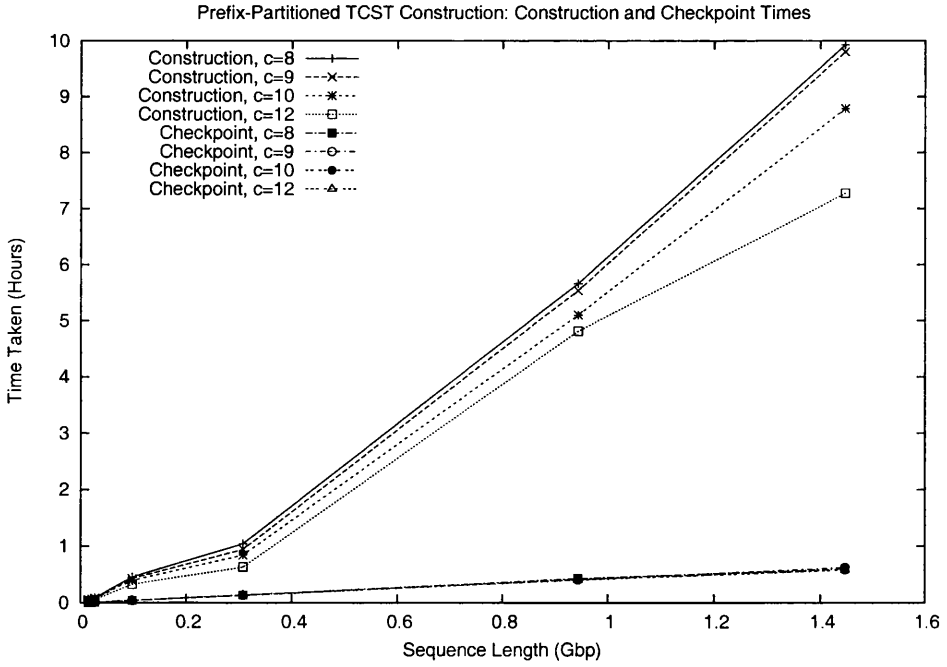

Figure 4.12: Performance of Prefix-Partitioned construction phases.

10% of the total time taken to create the index. Additionally, it can be seen that the total amount of time spent on checkpointing is directly proportional to the sequence length (as would be expected given that the amount of data being written to disk is directly proportional to the sequence length and random disk access is not required).

4.4.4 Improved Prefix-Partitioned TCST Construction

Figure 4.13 shows how the performance of the Improved Prefix-Partitioned TCST Construction Algorithm compares with the original prefix-partitioned algorithm (both using a compressed depth equal to 8). For sequences of up to length 316 Mbp (which could be indexed using no more than 11 partitions), the Improved Prefix-Partitioned TCST Construction Algorithm was typically about 15% slower than the original algorithm. This is largely due to the additional cost of the pre-processing step. However, for the two longest sequences indexed, overall construction time was reduced by 6% and 25% respectively. This difference in performance will continue to grow as larger sequences, requiring more partitions, are indexed. This shows that the time taken to perform the pre-processing step can be recouped over the remainder of the index construction and that use of the Improved Prefix-Partitioned TCST Construction Algorithm is advantageous whenever it is necessary to use a large number of partitions. Similar trends were observed when using other values for the compressed depth. Here, the difference in times observed for different runs of each experiment was approximately 2.5%. This increased variance in the times observed (when compared to previous experiments) is likely to be due to the increased amount of on-disk data being accessed.

The improvement in construction performance observed when using the Improved Prefix-Partitioned TCST Construction Algorithm confirms the analysis of the Prefix-Partitioned TCST Construction Algorithm given in Section 2.4.4. As both algorithms insert the suffixes into the tree in an identical order (and make use of the same insertion code), the relative performance of the two algorithms confirms that repeatedly scanning the sequence is an inefficient technique for identifying the required suffixes for a given partition.

Construction Phases Figure 4.14 shows the total amount of time spent in each phase of the Improved Prefix-Partitioned TCST Construction Algorithm (suffix grouping, in-memory construction and checkpointing). As was found with the Prefix-Partitioned TCST Construction Algorithm, construction time is dominated by the creation of the in-memory index sections. Suffix grouping and checkpointing each typically accounted for 8% of the overall time to create the index. The time taken by the grouping algo-

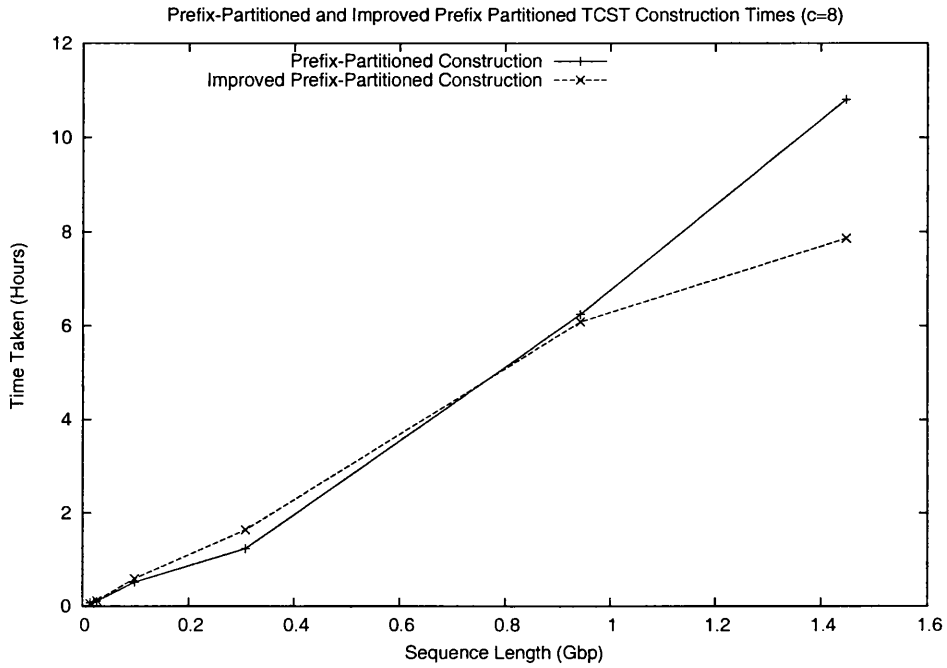

Figure 4.13: Top-Compressed Suffix Tree construction times using both the Prefix-Partitioned and the Improved Prefix-Partitioned construction algorithms.

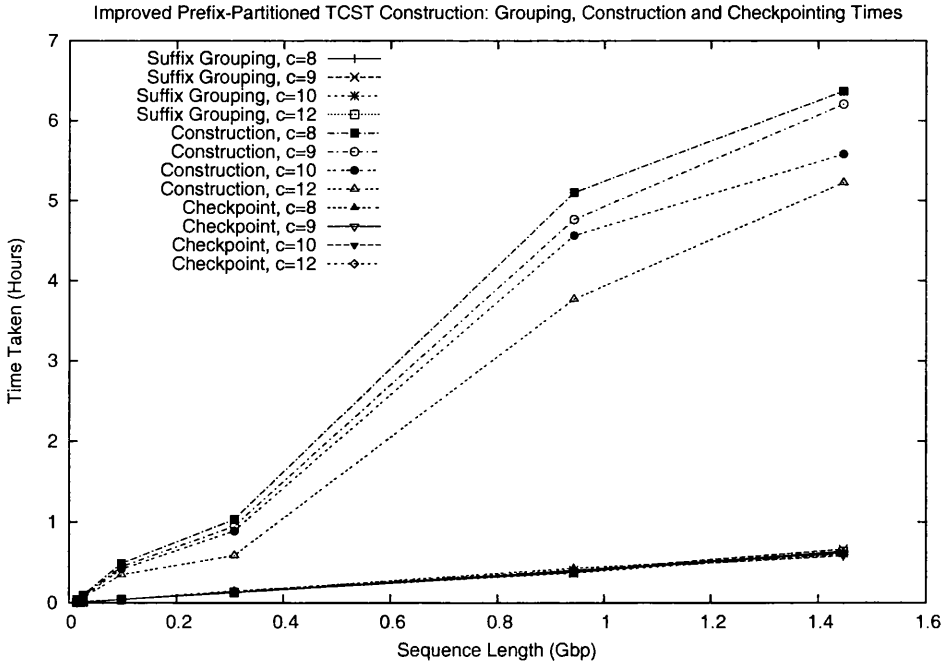

Figure 4.14: Improved Prefix-Partitioned TCST construction phases.

rithm was found to be directly proportional to the sequence length (as predicted in Section 3.5.2), with the performance of the checkpointing phase being identical to that of the prefix-partitioned algorithm (as would be expected since they are performing identical tasks).

4.4.5 Parallel TCST Construction

The final aspect of construction performance to be explored is that of the two parallel TCST construction techniques discussed in Section 3.5.3. These techniques require that access to more than one processor (either on a single computer, or over several computers) is available.

Multi-Threaded Construction

Table 4.3 shows the performance gain achieved using prefix-partitioned double-threaded construction on a two processor computer. For the shortest sequence (28 Mbp), overall construction time was reduced by 40% when using double-threaded construction. This result was achieved using twelve partitions (compared to one partition for single-threaded construction). For the larger sequence (316 Mbp), the reduction in overall construction time was 31% and was achieved using 25 partitions (compared to 11 for single threaded construction). In both cases, marginally slower times were observed using other numbers of partitions. As both construction threads share one heap space, it is necessary to use more partitions than would be the case for single threaded construction. This effectively restricts the use of multi-threaded construction to relatively short sequences: increasing the already large number of partitions required to index the longest sequences would become impractical (see Section 4.5). This limitation does not apply to distributed construction, where several independent heap spaces are used, and is therefore a more attractive choice for indexing large sequences. Allowing for the overheads associated with initialising the data structure, contention for writing to disk, and the fact that both threads will not finish simultaneously, the observed reduction in construction time is in accordance with what would be expected for double-threaded construction. The use of additional processors will yield additional improvements in performance (assuming that the number of jobs is not less than the number of processors).

Sequence Length (Mbp)	Single-Threaded Build (Seconds)	Double-Threaded Build (Seconds)	Percentage Reduction
28	383	231	40%
100	1854	1160	38%
316	4453	3078	31%

Table 4.3: Multi-threaded TCST construction performance.

Distributed Construction

Using two similarly equipped computers, TCSTs were constructed using the distributed variants of both the prefix-partitioned and improved prefix-partitioned construction algorithms. From Table 4.4, it can be seen that the distributed version of the Prefix-Partitioned TCST Construction Algorithm reduces construction time by an average of 45% (when compared to the single-threaded non-distributed version of the same algorithm) and that the reduction in construction time does not significantly deteriorate as sequence length increases (as was the case with multi-threaded construction).

Sequence Length (Mbp)	Single Computer Build (Seconds)	Distributed Build (Seconds)	Percentage Reduction
315538525	3272	1691	48%
965390695	19316	10348	46%
1482254280	39696	22944	42%

Table 4.4: Distributed prefix-partitioned TCST construction performance using two computers.

Table 4.5 shows how construction time is reduced when using the distributed variant of the Improved Prefix-Partitioned TCST Construction Algorithm. Here, the average reduction in construction time was 42%. As expected, this is slightly less than that of the prefix-partitioned algorithm. This is because the implementation of the pre-processing step is single threaded and does not take advantage of the second computer. Again, note that the improvement in performance does not deteriorate as sequence length increases.

For each algorithm, adding additional distributed build clients will further reduce construction time, assuming that the number of partitions is greater than the number of build clients, with the total time taken being inversely proportional to the number of build clients used. Distributed construction can be combined with multi-threaded construction (i.e. each distributed build client has more than one construction thread)

Sequence Length (Mbp)	Single Computer Build (Seconds)	Distributed Build (Seconds)	Percentage Reduction
315538525	3619	2206	39%
965390695	18744	10604	43%
1482254280	28692	16450	43%

Table 4.5: Distributed improved prefix-partitioned TCST construction performance using two computers.

and may be useful if suitable resources are available, although it is likely that this would require an increased number of partitions to be used.

4.5 Minimum Number of Partitions

For a fixed amount of main memory, we define the minimum number of partitions to be the least number of equal-sized prefix partitions that can be used in order to successfully index a given sequence; that is, the least number of such partitions where the largest sub-section of the index contained in a single partition can be created entirely in main memory. In Section 2.4.4, it was argued that when using a fixed amount of main memory, the minimum number of partitions required must grow super-linearly as the size of the sequence increases (c.f. Hunt et al. [58]). Figure 4.15 shows how the minimum number of partitions grows with sequence length when using a fixed heap size of 2 GB and a compressed depth of 8. In each case, the minimum number of partitions was established by approximating the number of partitions required, and subsequently refining this value through a process of trial and error. It can be seen that the growth in minimum number of partitions is not linear with respect to the sequence length. For example, the 316 Mbp sequence required 11 partitions in order to be indexed, whereas the 965 Mbp sequence required 64 partitions and the 1.5 Gbp sequence required 260 partitions. Comparing the 316 Mbp sequence with each of the two larger sequences, the following ratios are observed. For sequence length, the ratios are 1:3.05 and 1:4.75 whereas the ratios between the number of partitions required are 1:5.8 and 1:23.6. This clearly illustrates that the minimum number of partitions grows super-linearly.

Although the minimum number of partitions required to index a given sequence is primarily determined by the sequence length, the size of the available main memory and the distribution of suffixes over the partitions (see Section 4.5.1), other factors can influence the exact number of partitions required. Any parameter which affects memory usage can alter the minimum number of partitions required. For example,

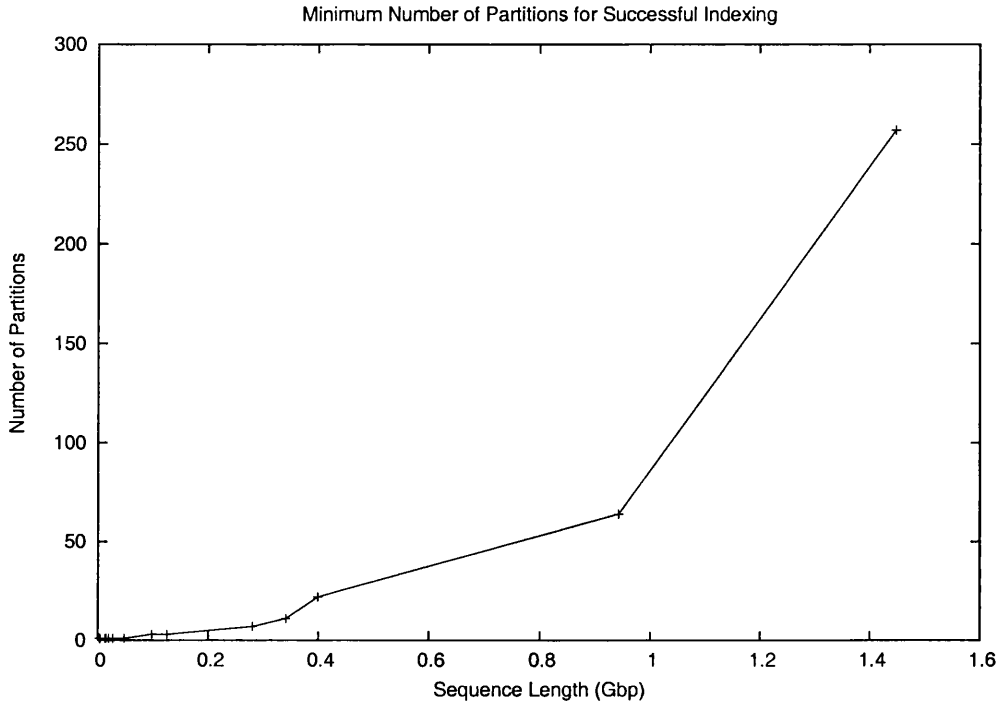

Figure 4.15: Minimum number of partitions required for successful index creation using a 2 GB Heap and a compressed depth of 8.

increasing a buffer size will reduce the amount of the heap space that is available for index construction and therefore may increase the number of partitions required (and vice versa). The size of the compressed depth, c, also affects the minimum number of partitions. For larger sequences, the two-level array can occupy notably less space than the equivalent suffix tree nodes. Therefore increasing c can decrease the amount of space the in-memory index occupies and thus reduce the number of partitions required. For the longest sequence indexed, increasing c from 8 to 12 resulted in the minimum number of partitions dropping from 260 to 244. Finally, as discussed in Section 3.5.3, all partition boundaries must correspond to the start and end of the relevant ribs. Therefore, altering the rib size can alter the distribution of suffixes of partitions and thus subtly alter the spread of suffixes over partitions.

4.5.1 Distribution of Suffixes over Partitions

The distribution of suffixes over prefix partitions is one of the main factors determining how many partitions are required to successfully index a given sequence. In particular, the first reported use of prefix-partitioned suffix tree construction assumes that the *'tree*

is uniformly populated' [58], i.e. that each partition should contain approximately the same number of indexed suffixes. However, it can be demonstrated that this assumption does not necessarily hold when suffix trees are used with DNA sequence data and that this assumption is not necessary for successful index creation using prefix-partitioning.

Figures 4.16, 4.17 and 4.18 illustrate the number of suffixes lying within each partition for indexed sequences of lengths 316 Mbp, 965 Mbp and 1.5 Gbp respectively. In each case, it can be seen that there is considerable variation in the number of suffixes that lie within each partition. Table 4.6 summarises this data for four different DNA sequences. As sequence length and the corresponding minimum number of partitions increases, the average number of suffixes in each partition decreases, whereas the number of suffixes within the largest partition remains fairly stable. For each sequence, the partition containing the largest number of suffixes was found to be the first partition. The range of prefixes that lie within the first partition will always start with the letter **A** repeated c times, with the end of the partition varying according to choice of c and number of partitions. This suggests that suffixes beginning with the letter **A** repeated are the most commonly occurring in DNA sequences. Thus, for each of the sequences used in this experiment, the task of determining the minimum number of partitions required was reduced to finding the minimum number of partitions such that the number of suffixes within the first partition was small enough for successful indexing. This observation was found to hold true for each of the sequences used throughout this work. However it may not be possible to assume that this will apply to all genomic data.

Sequence Length	Partitions Used	Number of Suffixes per Partition		
		Smallest	Biggest	Average
100 Mbp	3	22837291	45068981	33425957
316 Mbp	11	41896	53566042	28685319
965 Mbp	64	21656	45265855	15084229
1482 Mbp	260	0	38176065	5700977

Table 4.6: Summary statistics for the distribution of suffixes over prefix partitions.

The inclusion of the character **N** (which represents one or more unknown characters) in the alphabet used when indexing DNA sequences has a notable effect on the distribution of the suffixes. Note that substrings consisting solely of the character **N** repeated one hundred or more times have been removed from the sample data (see Section 4.2). For sequences that are either completed, or nearing completion, this character will occur infrequently. However, this character will occupy the same amount of space within the prefix ranges as each other character in the alphabet. This results

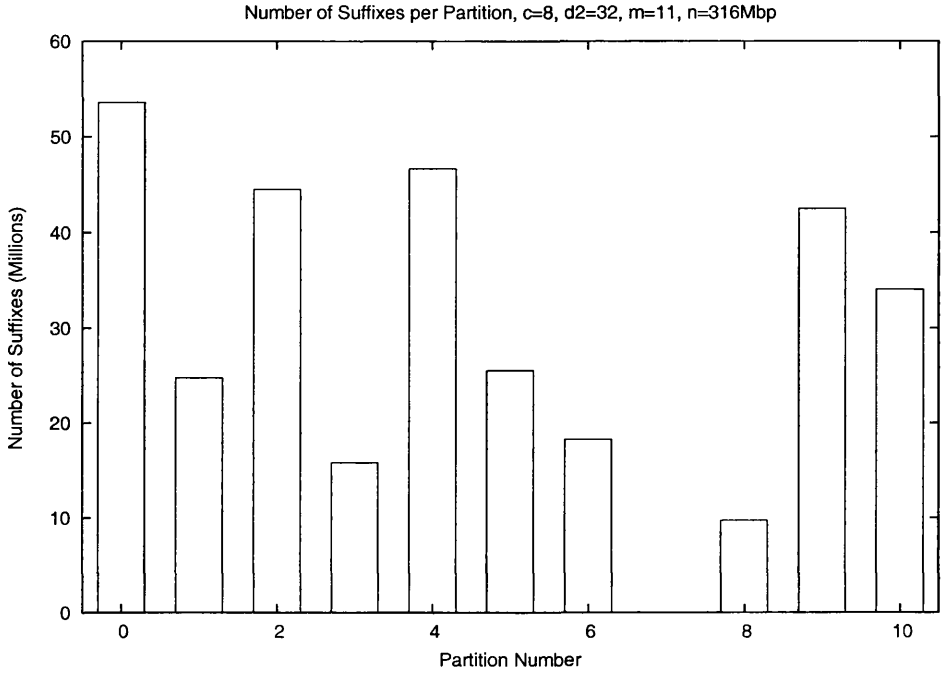

Figure 4.16: Distribution of suffixes over 11 partitions for a 316 Mbp sequence.

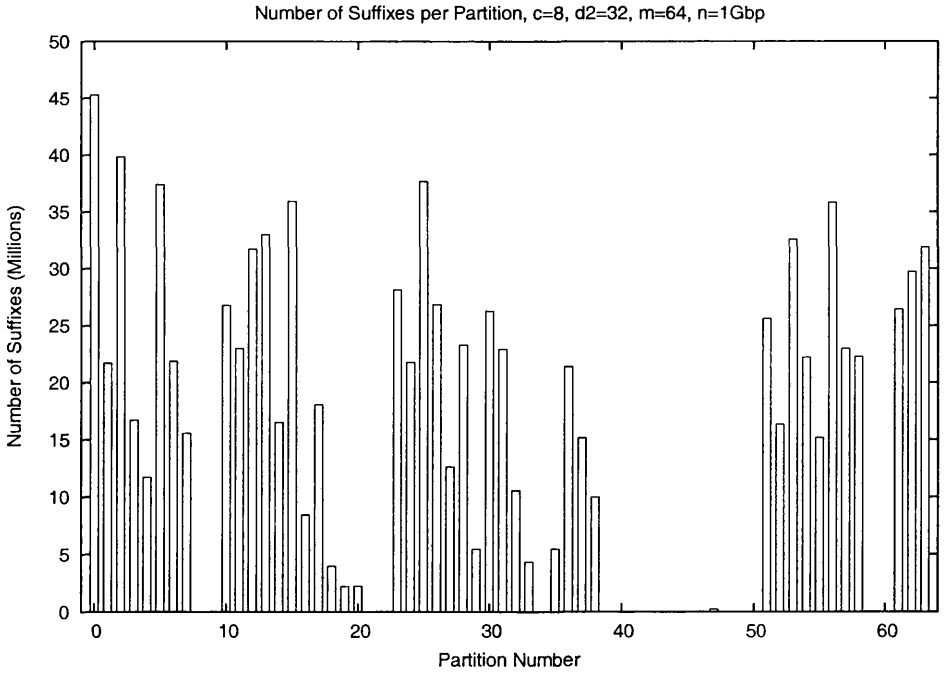

Figure 4.17: Distribution of suffixes over 64 partitions for a 1 Gbp sequence.

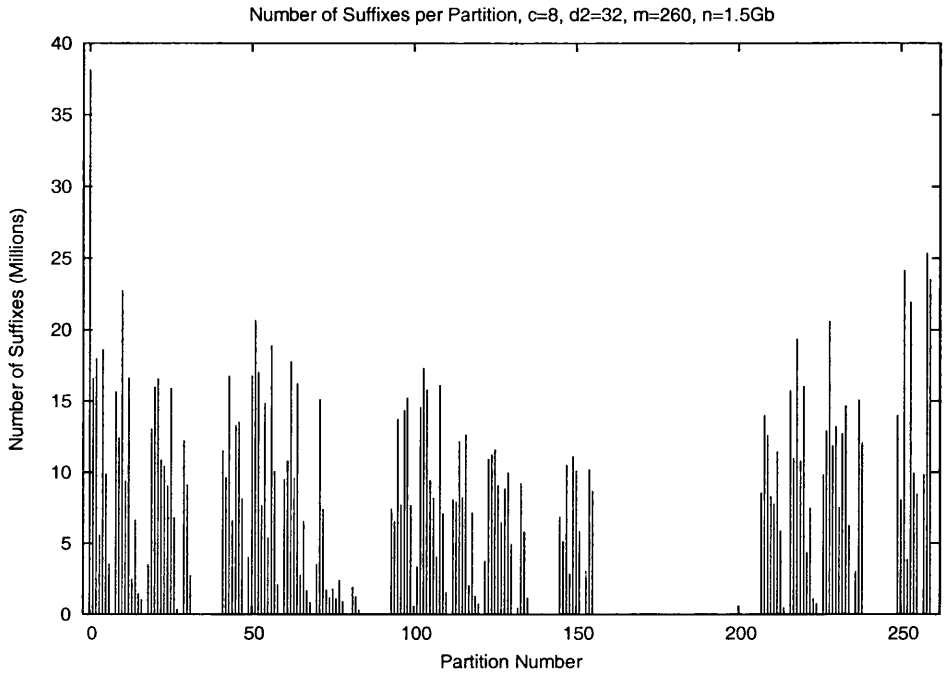

Figure 4.18: Distribution of suffixes over 260 partitions for a 1.5 Gbp sequence.

in a number of partitions that are either sparsely populated, or completely empty. This effect can be seen in Figures 4.16, 4.17 and 4.18, where the range of partitions corresponding to those prefixed by the letter **N** are almost completely empty (these partitions can be found in the fourth fifth of the range of partitions shown). Obviously, if a given sequence did not include this character then it is possible to omit **N** from the alphabet, thus giving a more even distribution of suffixes over the partitions. This would also reduce the range of possible prefix values to be supported by the two-level array, allowing for a more compact representation.

4.5.2 Limitations of Prefix-Partitioned Construction

So far we have demonstrated that the prefix-partitioned approach to the construction of suffix trees is a viable technique for indexing large sequences. Use of this technique assumes that the data being indexed can be successfully partitioned, i.e. that it is possible to split the index into a number of independent sections that are small enough to be created entirely in main memory. It was shown in the previous section that even though the suffixes of a typical DNA sequence are not uniformly distributed over the range of prefix partitions, the nature of the distribution is such that prefix-partitioned

construction is still possible. However, if a suitable partitioning cannot be achieved then construction will not be possible using this technique. This situation can only occur where there are one or more long substrings consisting solely of a very short repeated pattern.

Consider a text containing a long substring comprised entirely of the character **N** repeated x times (for some value of x). Of the suffixes corresponding to this substring, the first $x - c$ will all share a common c character prefix and therefore must lie within the same prefix partition (regardless of the number and size of the partitions used). Therefore, if all $x-c$ suffixes cannot be indexed in main memory then prefix-partitioned construction will fail. A similar problem could arise if a substring contained a short pattern (such as **AC**) repeated x times. In this case, the suffixes drawn from this substring will lie alternately in two partitions (one corresponding to **ACAC...** and the other corresponding to **CACA...**). Again, if the substring is suitably long then prefix partitioning could fail. Although it is unlikely that the latter case would occur, the former can be present in DNA sequences. However, such use of the character **N** denotes a section of the sequence where the sequencing process has not been completed, so providing an index over such sub-strings is likely to be of limited use. In practice, this limitation of prefix partitioning is unlikely to be an obstacle to the use of this construction technique.

4.6 Matching Performance

The final aspect of Top-Compressed Suffix Tree performance to be explored is that of matching target strings against the sequence using the index. Support for efficient queries is a vital property of any index. If the TCST is to be used in preference to the suffix tree, then the time taken to complete a typical query using the TCST must either be equal to or less than the time taken using a suffix tree.

The basic matching operation over the suffix tree consists of matching characters from the query string one at a time against the labels associated with the nodes of the tree. This operation forms the basis of many algorithms that operate over the suffix tree. We explore the performance of two versions of this basic matching technique. The first is the exact matching problem (where all occurrences of a given query string are located), the second simply tests for the presence of the query string (i.e. it simply returns true or false depending on whether the query string is present or not). Although inherently similar problems, testing for the presence of a string will not require that the complete sub-tree be traversed below the point of the match and is likely to have

different performance characteristics.

This section discusses several aspects of matching performance. As was the case with construction performance, for sequences that can be indexed entirely in main-memory we compare the time taken to complete a set of queries using the TCST against that of suffix trees constructed using the naïve algorithm and Ukkonen's algorithm.[16] We also explore how the choice of compressed depth affects query performance, including a detailed exploration of the performance of searching for patterns that are shorter than the compressed depth. Finally, we explore query performance over persistent TCSTs.

4.6.1 Generating Target Strings

In order to generate sets of target strings, substrings were drawn at random from two sequences that were not being used as part of the index evaluation. For each set, a minimum and maximum length (inclusive) was specified, with the length of each substring being a random value from this range. The sets of target strings were stored as plain text files in the format specified in Appendix A.

4.6.2 In-Memory Matching Performance

The first aspect of query performance to be explored is how the performance of TCSTs of varying compressed depth compares with that of suffix trees constructed with the naïve algorithm and Ukkonen's algorithm (for indexes that can be held entirely in main memory). Figure 4.19 shows how the time taken to complete a set of one million potential exact matches (minimum length four characters, maximum length one hundred characters) varies with sequence length and index type. The time taken includes, for each query, matching the target pattern, locating each occurrence and traversing the result set. Each experiment was repeated four times, with the times shown in Figure 4.19 being the average of the total times taken for each of the four runs. To prevent the first experiment to use a particular set of queries being unfairly penalised (due to the later runs performing better due to caching of the query list) the results of the first experiment were ignored, and the experiment repeated.

[16]Note that by comparing the results obtained from each of the different index implementations it is possible to confirm, within reason, the correctness of the implementations used—if all the indexes give identical results we can be confident that the results are accurate. Additionally, for shorter runs, the search results produced by using the TCST where compared against those obtained using an implementation of the Boyer-Moore search algorithm. This served as a further test of correctness, but, given the computational time required by Boyer-Moore, was restricted to sequences of lengths up to 100 MB.

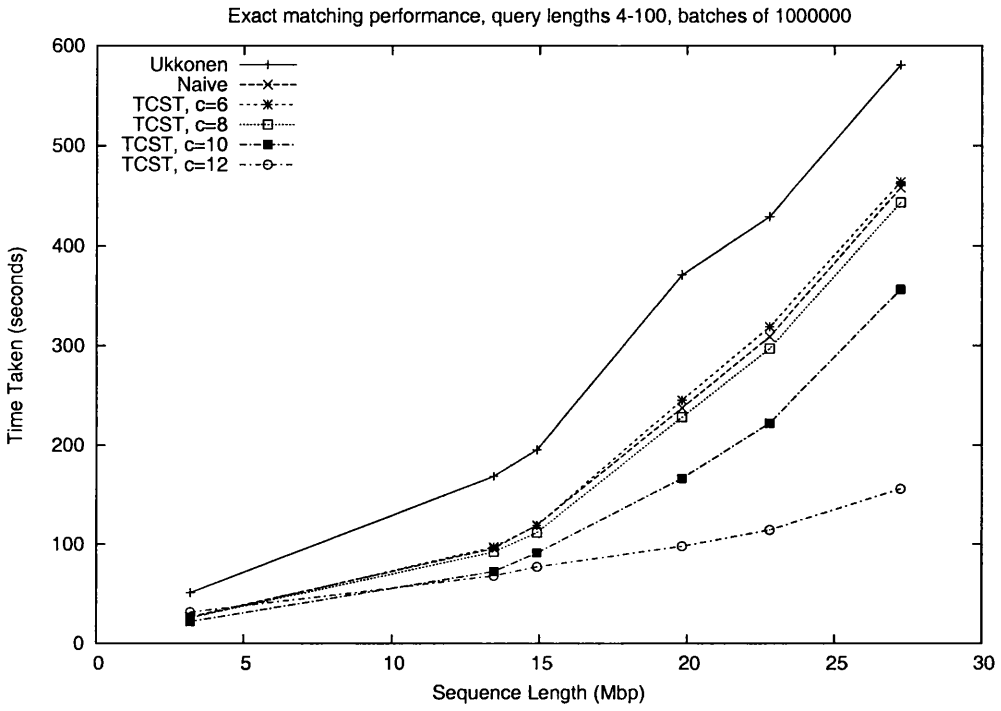

Figure 4.19: In-memory exact matching performance using suffix trees.

The first result of note is that our implementation of a naïvely constructed suffix tree outperforms the equivalent suffix tree constructed using Ukkonen's algorithm. This improvement is almost certainly due to improved cache interaction resulting from the different node representation and clustering achieved with the naïve algorithm. Our naïve tree uses a more compact node representation than that required by Ukkonen's algorithm. In particular, no suffix link is required and the suffix number and right label are calculated on demand. This more compact representation allows more nodes to stored in any region of memory, thus increasing the likelihood of the required node being cached. Additionally, the top-down insertion of suffixes into the tree during construction is more likely to have nodes that are accessed together during search being located close together in main memory (thus reducing the amount of random memory access). Here, the overhead associated with the on-demand calculation of the suffix number and right label (see Section 2.4.1) is entirely recouped by the performance boost obtained due to greater locality of reference.

Over the complete range of sequences indexed, increasing the compressed depth, c, of a TCST improves the performance of the exact matching algorithm. On average, when using a TCST with a compressed depth of 12, the total time taken to complete

a set of exact matches was 63% less than the time taken when using a suffix tree constructed with Ukkonen's algorithm and 39% less than a naïvely constructed suffix tree. If we consider only the three longest sequences that could be indexed entirely in main memory, then the average percentage reductions further improve to 73% and 62% respectively. The only exception to this trend was with the shortest sequence indexed (3 Mbp), where the performance improved when increasing c from 6 through to 10, but further increasing c to 12 actually worsened the performance. This is almost certainly due to the two-level array being excessively sparse, resulting in poorer locality than that obtained with lower values of c (see also Section 4.3.1).

As with index construction, the improvement in performance associated with increasing c can be attributed to the presence of the two-level array. Matching the first c characters of a query using the two-level array will be quicker than traversing the corresponding section of a suffix tree. This is due to fewer character comparisons being required, greater locality of access to both query and sequence strings and fewer nodes being accessed (see Section 4.4.1 for a fuller explanation). For index construction, it was observed that as c decreased, TCST construction performance tended to that of naïve suffix tree construction. If we consider only query patterns that are longer than c, then this observation holds for exact matching performance, however when query patterns shorter than c are included such a correlation cannot be established. This is confirmed by the results shown in Figure 4.19, where the performance of the naïvely constructed trees was found to be somewhere between that of the TCSTs with compressed depths of 6 and 8. The performance of short queries is explored in detail in the Section 4.6.3.

Figure 4.20 shows the results of the experiment described above repeated with the exact matching operation replaced by simply testing for the presence of the query string (and not locating the occurrences of it in the sequence). With this operation it is seen again that increasing c yielded improved performance. The performance gain achieved was consistent over all the lengths of sequence used, with the best performing TCST (compressed depth of 12) being on average 37% and 33% faster than the suffix trees constructed with Ukkonen's algorithm and the naïve algorithm respectively. Note that each experiment described here was repeated in its entirety (meaning that the complete set of four runs was repeated), and the average times observed for each experiment differed by less than 1%.

Rib Size All the TCST results presented above used a constant rib size of 32. Each of these experiments was also undertaken using rib sizes of 16, 64 and 128. For equivalent

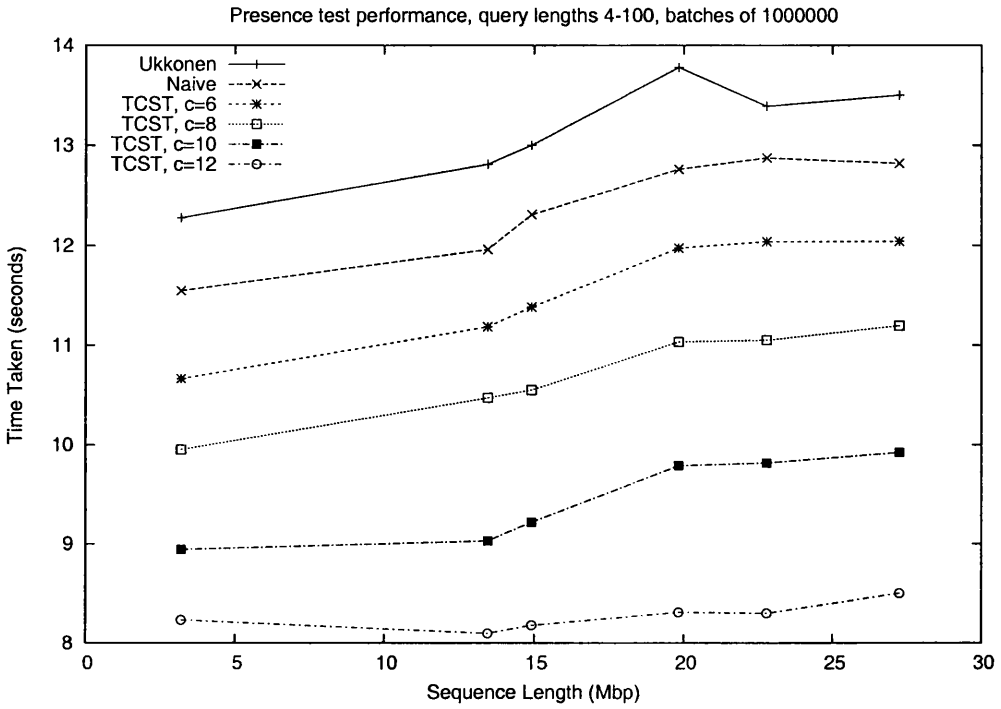

Figure 4.20: In-memory presence test performance using suffix trees.

indexes using different rib sizes, the largest observed difference in the time taken to execute a set of queries was 10%. However it was found that the typical difference was approximately 2%. Although altering the rib size did have a marginal effect on performance, the effect was not predictable—the optimum choice of rib size varied not only between indexes but also between sets of queries. Therefore, a rib size of either 32 or 64 is recommend as these values yielded the best index construction performance.

4.6.3 Short Query Performance

The second aspect of query performance to be explored is that of matching short query patterns using the TCST. This is of interest as the nature of searching a TCST for a query pattern shorter than the compressed depth is quite different from the same operation on a suffix tree. In Section 3.6.2 it was observed that searching for particularly short queries may take longer using TCST than when using a suffix tree.

Several batches of query patterns were generated using the technique described in Section 4.6.1. Each batch consisted of queries of a fixed length, with one batch created for each possible query length between one and twelve. For queries of length 5 or greater, each batch consisted of one million queries. However, it would be impractical

to perform exact matching for one million queries for each query length less than five as this would take an excessive amount of time to process (such a list would also contain many repeated target patterns and is unlikely to be used in practice). Therefore, shorter lists were used for each query length less than five, with all times being presented as averages. This allowed an exploration of which index was most efficient for each potential query length. Again, the performance of both exact matching and testing for the presence of the given string were explored.

Figure 4.21 shows the performance of the exact matching algorithm over a sequence of length of 3 Mbp. Again, it is seen that the suffix tree constructed using Ukkonen's algorithm is the worst performing index in all cases—taking up to three times as long to execute the queries as the best performing TCST. For queries of lengths ten to twelve, the TCST with a compressed depth of twelve was found to be the best performing index and was found to be on average 27% quicker than the naïvely constructed suffix tree. For queries shorter than ten, the TCST with a compressed depth of ten was found to be the best performing index, being just over 10% quicker than the naïvely constructed suffix tree. However, when this experiment is repeated with longer sequences, the TCST with a compressed depth of twelve becomes the best performing index for all query lengths. When searching for matches in a 27 Mbp sequence (see Figure 4.22), the TCST with a compressed depth of twelve was on average 67% faster than the suffix tree constructed with Ukkonen's algorithm and 53% faster than that naïvely constructed suffix tree.

The benefits of the TCST over the suffix tree are different when searching for query patterns shorter than the compressed depth. The advantages of matching against the two-level array (as opposed to using the tree) are diminished by the fact that we must scan several locations in the array. However, there are some benefits to this technique. Firstly, this technique does not require the indexed sequence to be accessed when looking for a match. Also, much of the access to the two-level array will be contiguous, allowing for good cache interaction. Finally, the presence of the two-level array will result in the sub-trees that are to be traversed (i.e. those that lie within the matching prefix range) being much smaller than the corresponding section of suffix tree. This will mean that fewer nodes are traversed, and less stack space used during recursion. Essentially, optimum TCST performance for short queries will be obtained by finding a suitable balance between reducing the size of the sub-trees and avoiding making the two-level array too sparse (as was the case when a compressed depth of twelve was used with the 3 Mbp sequence).

Figures 4.23 and 4.24 show the results of performing a similar experiment to that

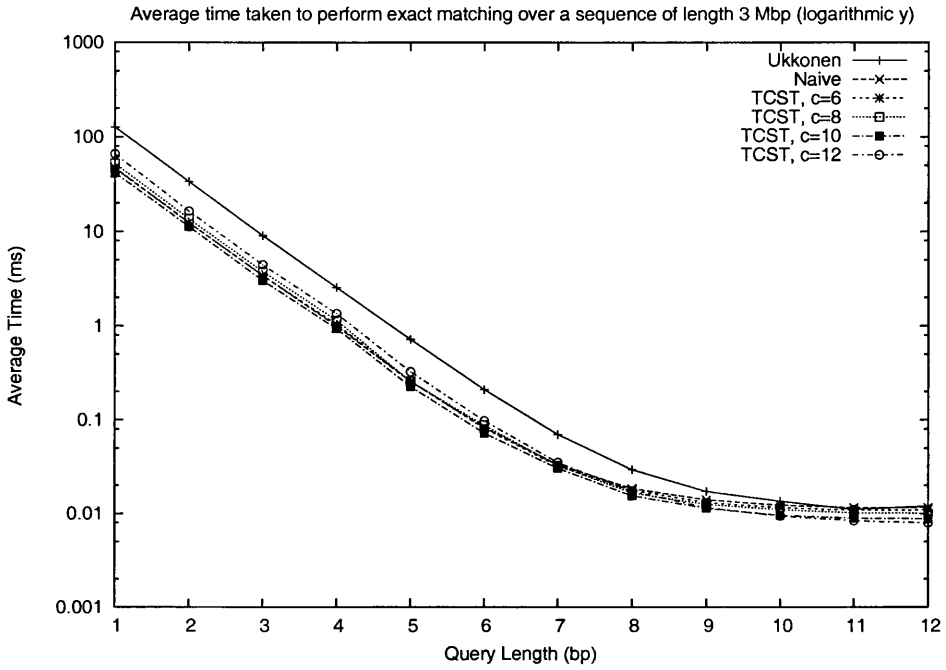

Figure 4.21: Average in-memory exact matching performance for short queries over a 3 Mbp sequence (logarithmic y axis).

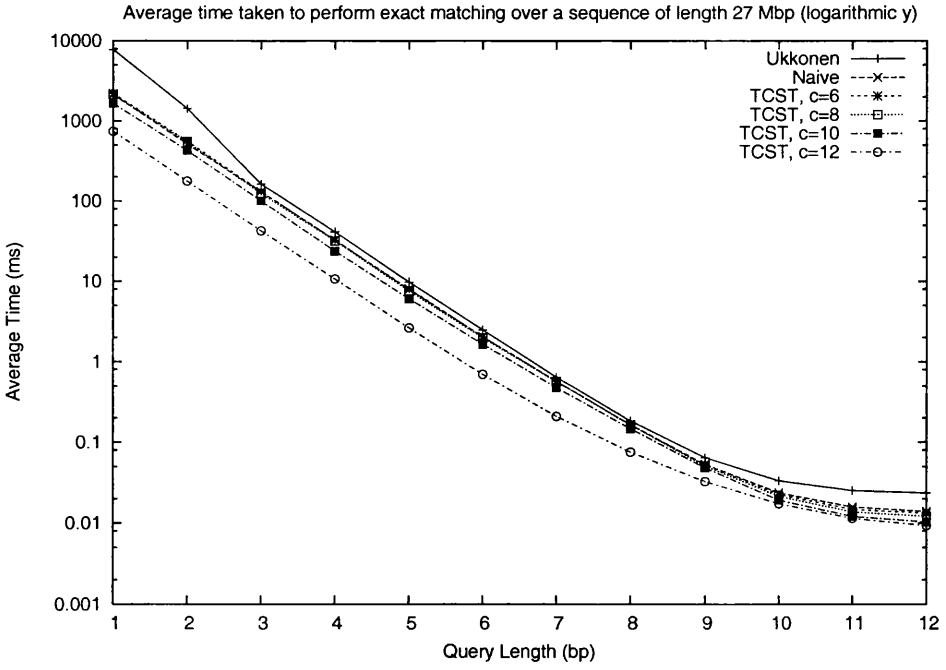

Figure 4.22: Average in-memory exact matching performance for short queries over a 27 Mbp sequence (logarithmic y axis).

described above, where the exact matching operation is replaced by testing for the presence of the given query string. Note that each batch of queries used for this experiment contained exactly one million queries. With the suffix tree, establishing the presence of a query pattern of length l is an $O(l)$ operation. This is also the case for the TCST when l is greater than c. However, once l is less than c we will have to scan a section of the two-level array, therefore such a guarantee cannot be given. From Figures 4.23 and 4.24 it can be seen that once the query length is less than c, the total time taken by this matching operation over a TCST becomes reasonably constant (differing by no more than 5%) and is not directly proportional to the query length. This suggests that the number of two-level array entries that must be scanned in order to locate the first match does not increase significantly as query length decreases. Once l is greater than c, the time taken by a TCST will grow at a similar rate to that of the suffix tree. This can be seen from Figures 4.23 and 4.24, where the lines corresponding to the various TCSTs only begin to rise once the query length exceeds c.

For each query length l, the best performing index was found to be the TCST with the smallest possible value of c that is not less than l. However, the performance of both implementations of the suffix tree were consistently worse than the best performing TCST, suggesting that if c is further reduced there will be no performance gain (even for the shortest of queries). If searching for the presence of such short strings is of particular interest to a client of this index, then it may be beneficial to choose a suitable value for c based upon these observations (although this may be to the detriment of other aspects of performance). As with the previous experiment, each experiment was repeated in its entirety (meaning that the complete set of four runs was repeated), and the average times observed for each experiment differed by less than 1%.

Summary Contrary to the observations made in Section 3.6.2, on average, searching for short query strings takes less time when using a TCST index than a suffix tree. For exact matching, a significant time saving is made by only traversing sub-trees. When simply testing for the presence of a given string, the time taken by all the indexes tested was comparable, with the best performing index found to be the TCST with a suitably chosen compressed depth.

4.6.4 Persistent Matching Performance

The final aspect of query performance to be explored is that of the matching algorithms when operating over a persistent TCST. When operating the index from disk, the amount of time spent on faulting (that is, the amount of time spent transferring sections

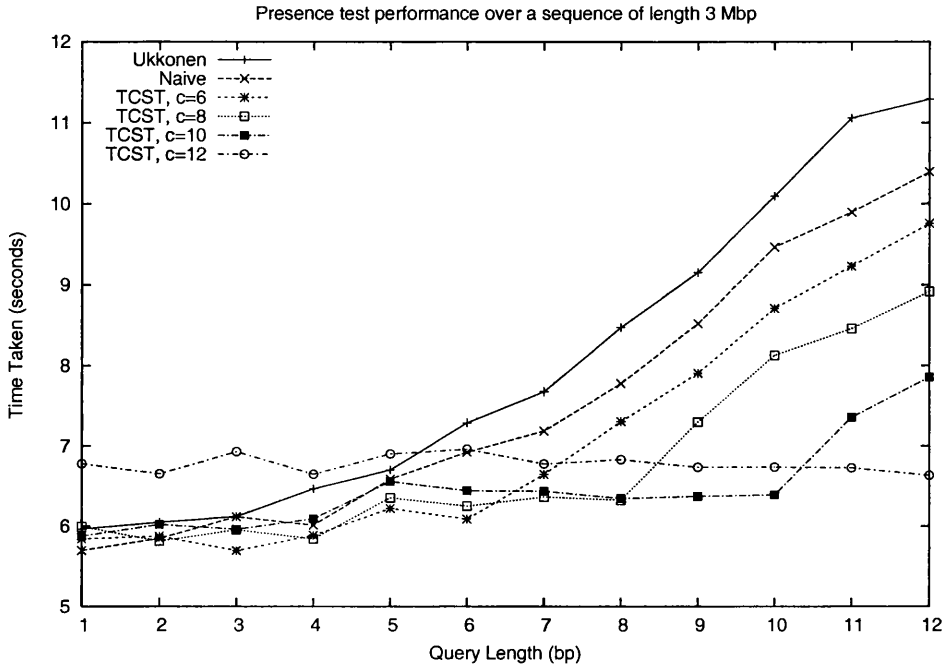

Figure 4.23: In-memory presence test performance for short queries over a 3 Mbp sequence.

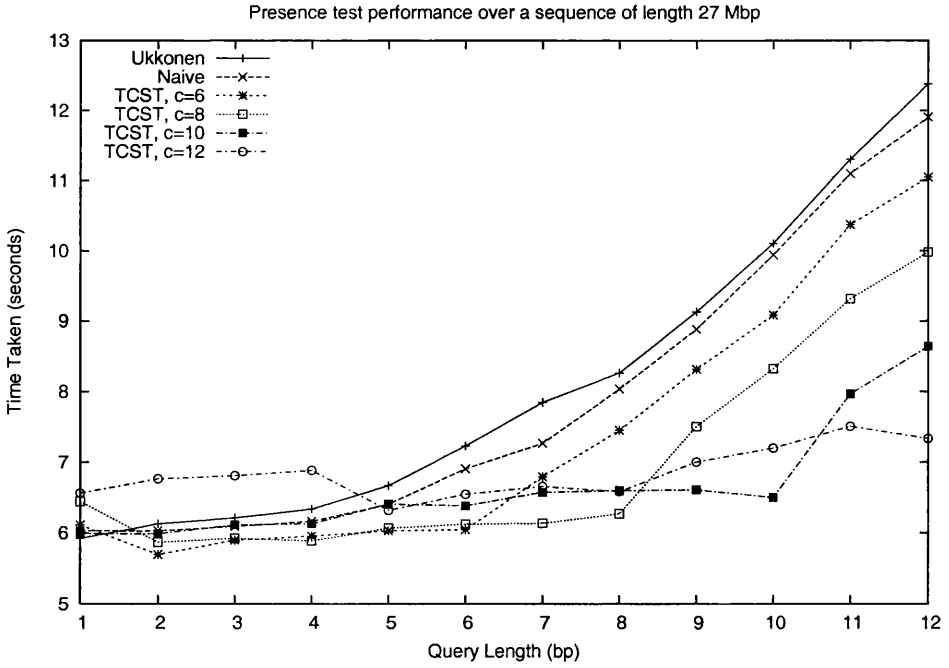

Figure 4.24: In-memory presence test performance for short queries over a 27 Mbp sequence.

of the index from disk to main memory) is expected to have a significant impact on index performance—particularly so when using indexes that are several times larger than the available main memory. Additionally, with a persistent index there are certain tests that must be performed before accessing certain parts of the index, and such tests may also impact upon performance. The impact of these overheads must be balanced against the benefits of using a persistent index, namely that we can index sequences many times larger than is possible with main-memory techniques and that we do not have to re-build the index each time a batch of queries is to be performed.

In order to examine different aspects of the overhead associated with using the index from disk, times will be recorded for *cold*, *warm* and average performance. By *cold* it is meant that the query is being executed for the first time, so the relevant section of index is less likely to be in main memory. By *warm* it is meant that the query is being repeated, increasing the likelihood of the relevant section of index already being present in main memory. *Average* query performance will also be considered when discussing the effects of caching and eviction, in such cases the average query performance will be the mean time taken to perform a query, for the complete set of queries performed (which will include examples of both warm and cold queries). Such times are now explored for a variety of different sequences and queries.

Warm Index Performance

Comparing the warm performance of queries over both main-memory only and small persistent TCSTs allows the impact on performance due to the necessary differences in implementation to be explored. Firstly, there is the impact of the *residence tests*, these are the various tests that must be performed to establish if a required section of the index is resident in main memory or not. Secondly, there will be some differences in the main-memory layout of the two types of index: for the main-memory only TCST the index layout will be determined by the construction algorithm, for the persistent TCST the main-memory layout will be determined by the order in which the relevant index sections are faulted. To ensure that any observed difference in performance is due to the factors described above, the times compared will be warm times—this ensures that all necessary parts of the persistent index will have been faulted into main-memory (as all of the indexes used here could fit entirely in main-memory, no part of the index will have been evicted during these experiments). Both the exact matching algorithm and testing for the presence of the given string are considered here. Furthermore, the queries are split into two categories: those that may be shorter than the compressed depth and those that are always longer than, or equal to, the length of the compressed

depth. This distinction is made as queries that are shorter than the compressed depth may invoke many residence tests, whereas with longer queries this will only happen once.

Figure 4.25 shows the performance of warm exact matching over both main-memory only and persistent TCSTs (over a variety of sequence lengths). All query strings had a length between 12 and 100 (inclusive). As would be expected, the time taken by the persistent TCSTs is, on average, marginally slower than that of the main memory only trees. A similar performance overhead was observed when the exact matching operation is replaced with testing for the presence of the string (not shown). In both cases, the persistent TCSTs were, on average, one percent slower than the main-memory only trees for warm queries.

Repeating the above exact matching experiment using query strings of lengths between 4 and 12 (inclusive) yielded significantly different results (see Figure 4.26). Here, the residence tests can have a significant impact on the performance of the exact matching algorithm. From Figure 4.26, it can be seen that increasing the compressed depth actually worsened the performance of the warm persistent TCST, significantly so in the case of the TCST with a compressed depth of 12. This is due to the large number of two-level array entries that must be scanned in order to satisfy such queries. Scanning a large number of two-level array entries will lead to a large number of residence tests being performed: one such test is required for each rib in the given range, and, for non-null ribs, there will be a similar test performed on each rib entry. Increasing the compressed depth will, for short queries, increase the number of two-level array entries that must be scanned, and the cost of performing the additional residence tests is, in some cases, counteracting the benefits of using a higher value for the compressed depth. However, it was also found that the rate at which the average time to perform a query grows with sequence length decreased as the compressed depth increased, suggesting that as sequence length increases it is likely that a higher value for the compressed depth will be more suitable.

A more surprising result is that for all the indexes with a compressed depth less than 12 it was found that the warm persistent index performed significantly better than the main-memory only equivalent—in some cases the warm persistent index was five times quicker than the corresponding main-memory index. This improved performance is due to the effects of clustering. With the persistent index the in-memory sub-trees will be arranged in the order that they were faulted from disk, i.e. the nodes of the sub-trees will be arranged in depth first order (as dictated by the marshalling algorithm) and collections of ribs and sub-trees are likely to be located closely in memory (as dic-

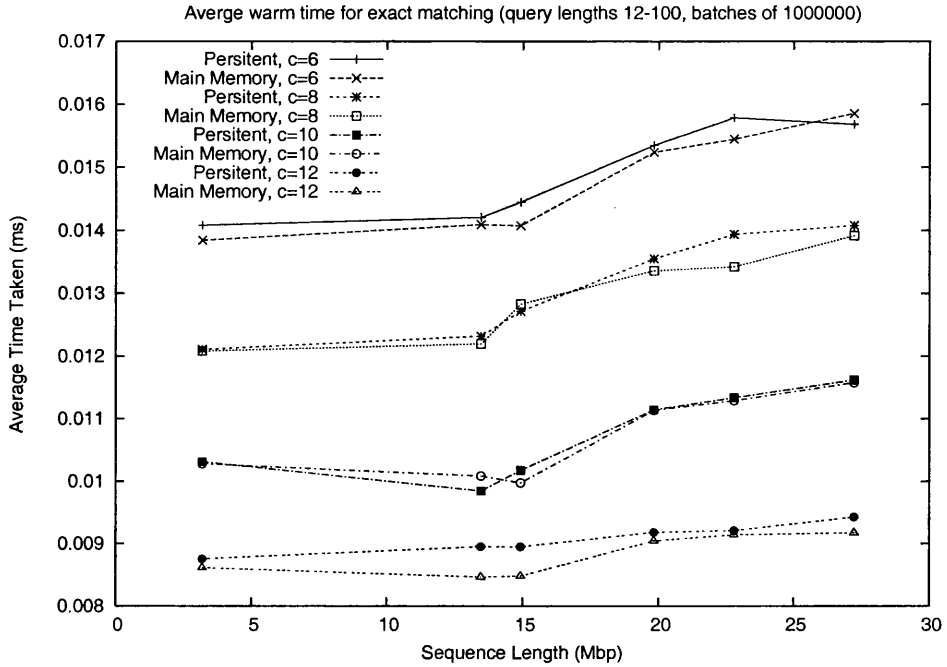

Figure 4.25: Average warm exact matching performance for main-memory only and persistent TCSTs over a variety of lengths of sequence (query lengths 12–100).

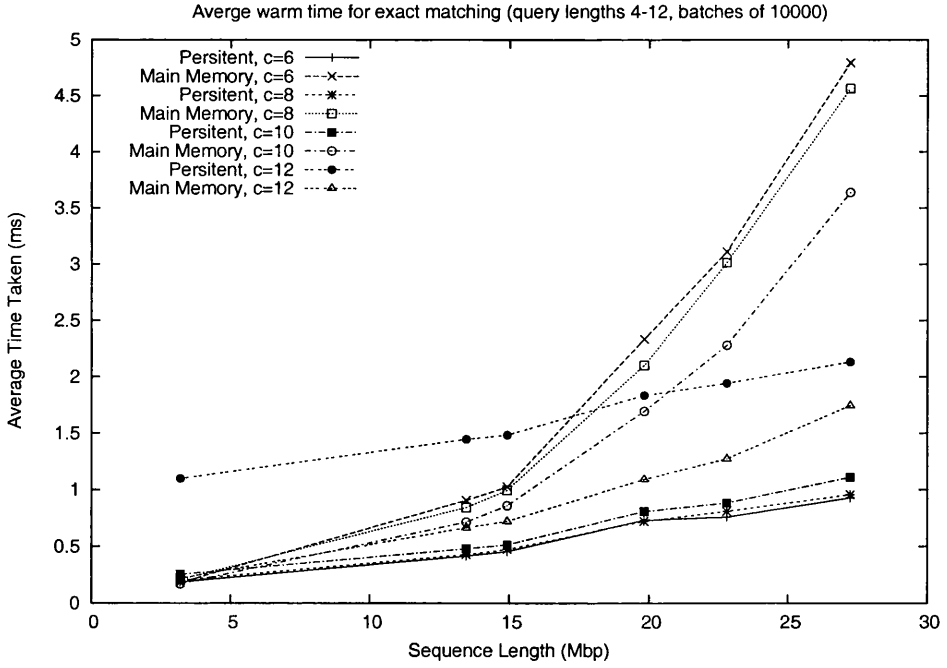

Figure 4.26: Average warm exact matching performance for main-memory only and persistent TCSTs over a variety of lengths of sequence (query lengths 4–12).

tated by the order in which they are faulted as the list of two-level array elements is scanned). In contrast, the ordering of all objects within the main-memory only TCSTs will be dictated solely by the construction algorithm. As was found when comparing main-memory only trees (see Section 4.4.1), clustering has a dramatic impact upon the performance of such indexes. This result justifies the use of measured execution time as a means of comparing index performance—the substantial difference in the time taken by equivalent indexes could not be predicted through traditional analytical techniques. Repeating this experiment with the presence test replacing the exact matching algorithm yielded the same one percent slow down discussed above (such queries will typically only require a few two-level array entries to be scanned, regardless of query length). Note that for all of the warm times presented in this section, the difference in times noted for repeated runs of the same experiment was approximately 2.5%.

Cold Index Performance

The next part of the persistence overhead to be examined is the cost associated with faulting required sections of the index. Accessing data on disk is typically several orders of magnitude slower than accessing data in main memory. Additionally, once the data has been read from the disk, the corresponding in-memory data structure must be created prior to executing the query. Given these necessary additional steps, it is expected that the performance of a cold TCST will be significantly slower than the warm performance of the same index—this is an unavoidable overhead of using an on-disk index. However, the impact of accessing the data on disk is lessened as much of the data is accessed sequentially (individual sub-trees are arranged as single 'blocks' of data) and caching is expected to improve the performance of the index when used for larger batches of queries (this is explored in the following section).

The cost of faulting required sections of the index can be established by comparing the cold and warm performance of the same persistent index. Such an experiment assumes that no eviction is necessary (i.e. all the required sections of the index can fit into the available main memory) and that the number of queries is small enough that almost all queries will require an index section to be faulted (i.e. the number of times a given sub-tree is accessed more than once should be negligible). These conditions ensure that the effects of caching do not impact upon the measured cold performance, and that eviction (and subsequent re-faulting) of sub-trees does not impact upon the measured warm performance. Furthermore, in order to minimise the impact on performance of the operating system's disk cache, different experiments that make use of the same on-disk index should not be performed sequentially.

Figure 4.27 shows the performance of exact matching over both cold and warm persistent TCSTs with compressed depths of 8, 10 and 12. The queries used were of lengths 12 to 100 (inclusive), and numbered 1000. The first result of note is that for a fixed compressed depth, the average time taken to perform exact matching over a cold section of the index grows more rapidly with sequence length than the average time taken to perform warm exact matching. This is because the nature of the TCST is such, that all sub-trees start at the same prefix depth; meaning that in many cases the faulted sub-tree will contain parts of the index that are not on the path corresponding to the current query. For example, if we consider a query of length 50 and a TCST with a compressed depth of 10, then the complete sub-tree corresponding to the first 10 characters of the query may need to be faulted, but only the nodes on the path corresponding to the query (and, if a match is found, those below the point of the match) are subsequently accessed. However, such costs must be amortised over the lifetime of the index: subsequent queries may access other parts of such a sub-tree, which can then be done without requiring access to the on-disk index (assuming the sub-tree has not been chosen for eviction).

The second trend worth noting is that the average time taken to perform a cold query grows less rapidly with sequence length as the compressed depth is increased. This is primarily due to the fact that increasing the compressed depth results in an index that consists of a greater number of sub-trees, with the sub-trees also being smaller than those found in TCSTs with lower values for the compressed depth. Hence, less data is accessed when faulting a typical sub-tree. Furthermore, increasing the compressed depth also increases the likelihood of encountering an entry in the two-level array that does not correspond to an entry in the original sequence (meaning that no sub-tree is to be read from disk). Overall, the best performing cold TCSTs were those with a compressed depth of 12. However, for the shortest sequences such an index would be excessively sparse, so using a lower value for the compressed depth may be advantageous in some circumstances. On average, for queries of a length greater than the compressed depth, the best performing TCSTs took approximately 300 times longer to perform a cold query than a warm query.[17]

The above experiment was repeated using a batch of 50 queries of lengths 4 to 12 (inclusive), the results of which are shown in Figure 4.28. When performing exact matching for queries that are shorter than the compressed depth, it is necessary to access all parts of the sub-trees that are faulted (i.e. no data will be read from disk

[17]It should be noted that the relative performance of cold and warm queries will vary greatly between different system environments.

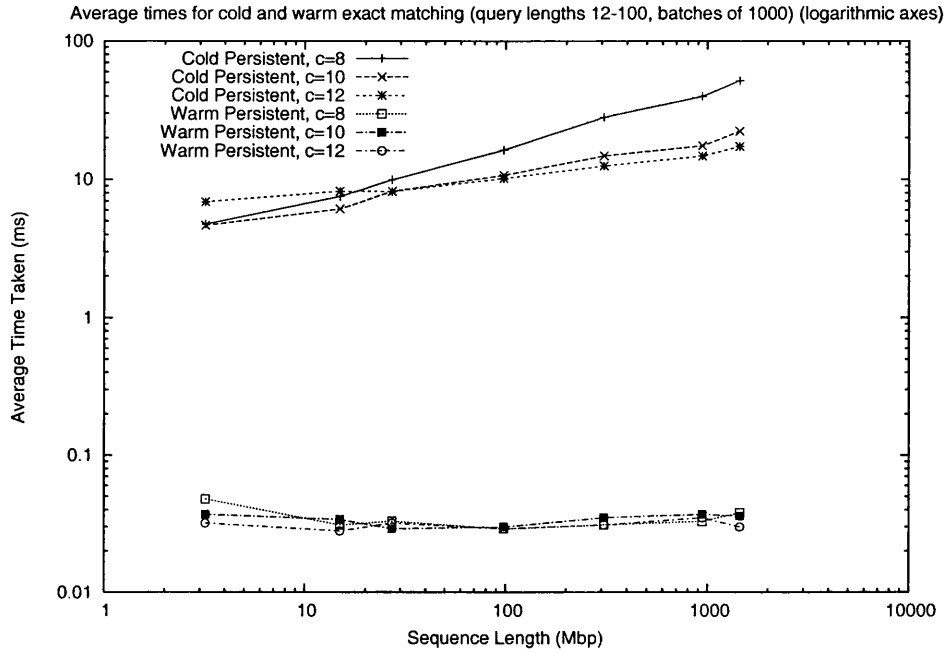

Figure 4.27: Average exact matching performance for both cold and warm persistent TCSTs, query lengths 12–100 (logarithmic axes).

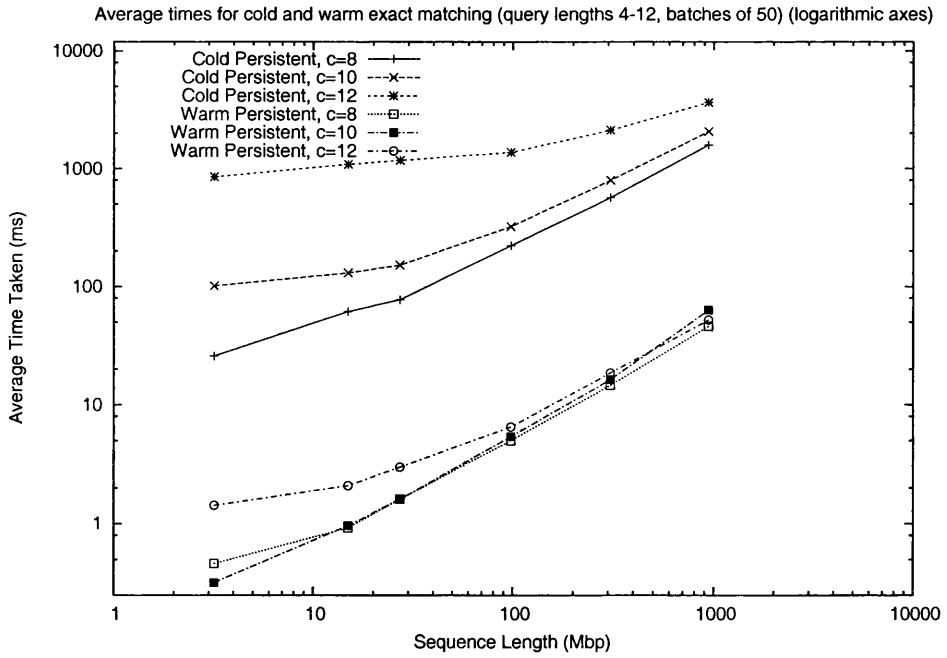

Figure 4.28: Average exact matching performance for both cold and warm persistent TCSTs, query lengths 4–12 (logarithmic axes).

and not subsequently accessed). This is reflected in the observed results, where it can be seen that the average time taken by cold and warm queries both grow with sequence length at a similar rate. As above, it can be seen that the average time taken to execute a cold query grows less rapidly with sequence length when a higher value is used for the compressed depth. However, for the range of sequences used, the best performing index in each case was the TCST with a compressed depth equal to 8 (although the difference in average time taken by the different indexes decreases significantly as sequence length increases). For shorter queries it was found that the best performing TCSTs took approximately 48 times longer to perform a cold query than a warm query.

The previous two experiments were also performed with exact matching being replaced by testing for the presence of the desired string. For queries of length 12 or greater, the cold performance of the presence test over persistent TCSTs yielded near identical results to those given in Figure 4.27. This is what would be expected, given that the same volume of data is read from disk in both cases (for queries of length greater than the compressed depth, the relevant sub-tree must be examined to establish whether or not the query string is present in the index). However, the behaviour of the index for the shorter queries is notably different. For queries that are shorter than, or equal to, the length of the compressed depth it is not necessary to access any sub-trees in order to establish the presence of the string (this can be by simply establishing if the relevant sub-tree is present or not).

Comparing Figure 4.29 to Figure 4.28, it can be seen that the cold performance of the presence test grows with sequence length at a much slower rate than that observed with the exact matching operation (when executing queries of lengths close to that of the compressed depth). Exact matching of short target patterns may require numerous sub-trees to be faulted, whereas at most one sub-tree is faulted when performing the presence test operation. Hence, the rate of growth in time taken with sequence length is relatively slow when performing the presence test. For the TCSTs with compressed depths of 8 and 10, some of the queries in the sample batch will require sub-trees to be faulted from disk, the size of which will grow as sequence length increases. For the TCSTs with a compressed depth of 12, the rate of growth in time taken is much slower as no sub-trees need to be faulted. However the average time taken to perform a cold query was still seen to grow with sequence length. This is likely to be a result of the increasing size of the on-disk index and the fact that a greater number of partitions (files) are used: the overhead of accessing the required parts of the on-disk index is likely to be slower as index size, and the number of partitions, increases.

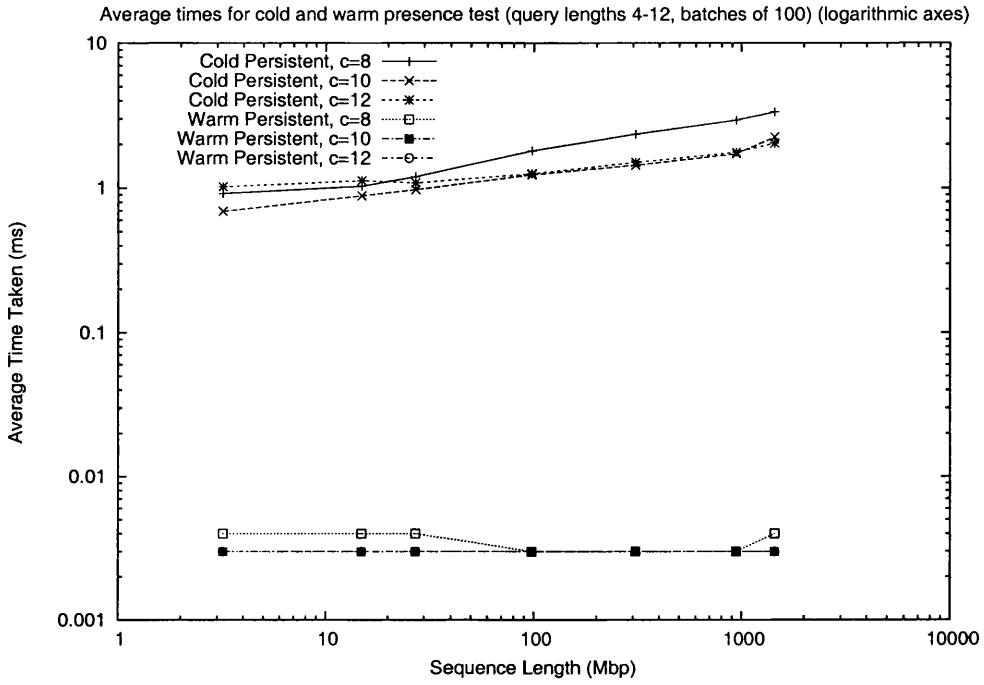

Figure 4.29: Average presence test performance for both cold and warm persistent TCSTs, query lengths 4–12 (logarithmic axes).

Although cold queries are, by their very nature, significantly slower than the corresponding warm queries, this overhead must be balanced against the benefits of using a persistent index. Firstly, with a persistent index it is not necessary to construct the data structure each time a batch of queries is to be performed. Secondly, persistent indexes allow longer sequences to be indexed than is possible with main-memory only index techniques. Furthermore, as more queries are performed using a given index, a greater part of that structure will become cached in main-memory, thus decreasing the likelihood that the required part of the index will need to be fetched from disk (although in some cases it will be necessary to subsequently evict parts of the index from main-memory to accommodate further queries). In most cases, it can be seen that increasing the compressed depth results in improved performance, the two exceptions to this being, when sequences of lengths less than 30 Mbp are indexed and when performing exact matching for queries of lengths less than 12.

The Effects of Eviction and Caching

The final aspect of query performance to be explored is that of executing batches of queries over large persistent indexes. It was shown in Section 4.4 that TCSTs can be constructed over large sequences. However, in order to demonstrate the utility of such an index it is also necessary to show that relevant query algorithms can operate efficiently over the completed index. Furthermore, such a demonstration also allows an exploration of how performance is affected by the frequency of eviction cycles and what percentage of sub-tree accesses require the on-disk index to be examined. All of the experiments presented in this section were performed using TCSTs with compressed depths of 10 or 12, built over the three longest sequences in our data set (that is, the sequences of lengths 315 Mbp, 963 Mbp and 1.482 Gbp). Three batches of queries were used: the first contained one million queries of lengths 12–100; the second contained one million queries of lengths 4–12; and the final set contained one thousand queries of lengths 4–12.

For the purposes of these experiments, an eviction cycle is triggered whenever the amount of available main memory falls below 5% of the total main memory, with eviction continuing until 20% of the main memory becomes available. Sections of the index are selected for eviction at random, with each section corresponding to one-sixth of the total prefix range (more than one such section may need to be evicted during each cycle). For each selected section, all backbone entries are set to null, with the exception of those marking known-null ribs. The parameters used here were selected as they were found to provide adequate performance over the range of experiments performed but may need to be adjusted for other workloads.

Table 4.7 shows the average query time and number of eviction cycles required when performing exact matching over large persistent TCSTs. For the first batch of queries used (one million queries of lengths 12–100), increasing the compressed depth from 10 to 12 halved the average query time and significantly reduced the number of eviction cycles invoked. However, for the second batch of queries (one thousand queries of lengths 4–12), it was found that increasing the compressed depth had little effect on the number of eviction cycles invoked and caused an increase in the average time taken to perform a query. This corresponds to the trends observed in the previous section regarding cold query performance: cold query performance and the frequency of eviction are both largely determined by how much data must be read from disk in order to satisfy the given queries. For both sets of queries it can be seen that the number of eviction cycles required increases as sequence length increases. This is what would be expected given that the number of eviction cycles is influenced by the same

Query Set	Sequence Length (Gbp)	c=10 Number of Evictions	c=10 Avg Time (ms)	c=12 Number of Evictions	c=12 Avg Time (ms)
12–100	0.315	7	2.462	0	1.378
12–100	0.965	22	4.605	0	2.547
12–100	1.482	43	6.775	1	3.597
4–12	0.315	9	841	7	2385
4–12	0.965	25	1926	27	4330
4–12	1.482	41	3041	48	5091

Table 4.7: Exact matching performance over large sequences for two separate query batches (one million queries of lengths 12–100, and one thousand queries of lengths 4–12).

factors that impact upon the number of partitions required in order to successfully index a given sequence.

Table 4.8 shows the above experiment repeated for the presence test operation. For the first batch of queries, the results obtained are equivalent to those for exact matching. However, for the second batch (one million queries of lengths 4–12), the results are notably different from those observed for exact matching. With the presence test, it was found that increasing the compressed depth reduces the number of eviction cycles triggered and, for the longest sequence, improves query performance. Again this corresponds to the trends observed previously for cold query performance.

Similar trends are also observed when the effect of caching is examined. Here, a cache *miss* is defined as being an access to a two-level array element that results in the on-disk index being examined, and a cache *hit* is where accessing a two-level array

Query Set	Sequence Length (Gbp)	c=10 Number of Evictions	c=10 Avg Time (ms)	c=12 Number of Evictions	c=12 Avg Time (ms)
12–100	0.315	7	2.328	0	1.492
12–100	0.965	24	4.622	0	2.507
12–100	1.482	41	6.965	1	3.590
4–12	0.315	0	0.820	0	0.841
4–12	0.965	1	1.246	0	1.328
4–12	1.482	5	1.908	0	1.680

Table 4.8: Presence test performance over large sequences for two separate query batches (one million queries of lengths 12–100, and one million queries of lengths 4–12).

element does not require the on-disk index to be examined (i.e. the required sub-tree is already in main memory, or has been marked as 'known null' or 'known non-null'). Note that certain queries require several two-level array locations to be accessed and may result in several cache hits or misses. Figures 4.30 and 4.31 show the ratio of cache hits and misses for the experiments described above. For batches of queries where eviction is required, the percentage of cache hits is seen to drop as sequence length (and the number of eviction cycles) increases. For batches where little, or no, eviction takes place, the percentage of cache hits stays reasonably constant as sequence length increases. Over all of the exact matching experiments, the percentage of two-level array accesses resulting in a cache hit ranged from 43% to 8%, depending on the size of the index and the nature of the queries. For the presence test operation, the cache hit rate ranged from 43% to 15% for the batch of longer queries and was found to be as high as 72% for the batch of shorter queries (which, given their length, is likely to contain many repeated queries). Given that each experiment begins with an empty in-memory index, and that queries are not expected to exhibit good locality, such modest cache hit rates are to be expected. However, all techniques that can reduce the number of times the on-disk index is accessed will improve index performance.

Summary

As would be expected, the performance of a persistent TCST is notably slower than its main-memory only equivalent. However, this overhead must be balanced against the fact that using an on-disk index allows a far greater volume of data to be indexed. The observed performance overhead is due to the costs associated with transferring sections of the index from main-memory to disk (a necessary operation with an on-disk index). However, the impact of this overhead is reduced by the effect of caching: as the number of completed queries grows, a greater percentage of the available main-memory will be occupied by faulted index sections, thus reducing the number of times the on-disk index is accessed. Additionally, when comparing the performance of a main-memory only index and a warm persistent index, it was found that the object layout used for the warm persistent index allowed for better performance than that of the main-memory only index, thus further reducing the impact on performance of using a persistent TCST. Overall, it was found that the TCST with a compressed depth of 12 gave the best performance, the only exception being, when exact matching is performed for short queries, in which case using a compressed depth of 8 was found to give better performance.

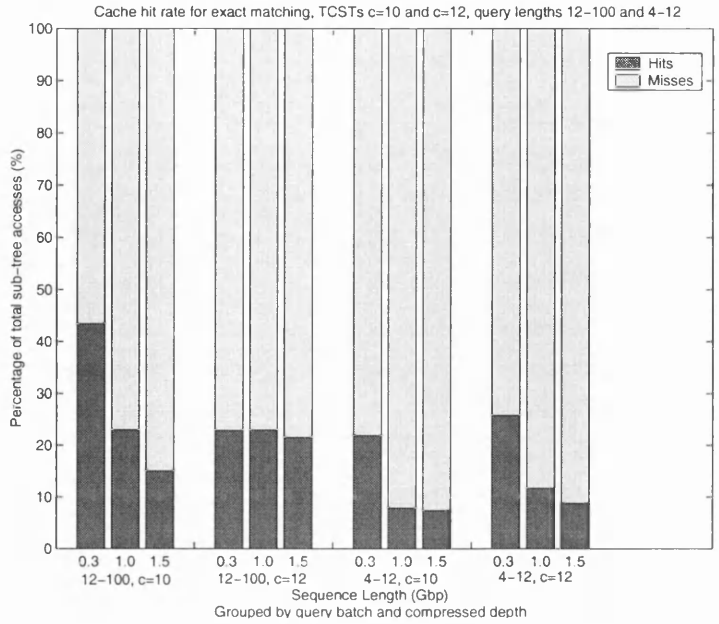

Figure 4.30: Cache hit rate for exact matching over large sequences using two query batches (one million queries of lengths 12–100, and one thousand queries of lengths 4–12).

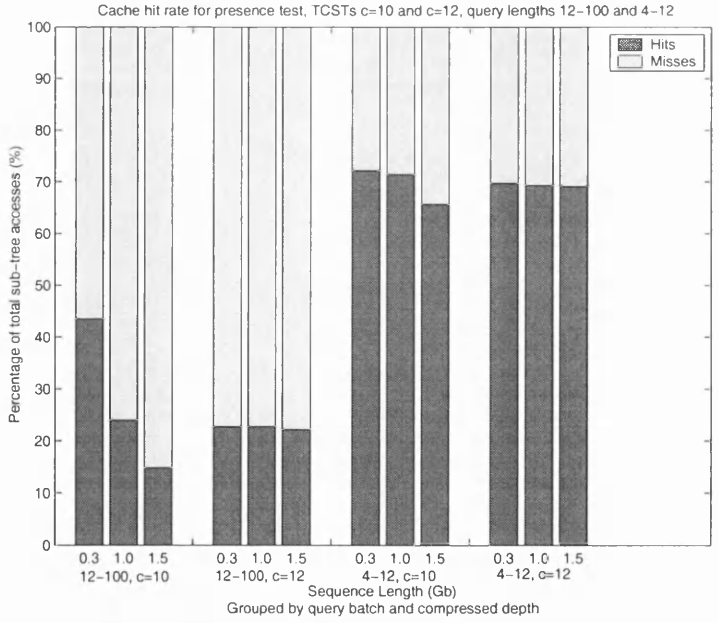

Figure 4.31: Cache hit rate for the presence test over large sequences using two query batches (one million queries of lengths 12–100, and one million queries of lengths 4–12).

4.7 Summary of Results

We now summarise the main results presented in this chapter. In Section 4.3 it was shown that the TCST can provide a more compact on-disk index than all previously published main-memory or disk-resident implementations of the suffix tree. Furthermore, it was demonstrated that it is possible to increase the compressed depth of the index as sequence length increases without incurring any space overhead. Section 4.4 discussed the performance of various techniques for constructing TCSTs. It was shown that in-memory TCSTs can be constructed in less time than that taken by more traditional suffix trees using Ukkonen's linear time construction algorithm. Additionally, it was shown that index construction time can be significantly decreased by employing, where appropriate, the Improved Prefix-Partitioned TCST Construction Algorithm or parallel construction. Section 4.6 explored the performance of both exact matching and testing for the presence of a given string over disk-resident and main-memory TCSTs. Again, it was found that in-memory TCSTs performed better than equivalent implementations of more traditional suffix trees. For disk-resident indexes, the overheads associated with using an on-disk index were explored. Overall, it was found that increasing the compressed depth of the TCST improved query performance, with the only exception being when exact matching is performed over a disk resident TCST for short queries. Throughout this chapter, we have seen that the operational parameters associated with the TCST can have a great impact on the performance of the index. In the next chapter we introduce a framework and toolkit that can assist developers and clients of novel index technology in the management of operational parameters.

Chapter 5

Generic Index Development and Operation Framework

This chapter introduces the Generic Index Development and Operation Framework (GIDOF): a framework and toolkit that aids both developers and clients working with novel indexing technology.[1] More specifically, GIDOF addresses many aspects of the management of operational parameters—a task that is of fundamental importance to the successful implementation and deployment of bespoke indexing technology, and is often non-trivial. Section 5.1 motivates the need for such a parameter management framework, drawing on the experiences of designing and evaluating the TCST (as presented in the preceding chapters). Sections 5.3 and 5.4 give the core functionality of the GIDOF parameter model. Section 5.5 defines the representation of the supported parameter types in XML. Finally, Section 5.6 describes the tools provided for use by developers and clients. Examples of the use of this framework will be given in the following chapter.

5.1 The Role of Parameter Management in the Process of Index Development and Use

Operational parameters play a significant role in the successful deployment of many applications, particularly so in the case of performance critical applications, such as indexes. In this work, the term *operational parameter* is used to refer to any option

[1] In this chapter, the term *developer* is used to refer to the person, or persons, involved in the design and implementation of the index technology in question; with term *client* used to refer to those who use the completed implementation (for example, those involved in the deployment of a server based on the index technology).

within the implementation that can, when altered, affect some aspect of the performance of the index. For example, in the previous chapter the effect of some of the operational parameters relating to the TCST were explored, namely, the compressed depth, rib size and choice of construction algorithm. Each was found to influence the performance of the index in such a way that choosing the optimum values for the parameters cannot be done without knowledge of the workload (i.e. it is not possible to simply choose values that would be appropriate in all circumstances). Furthermore, there are numerous other parameters associated with the implementation of the TCST that were not explored in detail. For example, several I/O buffers were used, the sizes of which will affect the performance of many aspects of the index. With such parameters, it is possible to choose values that are likely to give acceptable levels of performance in most circumstances, however, the value chosen may not be optimal in all cases. Hence, allowing the client of an index application to tune selected parameters may result in improved index performance.

In order for a client of a particular index technology to be able to optimise the performance of their index, it is necessary that they can change the values of the index implementation's operational parameters. This requires that some sub-set of the parameters are presented to the client in such a way that they can sensibly alter the parameters that they deem to be of interest. Additionally, it will be required that some form of documentation is provided, as the client will need to be informed as to the form and purpose of the parameter. Such tuning is more likely to be of interest when an index is to be long-lived and heavily used as this allows the costs associated with tuning the index to be recouped.

So, why is it beneficial to provide a framework and toolkit to support the tasks associated with the management of operational parameters? Firstly, the need for careful parameter management is not unique to the implementation of the TCST. It is a feature of many persistent systems: for example, the SPHERE persistent object store [93] and the Oracle family of databases [84] both require some degree of parameter tuning if adequate levels of performance are to be achieved. Any index system of even moderate complexity is likely to require similar tuning if implemented using general purpose persistence systems, or when implemented using bespoke techniques. This suggests that such a framework and toolkit is likely to be of use to developers of such systems. Secondly, developing 'ad hoc' techniques to manage parameters is likely to result in redundant development effort, with each index implementation providing its own variation on this task. Provision of a toolkit for the management of operational parameters could lessen the development effort required when implementing indexes with tunable

parameters. Finally, by providing tools that allow a given set of parameters to be manipulated in a structured manner, the parameter management tasks performed by both developer and client may be simplified.

In addition to the benefits described above, a suitable parameter management framework and toolkit can also provide a means of communication between the developer and client. Such communication is necessary if the client is to successfully tune their index: it is essential that the role of a parameter, and what effect changing it will have, is known before a value can sensibly be chosen. Before discussing the specification of the GIDOF parameter model, the tasks associated with parameter management at both the development stage and the deployment stage are discussed in detail.

5.1.1 Development: Parameter Identification and Experimentation

Identifying parameters that affect index performance is a fundamental part of the successful implementation of any index technology. As implementation progresses, the developer will introduce and specify parameters as necessary, gradually building the complete list of parameters associated with the index application. This iterative specification of the parameter list reflects the piecemeal nature of software development, and it is vital that any parameter model and toolkit supports such use.

Once a given parameter has been identified, the developer must then decide how this parameter is to be treated. The simplest way of doing this would be to simply choose a suitable value, based on past experience and assumptions about the use of index, and then fix this choice in the code. This approach is clearly limited in that the assumptions made may not accurately reflect the use of the index and that changing the value of the parameter requires knowledge of the development process. Similarly, the developer could explore various performance measurements before fixing the parameter's value, or possibly establishing rules to determine the value. Although such experimentation would provide additional insight into the effect of changing the parameter's value, it still relies on certain assumptions about the final use of the index (for example, both the workload and system environment used may differ from that of the client of the index). In both cases, the existence of the parameter, the value chosen and the implications of this choice will be 'hidden' in the details of the code. Any party wishing to tune the performance of the index would not readily have access to such a parameter and would be unable to fully address the aspects of performance influenced by this parameter. Additionally, if the developer wishes to change the value of this parameter, they must locate and alter the correct part of the source code. Separating the parameter specification from the implementation would allow parameters to be

tuned more readily, and this is how parameters are to be treated by both the framework and toolkit.

Once a parameter has been identified, the developer could decide to present it in such a way that the client may alter its value. This may be because the parameter has a significant impact upon performance and that its value cannot be determined without knowledge of the client's workload. Exposing a parameter in this manner has the advantage of allowing the client to tune the index implementation to suit their needs. However, this also requires that the developer provides a suitable mechanism for altering the parameter's value and that the developer fully documents the role of the parameter. In particular, documenting such parameters is of importance as it is only the developer who will have sufficient knowledge of the index to be able to judge which parameters will affect which aspects of performance. Given this observation, it is essential that the parameter model provided by the framework allows each parameter to be annotated, thus providing a mechanism for the developer to communicate their knowledge of the parameter's function to the client.

In order to be able to fully document the parameter list associated with an index implementation, the developer may wish to explore the effect of changing certain parameters. This allows performance trends to be established and parameters to be annotated accordingly. In some cases the developer may use this information to tune certain aspects of the index, or restrict the range of values that a parameter might take to those identified experimentally as providing acceptable performance. This process of experimentation is equivalent to that which was discussed in the previous chapter, where the effects of changing several operational parameters related to the TCST were explored, and resultant trends identified. Such experimentation requires that the developer can easily edit and refine the parameter list associated with the index.

In summary, as part of the development process the developer will identify and annotate the various operational parameters associated with the implementation of the index in question. This process will be iterative, with the parameter list evolving as implementation progresses. Furthermore, the developer may also undertake a performance evaluation to establish the effects of altering certain operational parameters. This information can then be used by both the developer and client to tune the index implementation.

5.1.2 Index Operation: Tuning

If a particular data set is likely to be queried extensively then it may be beneficial to tune the indexes used to access this data. In particular, the client of the index

technology may wish to optimise the index for a given workload and system environment, potentially after undertaking some performance evaluation. It is likely that only certain aspects of performance, and therefore certain operational parameters, will be of interest. For example, if the data set and index are to be long-lived, then query performance is more likely to be explored than index construction performance. In order for index tuning to be performed effectively, the client will need to know what parameters are available for tuning, what these parameters correspond to, and what the effect of changing the parameters would be. This information is to be provided by the developer, as only the developer will have the necessary expertise. Correspondingly, any tool provided for tuning a given index implementation should be arranged in such a way that only those parameters of interest need be addressed.

The way in which operational parameters are presented to the client of the index will determine how any tuning or evaluation processes are undertaken. For example, it would be unrealistic to expect a client of the technology to alter operational parameters by directly editing the source code of the implementation as this would require significant knowledge of the design and implementation of the index. More common mechanisms for allowing operational parameters to be adjusted include the use of command-line arguments or configuration files. The former can be effective when only a small number of options are available, but quickly becomes inappropriate as the number of parameters rises; the latter is more appropriate where a larger number of parameters are available. Although these techniques allow an application to be tuned, they do not readily allow for detailed annotations to accompany parameter definitions, an important property if tuning operations are to be successful.

5.2 Examples of Performance Tuning Tools

The increasing complexity of modern data management systems has seen a growth in the demand for applications that allow end-users to tune a given system to suit the needs of their applications [26]. In particular, many database management systems are supplied with tools that allow operational parameters to be altered by the end-user. For example, the *Oracle Tuning Pack* [91] contains several applications that allow operational parameters to be adjusted, each addressing a specific aspect of the database implementation. Similar tools are available for other commercial databases, such as DB2 [95]. Systems of this type often make use of a set of rules to in order to allow past experience of tuning the system to be reflected in the values of operational parameters (such parameters will not be exposed to the end-user) [73]. This is similar

to the use of rules within expert systems [27].

The tuning mechanisms supplied with commercial databases are each specific to a single database platform, thus they are not suitable for use by developers of bespoke applications. Examples of more general parameter configuration tools include the package `java.util.prefs`, which is supplied as part of the *Java 2* Platform[2] [38], and numerous commercially available XML property editor applications. However, these more general tools are aimed at allowing users to manage simple application preferences and do not address the wider task of parameter management: for example, they do not support the use of rules to allow previous tuning experience to be readily exploited. Parameter management tools also have applications in many other areas of computing science, including information retrieval [53] and parallel programming [5].

5.3 Overview of Functionality

The Generic Index Development and Operation Framework consists of an implementation of a flexible and comprehensive parameter model, together with two applications that allow parameter lists to be defined and edited. The design and implementation of GIDOF has been independent of the TCST, and is such that it is possible to make of use of this technology when implementing any application. However, GIDOF has not been evaluated using any other application implementation, and, as such, proof that the implementation is generic is not provided. The primary purpose of the implementation of the parameter model is to provide a means by which an index implementation can access and manipulate the parameters associated with it. The key features of the parameter model, and its implementation within GIDOF, are given below.

- *Parameter Grouping* Parameters are split into two groups: the first is a flat list of parameters that are shared amongst all aspects of the implementation; the second group is arranged hierarchically and can be used for parameters that have more limited scope.

- *Phases* Parameters can have a fixed lifespan associated with them, which is defined in terms of phases: this ensures that parameters with values that are dependent on other parameters are not accessed before they have a meaningful value.

[2]The package `java.util.prefs` is available as part of the Java 2 Platform as of version 1.4.0, see `http://java.sun.com/j2se/` for details.

- *Rule-Based Parameters* Parameters can be defined in such a way that their value is derived from a sequence of rules (conditions and corresponding actions), allowing a parameter's value to be derived at run-time.
- *Event-Based Sharing* Events are triggered whenever a parameter's value changes, allowing components that make use of the parameter to respond, if necessary, to the change.
- *XML Representation* Use of XML allows the parameter model to be represented in a structured textual format, which can then be used to provide a means of persistence for the definitions and values of the current parameter list.

Two applications are provided as part of GIDOF. The first is for use by those involved in the implementation of the index, who are responsible for identifying, defining and annotating all of the parameters associated with the index. The second application is for use by clients of the index, who can, if desired, explore the effects of changing the values of certain parameters. The input, where appropriate, and output from these applications will take the form of the XML representation of the parameter list. If desired, it would be possible to create, or edit, this XML file using third party tools. However, it is recommended that the supplied applications are used as they guide the user through the task and ensure that specified parameters are valid. The two applications are described below.

- *Developer's Parameter Tool* A graphical application that allows the complete annotated parameter list associated with an index to be defined: the parameter list is built up iteratively as implementation (and possible experimentation) progresses, with some parameters identified as being 'tunable' (i.e. its value may be altered by the client).
- *Client's Parameter Tool* A graphical application that allows those deploying the index to tune selected parameters associated with the index implementation (the tunable parameters having been selected and annotated previously by the developer).

Together, the two applications provide a means for supporting parameter management at all stages of index development and use. The developer's application supports both initial development and experimentation, and is expected to be used repeatedly as development progresses: new parameters can be added to the existing model as and when they are required, and the nature of existing parameters can be altered. The

same tool can also be used to support a possible experimentation, or evaluation, phase, with the existing parameters updated according to the results of any experimentation. The client's application allows the parameters of the index application to be tuned to suit the given workload and environment. This tool can also be used to support experimentation, albeit in a more restricted way than is supported by the developer's application.

5.4 Parameter Model

The first aspect of GIDOF to be discussed in detail is the parameter model and its implementation in Java. This model has been designed to be flexible and provides the developer with a variety of ways to structure the parameter list for their index, while still providing the application with a straightforward view of the parameters available to it.

5.4.1 Parameter Styles

Three styles of parameter are supplied with GIDOF, with each style corresponding to a different way in which the developer may want to use a given parameter. However, all parameter styles have certain common attributes, allowing the application to view different parameter styles uniformly (allowing the style of a parameter to be changed without requiring any significant changes in the corresponding application). Correspondingly, the implementations of the three parameter styles all extend the same abstract class defining the common attributes and accessor methods. The following attributes are associated with every parameter.

- *Name* A unique name that is used to identify the parameter (see also Section 5.4.2): this identifier is used by the application to select a required parameter.
- *Type* A parameter's value can be specified as being of one of the following types: `String`, `Character`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` or `Double` (i.e. object representations of all primitive types supported by Java, with the addition of the type `String`).
- *Value* The current value associated with the parameter, which will be an object of the specified type when used by the index application, and will be treated as text when manipulating the XML representation.

- *Description* A textual description of the parameter's purpose, and, where appropriate, what the implications of changing the parameter's value would be.
- *User Mode* States whether or not the parameter's value can be altered using the client's parameter tool, i.e. whether or not the parameter is tunable.
- *Constant Value* States whether or not the parameter's value can be changed by the index application. Such a parameter may still be tunable prior to invoking the application, but its value may not subsequently be altered once the program is running.

In addition to the above attributes, parameters will also have attributes specific to the style of parameter used. The three parameter styles, their attributes and predicted use are now discussed.

Simple Parameters This represents the most basic form of parameter supported by GIDOF. The value of such a parameter may be any value of the specified type.[3] In the case of the numeric data types, this may correspond to a particularly wide range of values. As such, Simple Parameters are only of limited use for representing tunable parameters—it is unlikely that such a wide range of values would be sensibly used with the application, although such situations may still occur. However, they are useful for representing parameters that are modified by the index application, or parameters that the developer may be exploring.

Fixed-Choice Parameters Associated with this form of parameter is a pre-defined list of values, with the range of values supported by the parameter being limited to those on this developer specified list. Fixed-Choice Parameters are most likely to be of use where the developer has identified a range of suitable values for a given parameter, and then wishes to allow the client of the index application to choose which value best suits their need. Examples of the type of parameters that may be represented in this manner are parameters that correspond to different algorithm choices or different buffer sizes.

Rule-Based Parameters This form of parameter is distinct from the previous two in that the value associated with a given parameter is calculated at run-time according to a set of conditions and actions specified by the developer (a single rule is composed

[3]When dealing with the XML representation, it is more accurate to say that a Simple Parameter can take any value that can be parsed as the stated type.

of one condition and a corresponding action). The purpose of such a representation is to allow parameters to be specified in such a way that their value is dependent on the current workload and system environment, allowing the most appropriate value to be used without need for manual tuning. This allows performance trends identified by the developer to be exploited. For example, if a relationship was found between buffer size and input size, such a relationship could be expressed using a Rule-Based Parameter. A full description of the conditions and actions associated with Rule-Based Parameters is given in Section 5.4.3.

Accessing a Parameter's Value

All three parameter styles share common accessor methods to enable the parameter's value to be retrieved and, where appropriate, modified. In all three cases, retrieving a parameter's value using the `getValue` method simply results in an object of the correct type being returned—it is up to the application how this value is subsequently used. Note that in the case of the Rule-Based Parameter the value returned will be that which was calculated previously by evaluating the associated conditions and actions, i.e. the rules are not re-evaluated upon each access to the parameter (this behaviour can, if desired, be achieved by explicitly invoking the `tune` method before accessing the parameter). Unless the parameter is specified as having a constant value, or is an instance of a Rule-Based parameter, it is possible to modify its current value using the `setValue` method. When invoked, this method will check that the supplied value is of the correct type and, in the case of a Fixed-Choice Parameter check that the value matches one of the pre-defined values, before updating the parameter's current value.

5.4.2 Parameter Grouping

The parameters within a GIDOF parameter list can be split into two groups. The first grouping, referred to as the *shared parameters*, will contain the parameters that are fundamental to the index implementation and have values that will be needed by more than one component within the application implementation. Parameters within the shared grouping are stored as an unordered set, each being identified by a unique name. It is expected that there will be few such parameters in any given parameter list. The second grouping of parameters contains those of more limited scope; parameters that will typically only be accessed by one component. This second grouping, referred to as the *local parameters*, is expected to contain a greater number of parameters than the first grouping and is arranged hierarchically. This hierarchy allows the parameters

to be classified according to use, thus simplifying the task of identifying parameters of interest. Note that the nature of this hierarchy is essentially arbitrary, allowing the developer to classify parameters according to any suitable criteria. For example, it may be decided to classify parameters according to function, such as grouping I/O related parameters together, or it may be decided that the grouping of parameters should closely mimic the package and class hierarchy of index implementation. All parameters at a given point in the hierarchy must have unique names, although it is possible to have parameters at different points in the hierarchy sharing a common name. By allowing the developer to specify the exact nature of the hierarchy, and deciding which parameters belong in the shared grouping, it is possible for the developer to structure their parameter list in such a way that the client will only need to browse a sub-set of the parameters in order to establish which are of interest.

The main purpose of splitting the parameters into two distinct groups (and then further classifying the local parameters) is to simplify the processes of identifying and manipulating required parameters, particularly so in the case of a client of the index application who is attempting to tune the index. While such grouping may not be necessary if only a handful of parameters are associated with a given index, it is expected that allowing parameters to be organised in this manner will become increasingly important as the number of parameters associated with a system grows.

5.4.3 Rule-Based Parameters

Rule-Based Parameters are parameters that can be specified in such a way that their value is derived at run-time dependent on the value of other parameters. This can allow relationships between workload, parameter choice and performance to be exploited without requiring manual tuning. Furthermore, it means that such relationships are expressed in a manner that is independent of the code. This allows the conditions and actions of a parameter to be adjusted by the developer without the need for source code to be re-compiled (as would be the case if equivalent rules were simply specified as part of the implementation of index application).[4] Thus, parameter exploration and establishing associated rules can be performed independently of the implementation process.

The technique used to represent Rule-Based Parameters in GIDOF is as follows. Associated with each Rule-Based Parameter is a list of *conditions* and corresponding

[4]Consider the case of self-organising data structures, such as self-balancing B-Trees [2]: the rules that determine the behaviour of such structures is typically embedded in the implementation of the algorithm and cannot be altered without making changes to the source code.

actions. To determine the value of the parameter, each condition will be taken in the order specified by the developer until the first one that evaluates to `true` is encountered.[5] The action corresponding to this condition will then be evaluated, and the resulting value used as the parameter's new value. If none of the conditions is found to hold true then the parameter is not updated, thus the developer should ensure that all possibilities are covered. This is somewhat similar to the *case statement*, as found in numerous programming languages, albeit in a more specialised form. Three variations of Rule-Based Parameters are provided in GIDOF, each invoking the evaluation of the parameter's value under different circumstances. The three variations are described below.

- *Evaluate On-Demand* The value of the parameter will only be evaluated when explicitly requested by the index application.
- *Evaluate Once* The value of the parameter will be evaluated only once at a specified point in the lifetime of the index (see Section 5.4.4): this behaviour can also be achieved using the on-demand variation of the Rule-Based Parameter.
- *Self-Adjusting* The value of the parameter will be updated whenever the value of any parameter that it is dependent upon (i.e those that feature in any of its conditions or actions) changes.

Defining Conditions and Actions

The conditions and actions corresponding to a Rule-Based Parameter need to be expressed in such a way that they have a straightforward text based representation and that they can be evaluated simply at run-time. The need for a textual representation is to allow the conditions and actions to be input directly by the developer when using the Developer's Parameter Tool. Additionally, this also allows the conditions and actions to be stored simply as strings within the XML representation of the parameter list. The package used for this purpose was JEP (Java Expression Parser).[6] This package allows a variety of mathematical expressions to be parsed and evaluated, and supports the use of a number of common functions and constants. Furthermore, JEP supports the use of user defined variables, which can be used to represent the values of other parameters within GIDOF.

[5] Note that any non-zero value is taken to be `true`, with `false` corresponding only to the value zero.

[6] JEP - Java Mathematical Expression Parser, `http://www.singularsys.com/jep/`, as accessed September 2004.

Both conditions and actions are given as JEP expressions. Conditions will be evaluated using JEP, with the current value of any referenced variables previously being loaded into the JEP environment (and updated as necessary). This result will then be treated as a boolean value with all non-zero results being interpreted as being true. Actions are also specified as JEP expressions and will be evaluated if the corresponding condition is found to hold. Note that literal values are valid JEP expressions, and, in the case of `String` or `Character` types, only literal values are allowed as actions. A description of the operators, functions and constant values available for use within JEP expressions is given in Appendix C.4.4.

5.4.4 Phases

Given that Rule-Based Parameters can be specified in such a way that they are dependent on the values of other parameters, it is necessary to provide a mechanism to prevent such parameters from deriving their value from parameters that have yet to be given a meaningful value. This can be achieved through the use of *phases*. Phases can be used to mark significant milestones in the lifetime of the index and can be used to signal that a Rule-Based Parameter can meaningfully derive its value. Phases are specified by the developer as an ordered list, with the application specifying when a particular phase is to be entered or exited. As was the case with parameter grouping, the nature of the phases used is arbitrary, and can be omitted if they are not deemed to be necessary. No aspect of the use of phases is explicitly exposed to the client of the index application. Example uses of phases could be to distinguish between index construction and use of the completed index as a server, or to separate initialisation from the main algorithm.

Phases are primarily intended for use with Self-Adjusting Parameters; however they are also of use when dealing with Rule-Based Parameters that are to be evaluated only once at a specific point in the index's lifetime. In the first case, the temporal scope of a Self-Adjusting Parameter is defined in terms of a start and end phase (inclusive). When a particular phase is entered, any Self-Adjusting Parameter with that phase as its start phase will begin to auto-tune (i.e. its value will be recalculated whenever a value that it is dependent upon changes). This will continue until the end phase has passed, with several phases potentially passing in between. In the second case, a Rule-Based Parameter that is only to be evaluated once can be set at either the start or end of a specified phase.

5.4.5 Change Events

If shared parameters are used in a multi-threaded environment then it is necessary to provide a mechanism to alert a component that the value of a parameter that it uses has been changed. Furthermore, such a mechanism is also necessary to allow Self-Adjusting Parameters to be notified of changes to the parameters on which their value is dependent. This can be achieved by triggering a `ValueChangedEvent` whenever a shared parameter's value is changed. Any component that wishes to be notified of such changes can implement the `valueChanged` method and register a suitable implementation of the `ValueChangedListener` interface. The component can then process, or ignore, the updated value as necessary. A similar mechanism is also used to allow notification of changes in phase to be propagated.

A parameter that has been specified as self-adjusting is to be updated whenever a parameter that features in any of its conditions or actions has its value changed. Notification of such a change is achieved via the event mechanism described above. However, rather than the application implementing a suitable event listener, such a listener is provided by GIDOF. This component is initiated whenever the parameter list is opened by the application and is responsible for updating all Self-Adjusting Parameters that are affected by any given parameter value change.

Update Cycles

Given that events are used to initiate the updating of Self-Adjusting Parameters, and that any resulting update will also cause such an event, it is possible that a cycle of updates may be created. Such a cycle may be self-terminating: after a small number of iterations, the parameters may all reach a stable state. However, it is also possible that such a cycle may result in infinite propagation of updates, which could result in degradation of application performance. In GIDOF, such cycles are permitted and it is left to the judgement of the developer to establish if the dependencies are suitable. However, as an aid to the developer, the presence of such a cycle will be flagged by the Developer's Parameter Tool.

Detecting an update cycle is achieved by first creating a directed graph corresponding to all the Self-Adjusting Parameters. Each parameter will become a vertex on the graph, and each dependency will be represented as an edge directed in the order of the dependency. Note that such a graph typically will not be a connected graph. Given the completed graph, detecting a cycle can be achieved using a suitable implementation of a depth first traversal. A thorough introduction to directed graphs and corresponding

algorithms is given by Manber [79].

5.4.6 Summary

The GIDOF parameter model supports a wide variety of parameter types and provides various mechanisms to aid the developer in classifying and managing the parameter list associated with an index application. Developers need only use the features that meet their needs, allowing parameter lists to be as simple, or as complex, as necessary. Clients of the index application will see only a sub-set of the complete parameter list, with implementation details, such as the use of phases or Rule-Based Parameters, being hidden. The following section gives the schema for the XML representation of the parameter model.

5.5 XML Representation

This section defines the structure of the XML representation of the GIDOF parameter model. XML is used to provide a means of persistence for the parameter definitions and values, thus preserving this information between application invocations and providing a representation of the parameter list that is independent of the application source code. The internal representation of the parameter model can be created from the corresponding XML file when an application is invoked, with all subsequent changes made to the parameter values affecting only the internal model. If it is required that these changes be recorded, then the application must request that the internal model is written to an appropriate file. Selected elements from the XML Schema definition of the parameter list representation are given below, with the complete schema being presented in Appendix B.1.

5.5.1 Schema

The schema for the GIDOF parameter model is defined using XML Schema, and can be used to validate XML documents corresponding to parameter lists. XML documents, and corresponding schemas, are hierarchical in nature, thus the schema is presented in a top-down manner. The schema is self-contained, meaning that it does not refer to any elements defined elsewhere (with the exception of the constructs used to define the schema itself) and that all elements are part of a single namespace. Note that, by default, each element defined in the schema will appear exactly once as a child of the enclosing element, with all exceptions being explicitly annotated.

Top-Level Element The root element, named `properties`, encompasses the complete parameter model and is defined in Figure 5.1. This element simply consists of three child elements, respectively representing the list of phases, the shared parameters and the local parameters.

```
<xs:element name="properties">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="phaseList"/>
         <xs:element ref="sharedParameters"/>
         <xs:element ref="localParameters"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.1: Root element of the parameter model schema.

Phase List The `phaseList` element represents the list of phases used in the parameter list, and is defined as an unbounded sequence, or list, of `phase` elements (see Figure 5.2). Given that the use of phases is optional, the list is allowed to be empty. Each `phase` element has four attributes: the first corresponds to the name of the phase, with the remaining three solely being used for the purposes of tracking which phases have been completed, and the order in which they are to be processed.

Shared Parameters The list of shared parameters is represented by the element `sharedParameters`, as defined in Figure 5.3. There are no constraints on the number of shared parameters that may be present in a given parameter list, thus the number of child elements is unbounded and may be empty. Each parameter is represented using the `property` element (discussed later), this element can be used to represent all forms of parameter supported by GIDOF.

Local Parameters The representations of both the group hierarchy and the local parameters are encompassed by the `localParameters` element. This element consists of a potentially empty, unbounded list of `group` elements. In turn, each `group` element is comprised of a number of `property` elements, corresponding to the pa-

```
<xs:element name="phaseList">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="phase"
                     minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="name" type="xs:string"
                             use="required"/>
               <xs:attribute name="donePre" type="xs:boolean"/>
               <xs:attribute name="donePost" type="xs:boolean"/>
               <xs:attribute name="number" type="xs:integer"
                             use="required"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.2: Element representing the list of phases in the parameter model schema.

```
<xs:element name="sharedParameters">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="property"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.3: Element representing the list of shared parameters in the parameter model schema.

rameters associated with that point on the group hierarchy, and a number of group elements, corresponding to the sub-groups of the group represented by the current element. Both lists are unbounded and are allowed to be empty. Figure 5.4 defines both the localParameters element and the group element.

```
<xs:element name="localParameters">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="group"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="group">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="property"
          minOccurs="0" maxOccurs="unbounded"/>
         <xs:element ref="group"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complex
```

Figure 5.4: Elements representing the local parameters and groups of the parameter model schema.

Property Each parameter within a parameter list is represented by the property element (see Figure 5.5). The attributes of this element correspond to the generic attributes of a parameter, with the specifics of the actual parameter being given as a child element (which is defined as being a choice of one of the supported parameter types). The four attributes of the element correspond to the parameter's name, user mode, type and constant value status (as described in Section 5.4.1). The values that can be taken by the attributes userMode and valueType are constrained to being those supported by GIDOF (see Appendix B.1 for details). The parameter's description is given as a child element that has a value of the type string. The parameter's value is

not given by the property element, instead it is found in the child element representing the details specific to the given parameter type.

```
<xs:element name="property">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="description" type="xs:string"/>
         <xs:choice>
            <xs:element ref="simpleParameter"/>
            <xs:element ref="fixedChoiceParameter"/>
            <xs:element ref="ruleBasedParameter"/>
         </xs:choice>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
                    use="required"/>
      <xs:attribute name="userMode" type="userMode"
                    use="required"/>
      <xs:attribute name="valueType" type="supportedTypes"
                    use="required"/>
      <xs:attribute name="final" type="xs:boolean"
                    use="required"/>
   </xs:complexType>
</xs:element>
```

Figure 5.5: Property element of the parameter model schema.

Simple Parameter Figure 5.6 defines the XML representation of a Simple Parameter. This element, `simpleParameter`, is comprised solely of a single child element giving the parameter's value, which is stored as type String (all values in GIDOF are represented using text).

Fixed-Choice Parameter Figure 5.7 defines the XML representation of a Fixed-Choice Parameter. This element, `fixedChoiceParameter`, is comprised of an unbounded list of `choice` elements, each corresponding to a single value in the list of potential values associated with the parameter. At least one `choice` element must be present (otherwise, it would not be possible to associate a value with the parameter). The current value of the parameter is not stored explicitly, instead the number cor-

```
<xs:element name="simpleParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="value" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.6: Element representing a simple parameter in the parameter model schema.

responding to the currently selected value is recorded as an attribute of the element (with a negative number used to indicate that no value has been selected).

Rule-Based Parameter The final element in the schema to be given here is the representation of a Rule-Based Parameter. The definition of the `ruleBasedParameter` element is given in Figure 5.8. Associated with each Rule-Based Parameter is a list of conditions and actions. These are represented by two unbounded lists: the first list is comprised of `condition` elements and the second list is comprised of the corresponding `action` elements. Both are unbounded and must have at least one element present. Both types of element, `condition` and `action`, have a single child element which simply holds the string corresponding to the JEP expression.[7] The current value of the property is stored as a child element of type String: this is required to enable derived values to be preserved between program executions. Finally, each Rule-Based Parameter operates in one of the three modes described in Section 5.4.3. Hence, each `ruleBasedParameter` element has one child element corresponding to the selected mode of operation for the parameter. These child elements record (where necessary) details of the phases corresponding to the evaluation of the parameter.

[7]The definition of both `condition` and `action` elements are omitted here for brevity, their definitions are given in the complete schema presented in Appendix B.1.

```
<xs:element name="fixedChoiceParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="choices">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="choice"
                              minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="value"
                                       type="xs:string" />
                        </xs:sequence>
                        <xs:attribute name="number"
                                      type="xs:integer"
                                      use="required"/>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
               <xs:attribute name="currentChoice"
                             type="xs:integer"
                             use="required"/>
               <xs:attribute name="numberOfChoices"
                             type="xs:integer"
                             use="required"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.7: Element representing a Fixed-Choice Parameter in the parameter model schema.

```
<xs:element name="ruleBasedParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="rules">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="condition"
                              minOccurs="1" maxOccurs="unbounded">
                  <xs:element ref="action"
                              minOccurs="1" maxOccurs="unbounded">
               </xs:sequence>
               <xs:attribute name="numberOfRules" type="xs:integer"
                             use="required"/>
            </xs:complexType>
         </xs:element>
         <xs:choice>
            <xs:element name="onDemand" />
            <xs:element name="evaluateOnce">
               <xs:complexType>
                  <xs:attribute name="phaseName"
                                type="xs:string" use="required"/>
                  <xs:attribute name="isPre" type="xs:boolean"
                                use="required"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="selfAdjusting">
               <xs:complexType>
                  <xs:attribute name="startPhase"
                                type="xs:string" use="required"/>
                  <xs:attribute name="isStartPre"
                                type="xs:boolean" use="required"/>
                  <xs:attribute name="endPhase" type="xs:string"
                                use="required"/>
                  <xs:attribute name="isEndPre" type="xs:boolean"
                                use="required"/>
               </xs:complexType>
            </xs:element>
         </xs:choice>
         <xs:element name="currValue" type="xs:string" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 5.8: Element representing a Rule-Based Parameter in the parameter model schema.

5.5.2 Examples

Here, we present an example of each of the three forms of parameter used within the GIDOF parameter model. Each example is drawn from the TCST configuration file (see Appendix B.2), with a fuller discussion of the TCST operational parameters being given in the following chapter. Figures 5.9 and 5.10 show the XML elements corresponding to the TCST operational parameters representing the 'number of partitions' and the choice of 'alphabet'. The former is an example of Simple Parameter and uses the `simpleParameter` element, as defined in Figure 5.6. The latter is an example of a Fixed-Choice Parameter and uses the a `fixedChoiceParameter` element, as defined in Figure 5.7. Both parameters also make use of the more general element `property` (defined in Figure 5.5).

```
<property final="false" name="numberOfPartitions"
      userMode="User Defined" valueType="java.lang.Integer">
  <description>
     The number of partitions to be used when constructing the
     index.  For larger sequences it will be necessary to increase
     the number of partitions used in order to achieve successful
     index construction.  Use of multi-threaded construction will
     require more partitions.
  </description>
  <simpleParameter>
     <value>50</value>
  </simpleParameter>
</property>
```

Figure 5.9: A Simple Parameter representing the number of partitions to be used for a given TCST.

Figure 5.11 shows the XML element corresponding to the TCST operational parameter that represents the 'choice of construction algorithm'. This is an example of a Rule-Based Parameter. As was the case with the two examples given above, the main element is an example of the `property` element. Here, the child element representing the details of how to derive the parameter's value being an example of a `ruleBasedParameter` (as defined in Figure 5.8). The given conditions are textual representations of expressions that can be parsed and evaluated using JEP, whereas the actions are simply literal values of the type `java.lang.String`. This Rule-Based Pa-

```
<property final="false" name="alphabet"
          userMode="User Editable" valueType="java.lang.String">
  <description>
     The alphabet from which the sequence file is constructed.
     The supported alphabets are: DNA {A,C,G,N,T}, RNA {A,C,G,N,U}
     and protein {A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z}.
  </description>
  <fixedChoiceParameter>
    <choices currentChoice="0" numberOfChoices="3">
      <choice number="0"> <value>DNA</value> </choice>
      <choice number="1"> <value>RNA</value> </choice>
      <choice number="2"> <value>protein</value> </choice>
    </choices>
  </fixedChoiceParameter>
</property>
```

Figure 5.10: A Fixed-Choice Parameter representing the chosen alphabet for a given TCST.

rameter has been classified as an 'Evaluate Once' parameter, hence the child element `evaluateOnce`, which defines the point at which the parameter's value is to be derived, is present.

5.6 Toolkit

The toolkit associated with GIDOF consists of two applications: the *developer's parameter tool* and the *client's parameter tool.* The developer's tool allows parameter lists to be created, with both applications allowing such lists to be edited. Although it is possible to edit, or create, an XML document using a basic text editor or general purpose XML manipulation tool, there are advantages to using the applications provided. Firstly, all user's input is validated, ensuring that both the parameter list and XML representation are valid. Secondly, both applications are structured in such a way that the user will be guided through their task. Finally, such tools suppress much of the detail of the parameter list's XML representation, thus reducing the amount of expertise required in order to make use of GIDOF (this is especially important in the case of the client's tool). The developer's tool and client's tool are discussed in the following two sections, and a tutorial giving examples of the use of both tools can be

```
<property final="false" name="constructionAlgorithm"
          userMode="User Editable" valueType="java.lang.String">
   <description>
      Determines the version of the construction algorithm to used
      with the TCST.  If set to 'skip', then the suffix grouping
      phase is bypassed and the prefix-partitioned algorithm is
      used.  If set to either 'true' or 'false', then the improved
      prefix-partitioned algorithm is used.  'True' states that
      prefix codes should be stored in addition to suffix numbers,
      'false' states that they should be ignored.
   </description>
   <ruleBasedParameter>
      <rules numberOfRules="2">
         <condition number="0">
            <expression>
               minimiseDiskUsage || seqLength < 900000000
            </expression>
         </condition>
         <condition number="1">
            <expression>seqLength >= 900000000</expression>
         </condition>
         <action number="0">
            <expression>"skip"</expression>
         </action>
         <action number="1">
            <expression>"true"</expression>
         </action>
      </rules>
      <evaluateOnce isPre="true"
                    phaseName="initiateConstruction"/>
         <currValue>null</currValue>
   </ruleBasedParameter>
</property>
```

Figure 5.11: A Rule-Based Parameter representing the choice of TCST construction algorithm.

found in Appendix C.

5.6.1 Developer's Parameter Specification Application

The developer's parameter tool serves two purposes: it allows the developer to create new parameter lists, and it allows existing lists to be edited or extended. Upon starting the application, the user is asked to select whether or not they wish to edit an existing parameter list. If so, the specified parameter list is located, and the details of this list added to the list being edited. If not, an empty parameter list is created. After this point, all aspects of the application's behaviour are identical, regardless of whether a new list is being created, or an existing list is being edited. The four steps involved in specifying a parameter list are as follows:

1. The list, and order, of any phases used is specified.
2. Shared parameters are defined.
3. The group hierarchy is created.
4. Local parameters are defined and added to the corresponding group.

The above steps are presented in this order as certain stages are dependent on the completion of others. For example, Rule-Based Parameters can be associated with phases, hence the list of phases must be complete before such parameters are added. Furthermore, any local parameter that is defined as a Rule-Based Parameter can reference the list of shared parameters, hence shared parameters are defined before local parameters. Finally, all local parameters must be associated with a group, so the group hierarchy must be specified before the local parameters are defined. The four stages involved are now discussed in the order that they are presented to the user.

Phase List This step consists of specifying the names and the order of the phases to be used with the current parameter lists. Associated with each phase is a unique identifier, this is specified by the developer and consists of an arbitrary string. Phases can be added or deleted at this stage.

Shared Parameters This step allows the shared parameters in the current list to be added, edited or deleted. When adding a parameter, the style, type and mode of the parameter are selected from the fixed set of options provided. A unique name must also be given, as must a description of the parameter's role. For Simple Parameters

it is optional to provide a value, whereas for Fixed-Choice Parameters a list of values must be specified. In both cases, all values provided are checked to ensure that they can be parsed as the selected type. For Self-Adjusting Parameters, a list of conditions and actions must be provided (specified as a JEP expression), all of which will be verified before being added to the current parameter list. If a Self-Adjusting Parameter references another parameter, then the parameter upon which it depends must be specified first. If any form of circular dependency is required, then it will be necessary to first add the parameters without the necessary rules, and then edit the conditions and actions once all relevant parameters have been added. The application will warn the user if an update cycle has been specified.

Group Hierarchy At this point, the group hierarchy is specified. The hierarchy is shown as a tree, from which groups (nodes) can be added, edited or deleted (if a group is deleted, all associated groups and parameters are also deleted). Associated with each group is an arbitrary name, which must not be the same as that of any other group sharing a common parent.

Local Parameters The final step is where local parameters are specified. The group hierarchy is displayed, allowing the user to specify which group is of interest. Once the required group has been identified, the process of editing the local parameters proceeds in the same manner as that described for shared parameters. One limitation of the GIDOF implementation is that Rule-Based Parameters can only refer to shared parameters in their conditions and actions, so it is not possible to create an update cycle at this point. Once this stage is complete, the parameter list is output to the chosen file in XML form.

5.6.2 Client's Index Tuning Application

The client's tuning application is, in several ways, similar to the developer's tool described above. It allows a given parameter list to be explored and edited, and can be used iteratively as the process of tuning progresses. However, this tool presents only the aspects of the parameter list that the client can alter. In particular, the definition of phases and Rule-Based Parameters are not shown, as they can be edited only by the developer. Furthermore, only the parameters that the user can edit are displayed—these will be the parameters that the developer has specified as being 'tunable.' The parameter list is presented by the application in two different stages. The first step is where

the shared parameters can be edited, and the second is where the local parameters can be edited. Both steps are now discussed.

Shared Parameters The user will be presented with a list of the shared parameters that have been identified by the developer as being tunable. A selected parameter's attributes, including its description, can be displayed and its value altered (if desired). In the case of a Simple Parameter, any value entered will be checked to ensure that it can be parsed as the given type. For Fixed-Choice Parameters, the value will simply be selected from the list provided. Note that no attribute of a parameter, other than the value, can be edited using this tool. Details of the parameters that are not tunable (including all Rule-Based Parameters) are not displayed.

Local Parameters In order to allow the user to browse the local parameters that have been identified as 'tunable,' the group hierarchy is displayed (although it cannot be edited by this application). When a given group is selected, the corresponding list of tunable parameters (if any) is then displayed. These parameters can then be edited, as described above.

5.7 Summary

This chapter has introduced GIDOF, a framework and toolkit for the management of operational parameters. Section 5.1 discussed the role of operational parameters during both development and use of bespoke indexing technology, concluding that a suitable toolkit may be of benefit to both developers and clients of such indexes. Section 5.3 defined the areas of parameter management that were to be addressed by the GIDOF framework and toolkit. Details of the GIDOF parameter model were given in Section 5.4: this included several categorisations of parameter use and support for Rule-Based Parameters (the values of which are derived at run-time). The XML representation of this parameter model was defined in Section 5.5, with the tools that can create and manipulate the parameter model being discussed in Section 5.6. The following chapter explores the use of GIDOF within the implementation of the TCST, demonstrating how the concepts introduced in this chapter can be used when creating a tunable index implementation.

Chapter 6

GIDOF: Implementing a Tunable Index

This chapter describes how the GIDOF parameter model was used in the creation of a tunable implementation of the Top-Compressed Suffix Tree. As part of the process of undertaking the performance evaluation discussed in Chapter 4, a parameter specification was created for the TCST: the operational parameters identified during the implementation of the techniques described in Chapter 3 were added to the parameter list, with the results of the performance evaluation allowing a description of their impact to be given. This parameter specification serves as a complete description of all the operational parameters associated with the TCST implementation, and, when used with the GIDOF client's tuning application, allows the performance of the TCST to be tuned to suit the specific needs of a given workload. In this chapter, the nature and classification of the TCST operational parameters are discussed, demonstrating how the parameter model discussed in the previous chapter can be used with this index implementation. The complete XML parameter specification for the TCST is given in Appendix B.2.

6.1 Implementation Notes

Prior to discussing the operational parameters associated with the TCST, it is necessary to describe how the parameter list is manipulated by the TCST implementation. More specifically, the implementation of the TCST discussed in Chapter 3 makes use of several applications in order to construct and query a particular index; therefore, it is necessary to understand how each application accesses the list of available parameters.

Three applications are used when constructing a persistent TCST, they are: the *job server*, which co-ordinates index construction; one or more *sub-tree builders*, which are used to construct index partitions; and finally, *create tree*, which initiates the construction of a given index (see also Appendix A). When index construction is initiated, all the applications involved will access a single XML document containing the parameter list; each application will then create and retain a main-memory copy of the list. This allows each application to access the parameters specific to its operation, while still having all parameters contained in a single file (having the complete parameter list in a single file simplifies the process of tuning the index). However, only one application is responsible for the values of the shared parameters: the create tree application will access these values and submit the relevant details to the job server, which, in turn, can supply the information to the sub-tree builders. This prevents inconsistencies in the values used for parameters that are derived at run-time (as could happen if each application independently derived its own value for such a parameter). Correspondingly, once index construction is complete, it is the responsibility of the create tree application to ensure that the parameter list is saved to disk, thus preserving the derived values of the Rule-Based Parameters. This copy of the parameter list is saved in the same directory as the completed index, and it is this copy of the parameter list that will be used by the *query engine* application.

6.2 Shared Parameters within the TCST

The first set of operational parameters to be discussed are those that were classified as shared parameters. This set includes all parameters corresponding to the basic configuration of the index; covering details of the sequence to be indexed and the form of TCST to be created. Table 6.1 lists the user-editable shared parameters associated with the implementation of the TCST.[1] The first two parameters provide details of the sequence to be indexed (respectively giving the location of the sequence file and the corresponding alphabet). This information is required by all of the TCST applications; hence, shared parameters were used. The location of the sequence file is represented using a Simple Parameter and is of type String. The choice of alphabet is also represented as a string, but, as the choice of alphabet is restricted to being one of those corresponding to biological sequence data, a Fixed-Choice Parameter is used. The next two parameters respectively represent the location of the directory where the persistent

[1]Note that the naming conventions commonly associated with the Java language have been used when naming parameters—this is not compulsory and names may be chosen to be any suitable string.

index is to be located and the hostname of the computer hosting the job server. Again, this information is required by more than one of the TCST applications, hence the use of shared parameters. Both of these parameters are represented as Simple Parameters of type String.

The final three shared parameters define the style of TCST that is to be used. These parameters specify the number of partitions that the index should be split into; the size of rib to be used in the two-level array[2]; and finally, the size of the compressed depth. The number of partitions is specified as a Simple Parameter of type Integer. The remaining two parameters are both represented as Fixed-Choice Parameters of type Integer. Fixed-Choice Parameters are used for these two parameters as, unlike the number of partitions, the range of suitable values for these parameters was identified experimentally and shown to be small (see Chapter 4 for details).

Parameter Name	Type	Parameter Style	Purpose
seqFile	String	Simple Parameter	Sequence file to be indexed.
alphabet	String	Fixed-Choice	Alphabet from which the sequence is drawn.
destinationDir	String	Simple Parameter	Directory to contain the index.
jobServerHostName	String	Simple Parameter	Hostname of server running the Job Server.
numberOfPartitions	Integer	Simple Parameter	Number of partitions into which the index is split.
ribSize	Integer	Fixed-Choice	Size of ribs in the two-level array.
compressedDepth	Integer	Fixed-Choice	The compressed depth.

Table 6.1: Shared parameters of the TCST implementation.

6.2.1 Automating Parameter Choice

The parameter list presented in the previous section gives the basic list of shared parameters used by the TCST implementation. In this section, the definition of the shared parameters is adapted to support the automation of parameter choice through the use of Rule-Based Parameters.

Of the seven shared parameters listed in Table 6.1, only two are potentially suitable

[2]It may appear that the choice of rib size should be present as a local parameter, since it is a property specific to the two-level array. However, the value chosen impacts upon the partitioning scheme (partitions must be aligned to the start and end of ribs) and it also affects more than one aspect of index performance, hence it represented as a shared parameter.

for representation with Rule-Based Parameters. These parameters are the compressed depth and the number of partitions. For the remaining shared parameters, the appropriate values will either be specific to the user's environment (as is the case with file or machine names), or the optimum value is not predictable (as is the case with rib size[3]). Given the evidence presented in Section 4.5, it would be possible to approximate the number of partitions required for a given sequence using a Rule-Based Parameter. However, successful definition of such a Rule-Based Parameter would require more detailed knowledge of the relationship between sequence length and the minimum number of partitions. In addition, it is possible that, due to the presence of highly repetitive substrings, certain sequences would require more partitions to be used than such a rule may predict (see Section 4.5.1). Therefore, the use of a Rule-Based Parameter was deemed inappropriate in this case. On the other hand, ample experimental evidence has been gathered regarding the impact of the compressed depth on index performance, thus making this parameter suitable for representation as a Rule-Based Parameter.

It was shown in Section 4.3.1 that, by starting with a compressed depth of 8, it is possible to increase the compressed depth of the TCST as sequence length grows without incurring a significant increase in the size of the disk-resident index. Therefore, it would be possible to define the compressed depth as being a Rule-Based Parameter that takes advantage of this result. However, the compressed depth also impacts upon the time taken to create the index and the time taken to perform various types of query. Typically, the best construction times were obtained when using the highest possible compressed depth (see Section 4.4). Similarly, using the highest value for the compressed depth also gave the best possible query performance, except where exact matching of short target patterns is of particular importance (see Section 4.6). When defining a Rule-Based Parameter for the compressed depth it is essential that all aspects of performance be considered.

In order for the compressed depth to be represented as a Rule-Based Parameter, it is necessary to introduce three further shared parameters. Firstly, if we are to define conditions based on the length of the sequence, then a parameter representing this value will need to be added to the list of shared parameters. The value for this parameter will be established by the application once the appropriate sequence file has been accessed. This parameter will be a Simple Parameter of type Integer and will be listed as not being editable by the user (i.e. it will not be presented by the client's tuning application as it is only used internally by the TCST implementation). It is also necessary to know

[3]Note that, of the two values identified as suitable rib sizes, both provide similar levels of performance, thus the default value will be adequate in most situations.

how the user is likely to use the completed index, thus two parameters are introduced to allow the user to state their preferences. These parameters are both of type boolean (and therefore Fixed-Choice Parameters) and allow the user to answer simple 'yes' or 'no' questions regarding the nature of the index's workload. The first parameter allows the user to state if exact matching with short queries is of particular importance (by default, this will be set to false). The second parameter allows the user to state if it is preferential for the use of disk space to be minimised (which can be to the detriment of performance), and will be initially set to false. The three additional shared parameters are given in Table 6.2.

Parameter Name	Type	Parameter Style	Purpose
seqLength	Integer	Simple Parameter	Represents the length of the sequence being indexed (not user editable).
shortExacts	Boolean	Fixed-Choice	Set to true if exact matching of short query patterns is a priority (false otherwise).
minimiseDiskSpace	Boolean	Fixed-Choice	Set to true if disk space usage is to be minimised (set to false otherwise).

Table 6.2: Supplementary shared parameters of the TCST implementation.

Table 6.3 lists the conditions and actions associated with the Rule-Based Parameter representation of the compressed depth. All rules are defined using JEP, as discussed in Section 5.4.3 and Appendix C.4.4. When deriving the value for the parameter, the conditions will be taken in the order given until the first condition evaluating to true is found. When this condition is found, the value of the parameter will be set to be the value found by evaluating the corresponding action (which is simply an Integer literal for all of the actions specified here). The first rule states that if exact matching over short queries is of importance, then the compressed depth will be set to 8 (regardless of sequence length). The following two rules apply when the user does not wish to minimise disk space: in this case the compressed depth is set to 10 for sequences of length less than or equal to 30 Mbp, and set to 12 if the sequence is found to be longer. These values were based on the findings presented in Section 4.6.4. The final four rules apply in the case that exact matching over short queries is not of importance and the user does want to minimise disk-space usage. Here, the value of the compressed depth increases with sequence length, with the values at which it increases taken from the

Condition	Action
`shortExacts`	8
`!minimiseDiskSpace && seqLength <= 30000000`	10
`!minimiseDiskSpace && seqLength > 30000000`	12
`seqLength <= 25000000`	8
`seqLength > 25000000 && seqLength <= 300000000`	9
`seqLength > 300000000 && seqLength <= 1400000000`	10
`seqLength > 1400000000`	12

Table 6.3: Conditions and actions defining the value of the compressed depth when represented as a Rule-Based Parameter.

results presented in Section 4.3.1.

The final part of the specification of this parameter concerns the point at which the value will be derived. As the value of the compressed depth is dependent on the length of the sequence, its value must only be derived once the `seqLength` parameter has been given a value. Additionally, once the value of the compressed depth has been established it will not subsequently alter. Therefore, this Rule-Based Parameter is an example of an *evaluate once* parameter. Such use of a Rule-Based Parameter requires that one, or more, phases have been specified. Here, two phases were added: one corresponding to the initialisation stages and a second corresponding to the submitting of the index details to the job server. The value for the compressed depth was chosen to be evaluated at the start of the second phase: at this point all necessary information is available, and the value of the compressed depth is about to be accessed for the first time.

Overall, two new editable parameters and one non-editable parameter were added to the list of shared parameters, with the compressed depth parameter being changed from an editable Fixed-Choice Parameter to a (non-editable) Rule-Based Parameter. Although it may seem that these changes complicate, rather than simplify, the process of tuning the index (which is, after all, the aim of using a Rule-Based Parameter), it is expected that the parameters presented will be more readily understood by a potential client than the single parameter they replace. Answering two simple 'yes' or 'no' questions about index usage is likely to be straightforward, whereas selecting the most appropriate value for the compressed depth would require a greater understanding of the design of the TCST. Thus, the detailed knowledge of how the compressed depth impacts upon performance (as discussed in Chapter 4) can be more readily exploited, without requiring the client of the index to have knowledge of the index's design.

6.2.2 Summary

The parameters discussed in this section are mainly those that the user will have to provide a suitable value for prior to initiating index construction. Access to these values is then required by a variety of components in the index implementation, thus the use of shared parameters is appropriate. It was shown that both Simple Parameters and Fixed-Choice Parameters can be appropriate for representing tunable operational parameters: in some cases it is preferential to limit the choice of values (hence the use of Fixed-Choice Parameters), however this is not always possible (therefore, Simple Parameters may be required). It can also be seen that the use of Rule-Based Parameters allows an appropriate value for a parameter to be identified without requiring significant input from the user—even when deriving the correct value is non-trivial. It is expected that a wide variety of applications are likely to have at least basic configuration needs that can be served by the aspects of the GIDOF parameter model used here.

6.3 Local Parameters within the TCST

The second set of parameters to be discussed are those that were classified as local parameters. Such parameters only affect specific aspects of an application's performance, and it is not expected that a client would provide values for them all. However, access to these parameters is of importance if thorough tuning of the application is to be undertaken.[4]

6.3.1 Group Hierarchy

Local parameters are grouped according to their purpose. This simplifies the task of tuning a particular aspect of index performance: only parameters that belong to the group, or groups, affecting that area of performance need to be explored. Figure 6.1 illustrates the group hierarchy used for the parameter specification of the TCST (note that all such hierarchies have the group 'local parameters' at the top-most level). The two principal groups used here correspond to the two main stages in the lifetime of the index, namely, constructing the index and querying the index. Additionally, both of these groups have sub-groups. In the case of the construction group, the two sub-groups correspond to the two phases of the Improved Prefix-Partitioned TCST Construction Algorithm, i.e. suffix grouping and TCST construction (see Section 3.5.2). If the suffix-

[4]Such tuning may be undertaken as part of the development process or immediately prior to index deployment.

grouping phase is not used (i.e prefix-partitioned construction is favoured) then the parameters relating to suffix grouping are simply ignored. For the group corresponding to the query server, only one sub-group is present, representing the parameters that are specific to the processing of the result set.

An important point to note is that the groups have been chosen to reflect index use, and do not necessarily correspond to the structure of the index's implementation. For example, the component implementing the two-level array of the TCST has several operational parameters, some of which affect construction performance, with the remainder affecting query performance. Within the group hierarchy, the parameters used by the two-level array component are split across both of the main groups, as it is expected that this is how they would be accessed when tuning the index.

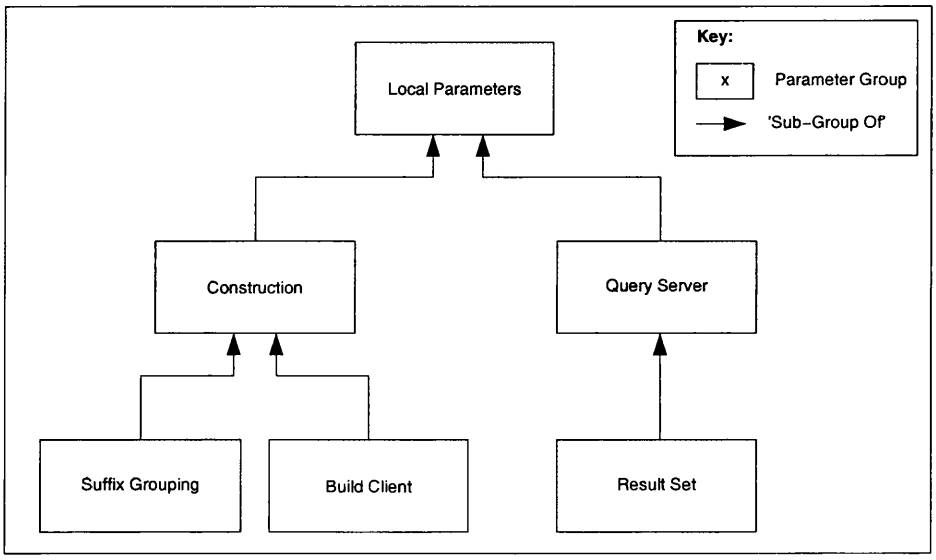

Figure 6.1: The hierarchy of groups for local parameters within the TCST.

6.3.2 Parameters

Taking each parameter group in turn, an overview of the local parameters associated with the implementation of the TCST is now given. Note that, each local parameter is given a suitable default value (derived through preliminary experimentation), thus tuning these parameters is optional for the user of the TCST.

Only one parameter is present in the parameter group 'construction' (the remaining local parameters that affect construction performance are located in the two sub-groups

'Suffix Grouping' and 'Build Client'). This parameter determines which version of the construction algorithm is to be used, and is represented as a Fixed-Choice Parameter of type String. The three possible values correspond to the use of the Prefix-Partitioned construction algorithm, the Improved Prefix-Partitioned construction algorithm (without storing prefix codes) and the Improved Prefix-Partitioned construction algorithm (storing prefix codes). The respective values are "skip", "false" and "true." This parameter is used by the create tree application, which will then instruct the sub-tree builder applications to use the specified algorithm.

The parameter group 'Suffix Grouping' contains the parameters that are specific to the implementation of the suffix-grouping algorithm (this algorithm is implemented in the create tree application). All are represented as Fixed-Choice Parameters of type Integer, and a range of suitable values is provided in each case. These parameters represent the following properties: the size of the buffers used when reading the sequence file and when writing out the grouped suffix lists; the capacity of each node of the linked lists representing the grouped suffix lists; and finally, the threshold at which the partial suffix lists should be written to disk (expressed as a fraction of the total available main memory). The other sub-group of 'construction' is the parameter group 'Build Client,' which represents the parameters that influence the performance of the distributed build clients. The first three of these parameters all correspond to buffer sizes and are represented as Fixed-Choice Parameters of type Integer. These three buffers are used, respectively, when writing completed sub-trees to disk, when writing a backbone section to disk, and finally, when reading the grouped suffix lists (Improved Prefix-Partitioned construction only). The final two parameters in this group correspond to the default number of build threads to be used by each distributed build client[5] and to the maximum interval between garbage collection cycles (see Section 3.9.2). Both are represented as Simple Parameters of type Integer. Table 6.4 summarises the specification of the local parameters that affect construction performance.

The final set of local parameters to be discussed are those that affect query performance. The first two parameters in the group 'Query Server' affect the nature of index eviction, corresponding respectively to the threshold at which eviction will take place (expressed as a fraction of the total available main memory) and the amount of the index to be evicted during each eviction cycle (expressed as a fraction of the total prefix-range). Both are represented as Fixed-Choice Parameters, and are of types Double and Integer, respectively. The final parameter in this group gives the size of the buffer to

[5]In order to allow different sub-tree builders to use different numbers of threads, the parameter giving the default number of threads can be overridden when invoking the application.

Parameter Name	Group	Type	Parameter Style
constructionAlgorithm	Construction	String	Fixed-Choice
prefixBufferSize	Suffix Grouping	Integer	Fixed-Choice
evictionThreshold	Suffix Grouping	Integer	Fixed-Choice
writeBufferSize	Suffix Grouping	Integer	Fixed-Choice
bucketBlockSize	Suffix Grouping	Integer	Fixed-Choice
numberOfThreads	Build Client	Integer	Simple Parameter
garbageCollectionInterval	Build Client	Integer	Simple Parameter
suffixesBufferSize	Build Client	Integer	Fixed-Choice
backBoneWriteBuffer	Build Client	Integer	Fixed-Choice
subTreeWriteBuffer	Build Client	Integer	Fixed-Choice

Table 6.4: Local parameters affecting construction performance of the TCST.

be used when reading a given sub-tree from disk. It is represented as a Fixed-Choice Parameter of type Integer. Two parameters are present in the 'Result Set' sub-group, and correspond to the capacity of the nodes used in the linked lists that store result sets (the capacity of the first node is smaller than other nodes as this was found to improve performance due to a large number of queries only giving a small number of results). Both are represented as Fixed-Choice Parameters of type Integer. Table 6.5 summarises the specification of the local parameters that affect query performance.

Parameter Name	Group	Type	Parameter Style
evictionThreshold	Query Server	Double	Fixed-Choice
evictionFraction	Query Server	Integer	Fixed-Choice
readBufferSize	Query Server	Integer	Fixed-Choice
firstBlockSize	Result Set	Integer	Fixed-Choice
blockSize	Result Set	Integer	Fixed-Choice

Table 6.5: Local parameters affecting query performance of the TCST.

Overall, the complete set of local parameters consists of the following: the sizes of six I/O buffers; the capacities of three types of linked-list nodes; three properties relating to memory management; the default number of threads to use during construction; and finally, the choice of construction algorithm. Although the default values given for these parameters are likely to provide adequate levels of performance, each parameter does impact upon the behaviour of the index and access to them is important if tuning is to take place.

6.3.3 Automating Parameter Choice

The only parameter discussed in the previous section that is a candidate for representation as a Rule-Based Parameter is the choice of construction algorithm. In Section 4.4.4, it was shown that Improved Prefix-Partitioned TCST construction was quicker than Prefix-Partitioned construction when a large number of partitions were used (the number of partitions required being determined by the length of the indexed sequence and the amount of available main memory). However, use of this algorithm is at the expense of disk space (albeit, temporarily), thus, if the user wishes to conserve disk space this algorithm should not employed. In this instance, use of a Rule-Based Parameter does not require that any additional parameters be specified: parameters representing both sequence length and whether or not disk space usage should be minimised are already present in the list of shared parameters.

Table 6.6 lists the conditions and actions associated with the Rule-Based Parameter corresponding to the choice of construction algorithm. Two rules are sufficient to cover all possibilities. The first condition will evaluate to true if it has been specified that disk-space usage should be minimised or if the sequence length is less than 900 Mbp, i.e. the Prefix-Partitioned algorithm should be used when disk space is at a premium or when the sequence is not long enough to justify use of the Improved Prefix-Partitioned algorithm. The second rule simply states that the Improved Prefix-Partitioned algorithm is to be used when the sequence length is long enough to justify it. In both cases, the action is simply a string that will indicate to the application which algorithm is to be used. As was the case with the compressed depth, this parameter is evaluated only once, and requires access to the value of the sequence length parameter; hence, this Rule-Based Parameter is evaluated at the start of the second phase (during which, index construction will be initiated).

Condition	Action
`minimiseDiskSpace || seqLength < 900000000`	`"skip"`
`seqLength >= 900000000`	`"true"`

Table 6.6: Conditions and actions defining the choice of construction style when represented as a Rule-Based Parameter.

6.3.4 Summary

The parameters discussed in this section all affect localised aspects of the performance of the TCST implementation, and represent values that a client of the index may

wish to alter, but do not need to do so in order to use the index. It was shown how parameter groups can be used to categorise the local parameters: parameters were grouped according to the aspects of index performance that they affect, thus allowing parameters of interest to be more readily identified when tuning the TCST implementation. Again, it was seen that both Fixed-Choice Parameters and Simple Parameters can be useful, although Fixed-Choice Parameters are preferable as they restrict the range values that can be used. Another example of exploiting the results of the extensive performance measurements presented in Chapter 4 was given; where the parameter representing the choice of construction algorithm was specified in such a way that its value can be determined automatically. Of the remaining local parameters, it may be possible to use Rule-Based Parameters to automatically select appropriate values (for example, increasing buffer sizes when more main-memory is available), but this would require further investigation into how they affect index performance and it may be that suitable trends cannot be identified.

6.4 Summary

This chapter has shown how many of the features of the GIDOF parameter model were used when providing a complete listing of the operational parameters affecting the performance of the TCST. This parameter list covers some twenty-five parameters, of which twenty-two can be tuned by a client of the TCST implementation. Given such a large number of parameters, it is important that the list of parameters is presented to the user in a structured manner if a client of the index is to locate parameters of interest. Parameters fundamental to the operation of the index were included in the list of shared parameters, with parameters deemed to be of less importance listed as local parameters (which were then further classified according to the area of performance that they impact upon). Where possible, experimental evidence was used to automate the selection of appropriate values for operational parameters, thus allowing some of performance trends identified in Chapter 4 to be exploited. The remaining parameters were simply made available for tuning by a user of the TCST implementation, the values of which can be accessed by using the toolkit described in the previous chapter.

Chapter 7

Conclusions and Future Work

This final chapter summarises the main contributions made within this thesis, demonstrating how the proposals given in the initial hypothesis have been addressed. Furthermore, areas of future research that could further the work presented are discussed.

7.1 Summary of Contributions

The primary contributions of this thesis are the provision of two novel technologies: the Top-Compressed Suffix Tree and the Generic Index Development and Operation Framework (GIDOF). These two technologies are complementary: the performance of the implementation of the TCST is dependent on the values of several operational parameters, with GIDOF providing a framework and toolkit for the management of such parameters.

The Top-Compressed Suffix Tree was introduced in Chapter 3. This data structure was designed as a scalable disk-resident index for sequence data and extends previous work on persistent suffix-based indexes. Such indexes allow greater volumes of data to be indexed than is possible with main-memory only techniques. Several techniques were given for constructing TCSTs, including the Improved Prefix-Partitioned TCST Construction Algorithm, which can better the performance of previously reported partitioned suffix tree construction algorithms. Techniques for parallel TCST construction were given, along with a description of how to query a disk-resident TCST.

Extensive measurements profiling various aspects of the performance of the TCST were discussed in Chapter 4, producing several notable results. It was found that the TCST can provide a more compact on-disk index than rival suffix tree implementations, occupying an average of only 8.17 bytes per character indexed. When large numbers of

partitions are used, the improved prefix-partitioned algorithm was found to out-perform prefix-partitioned construction, thus allowing partitioned indexes to be created in less time. Furthermore, distributed index construction can further reduce the time taken to create a given index by up to 48%. As an in-memory index, it was found that the TCST could be both constructed and queried more efficiently than traditional suffix tree implementations. Thus, it was shown that the TCST is a viable structure for indexing biological sequence data.

It was seen that the operational parameters associated with the implementation of the TCST impact greatly on the observed performance of the index. Thus, in Chapter 5 the GIDOF framework and toolkit were introduced to support some of the tasks associated with parameter management. Chapter 6 then demonstrated how the GIDOF parameter model was used with the TCST, showing how a framework such as GIDOF can be used in the specification of a tunable index. GIDOF introduces a number of ways of representing and classifying operational parameters, including rule-based parameters with values that are derived at run-time. GIDOF allows parameter specifications to be constructed iteratively, thus the parameter specification can be adapted as development and subsequent evaluation progress. The completed parameter specification can then be used to tune a given index for a particular workload. The design of GIDOF is such that the processes of parameter specification and tuning are largely independent of the application source code. This separation of the parameters from the implementation allowed the tasks associated with parameter management to be treated separately from application implementation; this is reflected in the design of the GIDOF toolkit.

7.2 Future Work

Several unexplored areas that could further the research presented in this thesis have been identified. These areas can be split into three categories. The first category consists of ways to extend the functionality of the implementations of the concepts presented here. The second category consists of areas that would benefit from additional evaluation. The final category is that of areas of, potentially substantial, future research. Such areas of future work are now discussed for the main concepts introduced in this thesis.

7.2.1 Extending the TCST Implementation

The implementation of the Top-Compressed Suffix Tree, although complete, could be extended in the ways described below.

64-bit Implementation The implementation of the TCST discussed here makes use of 32-bit values for both references and integers, limiting the size of sequence that can be indexed to 2 Gbp. In order to index larger sequences it would be necessary to change each index into the sequence (i.e. the left label of each node) to be 64-bit. This would allow all realistic sequences to be indexed, but at the cost of a significant increase in the size of both the in-memory and on-disk representation of the TCST. In turn, this would necessitate the use of a larger main memory object heap and require the use of 64-bit references (pointers).[1] Such changes would remove the 2 Gbp limit and, given a suitable amount of main memory, allow multi-gigabyte sequences to be indexed. Additionally, when indexing such large sequences it may be desired that a higher value for the compressed depth of the TCST be used, consequently this may result in a need for the indices to the two-level array to be 64-bit.

Optimising Disk Accesses All access to disk required by the TCST implementation was achieved using the standard I/O facilities of the Java language. Although this proved to be adequate for the purposes of our evaluation, it is likely that the performance of persistent indexes could be significantly improved by using alternative methods of accessing the disk-resident data. For example, replacing the language provided methods with *native methods*[2] might improve performance; as such methods allow more direct access to devices such as disks. An alternative approach would be to make use of an established platform for providing efficient access to large quantities of disk-resident data. This could be achieved by re-engineering the TCST in such a way that it can be implemented using a technology similar to the DataBlade framework or by storing sections of the index as ordinary data within a highly-tuned relational database, as demonstrated by Cooper et al. [32] (see Section 2.2.1).

Provision of a Query Server Providing a query server, accessible via any suitable network protocol, would serve two purposes. Firstly, it would allow the TCST indexing

[1] Such implementations of the Java Virtual Machine are currently available on some platforms, with more widespread support expected with the upcoming release of version 1.5.0 of the Java Runtime Environment (see `http://java.sun.com/`).

[2] The Java language provides the *Java Native Interface (JNI)* for allowing Java programs to invoke native methods (see `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`).

technology to be exploited more readily by those working with biological sequence data. Secondly, such a server would allow usage statistics to be gathered as to the nature of queries most commonly used, thus furthering the evaluation of the performance of the TCST. This data could, when combined with observations made in this thesis, be used to further tune the index implementation and, where appropriate, allow greater use of self-adjusting parameters. Additionally, a suitable server could be provided over other forms of textual data, for example, protein data or English text—both of which use a larger alphabet than that used with DNA sequences, and may require different tuning of the index.

7.2.2 Support for Approximate Matching Over the TCST

At present, the only forms of query supported by the TCST are variations of the exact matching problem. In order for the TCST to be used as a comprehensive means for indexing biological sequences it is likely that support for a wider variety of query algorithms will be required. In particular, support for some form of approximate matching would be of interest to those working with sequence data.

Applications that support searching DNA sequences to find areas of similarity to a given query (approximate matching) are amongst the most heavily used bioinformatics tools. By *similar*, it is meant that the sequence and query have a local alignment that scores above a certain threshold, given some application specific scoring scheme. However, the use of indexes to accelerate the performance of such queries is still an underdeveloped area of research. The suffix tree and suffix array have often been proposed as a suitable choice of index—it has been demonstrated that suffix based indexes can be used to accelerate the Smith-Waterman algorithm [88, 52]. The Smith-Waterman algorithm [96] is chosen as it is guaranteed to find all optimal scoring alignments. This is in contrast to heuristic based approaches such as BLAST [3, 4], which trades accuracy for performance. Although popular, the performance of BLAST is directly proportional to the size of the database (i.e. the collection of sequences against which the query will be executed) which could be a limiting factor as database sizes continue to grow. We now go on to describe how suffix based indexes can be used to accelerate this task and to discuss the applicability of the Top-Compressed Suffix Tree to this area.

It has been shown previously that suffix trees, and related data structures, can be used to accelerate the Smith-Waterman algorithm [88, 56, 52]. It therefore follows that if the TCST provides efficient support for the index query operations required by the accelerated implementations of the Smith-Waterman algorithm, then the TCST will also be a candidate index for this task. In this section, the functionality of the

TCST is compared to two such index implementations: the Suffix Sequoia [55, 56] and SPLAT (Suffix-tree Powered Local Alignment Tool) [52]. Both of these indexes have been successfully used to accelerate approximate matching, and both have functionality that overlaps with the TCST.

Finding Local Alignments

Locating the local alignments of a given query that score above a certain threshold when compared to a particular sequence can be characterised by two phases [69]. The first phase concerns locating the positions in the sequence where a potential local alignment is found to score above the given threshold, while the second phase concerns the delivery of final verified alignments. It is the acceleration of the first phase that is of greatest interest here. A thorough exploration of the alignment phase is given by Kent [69].

The Smith-Waterman algorithm [96] calculates a matrix of scores for all potential alignments between the sequences and the query using a set of recurrence relations together with dynamic programming. Naïvely evaluating this $l \times n$ matrix (where l is the length of the query and n is the size of the sequence) has a time complexity of $O(ln)$, and will quickly become impractical as sequence length grows. However, it is possible to reduce the amount of the matrix that is evaluated through optimisations such as those proposed by Myers and Durbin [86]. This technique made use of an index over the query and resulted in an algorithm that only required 4% of the matrix coefficients to be evaluated. One limiting factor of this technique is that the entire sequence must be scanned in order to execute every query. By employing a suitable index over the sequence, it is possible to further reduce the amount of the matrix that is evaluated in addition to removing the need to scan the entire sequence.

Suffix Sequoia

The *Suffix Sequoia* [55, 56] is an example of an index that can be employed to accelerate the first phase of finding local alignments. This data structure is of particular interest here as it uses the same prefix-coding scheme as the TCST, and all algorithms that operate over the sequoia can be used with the TCST. The suffix sequoia provides an index over the c-character prefixes of each of the suffixes of a given string (the remainder of each suffix is not indexed). The index consists of the three arrays described below (where a corresponds to the alphabet size, c is the size of the prefix and n is the length of the sequence):

- A bitmap of size a^c where each element is set to true if the corresponding c-

character prefix is present in the text, and set to false otherwise.

- An integer array of size a^c, with each entry corresponding to the location in the third array where a list of occurrences of the prefix can be found in the text.
- A final array consisting solely of the lists of occurrences of each c-character prefix pattern, this will be of size n and stored on disk.

In order to use the sequoia to accelerate approximate matching, it is necessary that the query be viewed as a series of (overlapping) c-character strings. The alignment process will be repeated using each c-character string obtained from the query. The data structure will then be used to retrieve the positions in the sequences where a c-character string present in one or more sequences is found to score above the stated threshold. It is claimed that using a value of c equal to 5 that the upper bound on the percentage of the matrix that must be evaluated is 1.6% [55].

The operations which an index must support in order to make use of this algorithm are that it can efficiently establish if a given c-character string occurs within the sequence and, if the c-character string does occur, that it can retrieve a list of positions where it appears. Additionally, the sequoia makes use of a lexicographical ordering to reduce the amount of the matrix that is evaluated. We now contrast the support for each of these requirements as provided by the TCST and by the sequoia.

Testing for String Presence Testing for the presence of c-character string using the sequoia is achieved by testing the boolean value stored at the relevant location in the bitmap. The same test can be achieved using the TCST by testing the relevant entry in the two-level array (during which, two arrays are accessed and two non-consecutive integers may need to be read from disk). Although it may seem that this operation will be slower using the TCST, the ability of the two-level array to compactly represent a sparsely populated array[3] will reduce this overhead. This will allow large un-populated areas of the index to be processed efficiently, particularly when operating over the index in lexicographical order. Additionally, the caching and eviction mechanisms used with the TCST could be suitably tuned to favour the retention of ribs (as opposed to suffix sub-trees), which would reduce (if not completely remove) the need to access the on-disk index in order to support this part of the algorithm.

Retrieving the List of Occurrences Using the sequoia, the list of occurrences of a particular c-character string is accessed by retrieving the file offset from the second

[3]Hunt comments on the need to support efficient operation over sparsely populated indexes [55].

array, and reading the list of occurrences from the relevant file. Using the TCST, it will be necessary to traverse the associated suffix sub-tree (faulting as necessary) in order to extract the positions. In both cases, this is an $O(o)$ operation (where o is the number of occurrences), with the performance of the TCST expected to be marginally slower due to the added overhead of reading in and traversing the tree.

Lexicographical Ordering Each potential c-character code is compared with the query string in lexicographical order: this minimises the amount of the matrix calculated as only the entries that change between two successive codes need to be computed. This can easily be replicated with the TCST, as accessing each entry in the two-level array in order is equivalent to traversing the index in lexicographical order.

SPLAT: Suffix Tree Powered Local Alignment Tool

An alternative approach to implementing approximate matching over the TCST would be to build upon the techniques described by Harding and Atkinson [52]. Harding and Atkinson describe SPLAT (Suffix-tree Powered Local Alignment Tool), a highly optimised version of the Smith-Waterman algorithm that has been accelerated through the use of a suffix tree index and through a number of optimisations that allow matrix calculations to be terminated early without missing any possible alignments. As with the algorithm described in the previous section, the index is used firstly to confirm the presence of a particular string that is of interest, and secondly to retrieve the positions of a particular occurrence. Within SPLAT, the suffix tree is accessed in a depth-first manner, with traversal only continuing deeper into the tree if the section of the matrix currently being calculated has the potential to score above the given threshold.

To use this algorithm with the TCST it is necessary to provide a depth-first traversal over the TCST. By scanning the two-level array from lowest entry to the highest, the sub-trees will be encountered in the order that they would be found during a depth-first traversal. As each sub-tree is associated with a c-character prefix, it is necessary to know the alignment score for the prefix before continuing (or abandoning) this possible alignment. This approach may limit the use of some of Harding and Atkinson's optimisations: when using a depth-first traversal starting at the root of the tree it is possible in some cases to abandon a low scoring alignment early, whereas using the TCST the calculation of the alignment score for the first c characters will always be completed. Whether or not this has a significant impact upon performance is yet to be explored. Other possibilities would include combining Harding's approach with the techniques associated with the suffix sequoia or possibly simulating a full depth-first

traversal by a carefully constructed scan of the two-level array.

Summary

The overlap in functionality of the TCST and the suffix sequoia suggests that the accelerations in the calculation of the dynamic programming matrix demonstrated with the sequoia are equally applicable to the TCST. While the sequoia has been designed solely to accelerate approximate matching algorithms, the TCST has been designed to be a more versatile data structure that supports a wider variety of algorithms (for example, the sequoia does not support exact matching as it only indexes fixed size windows over the text). However, it still provides an efficient implementation of the methods required by the approximate matching algorithm described by Hunt [55]. During approximate matching, it is the top of the index that is most commonly used, and this is exactly the area of the TCST that is likely to be resident in main memory and that benefits from a compressed representation. One potential advantage of the TCST over the sequoia is that the TCST does not have a maximum query window size (the sequoia processes each query using a sliding window of size c or less, whereas the TCST has no such limit—the suffix sub-trees can be used to test for the presence of patterns longer than c).

Adapting the SPLAT algorithm for use with the TCST is potentially an attractive solution as this algorithm has been heavily optimised. However, some of the optimisations proposed may not be directly applicable to the TCST and it may be required to provide additional index functionality in order to replicate such features. As stated above, the fact that the TCST aims to retain much of the top of the index in main-memory is likely to be of benefit with SPLAT. In either case, the applicability of approximate matching techniques to the TCST requires in-depth exploration.

7.2.3 Use of GIDOF with Other Indexes

In this thesis, it was shown that GIDOF can be used to manage a variety of parameters associated with the implementation of the TCST. It is expected that this framework is suitable for managing the kinds of parameters commonly associated with performance engineering. Use of the framework with further examples of indexing technology would be of interest, as this may uncover further styles of parameter use that are not directly supported—it is expected that all parameters can be managed with GIDOF, but that some, as yet undocumented, common forms of parameter use may benefit from more specialised support. Of particular interest would be the use of GIDOF by a greater

number of index developers. Such use is the most likely source of the identification of parameter management requirements that are not currently addressed within the framework. Although GIDOF has been designed specifically to support the management of parameters within bespoke index implementations, it is likely that similar technologies may be of interest in other areas of performance engineering, for example, applications that support the visualisation of large amounts of data. Parameter management in such contexts may be addressed through the use of GIDOF, although additional functionality may be required.

7.2.4 Lightweight Persistence for GIDOF

It is intended that parameter management will form only part of GIDOF, and that future revisions of the framework will address other aspects of index implementation and operation. One such example would be complementing the parameter management component with a parameter exploration component. This could then be used to automatically explore different combinations of parameters, establishing, for a given input and task, what combination of parameters resulted in the best performing index. Another component that could be added to GIDOF is support for *lightweight persistence for indexes*. By lightweight persistence, it is meant, generic persistence components that can be used to provide persistence for simple write-once read-many data structures (for example, indexes over reference data). Such a mechanism is termed *lightweight* as it only supports write-once read-many structures and no support is given for updating already disk-resident structures. An approach to implementing such a technology is now given.

The potential for such a technology can be seen by considering the representation of tree-based indexes as multi-layered data structures. With the Top-Compressed Suffix Tree, it was seen that persistence could be achieved by splitting the data structure into two layers, with faulting of sub-trees occurring when crossing from the top layer to the lower layer. A similar pattern was used in the work of Cooper et al., except that their tree was split into multiple layers [32]. By splitting a write-once read-many index into layers it is possible to implement persistence purely in terms of reading and writing sub-trees, with writing only occurring during index construction. Note that each layer can consist of more than one (non-overlapping) sub-tree. Implementing such a system requires the following properties to hold true:

- The system must know when to fault in a required section of the index, i.e. when an algorithm attempts to cross a boundary from part of the structure that is in

main memory to part of the structure that is not, the faulting mechanism must be invoked.

- The system must know what section of the index corresponds to the part that is being faulted, i.e. there must be a relationship between points on a layer boundary and on-disk index sections.
- The system must be able to keep track of what sections of the index are currently held in main memory; the main purpose of this is to support efficient selection of index sections for eviction.
- It is also required that techniques are given to support the transfer of index sections to and from disk, i.e. that the index sections can be marshalled and un-marshalled.

Additional requirements are that such a system would only involve a minimal development overhead, and that the impact on the in-memory performance would be minimal (obviously, faulting sections of index from disk will incur a significant overhead when compared with a main-memory implementation of the same structure). The advantages of such a system would be that an existing main-memory tree can easily be extended to allow its use as an on-disk index. Furthermore, by providing implementations of a variety of different eviction techniques (for example, first-in first-out, least recently used or random) the developer can experiment with different strategies without the development overhead of having to implement each one.

7.3 Concluding Remarks

To conclude, we return to the thesis statement given in Section 1.2.1. It was proposed that the Top-Compressed suffix tree could be used with large volumes of sequence data. In Chapter 4, it was shown that partitioned TCSTs can be created over sequences of up to 1.5 Gbp in length, resulting in an index that was several times larger than that possible with main-memory only techniques. Furthermore, it was then shown that it is possible to query the completed index, with index sections being transferred to (and subsequently evicted from) main memory as necessary, thus confirming that the TCST is a viable data structure for use with large biological sequences. It was then shown that main-memory TCSTs can be created and queried more efficiently than traditional suffix tree representations, thus demonstrating that the TCST is competitive with the suffix tree. The extensive performance measurements given in Chapter 4 demonstrate how

the values used for operational parameters can impact greatly upon index performance. The discussion of the design and use of the GIDOF framework and toolkit given in Chapters 5 and 6 demonstrate how such a framework and toolkit can be used in the implementation of a tunable index, confirming the final aspect of the original proposal; that such frameworks can be used during index development and operation.

Appendix A

Top-Compressed Suffix Tree: Usage Instructions

This appendix describes the steps required to successfully install, compile and use the Java™ implementation of the Top-Compressed Suffix Tree described in Chapters 3 and 4. This implementation makes use of the GIDOF toolkit to allow operational parameters to be tuned to suit the user's needs. A summary of the role of the more important parameters is given here—the role of other parameters can be explored through use of the GIDOF tuning application. For guidance on the use of the GIDOF framework, see Appendix C. The guidelines given here reflect the needs of the software when used with the largest sequence lengths supported. The amount of RAM and disk space required are dependent on the data set being indexed, therefore use of this software with more limited resources may be possible when indexing smaller quantities of data. All the example commands given use a command line with semantics similar to the BASH shell.[1] The exact form of these commands may vary on other platforms.

A.1 System Requirements

- Java 2 Runtime Environment (JRE), Standard Edition, version 1.4.0 (or higher).
- 2 GB of RAM, of which 1880 MB will be allocated to the Java Virtual machine.
- Disk space equal to approximately ten times that of the sequence data to be indexed (additional space will be required if the Improved Prefix-Partitioned TCST

[1] A description of the semantics of the BASH command line environment is available from `http://www.gnu.org/software/bash/bash.html`, as accessed August 2004.

Construction Algorithm is used).

- NFS (or equivalent networked file system), if distributed construction is used.
- Java 2 Software Development Kit (SDK), Standard Edition, version 1.4.0 (or higher), if compilation from source is required.
- Apache Ant version 1.5.1 (or higher) (required for both compilation from source, and for invoking the GIDOF tuning application).

Both the Java 2 Runtime Environment and Software Development Kit are available from `http://java.sun.com/`. Ant is available from `http://ant.apache.org/`. The GIDOF library should be supplied with the software.

A.2 Installation Guide

Prior to compilation, the directory containing the Top-Compressed Suffix Tree software should consist of four items: an Ant build script (`build.xml`), a directory named `libs` containing all necessary `jar` files, a directory containing the source code (simply named `java`) and a file named `properties.xml` (which specifies the operational parameters to be used with the index). It is assumed that both Java and Ant are already installed and configured.

The software can be compiled simply by invoking the command `ant`. This will create a directory named `classes` and compile all the relevant source code, placing the compiled classes in the newly created directory. If Javadoc documentation is desired, this can be obtained by running the command `ant doc`. Finally, in order to run the software, the location of the `classes` directory, and the various `jar` files located in the `libs` directory, must be added to the `CLASSPATH` environmental variable.

The GIDOF tuning application can be invoked by using the following command: `ant TuningWizard`. This application can be used to edit the file `properties.xml` prior to invoking index construction. Note that, at the end of index construction the current properties of the index will be saved in the index output directory, thus if further tuning is required prior to invoking the query server, the file `properties.xml` in the directory containing the persistent index should be edited (fewer parameters will be available for tuning at this stage).

A.3 Data Format

The TCST can be used to index sequence data comprised of elements drawn from any of the following alphabets: *DNA* {A,C,G,N,T}, *RNA* {A,C,G,N,U} or *protein* {A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z}. The data to be indexed should be stored in a single ASCII text file that has been stripped of all formatting (including all white space characters). That is, the file should contain only characters drawn from the chosen alphabet.

A.4 Index Creation

As the Top-Compressed Suffix Tree software developed, several different techniques for index construction were implemented. Thus, this implementation can be used to construct TCSTs in a variety of ways. Firstly, two distinct algorithms are used for the creation of the index sub-trees. These are, prefix-partitioned construction and improved prefix-partitioned construction. The latter should be used when disk-space is ample and a large number of partitions are required to create the index, the former should be used in all other circumstances. In addition, single-threaded, multi-threaded and distributed construction are all available. For simplicity, all of these options make use of the same components, meaning that Java RMI (Remote Method Invocation) is used even when only one CPU is being used for index construction.

A.4.1 TCST Job Server

The Job Server (class `suffixtrees.distr.JobServerImpl`) is the main co-ordinating class in the TCST. This server, which can only co-ordinate one index build at a time, must be running prior to starting the construction of a given index. If using distributed construction, this server must be accessible by all of the distributed clients; otherwise, it can simply be run on the local computer. Communication with the Job Server makes use of RMI; therefore, a remote object registry must be running on the computer hosting the Job Server. This registry can be invoked by the command `rmiregistry`.

Example Commands: Starting the Job Server

```
% rmiregistry &
% java -server suffixtrees.distr.JobServerImpl &
```

A.4.2 Initiating an Index Build

Assuming that the Job Server is already running, creating a TCST requires two programs to be run: a version of the Sub-Tree Builder (which constructs individual partitions of the index) and the `CreateTree` program (which initiates the construction of the index). Each program is now discussed in detail.

Sub-Tree Builder

The Sub-Tree Builder implements two versions of the TCST construction algorithm: the first is the Prefix-Partitioned TCST Construction Algorithm, and the second is the Improved Prefix-Partitioned TCST Construction Algorithm. The use of both algorithms is identical, however the latter expects the output of the suffix grouping to be complete prior to the creation of index partitions. The algorithm invoked is determined by the Create Tree application (see below). The work carried out by the sub-tree builders is controlled by the Job Server, and they can be started before the index is ready to be constructed (i.e. they will wait until details of the index are available). A sub-tree builder can make use of more than one construction thread (giving multi-threaded construction). However, each thread will share a common heap and therefore smaller partitions must be specified if this option is to be used. Distributed construction is achieved simply by starting a sub-tree builder on more than one computer. If desired, multi-threaded and distributed construction can be combined. The two operational parameters of most interest here are the location of the Job Server and how many threads to use (see Table A.1). Values for these parameters can be accessed via GIDOF, or set using the command line (which will override any GIDOF settings).

Parameter Name	Purpose
jobServerHostName	The hostname of the server where the Job Server is running (can be set to localhost).
numberOfThreads	The number of construction threads to use.

Table A.1: Usage of command-line arguments for Sub-Tree Builder.

Create Tree

The final stage in initiating the construction of a Top-Compressed Suffix Tree is to invoke the class `suffixtrees.topcompressed.CreateTree`. The primary purpose of this class is to co-ordinate index construction and ensure that all details of the index to

be constructed are related to the Job Server. In turn, the Job Server will supply this information to the sub-tree builders, but only when asked to do so by the Create Tree program. If the Improved Prefix-Partitioned TCST Construction Algorithm is being used, this class will also invoke the suffix-grouping algorithm. The suffix-grouping algorithm can be used in two different ways: storing the prefix code along with each suffix, or omitting the prefix codes. Storing the prefix codes requires additional disk space, but can give improved performance. The version of the grouping algorithm to be used (or if grouping should be skipped entirely) is specified as a command-line argument. Details of all the operational parameters used by this class are given in Table A.2. The first five parameters listed can be specified either using the command line or via GIDOF, with the final three parameters being accessed via GIDOF.

Parameter Name	Purpose
jobServerHostName	The hostname of the server where the Job Server is running (can be set to localhost).
seqFile	Location of the sequence file to be indexed.
alphabet	Alphabet form which the current sequence is composed. Must be one of {`DNA`,`RNA`,`protein`}.
destinationDir	Where the completed index will be stored.
numberOfPartitions	The number of partitions to use.
compressedDepth	The size of the compressed depth.
ribSize	The size of the ribs in the two-level array.
constructionAlgorithm	Determines which version of the suffix grouper should be used (or if it should be skipped). Must be one of {`skip`,`false`,`true`}.

Table A.2: Usage of command-line arguments for Create Tree.

Example Commands: Starting Index Construction

```
% java -server -Xmx1880m -Xms1400m -Xss1024k \
       suffixtrees.topcompressed.SubTreeBuilder kona &
% java -server -Xmx400m suffixtrees.topcompressed.CreateTree \
       kona ~/SeqIdx/Fruit_Fly.seq DNA ~/SeqIdx/FFIndex/ 26
```

A.4.3 JVM Options

In addition to the command-line arguments specified above, a number of arguments must be supplied to the Java Virtual Machine[2] in order to achieve the best possible performance. These parameters are specific to a given JVM implementation; therefore, the documentation for the chosen JVM should be consulted prior to using these options. Each of the JVM options used in the examples given is described in Table A.3.

JVM Argument	Purpose
`-server`	Signals that the Server variant of the virtual machine is to be used.
`-Xms`	Sets the minimum heap size to the value specified.
`-Xmx`	Sets the maximum heap size to the value specified.
`-Xss`	Sets the thread stack size to the value specified.

Table A.3: JVM command-line arguments.

A.5 Query Execution

A simple program for performing exact matches is supplied with the TCST. It is expected that this class, `suffixtrees.topcompressed.QueryEngine`, will be used in conjunction with client specific technology (i.e. that it be used as the basis for providing a query server). However, it can also be used as a stand-alone tool for performing batches of queries. The target patterns should be contained in a single file, one per line, and are subject to the same formatting constraints as the original sequence data. Table A.4 describes the parameters used with this class. Additional parameters can be accessed via GIDOF, with some parameters having had their values fixed during index construction. Note that the list of queries to be performed is always specified as a command-line argument.

Example Commands: Querying an Index

```
% java -server -Xmx1880m -Xms1400m -Xss1024k \
       suffixtrees.topcompressed.SubTreeBuilder \
       ~/SeqIdx/FFIndex/ ~/SeqIdx/Fruit_Fly.seq DNA \
       ~/SeqIdx/FlyQueries.qrs
```

[2]In the examples given, all JVM arguments are preceded by a single dash.

Parameter Name	Purpose
destinationDir	Location where the completed index is stored.
seqFile	Location of the indexed sequence file.
alphabet	Alphabet form which the current sequence is composed. Must be one of {`DNA`,`RNA`,`protein`}.
queryList	Location of the file containing the queries to be executed.

Table A.4: Usage of command-line arguments for Query Engine.

Appendix B

GIDOF: XML Parameter Model

B.1 XML Schema

```
<?xml version='1.0'  encoding='UTF-8' ?>
<!DOCTYPE  xs:schema PUBLIC '-//W3C//DTD XMLSCHEMA 200102//EN'
 'http://www.w3.org/2001/XMLSchema.dtd'   >
<!--Copyright: 1.0-->
<!--Title: GIDOF: Generic Index Development and Operation Framework-->
<!--Version: -->
<!--Author: Robert Japp-->
<!--Project Description: A framework and toolkit for the management of
    Operational Parameters.-->
<!--Document Description: GIDOF parameter model schema.-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="GIDOF"
           targetNamespace="GIDOF"
           elementFormDefault="qualified">

   <!--The parameter list-->
   <xs:element name="properties">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="phaseList"/>
            <xs:element ref="sharedParameters"/>
            <xs:element ref="localParameters"/>
         </xs:sequence>
      </xs:complexType>
```

```
</xs:element>

<!--The List of Phases-->
<xs:element name="phaseList">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="phase"
                     minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="name" type="xs:string"
                             use="required"/>
               <xs:attribute name="donePre" type="xs:boolean"/>
               <xs:attribute name="donePost" type="xs:boolean"/>
               <xs:attribute name="number" type="xs:integer"
                             use="required"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!--The Shared Property List-->
<xs:element name="sharedParameters">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="property"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!--Root element for the group hierarchy-->
<xs:element name="localParameters">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="group"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!--A Group element, contains a list of properties and sub-groups-->
<xs:element name="group">
   <xs:complexType>
```

```
      <xs:sequence>
         <xs:element ref="property"
          minOccurs="0" maxOccurs="unbounded"/>
         <xs:element ref="group"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>

<!--Java types supported by GIDOF-->
<xs:simpleType name="supportedTypes">
   <xs:restriction base="xs:string">
     <xs:enumeration value="java.lang.String"/>
     <xs:enumeration value="java.lang.Character"/>
     <xs:enumeration value="java.lang.Boolean"/>
     <xs:enumeration value="java.lang.Byte"/>
     <xs:enumeration value="java.lang.Short"/>
     <xs:enumeration value="java.lang.Integer"/>
     <xs:enumeration value="java.lang.Long"/>
     <xs:enumeration value="java.lang.Float"/>
     <xs:enumeration value="java.lang.Double"/>
   </xs:restriction>
</xs:simpleType>

<!--Parameter User Modes-->
<xs:simpleType name="userMode">
   <xs:restriction base="xs:string">
     <xs:enumeration value="User Editable"/>
     <xs:enumeration value="User Defined"/>
     <xs:enumeration value="Not User Editable"/>
   </xs:restriction>
</xs:simpleType>

<!--Property element (includes generic parameter attributes) -->
<xs:element name="property">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="description" type="xs:string"/>
         <xs:choice>
            <xs:element ref="simpleParameter"/>
            <xs:element ref="fixedChoiceParameter"/>
            <xs:element ref="ruleBasedParameter"/>
         </xs:choice>
```

```
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
                    use="required"/>
      <xs:attribute name="userMode" type="userMode"
                    use="required"/>
      <xs:attribute name="valueType" type="supportedTypes"
                    use="required"/>
      <xs:attribute name="final" type="xs:boolean"
                    use="required"/>
   </xs:complexType>
</xs:element>

<!--Representation of Simple Parameter-->
<xs:element name="simpleParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="value" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!--Representation of Fixed-Choice Parameter-->
<xs:element name="fixedChoiceParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="choices">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="choice"
                              minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="value"
                                       type="xs:string" />
                        </xs:sequence>
                        <xs:attribute name="number"
                                      type="xs:integer"
                                      use="required"/>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
               <xs:attribute name="currentChoice"
                             type="xs:integer"
                             use="required"/>
```

```
                <xs:attribute name="numberOfChoices"
                              type="xs:integer"
                              use="required"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!--Representation of Rule-Based Parameter-->
<xs:element name="ruleBasedParameter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="rules">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="condition"
                              minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="expression"
                                       type="xs:string" />
                        </xs:sequence>
                        <xs:attribute name="number"
                                      type="xs:integer"
                                      use="required"/>
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="action"
                              minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="expression"
                                       type="xs:string" />
                        </xs:sequence>
                        <xs:attribute name="number"
                                      type="xs:integer"
                                      use="required"/>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
               <xs:attribute name="numberOfRules" type="xs:integer"
                             use="required"/>
            </xs:complexType>
```

```
            </xs:element>
            <xs:choice>
               <xs:element name="onDemand" />
               <xs:element name="evaluateOnce">
                  <xs:complexType>
                     <xs:attribute name="phaseName"
                                   type="xs:string" use="required"/>
                     <xs:attribute name="isPre" type="xs:boolean"
                                   use="required"/>
                  </xs:complexType>
               </xs:element>
               <xs:element name="selfAdjusting">
                  <xs:complexType>
                     <xs:attribute name="startPhase"
                                   type="xs:string" use="required"/>
                     <xs:attribute name="isStartPre"
                                   type="xs:boolean" use="required"/>
                     <xs:attribute name="endPhase" type="xs:string"
                                   use="required"/>
                     <xs:attribute name="isEndPre" type="xs:boolean"
                                   use="required"/>
                  </xs:complexType>
               </xs:element>
            </xs:choice>
            <xs:element name="currValue" type="xs:string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

</xs:schema>
```

B.2 TCST Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<properties xmlns="GIDOF">
  <phaseList>
    <phase donePost="false" donePre="false" name="initialisation"
           number="0"/>
    <phase donePost="false" donePre="false" name="initiateConstruction"
           number="1"/>
  </phaseList>

  <sharedParameters>
    <property final="false" name="numberOfPartitions"
              userMode="User Defined" valueType="java.lang.Integer">
      <description>The number of partitions to be used when
      constructing the index.  For larger sequences it will be
      necessary to increase the number of partitions used in order to
      achieve successful index construction.  Use of multi-threaded
      construction will require more partitions.</description>
      <simpleParameter> <value>1</value> </simpleParameter>
    </property>
    <property final="false" name="destinationDir"
              userMode="User Defined" valueType="java.lang.String">
      <description>The directory where the completed index (and
      intermediary output) is to be stored.  Should be in a location
      accessible to all build clients.</description>
      <simpleParameter> <value>null</value> </simpleParameter>
    </property>
    <property final="false" name="smallExacts"
              userMode="User Editable" valueType="java.lang.Boolean">
      <description>If set to true, the index will be tuned to improve
      the performance of executing exact matching using short query
      performance.  By short, it is meant query strings of length less
      than 8.  If set to false, the index will be tuned to suit a
      greater variety of query type.</description>
      <fixedChoiceParameter>
        <choices currentChoice="1" numberOfChoices="2">
          <choice number="0"> <value>true</value> </choice>
          <choice number="1"> <value>false</value> </choice>
        </choices>
      </fixedChoiceParameter>
    </property>
    <property final="false" name="alphabet"
              userMode="User Editable" valueType="java.lang.String">
```

```
    <description>The alphabet from which the sequence file is
    constructed.  The supported alphabets are: DNA {A,C,G,N,T}, RNA
    {A,C,G,N,U} and protein
    {A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z}.</description>
    <fixedChoiceParameter>
      <choices currentChoice="0" numberOfChoices="3">
        <choice number="0"> <value>DNA</value> </choice>
        <choice number="1"> <value>RNA</value> </choice>
        <choice number="2"> <value>protein</value> </choice>
      </choices>
    </fixedChoiceParameter>
  </property>
  <property final="false" name="ribSize"
            userMode="User Editable" valueType="java.lang.Integer">
    <description>The size of the ribs to be used in the two-level
    array component of the Top-Compressed Suffix Tree.  Altering
    this value may give a slight improvement in query and
    construction performance.</description>
    <fixedChoiceParameter>
      <choices currentChoice="1" numberOfChoices="2">
        <choice number="1"> <value>32</value> </choice>
        <choice number="2"> <value>64</value> </choice>
      </choices>
    </fixedChoiceParameter>
  </property>
  <property final="false" name="minimiseDiskSpace"
            userMode="User Editable" valueType="java.lang.Boolean">
    <description>If set to true, the amount of disk space used by
    the index will be minimised.  This may reduce the performance of
    index creation and querying.  If set to false, more disk space
    may be used by the index, but this can allow greater
    performance.</description>
    <fixedChoiceParameter>
      <choices currentChoice="1" numberOfChoices="2">
        <choice number="0"> <value>true</value> </choice>
        <choice number="1"> <value>false</value> </choice>
      </choices>
    </fixedChoiceParameter>
  </property>
  <property final="false" name="compressedDepth"
            userMode="Not User Editable"
            valueType="java.lang.Integer">
    <description>The compressed depth to be used with the
    Top-Compressed Suffix Tree.  Lowering this value can decrease
```

```
    the amount of space occupied by the on-disk index, however this
    can be to the detriment of both construction and query
    performance.  If using a persistent index, and performing mainly
    short exact matches, choosing a lower value for the compressed
    depth may improve performance.</description>
    <ruleBasedParameter>
      <rules numberOfRules="7">
        <condition number="0">
          <expression>smallExacts</expression>
        </condition>
        <condition number="1">
          <expression>!minimiseDiskSpace &&
                       seqLength <= 30000000</expression>
        </condition>
        <condition number="2">
          <expression>!minimiseDiskSpace &&
                      seqLength > 30000000</expression>
        </condition>
        <condition number="3">
          <expression>seqLength <= 25000000</expression>
        </condition>
        <condition number="4">
          <expression>seqLength > 25000000 &&
              seqLength <=300000000</expression>
        </condition>
        <condition number="5">
          <expression>seqLength > 300000000 &&
                      seqLength <=1400000000</expression>
        </condition>
        <condition number="6">
          <expression>seqLength > 1400000000</expression>
        </condition>
        <action number="0"> <expression>8</expression> </action>
        <action number="1"> <expression>10</expression> </action>
        <action number="2"> <expression>12</expression> </action>
        <action number="3"> <expression>8</expression> </action>
        <action number="4"> <expression>9</expression> </action>
        <action number="5"> <expression>10</expression> </action>
        <action number="6"> <expression>12</expression> </action>
      </rules>
      <evaluateOnce isPre="true" phaseName="initiateConstruction"/>
      <currValue>null</currValue>
    </ruleBasedParameter>
  </property>
```

```
    <property final="false" name="seqFile"
              userMode="User Defined" valueType="java.lang.String">
      <description>The file name of the sequence to be indexes (can be
      relative or absolute).</description>
      <simpleParameter> <value>null</value> </simpleParameter>
    </property>
    <property final="false" name="seqLength"
              userMode="Not User Editable"
              valueType="java.lang.Integer">
      <description>Represents the length of the sequence being index.
      This parameter is not user editable, its value will be
      established at run-time from the given details of the sequence
      being index.</description>
      <simpleParameter> <value>null</value> </simpleParameter>
    </property>
    <property final="false" name="jobSeverHostname"
              userMode="User Defined" valueType="java.lang.String">
      <description>The hostname of the server where the JobServer is
      running (can be set to localhost).</description>
      <simpleParameter> <value>null</value> </simpleParameter>
    </property>
  </sharedParameters>

  <localParameters>
    <group name="Local Parameters">
      <group name="Query Server">
        <property final="false" name="evictionThreshold"
                  userMode="User Editable"
                  valueType="java.lang.Double">
          <description>The smallest amount of free memory before
          eviction is triggered.  Decreasing this vaule may result in
          more frequent evictions but may also improve the number of
          cache hits.</description>
          <fixedChoiceParameter>
            <choices currentChoice="1" numberOfChoices="6">
              <choice number="0"> <value>0.02</value> </choice>
              <choice number="1"> <value>0.05</value> </choice>
              <choice number="2"> <value>0.07</value> </choice>
              <choice number="3"> <value>0.1</value> </choice>
              <choice number="4"> <value>0.15</value> </choice>
              <choice number="5"> <value>0.2</value> </choice>
            </choices>
          </fixedChoiceParameter>
        </property>
```

```
<property final="false" name="evictionFraction"
          userMode="User Editable"
          valueType="java.lang.Integer">
  <description>The amount of the index to be evicted whenever
  the amount of available main-memory drops below the
  specified threshold.  This value is the fraction of the
  total index (i.e. 6 -> 1/6).  Decreasing this value may give
  improved cache hit rates, but may also cause more frequent
  evictions (and thus degrade performance).</description>
  <fixedChoiceParameter>
    <choices currentChoice="0" numberOfChoices="6">
      <choice number="0"> <value>6</value> </choice>
      <choice number="1"> <value>8</value> </choice>
      <choice number="2"> <value>10</value> </choice>
      <choice number="3"> <value>12</value> </choice>
      <choice number="4"> <value>14</value> </choice>
      <choice number="5"> <value>16</value> </choice>
    </choices>
  </fixedChoiceParameter>
</property>
<property final="false" name="readBufferSize"
          userMode="User Defined"
          valueType="java.lang.Integer">
  <description>The size of buffer to be used when reading a
  sub-tree of the index.  Increasing this value may improve
  query performance, but at the cost of increased memory usage
  (hence more frequent evictions).</description>
  <fixedChoiceParameter>
    <choices currentChoice="0" numberOfChoices="4">
      <choice number="0"> <value>4096</value> </choice>
      <choice number="1"> <value>8192</value> </choice>
      <choice number="2"> <value>16384</value> </choice>
      <choice number="3"> <value>32768</value> </choice>
    </choices>
  </fixedChoiceParameter>
</property>
<group name="Result Set">
  <property final="false" name="blockSize"
            userMode="User Defined"
            valueType="java.lang.Integer">
    <description>The size of each block in the linked list
    used by the result set.  Increasing this value can give
    improved performance when queries produce a large number
    of results (e.g. when performing exact matching for short
```

```
        query patterns).</description>
        <fixedChoiceParameter>
          <choices currentChoice="0" numberOfChoices="4">
            <choice number="0"> <value>100</value> </choice>
            <choice number="1"> <value>200</value> </choice>
            <choice number="2"> <value>500</value> </choice>
            <choice number="3"> <value>1000</value> </choice>
          </choices>
        </fixedChoiceParameter>
      </property>
      <property final="false" name="firstBlockSize"
                userMode="User Defined"
                valueType="java.lang.Integer">
        <description>The size of the first block in the linked
        list used by the result set.  Increasing this value can
        give improved performance when queries produce a large
        number of results (e.g. when performing exact matching for
        short query patterns).  This value is typically lower than
        the block size as many queries will only produce a
        handfull of results.</description>
        <fixedChoiceParameter>
          <choices currentChoice="1" numberOfChoices="6">
            <choice number="0"> <value>10</value> </choice>
            <choice number="1"> <value>20</value> </choice>
            <choice number="2"> <value>30</value> </choice>
            <choice number="3"> <value>40</value> </choice>
            <choice number="4"> <value>50</value> </choice>
            <choice number="5"> <value>100</value> </choice>
          </choices>
        </fixedChoiceParameter>
      </property>
    </group>
  </group>
  <group name="Construction">
    <property final="false" name="constructionAlgorithm"
              userMode="User Editable"
              valueType="java.lang.String">
      <description>Determines the version of the construction
      algorithm to used with the TCST.  If set to 'skip', then the
      suffix grouping phase is bypassed and the prefix-partitioned
      algorithm is used.  If set to either 'true' or 'false', then
      the improved prefix-partitioned algorithm is used.  'True'
      states that prefix codes should be stored in addition to
      suffix numbers, 'false' states that they should be ignored.
```

```
    </description>
    <ruleBasedParameter>
      <rules numberOfRules="2">
        <condition number="0">
          <expression>minimiseDiskUsage ||
                      seqLength < 900000000</expression>
        </condition>
        <condition number="1">
          <expression>seqLength >= 900000000</expression>
        </condition>
        <action number="0">
          <expression>"skip"</expression>
        </action>
        <action number="1">
          <expression>"true"</expression>
        </action>
      </rules>
      <evaluateOnce isPre="true"
                    phaseName="initiateConstruction"/>
      <currValue>null</currValue>
    </ruleBasedParameter>
  </property>
  <group name="Build Client">
    <property final="false" name="garbageCollectionInterval"
              userMode="User Editable"
              valueType="java.lang.Integer">
      <description>The time (in milliseconds) between
      invocations of the garbage collector (this is a system
      property).  Increasing this value may improve construction
      performance, however having too large a gap between
      garbage collections may degrade performance.</description>
      <simpleParameter> <value>900000</value> </simpleParameter>
    </property>
    <property final="false" name="backBoneWriteBuffer"
              userMode="User Editable"
              valueType="java.lang.Integer">
      <description>The size of buffer used when writing
      back-bone sections to disk.  Increasing this value may
      improve performance when creating large indexes, however
      this will increase memory usage, and may result in more
      partitions being used.</description>
      <fixedChoiceParameter>
        <choices currentChoice="1" numberOfChoices="5">
          <choice number="0"> <value>131072</value> </choice>
```

```
      <choice number="1"> <value>262144</value> </choice>
      <choice number="2"> <value>524288</value> </choice>
      <choice number="3"> <value>1048576</value> </choice>
      <choice number="4"> <value>2097152</value> </choice>
    </choices>
  </fixedChoiceParameter>
</property>
<property final="false" name="numberOfThreads"
          userMode="User Editable"
          valueType="java.lang.Integer">
  <description>The default number of construction threads to
  be used by sub-tree builders.  Increasing the number of
  threads can improve construction performance, but at the
  cost of additional partitions being used.  Note that this
  value can be overridden at the command line.</description>
  <simpleParameter> <value>1</value> </simpleParameter>
</property>
<property final="false" name="suffixesBufferSize"
          userMode="User Editable"
          valueType="java.lang.Integer">
  <description>The number of suffixes to be accessed at one
  time when processing the grouped suffix list (if
  appropriate).  Increasing the number of suffixes increases
  the buffer size and may improve construction performance,
  but at the cost of increased memory use (which potentially
  requires additional partitions to be used).</description>
  <fixedChoiceParameter>
    <choices currentChoice="2" numberOfChoices="4">
      <choice number="0"> <value>32768</value> </choice>
      <choice number="1"> <value>65536</value> </choice>
      <choice number="2"> <value>131072</value> </choice>
      <choice number="3"> <value>262144</value> </choice>
    </choices>
  </fixedChoiceParameter>
</property>
<property final="false" name="subTreeWriteBuffer"
          userMode="User Editable"
          valueType="java.lang.Integer">
  <description>The size of buffer used when writing
  individual sub-trees to disk.  Increasing this value may
  improve performance when creating large indexes, however
  this will increase memory usage, and may result in more
  partitions being used.</description>
  <fixedChoiceParameter>
```

```
        <choices currentChoice="1" numberOfChoices="5">
          <choice number="0"> <value>131072</value> </choice>
          <choice number="1"> <value>262144</value> </choice>
          <choice number="2"> <value>524288</value> </choice>
          <choice number="3"> <value>1048576</value> </choice>
          <choice number="4"> <value>2097152</value> </choice>
        </choices>
      </fixedChoiceParameter>
    </property>
  </group>
  <group name="Suffix Grouping">
    <property final="false" name="evictionThreshold"
              userMode="User Editable"
              valueType="java.lang.Integer">
      <description>The amount of available main memory drops
      below this value (where the value is to be read as (1/x),
      eviction will take place.  Increasing this value will
      cause fewer eviction to take place, but having too small a
      fraction can cause the algorithm to fail.</description>
      <fixedChoiceParameter>
        <choices currentChoice="2" numberOfChoices="7">
          <choice number="0"> <value>30</value> </choice>
          <choice number="1"> <value>35</value> </choice>
          <choice number="2"> <value>40</value> </choice>
          <choice number="3"> <value>45</value> </choice>
          <choice number="4"> <value>50</value> </choice>
          <choice number="5"> <value>55</value> </choice>
          <choice number="6"> <value>60</value> </choice>
        </choices>
      </fixedChoiceParameter>
    </property>
    <property final="false" name="writeBufferSize"
              userMode="User Editable"
              valueType="java.lang.Integer">
      <description>The size of buffer to be used when writing
      out an individual bucket.  Increasing the size of this
      value may improve grouping performance, but at the cost of
      increased memory usage.</description>
      <fixedChoiceParameter>
        <choices currentChoice="0" numberOfChoices="3">
          <choice number="0"> <value>1048576</value> </choice>
          <choice number="1"> <value>2097152</value> </choice>
          <choice number="2"> <value>4194304</value> </choice>
        </choices>
```

```
            </fixedChoiceParameter>
          </property>
          <property final="false" name="bucketBlockSize"
                    userMode="User Editable"
                    valueType="java.lang.Integer">
            <description>The size of each block in the buckets used to
            collect grouped suffixes.  There will be one bucket per
            partition.  Increasing the size of the buckets will result
            in fewer 'eviction' phases, and may result in increased
            performance.  However, if the bucket sizes become too
            large, suffix grouping may fail due to inadequate memory
            resources.</description>
            <fixedChoiceParameter>
              <choices currentChoice="1" numberOfChoices="4">
                <choice number="0"> <value>32768</value> </choice>
                <choice number="1"> <value>65536</value> </choice>
                <choice number="2"> <value>131072</value> </choice>
                <choice number="3"> <value>262144</value> </choice>
              </choices>
            </fixedChoiceParameter>
          </property>
          <property final="false" name="prefixBufferSize"
                    userMode="User Editable"
                    valueType="java.lang.Integer">
            <description>The buffer size (in number of prefixes) to be
            used when reading in the sequence file.  Increasing this
            buffer size may improve performance, but at the cost of
            additional memory usage.</description>
            <fixedChoiceParameter>
              <choices currentChoice="1" numberOfChoices="4">
                <choice number="0"> <value>32768</value> </choice>
                <choice number="1"> <value>65536</value> </choice>
                <choice number="2"> <value>131072</value> </choice>
                <choice number="3"> <value>262144</value> </choice>
              </choices>
            </fixedChoiceParameter>
          </property>
        </group>
      </group>
    </group>
  </localParameters>
</properties>
```

Appendix C

GIDOF: User's Tutorials

This appendix describes the steps required to install, compile and use GIDOF for the purposes of specifying or tuning a parameter list associated with a given application implementation. Separate tools are provided for clients and developers of the technology. Clients of an application making use of GIDOF should ensure that their system meets the requirements specified in Section C.1 and then consult Section C.3. Developers making use of GIDOF should read all sections of this appendix.

C.1 System Requirements

- Java 2 Runtime Environment (JRE), Standard Edition, version 1.4.0 (or higher).
- Java 2 Software Development Kit (SDK), Standard Edition, version 1.4.0 (or higher), if compilation from source is required.
- Apache Ant version 1.5.1 (or higher),required for both compilation from source, and for invoking the GIDOF tuning application.
- Xerces Java Parser libraries: `xercesImpl.jar` and `xmlParserAPIs.jar`, February 2003 version or higher.
- Java Mathematical Expression Parser (JEP) library, version 2.3.0 or higher.

Both the Java 2 Runtime Environment and Software Development Kit are available from `http://java.sun.com/`. Ant is available from `http://ant.apache.org/`. Xerces is available from `http://xml.apache.org/xerces-j/`. Java Mathematical Expression Parser is available from `http://www.singularsys.com/jep/`.

C.2 Installation Guide

If GIDOF is being used with a completed application implementation, then the library file `gidof.jar` should be provided and no compilation will be required. If compilation from source is required, then the following steps should be followed. Prior to compilation, three items should be present in the directory containing the GIDOF software: an Ant build script (`build.xml`), a directory named `libs` (containing the libraries discussed in the previous section) and a directory named `java` containing the source code. It is assumed that both Java and Ant are correctly installed and configured.

The software can be compiled by invoking the command `ant`. This will create a directory named `classes` and compile all relevant source code (placing the compiled classes in the newly created directory). If Javadoc documentation is required, then it can be obtained by invoking the command `ant doc`. The location of the `classes` directory and the contents of the `libs` directory should be added to the `CLASSPATH` environmental variable. Finally, if a `jar` file is required, then this can be created by invoking the command `ant jar`.

C.3 Using the Client's Parameter Tuning Tool

The GIDOF client's tuning tool can be used to alter the values of selected operational parameters associated with a given application. This can allow certain aspects of the application's behaviour to be tailored to suit the specific needs of the environment it will be operating in and the dataset that it will be operating over. The parameters presented by this tool are those that have been identified by the developer of the software as being appropriate for tuning by the client of the application. This tool can be invoked using the command `ant TuningWizard`, or via the command `java gidof.properties.TuningWizard`. If the latter is used, then the location of the property file to be examined can be supplied as a command-line argument, otherwise the file can be selected using the file chooser window presented by the tool. The two steps involved in tuning an application are now discussed.

C.3.1 Tuning Shared Parameters

The first step in exploring (and tuning) the operational parameters associated with a given application is to browse the list of tunable shared parameters. Shared parameters are parameters that affect several aspects of the application's performance and are likely to include fundamental options about how the application is to be used. Fig-

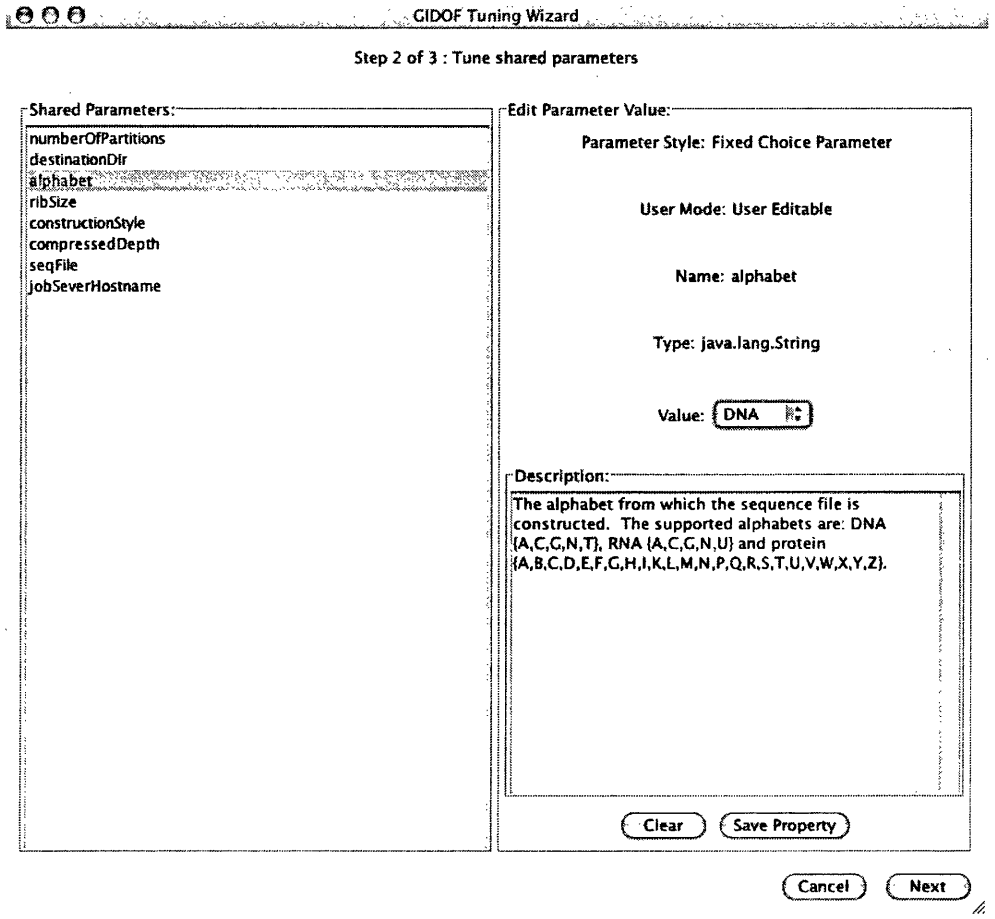

Figure C.1: Examining the value of a shared parameter using the GIDOF client's tuning tool.

ure C.1 shows this step of the tool. The list of available parameters is displayed on the left-hand side. When one of the parameters is selected, the right-hand panel of the tool will update to display the details of the parameter. This includes a description of the parameters purpose and, where appropriate, what the impact of changing the parameters value will have on the application's behaviour. At this point the value of the parameter can be altered. Depending on the nature of the parameter, the value will either be entered as text (which will subsequently be checked to ensure that it can be parsed as the appropriate type) or by selecting one of several possible values from the menu given (as is the case in Figure C.1). The value can then be saved and the values of other parameters explored. Once all parameters have been explored, the next step of the tuning process can be accessed.

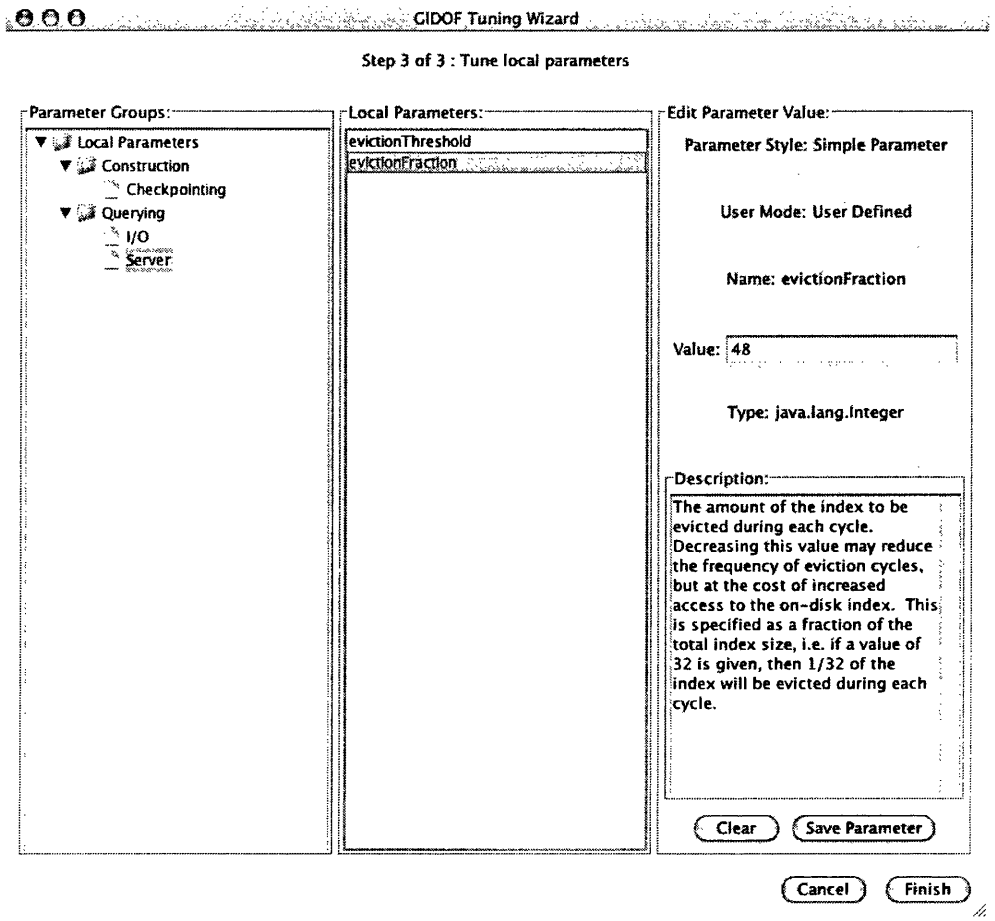

Figure C.2: Examining the value of a local parameter using the GIDOF client's tuning tool.

C.3.2 Tuning Local Parameters

The second step in exploring the operational parameters is to browse the groups of local parameters associated with the application. Although similar in many ways to the previous step, there are some notable differences in the way that the parameters are accessed and displayed. Local parameters are parameters that only affect the behaviour of limited areas of the application's behaviour. Therefore, parameters are organised in hierarchical groups, with each group corresponding to a particular aspect of the application. The exact nature of the grouping will be decided by the developer, but parameters should be organised in such a way that parameters affecting the same aspect of performance are grouped together. Figure C.2 shows how the group hierarchy and

list of parameters are presented by the tool. The left-most panel of the tool shows a tree representation of the available groups. Individual nodes of the tree can then be selected, with the middle panel updating to show the parameters associated with that group. A parameter can then be selected form the middle panel, with its details being presented in the right-most panel. The parameter's value can then be altered as before.

C.3.3 Saving Changes

Once the previous step has been completed, the user will be presented with a file chooser dialogue. Here, the user will select the file where the completed parameter is to be written. Applications may expect the property file to be stored at a specific location: this should be considered when selecting a location.

C.4 Developer's Parameter Specification Tool

The GIDOF developer's parameter specification tool can be used to define the parameters to be associated with a given application implementation. This tool is likely to be used iteratively, with the parameter list growing as implementation progresses. For example, as new components are added to the application, it is likely that further operational parameters will be required—these can simply be added to the current parameter list as necessary. Furthermore, it is possible to change the nature and values of the parameters with only minimal changes being required by the application implementation. For example, if it is found through experimentation, that a certain set of values produces the best possible performance, then the corresponding parameter can be restricted to allow only those values. The parameter specification tool can be invoked using the command `ant PropertyWizard`, or via the command `java gidof.properties.PropertyWizard`. If the latter is used, then the location of the property file to be edited can be supplied as a command-line argument, otherwise the file can be selected using the file chooser window presented by the tool. The steps involved in creating (or editing) a parameter list are now discussed.

C.4.1 Specifying Phases

The first part of the parameter list to be addressed is the specification of the phases associated with the application. Phases are used to mark significant points in the lifetime of the application, and can be used in conjunction with rule-based parameters to automate certain aspects of parameter management (see Section C.4.4). Each phase

is identified by a unique name, and the order in which they are presented corresponds to the order in which they are to be processed. The tool presents the current list of phases in the left-hand panel, allowing, where necessary, an existing phase to be removed from the list (see Figure C.3). New phases can be added to the end of the list. Once the phase list has been correctly specified, the user can proceed to the next step.

C.4.2 Specifying Shared Parameters

Shared parameters are parameters that are used throughout the application implementation, and are typically those that are fundamental to the operation of the application. For example, a parameter that represents the location of the completed index might be represented as a shared parameter, whereas a parameter representing an input buffer's size may be better represented as a local parameter as it will only be needed by one part of the application.

At this step in the use of the tool, shared parameters can be added or deleted from the current parameter list. Furthermore, existing parameters can be edited, allowing the nature of the parameter to change as application development progresses. Associated with each parameter are the following attributes: name, user mode, value, type, style, description and whether or not the value should be treated as a constant. The name and type of the parameter should correspond to the name and type used by the application implementation when accessing the parameter. The user mode determines whether or not a given parameter will be available for tuning, and can be changed as development progresses. The description should give a concise description of the role of the parameter within the application—this is the primary means by which a client of the technology will gain information on how to tune the application, therefore it is important that this information is provided.

In addition to the attributes described above, each parameter can be specified as being of the following three types: simple parameter, fixed-choice parameter, rule-based parameter. Each parameter style represents its value in a different manner. Simple parameters can have any value that can be parsed as the stated type. Such parameters are most likely to be of use when defining parameters are to be used by the developer (or internally by the application). Figure C.4 shows how the tool can be used to define a simple parameter. Fixed-choice parameters have a list of values associated with them, with the actual value of the parameter restricted to being one of those on the list. This style of parameter should be used wherever there are only a small number of values that a parameter may take (and are particularly suitable for use with client tunable parameters). The final style of parameter, the rule-based parameter, has a value that is

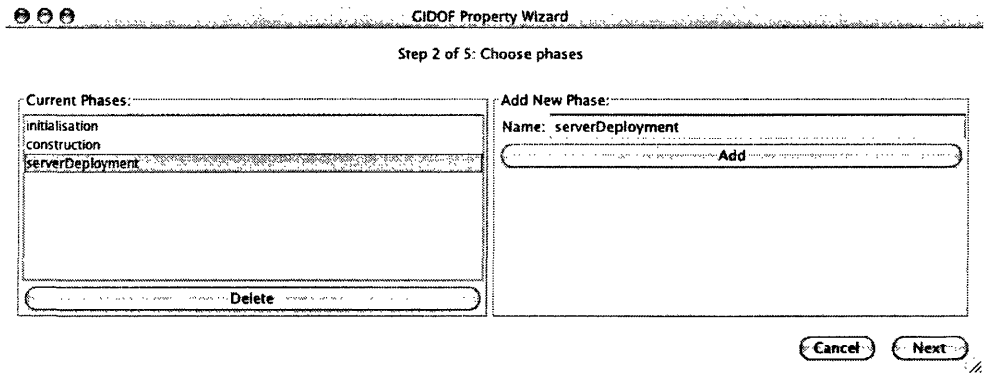

Figure C.3: Specifying the list of phases using the GIDOF developer's parameter tool.

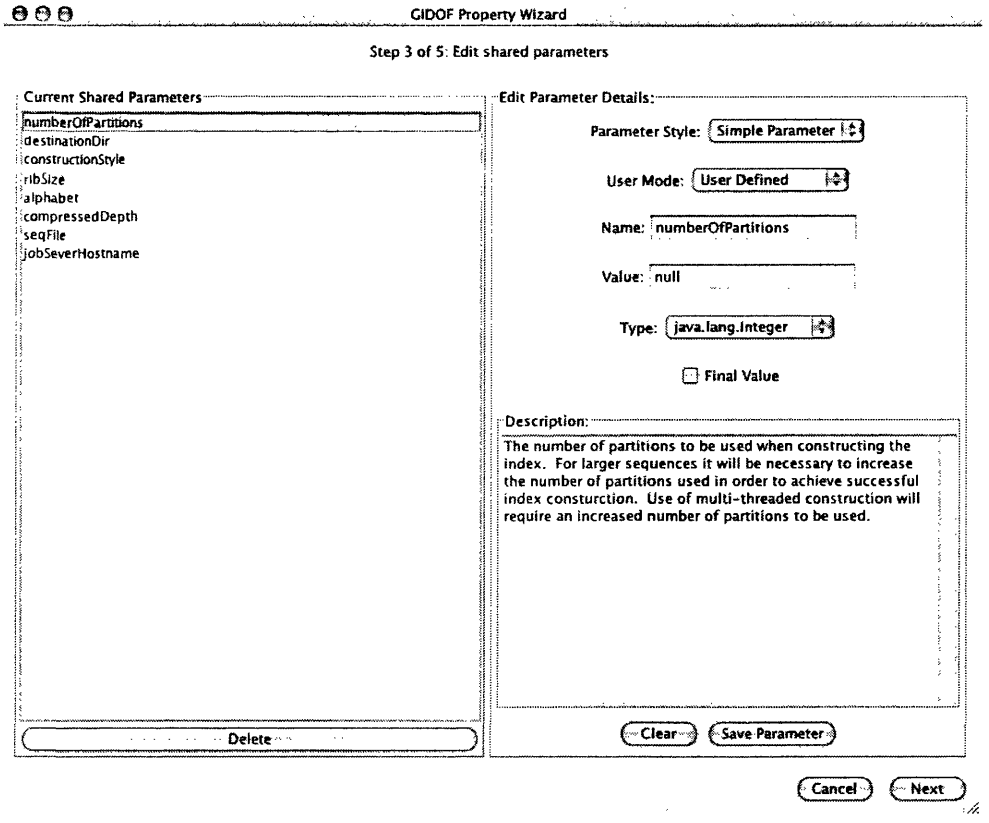

Figure C.4: Defining a shared parameter using the GIDOF developer's parameter tool.

defined in terms of a set of rules and corresponding actions and are discussed in detail in Section C.4.4.

C.4.3 Specifying the Group Hierarchy and Local Parameters

Parameters that only affect specific aspects of the application's behaviour, and are not fundamental to the operation of the application, should be defined as local parameters. Local parameters are arranged in groups, with all parameters in a given group influencing a certain aspect of the application's performance. For example, parameters may be grouped according to function, thus all parameters relating to, say, index construction may be taken together. This allows the parameters to be explored by the client in a structured manner—they need only examine the groups corresponding their particular areas of interest.

Prior to specifying the local parameters, the nature of the group hierarchy must be established. The tool presents the current group hierarchy as a tree where the user can add, delete or rename nodes as desired (see Figure C.5). Note that, deleting a group will result in all sub-groups and associated parameters also being deleted. Once the hierarchy has been specified, local parameters can then be added to the groups as necessary.

Defining the local parameters is the final step in the specification of a parameter list. As each local parameter is associated with a specific group, the first part in defining a local parameter is to locate the appropriate group. This is achieved by browsing the group hierarchy (again, presented as a tree) and selecting the appropriate node. Once a node has been selected, the current list of associated parameters is displayed and parameters can be added, deleted or edited as they were for shared parameters. Figure C.6 shows how this step is presented by the tool. The left-most panel presents the group hierarchy, the middle panel presents any parameters currently associated with the selected group, and the right-most panel shows the details of the parameter being added. Here, a fixed-choice parameter has been added, hence a dialogue requesting a list of values for the parameter has been presented. Once this step is complete, the parameter list can then be saved to an appropriate file, thus completing the specification of the parameter list (although further iterations of this process may be necessary).

C.4.4 Rule-Based Parameters

The final aspect of the developer's parameter specification tool to be discussed is the process of defining a rule-based parameter. A given parameter (which can be either a

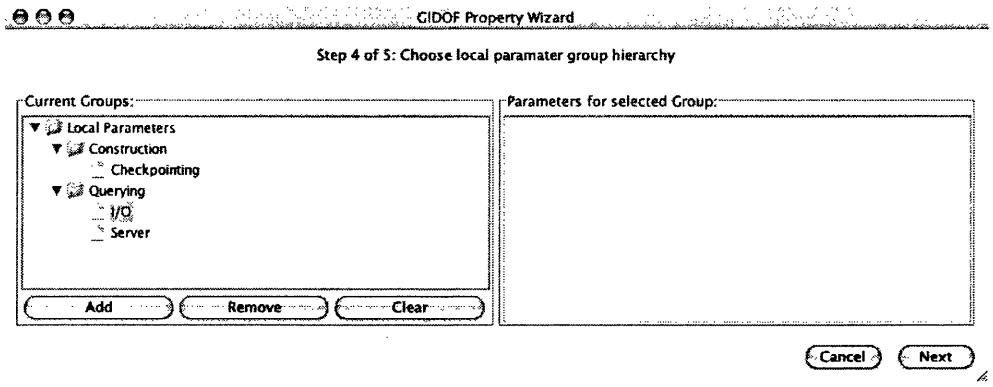

Figure C.5: Defining the group hierarchy using the GIDOF developer's parameter tool.

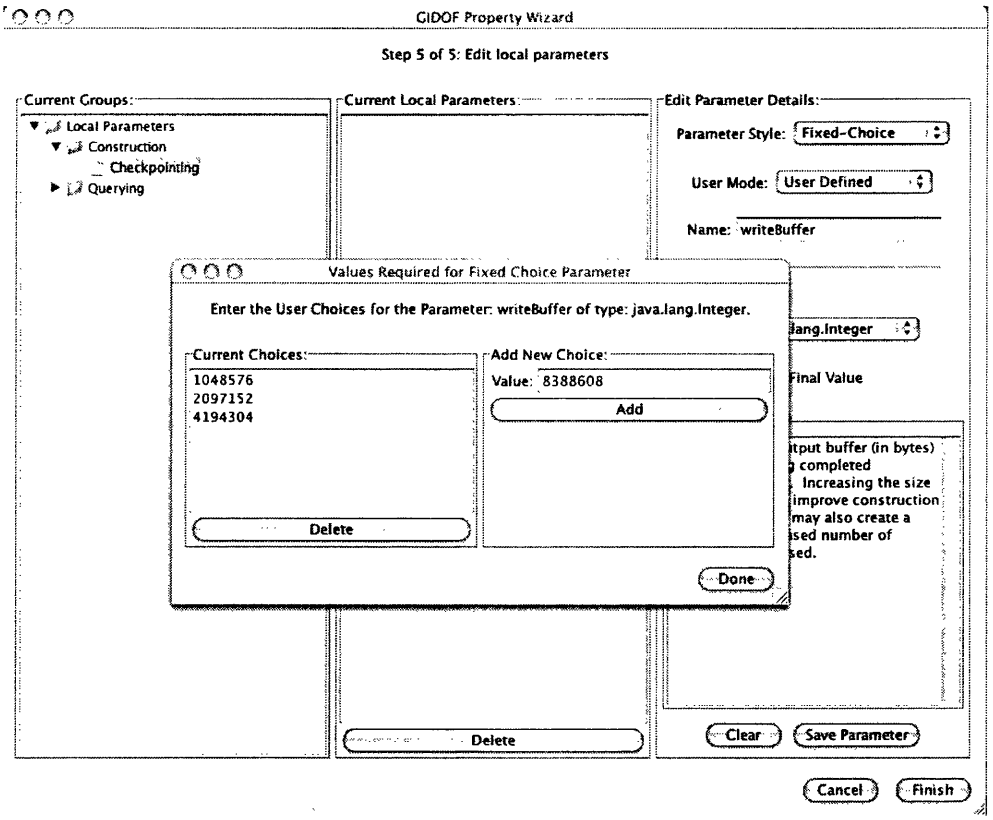

Figure C.6: Defining a local parameter using the GIDOF developer's parameter tool.

shared parameter, or a local parameter) can be defined as being a rule-based parameter, that is, a parameter with a value that is derived at run-time depending on the values of other parameters. This can allow parameters to be specified in such a way that their value is derived at run-time depending on the characteristics of the current system and workload—thus allowing parameters to be tuned without manual intervention.

Rule-based parameters are defined in terms of conditions and corresponding actions. When establishing the parameter's value, each condition is evaluated in turn, until a condition is found that evaluates to true (a non-zero value). When this condition is found, the expression of the corresponding action is evaluated, the result of which becomes the new value of the parameter. If no condition is found to hold true, then the parameter's value does not change. Three variations of the rule-based parameter are provided: those that are to be evaluated on-demand (i.e. the application will explicitly request that the parameter update its value); those that are to be evaluated once at a fixed point in the application's life-cycle; and those will update their value whenever a parameter that its value depends on updates (such parameters are referred to a self-adjusting parameters). Phases are used to determine the points at which self-adjusting parameters should start and stop tuning, and are used in the case of a parameter with a value that is derived at a fixed point. Both conditions and actions are specified using mathematical expressions and are parsed using the Java Mathematical Expression Parser (JEP), as described in the following section.

Figure C.7 shows how conditions and actions are specified using the developer's tool. In this example, the parameter `compressedDepth` is dependent on a parameter named `seqLength`; as `seqLength` increases beyond certain values, the value of `compressedDepth` is to increase accordingly. Once the attributes of a rule-based parameter have been set, the conditions and actions can be defined. This is presented as a simple window showing the current conditions and actions (which, if necessary, can be deleted) together with the option of adding a further condition and action. A label stating the current validity of both condition and action is shown, allowing the user to be sure that the specified condition and action are valid. Once all rules have been specified, the user simply closes the window, thus completing the specification of the parameter.

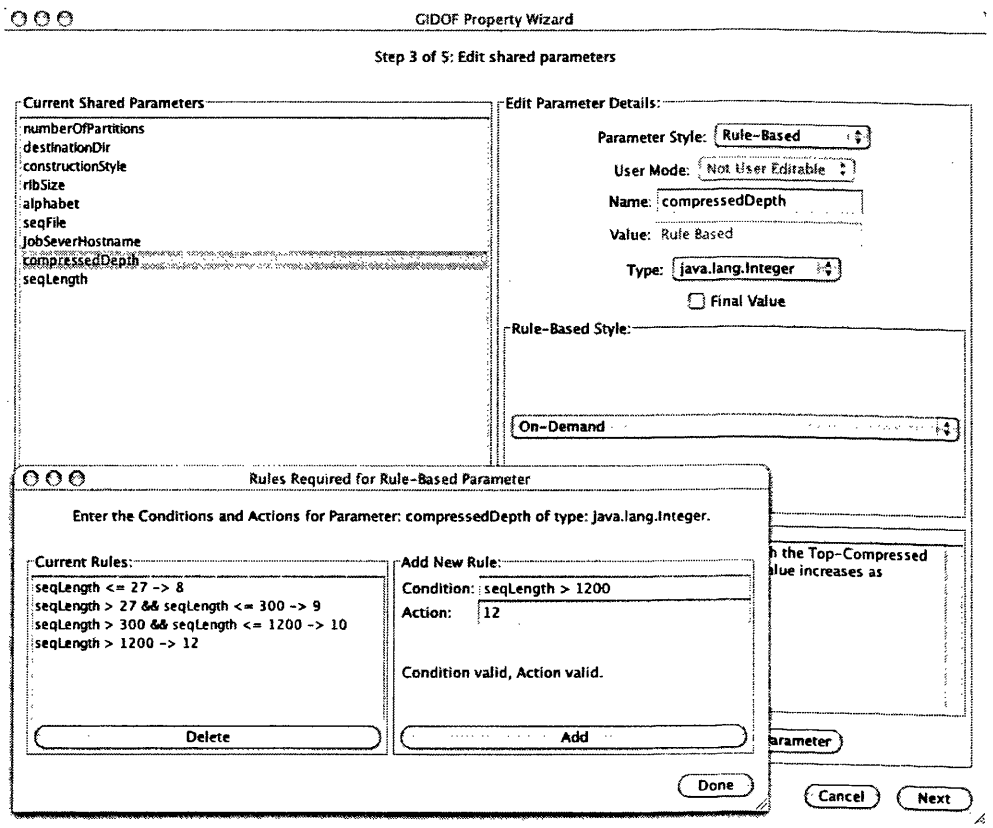

Figure C.7: Defining a rule-based parameter using the GIDOF developer's parameter tool.

JEP

The grammar associated with JEP (Java Mathematical Expression Parser)[1] is the method used to specify both conditions and actions of rule-based parameters. Valid JEP expressions are similar to valid Java expressions: variables, literals and function calls can be combined using a variety of operators, giving a powerful and flexible mechanism for specifying conditions and actions. The variables accessible when specifying a rule are all of the shared parameters that have previously been specified using GIDOF. In the example shown in Figure C.7, the conditions all reference the `seqLength` parameter, which is already in the list of shared parameters. Additionally, constant values for `pi`, `e`, `TRUE` and `FALSE` are provided, and can be accessed when defining a rule. The operators supported by JEP are listed in Table C.1, and include many of the operators

[1]JEP is developed by Singular Systems. The information given here is derived from the documentation provided at `http://www.singularsys.com/jep/`.

supported by Java (with the addition of the power operator). Furthermore, several common functions are also supported, including various trigonometric functions and logarithms. Note that literal values are valid expressions, thus actions need only be specified as the value that the parameter should take. In addition, variables of types String and Character are available when specifying rules and actions, so care should be taken when referencing these variables—while it is meaningful to compare to value of two strings, comparing a string with a numeric value will always return false.

C.5 Summary

This appendix has described the two applications that comprise the GIDOF toolkit. The process of tuning an application using the client's parameter tool was described, with illustrations of each of the key steps provided. The developer's parameter tool was then described, and again the key steps were discussed and illustrations provided.

Operator Name	Symbol
Power	^
Boolean Not	!
Unary Plus, Unary Minus	+x, -x
Modulus	%
Division	/
Multiplication	*
Addition, Subtraction	+, -
Less or Equal, More or Equal	<=, >=
Less Than, Greater Than	<, >
Not Equal, Equal	!=, ==
Boolean And	&&
Boolean Or	||

Table C.1: Operators available when specifying rules in GIDOF (JEP operators).

Function	Name
Sine	sin(x)
Cosine	cos(x)
Tangent	tan(x)
Arc Sine	asin(x)
Arc Cosine	acos(x)
Arc Tangent	atan(x)
Hyperbolic Sine	sinh(x)
Hyperbolic Cosine	cosh(x)
Hyperbolic Tangent	tanh(x)
Inverse Hyperbolic Sine	asinh(x)
Inverse Hyperbolic Cosine	acosh(x)
Inverse Hyperbolic Tangent	atanh(x)
Natural Logarithm	ln(x)
Logarithm base 10	log(x)
Absolute Value / Magnitude	abs(x)
Random number (between 0 and 1)	rand()
Square Root	sqrt(x)
Sum	sum(x,y,...)
Convert to String	str(x)

Table C.2: Functions available when specifying rules in GIDOF (JEP functions).

Glossary

Approximate Matching

The process of finding the positions in a text where a given pattern occurs, allowing a limited number of 'errors' in the matches. Such errors may include the deletion, insertion or replacement of one or more characters. Various techniques exist to solve this problem, although the number, and nature, of the errors allowed in a match differ with choice of algorithm (and also with the application-specific costs associated with each type of error).

Backbone

The first level array element in a *two-level array*. A two-level array will have exactly one backbone, which is used to maintain (and provide access to) the collection of *ribs* that make up the second level.

Base Pair

Two bases (nucleotides) that bond to form a complementary pair. In a *DNA sequence*, each base pair is represented by a single letter (drawn from the alphabet **{A,C,G,T}**).

BLAST

A software package that uses heuristics to compute local alignments (a form of *approximate matching*) over a DNA or protein sequence.

bp

An abbreviation of *base pair*.

Checkpointing

The process of transferring a section of a memory-resident data structure to secondary memory so that it can be accessed during subsequent program invocations (see also *eviction*).

Cold Performance

The performance of an algorithm when operating for the first time over a given data structure, meaning that no part of the data structure will have been cached before invoking the algorithm (compare with *warm performance*).

Densely Coding

The process of coding a given alphabet using sequential integers starting at zero, as opposed to using a standard coding (such as ASCII).

Disk-Resident

A *disk-resident* data structure is one that resides on secondary memory, with sections of the structure only being transferred to main memory as necessary.

DNA Sequence

A representation of a single DNA molecule. When given in textual format, it is comprised of a sequence of letters, each corresponding to one *base pair*.

Eviction

The process of removing a section of a data structure from main memory in order to allow the newly vacant section of main memory to be reclaimed and reused. Eviction may be used in conjunction with *checkpointing*.

Exact Matching

The process of finding all occurrences of a specific pattern in a given text.

Faulting

The operation of transferring a required section of a *disk-resident* data structure from secondary storage to main memory.

Fixed-Choice Parameter

A *GIDOF* parameter with an associated list of possible values, the value of which is then restricted to being one of those listed.

Generic Index Development and Operation Framework

A toolkit and framework to support the management of *operational parameters*; as is necessary when implementing or tuning a bespoke persistent index.

Genome

The entire genetic information contained in a single organism.

GIDOF

See *Generic Index Development and Operation Framework.*

Horner's Method

A method for evaluating a polynomial using repeated addition and multiplication.

Improved Prefix-Partitioned Construction Algorithm

A variation of the *Prefix-Partitioned Construction Algorithm* that uses a pre-processing stage to partition suffixes prior to creating a suffix tree.

Incremental Construction

The process of constructing a data structure in such a way that completed sections of the data structure may periodically be transferred to secondary storage and will not subsequently be accessed during construction.

JEP

Java Expression Parser: a package that can parse and evaluate a wide variety of mathematical expressions given in textual format.

JVM

Java Virtual Machine: a component of the Java runtime environment that interprets Java bytecode.

Known Non-Null

An element in a *two-level array* that is known not to be null. That is, it is known that there is a sub-tree corresponding to this element resident on disk, but that sub-tree is not currently held in main memory. Such a situation may arise when a two-level array element has been accessed on disk, but the corresponding sub-tree was not.

Known-Null

An element in a *two-level array* that is known to be null. That is, it is known that there is no sub-tree on secondary storage that corresponds to this element.

Local Parameters

The group of *operational parameters* in a *GIDOF* parameter model that only influence limited aspects of the applications behaviour.

Locality of Reference

An application is said to have good *locality of reference* if all accesses to memory are localised, thus allowing for good cache interaction.

Operational Parameter

A parameter that influences the behaviour of a given application.

Orthogonally Persistent Programming Language

A programming language that supports the automatic retention of main-memory data structures over several invocations of a given program.

Parameter Tuning

The process of tuning the *operational parameters* of a given application in order to improve selected aspects of performance.

PATRICIA

Practical Algorithm To Retrieve Information Coded In Alphanumeric: an indexing technique that supports efficient processing of textual data [85].

Persistence

The concept of retaining data values across multiple invocations of an application.

Persistence Layer

The section of code in the *TCST* responsible for invoking the *faulting* and *eviction* of index sections.

Prefix-Partitioned Construction Algorithm

A technique for constructing suffix trees (and related data structures) where the insertion of suffixes is grouped according to the prefix partition in which they lie.

Presence Test

Here, used to describe a simpler version of the *exact matching* problem where it is only required to confirm that the given search target is present in the text (i.e. it is not necessary to locate each occurrence of the target pattern).

Recursive Marshalling Algorithm

A recursive technique for serialising the main-memory representation of a tree, thus allowing it to be stored on disk as a linear sequence of bytes.

Reference Data

Data that is infrequently (possible never) updated.

Rib

A second level element in a *two-level array*. Ribs are allocated on demand, meaning that different two-level array instances of the same capacity may have different numbers of ribs.

RMI

Remote Method Invocation: a technique for communicating between process running on separate computers that uses the semantics of the method call.

Rule-Based Parameter

A *GIDOF* parameter with a value that is derived at run-time using a sequence of predefined rules.

Self-Adjusting Parameter

A variation of the *Rule-Based Parameter*, the value of which can update in response to any changes in the values upon which it depends.

Sequence Data

One, or more, instances of a *DNA sequence* represented in digital form.

Shared Parameters

A parameter grouping used in *GIDOF* to represent *operational parameters* that influence the behaviour of all aspects of a given index's behaviour.

Sibling Chain

A technique for representing a tree where a given node only has one child reference, with all other child nodes being accessed by first accessing the child of the current node and subsequently traversing the chain of sibling links starting at that node.

Simple Parameter

A *GIDOF* parameter that can take any value of the specified type.

SPLAT

Suffix-tree Powered Local Alignment Tool, a software package that employs a suffix tree index to accelerate the calculation of local alignments.

Suffix Sequoia

An array based data structure that can be used as an index of the prefixes of the suffixes of a given text.

Suffix Sub-Tree

A sub-tree in a *TCST*, and is equivalent to a suffix tree containing all the suffixes (minus the first c letters) corresponding to a particular element in the TCST.

Suffix Tree

An data structure, similar to a *PATRICIA* tree, that indexes all the suffixes of a given text in order to allow efficient matching of target patterns.

TCST

See *Top-Compressed Suffix Tree.*

TDD

A package to allow Top Down Disk-based suffix tree construction [99].

Top-Compressed Suffix Tree

A variant of the suffix tree that represents the first c characters of each suffix using a *two-level array.*

TOP / TOP-Q

A paging policy that claims to allow for efficient access to disk resident nodes when constructing a suffix tree using Ukkonen's construction algorithm [15].

Transient

A data structure is said to be *transient* if it is not retained once the program has finished executing (i.e. does make use of *persistence*).

Two-Level Array

A technique for representing a sparsely populated array using one *backbone* and some number of *ribs.*

Two-Pass Suffix Grouping Algorithm

An implementation of the pre-processing stage of the *Improved Prefix Partitioned Construction Algorithm.*

Ukkonen's Algorithm

A linear time (with respect to sequence length) suffix tree construction algorithm [100].

Warm Performance

The performance of an algorithm over a given data structure, where the algorithm has already been executed at least once over this data structure; thus increasing the likelihood of the data already being cached (compare with *cold performance*).

Bibliography

[1] Nabil R. Adam and Aryya Gangopadhyay. *Database Issues in Geographic Information Systems*, volume 6 of *Advances in Database Systems*. Kluwer Academic Publishers, 1997.

[2] Susanne Albers and Jeffery Westbrook. Self-organizing data structures. In *Online Algorithms, The State of the Art*, number 1442 in Lecture Notes in Computer Science, pages 13–51. Springer, 1998.

[3] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[4] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, X. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search tools. *Nucleic Acids Research*, 25:3389–3402, 1997.

[5] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 115–125. ACM, May 1990.

[6] A. Andersson, N.J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[7] Alberto Apostolico and Wojciech Szpankowski. Self-alignments in words and their applications. *Journal of Algorithms*, 13(3):446–467, 1992.

[8] Malcolm P. Atkinson. Programming languages and databases. In S. Bing Yao, editor, *Very large data bases: Fourth International Conference on Very Large Data Bases, West Berlin, Germany, September 13–15, 1978*, pages 408–419. IEEE Computer Society Press, 1978.

[9] Malcolm P. Atkinson. Databases and the grid: Who challenges whom? In Anne E. James, Brian Lings, and Muhammad Younas, editors, *New Horizons in Information Management, 20th British National Conference on Databases, BNCOD 20, Coventry, UK, July 15-17, 2003, Proceedings*, volume 2712 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2003.

[10] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical Report TR-2000-90, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, 901 San Antonio Road, Palo Alto, CA 94303, USA and Glasgow, G12 8QQ, Scotland, 2000.

[11] M.P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[12] M. Ayala-Rincón and P.D. Conejo. A linear time lower bound on McCreight and general updating algorithms for suffix trees. *Algorithmica*, 37(3):233–241, 2003.

[13] Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[14] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1), 1977.

[15] Srikanta J. Bedathur and Jayant R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 720–731. IEEE, 2004.

[16] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler. Genbank. *Nucleic Acids Research*, 30(1):17–20, 2002.

[17] Elisa Bertino. Index configuration in object-oriented databases. *VLDB Journal: Very Large Data Bases*, 3:355–399, July 1994.

[18] M. Birbeck, editor. *Professional XML*. Programmer to Programmer. Wrox Press Inc., second edition, 2001.

[19] R. Bliujute, S. Saltenis, G. Šlivinskas, and C.S. Jensen. Developing a DataBlade for a new index. In *15th International Conference on Data Engineering (ICDE '99)*, pages 314–327, Washington - Brussels - Tokyo, March 1999. IEEE.

[20] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, July 1985.

[21] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[22] C.B. Boyer and U. Merzbach. *A History of Mathematics*. Wiley, second edition, 1991.

[23] Alvis Brāzma, Inge Jonassen, Ingvar Eidhammer, and David Gilbert. Approaches to the automatic discovery of patterns biosequences. *Journal of Computational Biology*, 5(2):277–303, 1998.

[24] A.L. Brown. Constructing chromosome scale suffix trees. In Yi-Ping Phoebe Chen, editor, *Bioinformatics 2004*, volume 29 of *CRPIT*, pages 105–112, Dunedin, New Zealand, 2004. Australian Computer Society Inc.

[25] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2002.

[26] Surajit Chaudhuri. Letter from the special issue editor. *IEEE Data Engineering Bulletin*, 22(2):2, 1999.

[27] Surajit Chaudhuri, Eric Christensen, Goetz Graefe, Vivek R. Narassayya, and Michael J. Zwilling. Self-tuning technology in Microsoft SQL server. *IEEE Data Engineering Bulletin*, 22(2):20–26, 1999.

[28] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[29] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[30] L. Colussi and A. De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, June 1996.

[31] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[32] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases(VLDB '01)*, pages 341–350, Orlando, September 2001. Morgan Kaufmann.

[33] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[34] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Pearson Education International, fourth edition, 2004.

[35] Huw Evans. Why object serialization is inappropriate for providing persistence in Java. Technical report, Department of Computing Science, Glasgow University., 2000.

[36] Martin Farach, Paolo Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[37] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[38] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., fourth edition, 2002.

[39] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[40] Andrea Garratt, Mike Jackson, Peter Burden, and Jon Wallis. A comparison of two persistent storage tools for implementing a search engine. In *Persistent Object Systems, 9th International Workshop, POS-9, Lillehammer, Norway, September 6-8, 2000, Revised Papers*, volume 2135 of *Lecture Notes in Computer Science*, pages 177–186. Springer, 2001.

[41] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software—Practice and Experience (SP&E)*, 33(11):1035–1049, 2003.

[42] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structures & Algorithms.* Prentice-Hall, 1992.

[43] Jim Gray. On-line science: The world-wide telescope as a prototype for the new computational science. In Lise Getoor, Ted E. Senator, Pedro Domingos, and Christos Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, page 3. ACM, 2003.

[44] Jim Gray. The next database revolution. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 1–4. ACM, 2004.

[45] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. In *16th International Conference on Data Engineering (ICDE' 00)*, pages 3–12, Washington - Brussels - Tokyo, March 2000. IEEE.

[46] Jim Gray and Alex Szalay. The world-wide telescope, an archetype for online science. *Communications of ACM*, 45(11):50–55, 2002.

[47] Jim Gray, Alex Szalay, Ani R. Thakar, Christopher Stoughton, and Jan vandenBerg. Online scientific data curation, publication, and archiving. Technical Report TR-2002-74, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, 2002.

[48] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[49] D. Gusfield. Suffix trees (and relatives) come of age in bioinformatics. In *1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, page 3. IEEE Computer Society, 2002.

[50] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormack, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, Boston, MA, June 1984. ACM Press.

[51] C.G. Hamilton. Recovery Management for Sphere: Recovering a Persistent Object Store. Technical Report TR-1999-51, University of Glasgow, Department of Computing Science, December 1999.

[52] Nigel Harding and Malcolm Atkinson. SPLAT (suffix-tree powered local alignment tool): a full-sensitivity protein database search program that accelerates the smith-waterman algorithm using a generalised suffix tree index. Technical Report TR-2003-141, University of Glasgow, Department of Computing Science, February 2004.

[53] Ben He and Iadh Ounis. A study of parameter tuning for term frequency normaliztion. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana*, pages 10–16. ACM, 2003.

[54] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M.D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 562–573, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[55] Ela Hunt. The Suffix Sequoia Index for Approximate String Matching. Technical Report TR-2003-135, Department of Computing Science, University of Glasgow, March 2003.

[56] Ela Hunt. Indexed searching on proteins using a suffix sequoia. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 27:24–31, 2004.

[57] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A Database Index to Large Biological Sequences. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 139–148. Morgan Kaufmann, 2001.

[58] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. Database indexing for large DNA and protein sequence collections. *VLDB Journal*, 11(3):256–271, 2002.

[59] Ela Hunt, Robert W. Irving, and Malcolm Atkinson. Persistent suffix trees and suffix binary search trees as DNA sequence indexes. Technical Report TR-2000-63, University of Glasgow, Department of Computing Science, October 2000.

[60] Hyper-threading technology on the Intel Xeon^{TM} processor family for servers (white paper). Technical report, Intel Corporation, Santa Clara, CA 95052, USA, 2002. http://www.intel.com/business/bss/products/hyperthreading/server/.

[61] R.W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. *Journal of Discrete Algorithms*, 1(5–6):387–408, 2003.

[62] Robert Japp. First year report. Annual PhD progress report, Department of Computing Science, University of Glasgow, 2001.

[63] Robert Japp. Persistent indexes for data intensive applications. In Anne E. James, Muhammad Younas, and Mike Jackson, editors, *Technical Report of the 20th British National Conference on Databases, BNCOD 20, Coventry, UK, July 15–17, 2003, PhD Forum Papers*, pages 1–8, School of Mathematical and Information Sciences, Coventry University, Priory Street, Coventry, CV1 5FB, UK, 2003. Coventry University.

[64] Robert Japp. Persistent indexing technology for large sequences. In Anne E. James and Muhammad Younas, editors, *Technical Report of the 20th British National Conference on Databases, BNCOD 20, Coventry, UK, July 15–17, 2003, Poster Papers*, pages 8–11, School of Mathematical and Information Sciences, Coventry University, Priory Street, Coventry, CV1 5FB, UK, 2003. Coventry University.

[65] Robert Japp. The top-compressed suffix tree. In Lachlan M. MacKinnon, Albert G. Burger, and Philip W. Trinder, editors, *21st Annual British National Conference On Databases*, volume 2, pages 68–79, Department of Computer Science, School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton Campus, Edinburgh, EH14 4AS, UK, July 2004. Heriot-Watt University.

[66] Robert Japp. The top-compressed suffix tree: A disk-resident index for large sequences. Technical Report TR-2004-165, Department of Computing Science, University of Glasgow, July 2004.

[67] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In J.-K. Cai and C.K. Wong, editors, *Proceedings of the 2nd Annual International Conference on Computing and Combinatorics*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 1996.

[68] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *6th Symposium on Combinatorial Pattern Matching (CPM'95)*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 1995.

[69] J.W. Kent. BLAT - the blast-like alignment tool. *Genome Research*, 12:656–664, 2002.

[70] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[71] Marcel Kornacker. High-performance extensible indexing. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases, Edinburgh, Scotland, UK, 7–10 September, 1999*, pages 699–708, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers.

[72] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience (SP&E)*, 29(13):1149–1171, October 1999.

[73] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam Storm, and Leanne Wu. Automatic database configuration for DB2 universal database: Compressing years of performance expertise into seconds of execution. In *Datenbanksysteme für Business, Technologie und Web Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 620–629, 2003.

[74] C. Lefevre and J.-E. Ikeda. The position end-set tree: a small automaton for word recognition in biological sequences. *Computational Applied BioScience*, 9(3):343–348, 1993.

[75] Sungchae Lim, Joonseon Ahn, and Myoung-Ho Kim. A concurrent B^{link}-tree algorithm using a cooperative locking protocol. In Anne E. James, Brian Lings, and Muhammad Younas, editors, *New Horizons in Information Management, 20th British National Conference on Databases, BNCOD 20, Coventry, UK, July 15-17, 2003, Proceedings*, volume 2712, pages 253–260. Springer, 2003.

[76] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification.* The Java™ Series. Addison-Wesley, 1997.

[77] David B. Lomet. Simple, robust and highly concurrent B-trees with node deletion. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 18–28. IEEE Computer Society, 2004.

[78] L. Love. *The Suffix Binary Search Tree.* PhD thesis, Department of Computing Science, University of Glasgow, 2001.

[79] U. Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

[80] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. In *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 319–327. SIAM, 1990.

[81] E.M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[82] B. McLaughlin. *Java & XML*. O'Reilly & Associates, Inc., second edition, 2001.

[83] Jim Melton and Alan R. Simon. *SQL:1999 Understanding Relational Language Components*. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2002.

[84] S. Mittra. *Database Performance Tuning and Optimization: with examples from Oracle 8i*. Springer, 2002.

[85] D.R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[86] G. Myers and R. Durbin. Invited lecture - accelerating smith-waterman searches. In R. Guigo and D. Gusfield, editors, *Algorithms in Bioinformatics. Second International Workshop, WABI 2002, Rome, Italy, September 17-21,2002, Proceedings*, volume 2452 of *Lecture Notes in Computer Science*, pages 331–342. Springer-Verlag, September 2002.

[87] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, volume 1645 of *Lecture Notes in Computer Science*, pages 163–185, 1999.

[88] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[89] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[90] Naresh Neelapala, Romil Mittal, and Jayant R. Haritsa. SPINE: Putting backbone into string indexing. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 325–337. IEEE, 2004.

[91] Oracle Corporation. *Oracle Tuning Pack*, January 2004.

[92] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Software Practice & Experience*, 29(4):345–358, April 1999.

[93] Antonios Printezis. *Management of Long-Running, High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, 2000.

[94] Elzbieta Katarzyna Pustulka-Hunt. *Biological sequence indexing using persistent Java.* PhD thesis, Department of Computing Science, University of Glasgow, 2001.

[95] K. Bernhard Schiefer and Gary Valentin. DB2 universal database performance tuning. *IEEE Data Engineering Bulletin*, 22(2):12–19, 1999.

[96] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[97] Sun Microsystems Inc. *Java™ Object Serialization Specification*, November 1998.

[98] Sun Microsystems Inc. *Java™ Remote Method Invocation Specification*, October 1998. Revision 1.5.

[99] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical suffix tree construction. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 36–47. Morgan Kaufmann, 2004.

[100] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[101] Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242, Padova, Italy, June 1993. Springer.

[102] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[103] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.

[104] D.G. York. The sloan digital sky survey: Technical summary. *Astronomical Journal*, 120(3):1579–1587, 2000.