KALCHAS
a Dynamic Full-Text XML Search Engine

A Thesis Submitted to Aalborg University
in Partial Fulfillment of the Regulations
for the Degree of M.Sc. in Computer Science

by DENNIS ALEXANDER NØRGAARD
RASMUS CHRISTIAN KAAE
DUY THANH NGUYỄN

This document is typeset using LaTeX 2_ε with fonts Times, Platino, Courier, and generated by `latex`, `bibtex`, and `dvipdf`. The figures are drawn using METAPOST, SmartDraw, Xfig, Gnuplot, and the pseudocode is typeset using the `newalg` package.

Faculty of Engineering and Science
Aalborg University

Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg

TITLE: KALCHAS: a Dynamic Full-Text XML Search Engine

PROJECT PERIOD:
February 1^{st}, 2005 –
June 15^{th}, 2005

TERM:
DAT6

PROJECT GROUP:
d635a (room E4-119)

GROUP MEMBERS:
Dennis Alexander Nørgaard
Rasmus Christian Kaae
Duy Thanh Nguyễn

SUPERVISOR:
Albrecht Rudolf Schmidt

NUMBER OF COPIES: 7 (+ 1 online)

REPORT PAGES: 91

APPENDIX PAGES: 8

TOTAL PAGES: 102

SYNOPSIS:

This report documents our work on engineering a dynamic XML full-text index, usable for querying structured XML documents with simple syntax. We focus primarily on designing an index structure feasible for incremental updates while also discussing associated maintenance strategies for efficiently migrating data from an in-memory index into disk based B-trees. The proposed index structure is designed as a series of cascading inverted indexes: One index kept in main memory containing the working set of documents, one incrementally built disk based B-tree containing documents recently ruled out of the former index, and finally one static disk based B-tree containing documents that have remained static in a period long time.

The efficiency of minimizing the amount of data stored in the indexes is researched. We evaluate on various compression schemes in the context of compressing entire inverted lists vs. single postings. In extension, we propose a customized variable byte length encoding scheme for storing Dewey paths efficiently.

We facilitate the concept of *meet* operator in order to filter search results and return the most relevant XML elements. We refine our previously proposed algorithm for the *meet* operator in order to increase the relevance of result sets.

In conclusion we conduct empirical tests, showing that the implemented system performs reasonably within the intended environment.

Preface

This report serves as documentation of the second part of the master thesis project conducted by group D635A (E4-119) during the Spring 2005. The project was developed under supervision by Albrecht Rudolf Schmidt at the Database and Programming Technologies Unit at the Department of Computer Science, Aalborg University, Denmark.

The work presented in this project originates from work by Albrecht Rudolf Schmidt and previous projects conveyed by fellow students at the Database and Programming Technologies Unit. Our primary goal for this project is to implement a working full-text indexing service over XML documents for use in desktops or small networks. We show how the concept of the *meet* operator is used to gather information on how query results could be ranked and displayed.

Citations are marked with square brackets with an integer indicating the actual source which can be looked up in the bibliography section of this report. An electronic version of this thesis is available at `http://www.cs.auc.dk/library/`.

Acknowledgements

First, we would like to thank Albrecht Rudolf Schmidt, Ph.D, Assistant Professor, for his supervision and for providing the initial code base for implementing the *meet* operator.

Moreover, we would also like to acknowledge the creators and the following brands as registered trademarks for using their respective products: Sleepycat Berkeley DB, Microsoft Visual Studio .NET, Microsoft Windows XP, Doxygen, TortoiseCVS, Boost C++, CommonC++, Basic Compression Library, Expat, Apache, PHP, TEX/LATEX 2$_\varepsilon$, SmartDraw, METAPOST, Xfig, and Gnuplot.

Duy Thanh Nguyễn | duy@cs.aau.dk
Rasmus Christian Kaae | kaae@cs.aau.dk
Dennis Alexander Nørgaard | ear@cs.aau.dk

Summary

The amount of data stored on desktop computers is growing each day; therefore, the capability of conveniently searching documents based on their content and getting search results presented at a fine granularity is becoming more important to users. Realizing this fact, many companies such as Copernic, Google, Apple, Microsoft, and Yahoo! have put effort to develop full-text desktop search technologies, and recently they have launched their respective desktop search applications.

To be able to search for every word occurring in the content of each file, a *full-text index* is needed. A full-text index is somewhat like a reference book, where the location of most terms can be looked up. The concept of having a full-text index is taken from information retrieval systems, where applications are built in a client-server environment. In such environments, disk space is not an urgent matter, and thus the disk size of the inverted indexes is often sacrificed in favor of fast query evaluation.

This project focuses on the development of a *dynamic full-text search engine* targeted at desktop computers (and LANs), with particular emphasis on designing an efficient index structure. Working with desktop computers introduces new challenges, as compared to working with server applications which are capable of evaluating millions of queries per second. Some of the most interesting challenges when designing the index structure are: (i) how to efficiently index new documents and how to dynamically update the index to reflect changes on the indexed documents in the index, (ii) how to organize and represent data in the index to reduce its size, and (iii) how to efficiently query XML documents and present query results at a fine granularity.

The aforementioned questions have been addressed in our index structure design by employing a range of techniques to optimize the overall performance of the frequently updated index. One of the employed techniques is *index partitioning*. This is accomplished by having a cascade of three indexes, composed by an in-memory cached index, a small dynamic index and a large static index, instead of a single database index. Doing so, we are able to reflect updated files in-place and, at the same time, avoid frequent full index reconstructions. In-place indexing means whenever new documents are added or existing documents are modified/deleted they will be indexed without re-building the entire index, thus providing up-to-date information access for users. When the cached index becomes full, a batch of documents will be moved to the dynamic index, and the dynamic index will occasionally be merged with the static index. In addition to index partitioning, we also use *caching* and a number of strategies for moving data from the cached index to disk to optimize index updates.

Furthermore, we have utilized *Dewey encoding* and *Variable Byte Length encoding* to encode postings, consisting of terms and their respective location, to be stored in the underlying indexes. Additionally, other codecs like Run Length Encoding, Huffman coding, Rice, Lempel Ziv (LZ77), Burrows-Wheeler Transform and Unary have also been experimented with and compared in order to keep disk requirements at

a minimum.

We have primarily focused on indexing XML documents and other hierarchically structured data. In order to index non-structured data, a set of plugins that extracts meta-data and generates valid XML documents have been implemented.

In this project we want to facilitate *content-based search* on a collection of XML documents with *arbitrary schemas*; *i.e.*, when users formulate queries on XML documents which contain potentially relevant information, they need not to know about the mark-up structure that is used. To support this kind of search, we exploit the hierarchical structure of XML and its inherent fine granularity, so that keyword searches do not always return entire documents, but can return deeply nested XML elements containing the desired keywords.

When presenting query results, a number of challenges arises: (i) how should one result element be ranked over another, (ii) how should a union of two result sets associated with different terms be ranked, and (iii) how should the resulting XML elements be displayed (considering we allow arbitrary schemas), etc. To address these challenges we use *meet* operator which was originally introduced by Schmidt *et al.*.

In conlcussion, tests have been conducted to show the performance and the functionalities of the system developed.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Presenting data in both a human readable and machine interpretable way has been a challenge for developers in decades. As a result of these researches, a wide range of data formats has emerged. The root of most of these human and machine readable formats is the *Standard Generalized Markup Language* (SGML) [1]. Being one of the initial specified formats within its area, SGML was later outpaced by more modern formats such as *Hyper-Text Markup Language* (HTML) [2]. HTML is the original format for exchanging displayable data on the World Wide Web and later has become what common people often call the "Internet". In the recent years, a new format Extensible Markup Language (XML) [3] has emerged. XML is well suited for storing/exchanging database data and other structured data normally used in business applications. In comparison to HTML, XML offers the capability of keeping the presentation part and the data part of a document separated, while HTML merges these into one document.

Traditional database management systems (DBMS) generally have been the choice for persistent storage of structured data. Moreover, traditional DBMSs provide efficient mechanisms for modifying and retrieving data. However, they have a number of drawbacks when it comes to exchange of data as compared to XML. First, the stored data is tightly coupled with the database schemas and data type definitions, rendering data exchange difficult. In XML, on the other hand, this information is self-contained, and the presence of tags makes the documents self-documenting, *i.e.*, a schema need not be consulted to understand the meaning of the text. Second, data exported from a DBMS is often in a proprietary format, whereas the XML format is based on an international standard and is widely supported.

The increasing popularity of XML is partly due to the limitations of the other two major technologies for representing structured and semi-structured documents. HTML provides a fixed, predefined set of tags; these tags are mainly for presentation purposes and do not bear useful semantics. SGML is an international standard for the definition of device- and system-independent methods of representing text in electronic form. SGML differs from HTML in its emphasis on the semantics of document content (with user-definable, self-describing mark-ups) rather than presentation. However, the original SGML specification is too complex to be useful in many commercial applications [4].

XML documents are textual documents both human and machine readable. An XML document has a strict syntax, making parsing simple, and its hierarchical structure makes it suitable for many types of documents. On the downside, the verbose and redundant nature of the XML representation introduces an overhead which makes XML unsuitable as a general storage scheme.

Originally, XML was designed to meet the challenges of large-scale electronic publishing, but it is now also playing an important role in the exchange of a wide variety of data (*e.g.*, text, sound, images, etc.) on the Web. It is particularly useful as a data format when one application needs to communicate with another application, or integrate information from external sources. Since its initial recommendation [5] was published by the W3C in 1998, XML has now become the de facto standard format for web publishing and data transportation. This general acceptance can be attributed to two of the XML's core characteristics, namely flexibility and extensibility. Just as SQL is the dominant *language* for querying relational data, XML is becoming the dominant *format* for data exchange.

Furthermore, researchers in the field reckon that XML potentially will yield (i) more precise search by providing additional information in the elements, (ii) a better integrated search of documents from heterogeneous sources, (iii) a powerful search paradigm using structural as well as content specifications, and (iv) data and information exchange to share resources and to support cooperative search [6]. Interestingly, Microsoft has recently announced that they will adopt the industry-standard XML technology for the default file formats in the next version of Microsoft Office editions (currently code-named Office 12) to "give customers improved data interoperability and dramatically smaller file sizes." [7].

The remainder of this chapter is organized as follows. In the next section we analyze the problem of developing a dynamic XML full-text search engine. In Section 1.2 we formulate project objectives, and finally an outline of the report is given.

1.1 Problem Analysis

The widespread use of XML in digital libraries, product catalogues, scientific data repositories and across the Web arises the need for efficient management and retrieval of XML documents. In order to obtain adequate performance, the XML data need to be organized (indexed) in a way that facilitates efficient retrieval of the desired data from large repositories of XML documents. Without indexes, the database may be forced to conduct a full scan to locate the desired data records, which can be a lengthy and inefficient process. In this project we focus on the development of a *dynamic full-text search engine* targeted at desktop computers, with particular emphasis on designing an efficient index structure.

1.1.1 Full-Text Indexing

At the core of most search engines lies a *full-text index* [8]. In short, a full-text index is an index containing all words occurring in a collection of document, *e.g.*, the World Wide Web, and a reference to each of these occurrences. A full-text index can be compared to the index in a book; however, the latter is not full as the index does not

contain all words written in the book nor is each occurrence of a word referenced. Signature files, forward index and inverted index are three well-known kinds of full-text indexes; however, the underlying index structure for most XML search engines is the inverted index [6, 9] and the technique of choice for most Web search engines [8].

1.1.2 Updating Full-Text Indexes

When operating in a dynamic environment, XML documents are subject to frequent changes. In some applications, documents arrive at a high rate, and even within the context of a single desktop machine, a naive update strategy that consumes lots of CPU cycles and disk accesses would not be of value. Therefore, keeping inverted indexes always up-to-date with the content of documents at the XML element's granularity is a challenge. When designing a dynamic full-text index structure, one must consider the index update problem.

1.1.3 Encoding Data in Full-Text Indexes

Inverted indexes are generally large, ranging between 7% and 300% of the size of the indexed document collection, and often require several gigabyte of storage [10, 11], thus the data stored in inverted indexes must be encoded into a compact representation. In this project, the problem of data encoding and decoding must be considered.

1.1.4 Content-Based Keyword Search

Having data stored in XML documents rather than records in a database requires new approaches for querying. As a result many query systems, such as XQuery [12], XRANK [13], XXL [14] and XQL [15] have been developed. However, like their counterparts in the field of relational databases SQL, these systems are declarative languages relying on data schemas, thus an *a priori* knowledge of the document schemas is necessary in order to perform queries on the XML documents. While this approach ensures accurate results, a search however is restricted to collections of XML documents of the same schema.

Database schemas are used to constrain what and how information can be stored in the database and to constrain the data types of the stored information. In constrast, XML documents by default can be created without any associated schema, that is, an element may have any number (or type) of subelement or attribute. The *document type definition* (DTD) is a schema mechanism, included as part of the XML standard [16]. The DTD, however, is only an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain the information present in the document. However, the DTD in fact does not constrain types in the sense of basic types like integer or string. Instead, it constrains the appearance of subelements and attributes within an element. Document schemas are not our concern as the intention behind our XML search engine is to provide a schema-less querying facility that works solely on the hierarchical structure. This means that we may discard all DTDs and focus on the actual content of the XML documents when processing keyword searches.

Evaluating keyword search queries over hierarchical XML documents introduces

new challenges. First, XML keyword search queries do not always return entire documents, but may return deeply nested XML elements that contain the desired keywords. Second, the nested structure of XML implies that the notion of ranking is no longer at the granularity of a document, but at the granularity of an XML element. Finally, the notion of keyword proximity is quite complex in the hierarchical XML data model [13]. Traditional database methods for searching datasets using full-text indexes have been focusing on finding exact hits, whereas more recent systems (*e.g.*, [13] and [14]) rely on the XML hierarchy in order to find collections of relevant data.

As web search techniques can be employed to handle collections of flat documents, such as HTML, the assumption that they can be applied to a collection of XML documents seems reasonable since the former type can be seen as a special case of the latter. The hierarchical structure of XML documents, however, suggests finer granularity for indexing and retrieval, thus indexing and retrieval could be done on the basis of XML elements rather than documents. Searching the XML documents by translating XML elements into HTML format is not a plausible solution since it imposes a large computation overhead and leaves the process of displaying search results rather difficult [13].

In this project we want to facilitate *content-based keyword search* in a collection of XML documents with *arbitrary* schemas; *i.e.*, when users formulate queries on XML documents which contain potentially relevant information, they need not to know about the mark-up structure that is used. To support this kind of search, we exploit the hierarchical structure of XML and its inherent fine granularity, so that keyword searches do not always return entire documents, but can return deeply nested XML elements containing the desired keywords. In the case of queries of a single term, result extraction from the document collection is easy; however, ranking queries with multiple terms introduces several challenges. These challenges are met in the system to be devoloped by employing the *meet* operator which was originally introduced by Schmidt *et al.* [17, 18].

1.1.5 Retrospective

In our previously conducted work [19] within the field of XML full-text search engine developement, we have experienced a range of critical issues to be addressed in this project. The issues include optimization techniques, conceptual changes, and ideas for extending the search engine. Through the development of a search engine, we will address the following issues:

Optimizations. We want to experiment with *techniques for minimizing the storage space requirements* of our databases by coupling related data originally spanning multiple database records into single database records. More importantly, we want to implement a *caching functionality* to handle the problem of frequently modified files. The caching mechanism should focus on keeping recent files in memory to avoid disk I/O. Accompanying this caching mechanism, we must increase the use of the internal Berkeley DB cache, which is used for building and maintaining disk-based B-trees.

Conceptual changes. We want to redesign the implementation with a *modular structure* in mind. The modular design will make it possible to replace

essential components without loosing functionality. While redesigning the system architecture, *multi-threading* could be introduced in order to process files without interferring the user sitting in front of the PC.

Extensions. The system to be developed is intented for use in desktop environments, thus the need for integrating with third party applications arises. Instead of writing a multitude of scripts interacting with, for instance, MS Office products, we propose to integrate our system directly into the operating system. In addition, we want to develop a plugin structure to allow third party developers to develop file shredders for non-XML documents.

After having analyzed the problem area we are now in a position to formulate the project objectives.

1.2 Project Objectives

Based on the research results conducted in the previous project, this project continues the processs of developing a *dynamic full-text search engine*, called KALCHAS [1], targeted at desktop computers, with particular emphasis on designing an efficient index structure. Furthermore, we want to build from scratch a dynamic full-text search engine, facilitating content-based keyword searching in collections of XML documents. In this context, the system to be developed should address the following issues:

1. **Architecture design.** When designing the system architecture, the emphasis should be put on high modularity and extensibility. Further, the system should be made embeddable in other applications, accomplished by providing a simple API to allow easy integation for third party applications.

2. **Index structure.** When designing a dynamic full-text index structure, the focus should primarly be put on handling updates of indexed files in an efficient way. Moreover, the system should be able to handle both frequently modified files as well as static files.

3. **Index compression.** In order to minimize the data stored in the indexes, we want to experiment with different compression schemes to find the best candidate.

4. **Keyword search.** Our system is intended for searching in medium-sized collections of XML documents residing on typical desktop PCs with arbitrary schemas; that is, when users search for XML documents containing potentially relevant information, they need not to know about the mark-up structure used. To facilitate this kind of search and, moreover, to obtain results at the element granularity offered by the structure of the XML format, we will utilize the *meet* operator proposed by Schmidt *et al.* [17, 18].

In addition, we will reconsider and address the issues mentioned previously in Section 1.1.5.

[1] http://kalchas.dk/

1.3 Thesis Outline

The remainder of the report is organized as follows:

Preliminaries (Chapter 2) provides the reader a basic technical insight of the problem domain by introducing terminologies.

Dewey Encoding and Compression (Chapter 3) first introduces a method for capturing the hierarchical structure of XML documents. Afterwards, we describe the encoding scheme that is used and other codecs that have been experimented with.

***meet* Operator** (Chapter 4) describes the *meet* operator that will be used for content-based keyword searches.

System Architecture (Chapter 5) describes the overall system architecture of KALCHAS. Further, we describe how to embed KALCHAS in applications, and we present two applications having embedded.

Index Structures (Chapter 6) describes the structures used for storing inverted indexes and techniques used to maintain these.

Supported Operations (Chapter 7) describes the operations provided by the KALCHAS API. Here, we explain in details how the operations have been designed and implemented.

Tests and Evaluation (Chapter 8) describes the tests of the system, especially its performance. At the same time, we evaluate the tests of our system.

Conclusions (Chapter 9) concludes the report and evaluates our work. Furthermore, we will discuss topics to be investigated in the future work.

Chapter 2

Preliminaries

This chapter is intended to provide the reader with a basic technical insight into the problem domain by presenting terminologies and, at the same time, discuss some of the previous work directly related to ours. First, we present the XML data model (Section 2.1). Second, we explain how a full-text index building process is executed (Section 2.2). Finally, we discuss how to efficiently store the inverted indexes (Section 2.3).

2.1 XML Data Model

An *XML document* is a hierarchical structure consisting of nested elements which can be assigned attributes and values. All the XML documents start with a single root element that may contain any number of nested elements; however, elements cannot be interleaved. In addition, elements may contain data and must be delimited by a start and an end tag. Attributes, however, can only contain data; *i.e.*, they cannot contain elements nor have attributes. Child elements (or subelements) of a parent element are ordered whereas its attributes are not ordered. For a formal definition of the XML data model, the reader is referred to the XML specification [20] which describes precisely the XML data model through the XML language grammar.

Listing 2.1 shows an example XML document representing a small collection of scientific papers. The `<bibliography>` element is the root element, and it has `<paper>`, `<title>`, `<author>`, and `<year>` subelements nested under it. In this project we do not consider pointers pointing to internal/external XML elements and external XML documents.

By convention an XML document is modeled as a *tree* with a single root, where each element is mapped into a node, thus called a *document tree*. Textual content is assumed to be only in the leaf nodes, and internal nodes represent the structural relationships between elements. We formally define an XML document tree as follows:

```
<bibliography>
  <paper>
    <title>The Anatomy of a Large-Scale Hypertextual
    Web Search Engine</title>
    <author>Sergey Brin, Lawrence Page</author>
    <year>2000</year>
  </paper>
  ...
  <paper>
    <title>Building a Distributed Full-Text Index
    for the Web</title>
    <author>Sergey Melnik, Sriram Raghavan,
    Beverly Yang, Hector Garcia-Molina</author>
    <year>2001</year>
  </paper>
</bibliography>
```

Listing 2.1 : A structured XML document

Definition 2.1 (XML Document Tree). An XML document can be represented as a rooted tree $T = (V, E, r)$ where

- $V \equiv V_{ele} \cup V_{val}$ is a set of nodes represented in the XML documents. V_{val} are *value nodes* containing actual data, and V_{ele} are *element nodes* (or structural nodes), *i.e.*, nodes containing nested nodes but no data.
- $r \in V$ is the *root node* of the document, and
- $E \equiv \{(v_i, v_j) | v_i, v_j \in V \wedge parent(v_j) = v_i\}$.

Figure 2.1 depicts the document tree of Listing 2.1. Here, the nodes `<bibliography>` and `<paper>` are the element nodes, whereas `<title>`, `<author>`, and `<year>` are the value nodes. Throughout this report the terms "element" and "node" will be used interchangeably.

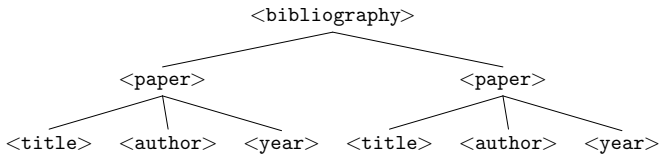

Figure 2.1 : Document tree of Listing 2.1

The HTML data model can also be perceived as a special version the XML data model: An HTML document is an XML document with only two value nodes $V'_{val} = \{head, body\}$, one element node $V'_{ele} = \{xhtml\}$ and two edges $E' = \{(xhtml, head), (xhtml, body)\}$, and its document tree is $T' = (V', E', xhtml)$.

Furthermore, a collection of XML documents can be defined as follows:

Definition 2.2 (XML Collections). A collection of XML documents is a forest, *i.e.*, an unordered set of XML document trees.

2.2 Building Full-Text Indexes

In the spirit of other search engines we choose to employ inverted index as the underlying index structure for our XML search engine. An *inverted index* is a collection of inverted lists, where each list is associated with a particular term. An *inverted list* for a given term is a collection of Unified Resource Identifiers (URI) [21] of those documents that contain that term. If the position of a term occurrence in a document is needed, each entry in the inverted list also contains a location offset. Positional information of terms is needed for proximity queries and query result ranking, and omitting this information in the inverted index imposes limitations [22]. An entry in an inverted list is also called a *posting*, and as a minimum it encodes the `(term,location)` information where `location` is a URI. We illustrate the structure of an inverted index in Figure 2.2, and the definition of these terms is shown Definition 2.3 and is used seamless in this report. A survey on indexing of XML documents can be found in [6].

Term	Inverted list
$term_1$	$loc \ldots, loc, loc$
$term_2$	$loc, loc, \ldots loc$
$\vdots$	$\vdots$
$term_{n-1}$	$loc, loc, \ldots$
$term_n$	$loc, loc, \ldots$

Figure 2.2 : Inverted index

Definition 2.3 (Terminology of inverted indexes). The following definitions will be used throughout the remainder of this report:

Term is a single word defined as a sequence of alpha-numerical characters. In full-text index almost all terms are subject to be indexed, except a set of trivial terms, *e.g.*, "the", "and", "or", "but", etc. They are often referred to as *stop words*.

Location is used to describe the position of a specific term.

Posting is a pair `(term,location)` describing a single occurrence of a term.

Inverted list of a given term is a list of locations describing all the occurrences of that term.

Inverted index is a collection of inverted lists, where each list is associated with a particular term.

Building an inverted index generally happens as follows. Given a collection of XML documents to be indexed, the parser scans one document at a time in order to strip all the metadata, such as XML tags, and extracts terms from the document to produce a set of `(term,location)` postings. After that, the `(term,location)` postings will be passed to the indexer which then inserts these postings into an inverted index. This process is repeated until the entire collection has been indexed as illustrated in Figure 2.3.

Figure 2.3 : The process of building an inverted index

2.3 Persistent Storage

Inverted indexes are generally large, ranging between 7% and 300% of the size of the indexed document collection, and often require several terabyte of storage [10, 11]. Due to the massive storage requirements, inverted indexes do not usually fit into main memory, but are kept partly on secondary storage. Additionally, building an inverted index is generally rather resource expensive in terms of CPU and disk usage. In this section we discuss the storage scheme that has been employed to store the inverted indexes.

When opting for a persistent storage system, one of two extremes can be taken; either implementing a special purpose, custom made system or using an existing database management system (DBMS). Using an existing DBMS has the advantage of making it possible to leverage on high level query languages, well-established data models, and advanced logical storage schemes; however, it has the downside of having a large footprint in terms of resource usage. A running DBMS often consists of a single process and multiple threads which are partly independent of the applications accessing the DBMS. In order to handle queries expressed in high-level languages (*e.g.*, SQL), the DBMS applies advanced techniques such as query parsing and query optimization. Custom implementations, on the other hand, can be tailored, making it possible to apply very specific optimizations; however, this approach may increase the overall complexity of the system as well as development time [23].

Somewhere in between existing DBMSs and specialized custom implementations lie *embedded databases* such as Berkeley DB [24]. An embedded database is a library that links directly into the application providing basic database functionalities, and as a result both database and application run in the same address space, avoiding network and inter-process communication [23].

Constructing databases for use in Berkeley DB requires low-level information of the database store, since Berkeley DB does not employ schemas. Instead, it only supports records consisting of `(key,data)` pairs, and compared to most database systems only a few simple operations are offered, namely insert, delete, retrieve and update. However, due to the nature of Berkeley DB, the fields in records can contain any type of low-level data, such as C structs, making it possible to store more advanced data structures. Additionally, both `key` and `data` fields can be of variable length, rendering efficient use of secondary storage possible. As Berkeley DB does not make use of schemas, it is unaware of the type and structure of data stored in records; it simply recognizes keys and considers data fields as simple payload. As a result, the stored data is tightly coupled with the application creating and accessing it. Additionally, since Berkeley DB does not offer any interfaces to the end user, all functionalities related to inserting, deleting and updating data on the logical level must be implemented by an application programmer, making the database an obvious

choice in scenarios where only a few specialized and predictable queries are needed, *e.g.* simple key lookups.

Chapter 3

Dewey Encoding and Compression

In order to represent the exact location of a term, we need to define a way of serializing XML documents. Several approaches suiting this purpose already exists (see [6]). One way of defining the location of a term is on a per-document basis, treating XML documents as flat documents and disregarding the hierarchical structure of XML; this method is often used in Web search engines such as Google [8]. Another approach is to treat each XML document as a document tree as defined in Definition 2.1 on page 8. However, [6, 13] mention that the perception of XML documents being trees is not entirely correct, since facilities such as XPath, XLink and XPointer introduce graph-like scenarios.

In this project, we have chosen the latter approach and utilized a so-called *Dewey encoding* which is a node ordering method used to serialize the hierarchical structure of XML documents. Using this method we are able to specify locations in a flat format without loosing structural information. Our use of Dewey encoding is described in detail in Section 3.1.

As a result of Dewey encoding, a set of Dewey paths is generated. Dewey paths make up the largest part of inverted indexes; thus it is desirable to represent them in a compact way. In addition to the reduced space utilization, using an efficient compression scheme results in faster processing time when storing and retrieving postings in the indexes [25]. Compression of Dewey paths is described in details in Section 3.2.

3.1 Dewey Encoding

Dewey encoding [26] is a node ordering method used to serialize the hierarchical structure of XML documents. This is accomplished by encoding each node's position in an XML document as a data value; each node is assigned a vector representing the *Dewey path* from the document's root to the node. We distinguish between "Dewey number" being a single integer value assigned to a specific node, and "Dewey path"

being a specific path from the document root to a given element.

We formally define Dewey paths as follows:

Definition 3.1 (Dewey Paths). Given a function $nid : V \mapsto \mathbb{N}$ to generate a Dewey number for every node. For any node $v \in V$, let $path(v) = \langle nid(r), \ldots, nid(v) \rangle$ be the Dewey path of v, where $\langle r, \ldots, v \rangle$ is the sequence of nodes found on the unique path from the document root r to v.

In this project Dewey paths are used as URI to describe `location` of the `(term,location)` postings, where `location` uniquely identifies the absolute location of a specific XML element in which `term` occurs.

Figure 3.1 shows a document tree of the sample XML document (from Listing 2.1) labelled with node position values, and Table 3.1 enumerates all the Dewey paths of leaf nodes in a flat representation form. Such a representation of Dewey paths is fairly easy to store since each Dewey number is local to its parent, thus the Dewey number values are normally quite small and need only few bytes to be represented. The use of Dewey paths is crucial in our project; without Dewey paths we would not be able to translate the hierarchy of XML documents into a flat format ready for storing in a database.

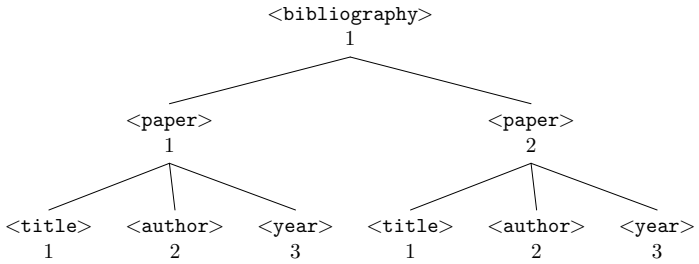

Figure 3.1 : A labelled document tree of Listing 2.1

Dewey path	Corresponding leaf node
/1/1/1	<title>
/1/1/2	<author>
/1/1/3	<year>
/1/2/1	<title>
/1/2/2	<author>
/1/2/3	<year>

Table 3.1 : Dewey paths of the labelled document tree

There are two major schemes to assign Dewey numbers to nodes: global ordering scheme and local ordering scheme. In the *global ordering scheme*, each element is assigned a globally unique Dewey number (see Figure 3.2). As opposed to this scheme, in *local ordering scheme*, only siblings should have unique Dewey numbers, thus quite a few elements could have the same Dewey number (see Figure 3.1). The former one has the drawback of requiring more storage space than the later one, because

in the global ordering scheme the value of assigned Dewey numbers may be very large. For this reason we employ the local ordering scheme. Despite each individual Dewey number assigned to the elements is non-unique, the calculated Dewey paths are always unique, *i.e.*, no two elements in the same document have the same Dewey path. In addition, Zobel *et al.* [27] claim that using the local ordering scheme indirectly improves the overall compression rate of succeeding compression codecs.

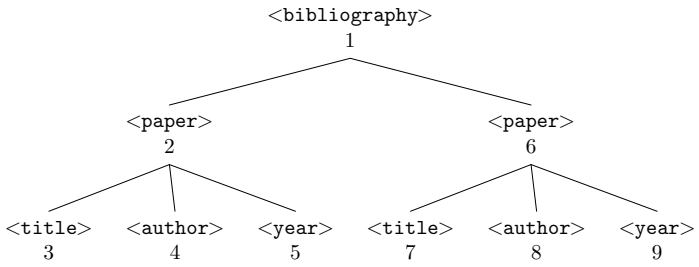

Figure 3.2 : Document tree of Listing 2.1 using a Dewey global ordering

Computing Dewey paths according to the local ordering scheme is carried out as follows. Since XML elements do not overlap, a stack structure can be used to keep track of them and compute current Dewey path. Once an XML start tag is encountered, the Dewey number stored in the top element is incremented by one and a new element is pushed onto the stack. Once an XML end tag is encountered, the top element is simply popped. At any given time, the Dewey numbers stored on the stack represent the Dewey path for the current XML element. When encountering a new start tag, a new Dewey number must be pushed onto the stack. In our implementation, computing Dewey paths is done at the same time as document shredding proceeds. For more information about computing Dewey paths the reader is referred to Section 7.2.1.1.

3.2 Compression

Once the Dewey paths are computed they are meant to be stored in either main memory or on disk. In the first case, compression is not an issue since we only store small amounts of data. However, the data stored on disk needs to be compressed efficiently in order to save disk space and thus increase the overall performance of the system.

Looking at the integral values of Dewey numbers within Dewey paths, we observe that the range of values decreases proportionally with the nesting level in the XML documents. This means that a Dewey number associated with a deeply nested XML element will be assigned a lower value than XML elements on higher nesting levels. Observing Listing 2.1, we see that (i) document ID of the XML document may range very high value in accordance with the number of indexed documents in the collection, (ii) a high number of `<paper>` nodes will result in high Dewey numbers, and (iii) inside a `<paper>` element the number of children decreases. This behavior is caused by the data-centric definition [28] of the XML file shown in Listing 2.1.

Reflecting on this we see that Dewey paths are generally made up of few large Dewey numbers and many small Dewey numbers. In order to compress these Dewey paths efficiently we must then be able to handle both high and low integral values.

The most time and space efficient way of doing so is by using a Variable Byte Length (VBL) [25, 29] compression scheme; this has been tested and is described in Section 8.5 on page 77.

Compressing inverted indexes is a well-investigated field, covered by many conference papers. However, most of the approaches described in [29, 30, 31] are targeted at collections of flat documents. The codecs[1] described in these papers rely on the ability to sort locations in a monotonically increasing order, which is straightforward since their locations are described using a single integer value, but not a sequence as in KALCHAS.

3.2.1 Variable Byte Length Codec

Our variant of a *Variable Byte Length* (VBL) codec is built on the idea of dividing a data value into a minimum of bytes, such that a minimum number of bytes will be allocated for representing a specific data value. Figure 3.3 illustrates that in our implementation, for each byte, the first 6 bits are used to represent *data value* (indicated by D) and the last 2 bits as *signal bits* (indicated by E and F). Here, E is used to indicate if the current byte is the last byte used to serially encode a Dewey number, and F is used to indicate if the current Dewey number is the ending entry of a Dewey path. The value ranges of Dewey numbers mapped to the size in bytes is shown in Table 3.2.

D	D	D	D	D	D	E	F
1	2	3	4	5	6	7	8

Figure 3.3 : Byte format: the first 6 bits used to represent data values, the last 2 bits used as signal fields

Min. value	Max. value	No. bytes needed
0	63 ($2^6 - 1$)	1 Byte
64	4.095 ($2^{12} - 1$)	2 Bytes
4.096	262.143 ($2^{18} - 1$)	3 Bytes
262.144	16.777.216 ($2^{24} - 1$)	4 Bytes

Table 3.2 : VBL byte sizes

Example 1. Given a Dewey path `/5000/700/63/` consisting of three Dewey numbers, 6 bytes must be allocated to represent the Dewey path using VBL encoding, because three bytes are needed to encode 5000, two bytes to encode 700, and one byte to encode 63. Now we analyze the value of signal bits in the six allocated bytes:

Byte 1–3. E is not set (*i.e.* 0) in the first and second bytes, but is set (*i.e.* 1) in the third byte to indicate that the current byte is the last byte used to serially encode Dewey number 5000. F is 0 in all of these bytes because 5000 is not the last entry of the Dewey path.

Byte 4–5. E is 0 in byte 4, but 1 in byte 5. F is 0 in all of these bytes because 700 is not the last entry of the Dewey path.

[1] A codec is a set of routines for compression and decompression.

Byte 6. Both E and F is 1, because byte 6 is the last byte used to represent 63, and the Dewey number 63 is also the last entry of the Dewey path.

The VBL codec is a *byte wise* codec which encodes in an 8 bit aligned fashion, making encoding and decoding optimized for standard CPUs. Other means of codecs relies on *bit wise* encoding and decoding, resulting in better compression ratios and longer execution times. The slow down of bit wise codecs is caused by missing alignments, since standard CPUs read data in 8-bit aligned chunks (8 bits, 16 bits, 32 bits, etc.) [25].

3.2.1.1 VBL Encoding

Pseudocode for encoding one Dewey number using the VBL codec is given in Figure 3.4. Encoding whole Dewey paths is done by sequentially encoding each Dewey number in the Dewey path and appending the data to the output data stream. The parameter n is the Dewey number to encode, and $LastDeweyNumber$ is a Boolean value indicating if n is the final Dewey number in the Dewey path. Notice that r is the output data stream, and $+$ is an overloaded operator used to append data. The $\gg$ operator is an ordinary right shift, and we access bits as vectors, *i.e.* referring to the 6 lower bits of v is written $v[1..6]$ and a single bit is referred to as $v[i]$, $i \in [1..8]$.

```
VBL-ENCODE(n, LastDeweyNumber)
 1  r ← ∅
 2  v ← n
 3  while (v ≥ 2^6 − 1) do
 4  r ← r + (v[1..6])
 5  v ← v ≫ 6
 6  if (LastDeweyNumber = true) then
 7     v[8] ← 1
 8  v[7] ← 1
 9  r ← r + v
10  return r
```

Figure 3.4 : VBL encoding

3.2.1.2 VBL Decoding

Pseudocode for decoding one encoded Dewey number using the VBL decoding is given in Figure 3.5. Decoding of a whole Dewey path is done by sequentially decoding each Dewey number in the Dewey path, and the VBL-DECODE function will report when the end of a Dewey path has been reached.

```
VBL-DECODE(b, i)
1  r ← 0
2  s ← 0
3  while (b_i[7] ≠ 1 AND b_i[8] ≠ 1) do
4  r ← r + (b_i ≪ s)
5  s ← s + 6
6  i ← i + 1
7  r ← r + (b_i ≪ s)
8  i ← i + 1
9  return (r, b_i[8] = 1)
```

Figure 3.5 : VBL decoding

3.2.2 Other Codecs

In addition to VBL, we will now introduce a set of compression and decompression schemes utilized in this project. In the tradition of other parts of the implementation, we have included an external library, the Basic Compression Library (BCL) [32], implementing a number of common codecs, such as *Run Length Encoding*, *Huffman*, *Rice* and *Lempel Ziv*. These codecs are all standard compression codecs. In addition to the BCL, we have implemented two customized codecs: *Burrows-Wheeler Transform* [33] codec, and *Unary* codec. All of the above codecs are lossless[2] and thus no information is lost in the process. In the following we will briefly discuss the differences between the mentioned compression schemes:

Run Length Encoding (RLE). The RLE codec is a general purpose codec used *e.g.* in JPEG compression. RLE encoding is done by reducing sequences of repeating values into a single value and a number indicating the length of the original sequence. RLE is efficient in cases where repeating sequences are long, this is often the case in very deep XML documents where each XML node contains very few children.

Huffman. The Huffman codec is one of the most common codecs and is more advanced than RLE. The basic principle behind Huffman compression is to perform a statistical analysis of the uncompressed data and then represent common values with a low amount of bits and not so common values with a high amount of bits. In order to decode the compressed data the Huffman codec stores a symbol tree, built by generating a histogram for all data values in the uncompressed data set. In the case of the BCL implementation of the Huffman codec, the tree is at most 384 bytes in the compressed data. The size of the tree is crucial for choosing when to use this codec, *e.g.* Huffman is not suited for data less than 500 bytes.

Rice. The Rice codec is much similar to the Huffman codec and thus tries to fit as many words into as few bits as possible. Unlike the Huffman codec the Rice codec will not build an expensive histogram for each value, but instead try to encode

[2] http://en.wikipedia.org/wiki/Category:Lossless_compression_algorithms

lower values with fewer bits and higher values with "enough" bits. This can be seen as a static Huffman tree. In comparison to the Huffman codec the main advantage of Rice is that Rice uses a static tree where Huffman builds a dynamic tree (which is expensive to build in some cases). In relation to Dewey paths, and the way KALCHAS generates them, the theory dictates that Rice would be well-suited since most Dewey paths will have all Dewey numbers to be relatively low, except for the document ID at the beginning of the Dewey path.

Lempel Ziv (LZ77). Along with the Huffman codec, the LZ77 codec is one of the most used codecs. The implementation provided by BCL is based on the original Lempel Ziv 1977 implementation. LZ77 is an extension to the very simple RLE codec and thus seeks to replace sequences of repeating values with fewer bits. In addition to the RLE codec, LZ77 references previous sequences of the same repeating values and thus is able to compress better than RLE in some cases.

Burrows-Wheeler Transform (BWT). The BWT codec is also known as a "block sorting algorithm". BWT works by performing a number of arithmetic operations leaving data in a more structured order. Encoding of data is done by first writing down all possible permutations and subsequently sorting the permutations lexically. This results in a matrix where the right most column comprises the final BWT code. Decoding is done by iteratively inserting the encoded data as the left most column of the matrix while sorting. After the final iteration the decoded data is located at the column starting and ending with the allocated delimiter symbol. As can be read from above this algorithm improves the overall compression ratio, however it also introduces a noticeable penalty in terms of execution time.

Unary. Unary coding is a way of representing non negative integrals by their ordinal values, *i.e.* the number 4 is represented as 1111. Representing values by their unary coding makes it easier for succeeding codecs to compress the data since the codes are highly redundant. A drawback of the unary coding is that high values consumes large amounts of storage, *i.e.* 2^{32} is represented by 2^{32} bits amounting to a total $536.870.912$ bytes while the same value easily could be represented by a 32 bit integer consuming a mere 4 bytes of memory.

The API documentation of BCL suggests to try out variations of sequential compression, *e.g.* first encode using RLE, then LZ77 and finally Huffman. In Section 8.5 on page 77 we will test and evaluate which of the above codecs, and sequences of codecs should be used in KALCHAS for optimal performance. An important fact about the above codecs is that they are all efficient at decoding which is crucial for the query performance [29, 30].

3.3 Summary

Dewey encoding is a generic way of serializing hierarchically structured data. We consider two types of Dewey encoding, namely the global ordering and the local ordering. The latter has been chosen in order to reduce the integral values of the generated Dewey numbers, and thus increasing the performance of the subsequent compression.

Compression of Dewey numbers is divided in to two special cases, namely the case of single postings (as seen in DI) and the case of compressing lists of postings (as seen in SI). Compressing single postings needs specially designed algorithms for encoding and decoding (such as the customized VBL codec) while lists of postings hold more data, which make it feasible to apply more generic compression schemes, such as Huffman, LZ77, and RLE. Test and evaluation of the aforementioned compression schemes are shown in Section 8.5 on page 77.

Chapter 4

Meet Operator

This chapter will define the concept of "meet" (Section 4.1) and provide algorithms for meet computation (Sections 4.2–4.3).

4.1 Definitions

Having the basic concepts of the XML data model (Section 2.1) and Dewey paths (Section 3.1) defined, we are now in a position to introduce the *meet* operator, which is the underlying algorithm of our keyword search processing. The idea of the *meet* operator was originally introduced by Schmidt *et al.* [17, 18]. Basically, the *meet* operator is a graph function that operates on a set of Dewey paths to compute the most "interesting" node containing specific search terms (keywords). Working with XML elements represented as nodes in the XML document tree, we want to rank specific nodes by their relevance. Given a non-empty set of nodes, the most interesting node in the set is the node shared by the majority of nodes in the set. Having paths in the document trees defined using Dewey paths we see that finding the union of Dewey paths is a special case of the more general problem of computing "longest common prefixes".

Before defining the concept of "meet" we now define the concept of "longest common prefixes".

Definition 4.1 (Longest Common Prefixes). Given two Dewey paths $p = [d_0, \ldots, d_n]$ and $p' = [d'_0, \ldots, d'_m]$, the prefix function *lcp* computes the *longest common prefix* of p, p' as follows:

$$lcp(p, p') = [d_0, \ldots, d_l],$$

where $l = max\{i | (d_0 = d'_0) \wedge (d_1 = d'_1) \wedge \ldots \wedge (d_i = d'_i), 0 \leq i \leq min(n, m)\}$.

Based on the above definition, the concept of "meet" is defined as follows:

Definition 4.2 (The Concept of "Meet"). Given two Dewey paths $p = path(v)$ and $p' = path(v')$, $v, v' \in V$, the *meet* of p, p' is the longest common prefix of p, p'; that is,

$$meet(p, p') = lcp(path(v), path(v')).$$

As can be seen from Definition 4.2, the *meet* operator computes the *longest common prefix* between two Dewey paths. Translating this problem into the domain of XML document trees, we find that the *meet* operator calculates the *lowest common ancestor*.

Ranking in the *meet* operator is executed on the theory that a deeply nested node containing many search terms is more important than a deeply nested node containing few search terms. This idea of ranking using the concept of the "lowest common ancestor" is also conducted in XRANK [13]. However, most traditional indexes provide ranking by means of global statistics for the index (*i.e.* IDF). In fact, Melnik *et al.* [10] argue that it is a necessity in any modern inverted index. Our tests, and the following example, demonstrate that ranking based on the functionality of the *meet* operator and the structural hierarchy of the XML documents suffices and provides accurate results.

Example 2. We search for the terms "Engine" and "Sergey" in an inverted index of the document shown in Listing 2.1. Initially, the index will return exact match on all search terms as postings, *i.e.* `<Engine,/0/0/0>` and `<Sergey,/0/0/1>`. Computing the *longest common prefix* of Dewey paths `/0/0/0` and `/0/0/1` we get `/0/0`. Therefore, the returned result is the `<article>` element (left sub-tree), since `<article>` is associated with the Dewey path `/0/0` and `<article>` is the deepest node in the document tree that contains both search terms. Here, the `<article>` element is said to be the *lowest common ancestor*.

Note that the `/` character is only used to separate the Dewey numbers from each other when they need to be displayed. Internally, in the implementation Dewey paths are represented as a vector of *unsigned integers*. While defining the *meet* terminologies above we implicitly introduced what this project defines as results from our *meet* operator.

4.2 Naïve Algorithm

Meets are computed by means of the *meet* operator. Finding meets using the *meet* operator actually amounts to finding the longest common prefixes of the input Dewey paths set, as stated previously. The problem of finding longest common prefixes [34, 35] is a general class of problem, thus having numerous ways of solving. In the DAT5 project, we have evaluated three different implementations of the *meet* operator (based on the naïve algorithm, graph-based algorithm, and line-based algorithm) [19, Chapter 3]. Previous tests have additionally shown that the line-based (now called "scan-based") algorithm is the most efficient one. To introduce the reader to the implementation of the *meet* operator we have included both the Naïve Algorithm and the optimized Scan-Based Algorithm.

One way to implement the *meet* operator is to compute "all" meets of an input set of Dewey paths, select and output only the most important nodes. The pseudocode of this

algorithm is shown in Definition 4.3. MEET-TWO takes as input two Dewey paths p, p' and outputs the longest common prefix of p, p' as defined in Definition 4.1 on page 20. MEET-SET takes as input a set of Dewey paths P, builds all possible combinations of meets using MEET-TWO and outputs them. If the number of input Dewey paths is n, the number of meets returned is n^2.

Definition 4.3 (Initial *meet* algorithm). Given two Dewey paths p_i and p_j we define the *meet* between them as follows:

- MEET-TWO(p_i, p_j) = *lcp*(p_i, p_j)

Given a set of Dewey paths P we define the *meet* of the set as follows:

- MEET-SET(P) = {MEET-TWO$(p_i, p_j)|p_i, p_j \in P$}

Given a set of Dewey paths P and a constant k we define the *meet* of P with relevance k as follows:

- MEET-K(P, k) = $\{p|(p \in$ MEET-SET$(P) \wedge$ *length*$[p] > k\}$

To prevent the document ID of XML documents to be returned as actual meets, we have implemented the MEET-K procedure to only return meets whose depth is greater than k. Selecting which "meets" should be returned in the output set is done by a simple selection rule $length[p] > k$. Due to the encoding format of Dewey paths described in Section 3.1 and Section 7.2.1.2 we define $k = 1$.

The time complexity of MEET-TWO is $O(l)$ where $l = min(length[p], length[p'])$ since comparing two Dewey paths p, p' takes linear time. Computation of MEET-SET takes $O(hn^2)$ because each Dewey path of the input set must be compared with each other, hence n^2, and $h = max\{length[p]|p \in P\}$ is the generic upper bound on all individual path comparisons.

This approach is rather naïve and not suitable for practical purposes since its time complexity is quadratic and its memory usage is linear in the size of the input. To compute meets, all nodes in P need to be loaded into main memory and compared with each other to find all meets, thus resulting in a huge amount of duplicates in the result set and rendering performance penalty. In addition to the inefficient memory utilization, no ranking is performed on the result set.

4.3 Scan-Based Algorithm

To eliminate the shortages of the algorithm presented above, we introduce another algorithm that is able to perform ranking and, at the same time, compute meets. However, before presenting the scan-based algorithm we need to discuss the problem of ranking.

4.3.1 Ranking Search Results

After being computed, the query result set must be ranked and displayed in a way that is relevant to users. Ranking is not within the scope of this project, however, we will

now try to outline some of the general ideas of ranking. When displaying the result set we want: (i) all hits in the result set must be *relevant* to the query at hand, and (ii) the search result must be displayed in an understandable and user-friendly way [36]. To meet these requirements we opted to rank XML elements of the result set according to the principle shown in Definition 4.4.

Definition 4.4 (Ranking principles). The ranking functionality embodied by the *meet* operator complies with the following rules:

Specific nodes first. Rank deepest nodes in document higher than nodes higher up in the document tree since in most cases they are more specific.

Short distance first. Rank XML elements by the node proximity (*i.e.*, the *distance* in the document).

The concept of "proximity" of keywords is defined in Definition 4.5.

Definition 4.5 (Node Proximity). Proximity of two nodes $prox(v, v')$ in the tree is defined as the length of the path between the nodes through their lowest common ancestor c:

$$prox(v, v') = (length[v] - length[c]) + (length[v'] - length[c]),$$

where $length[v_i]$ returns the number of edges found on the path from the root r to node v_i.

For more advanced ranking the reader is referred to [8, 13]. In the extreme case we would have to implement a static rank analyzer, as proposed in the XRANK system [13].

4.3.2 Scan-Based Meet Algorithm

Since the submission of [19] the need for a better implementation of the *meet* operator has shown. In result of this we have fine-tuned the ranking mechanism, which calculated real time, for better query results. The new algorithm is shown in Figure 4.1. MEET-SCAN takes as input a set of postings ordered by Dewey paths (cf. Figure 7.7). The algorithm visits each posting exactly once, comparing two Dewey paths with each other to compute the current meet and outputting whenever one of the following rules is satisfied:

Rule 1: Output the current meet if the Dewey path loaded from the input set originates from another document.

Rule 2: Output the current meet if the meet between p and p' has a depth shorter than k. This happens when two nodes have nothing else in common than the root and the document ID.

Rule 3: To provide basic ranking we calculate how many entities from the input set are intersected by the current meet (v), this is done by increasing the $hitcounter$ component of v. If the term of the newly loaded Dewey path is not already contained in v we increase the $hitcounter$ by 10 otherwise we increase by 1.

```
MEET-SCAN(P)
 1  R ← ∅
 2  v ← P.PeekTop()
 3  while (P ≠ ∅) do
 4  u ← P.PopTop()
 5  d ← prox(U, V)
 6  v ← lcp(U, V)
 7  Rule 1:
 8  if (d = −∞) then
 9     R ← R ∧ {v}
10     v ← u
11  Rule 2:
12  if (d < k) then
13     R ← R ∪ {v}
14     v ← u
15  Rule 3:
16  if (d ≥ k) then
17     if (v_term ∩ u_term = ∅) then
18        v_term ← v_term ∪ u_term
19        v_rank ← v_rank + 10
20  if (v_term ∩ u_term ≠ ∅) then
21     v_rank ← v_rank + 1
22  Rule 4:
23  if (v_rank ≥ 10) then
24     R ← R ∪ {v}
25     v ← u
26  return R
```

Figure 4.1 : Scan-based *meet* algorithm

This is done to rank nodes that contain many terms higher than nodes containing few terms.

Rule 4: When we have a sufficient amount of elements of the current meet (v) we output it and promote u to our new working meet.

Figure 4.1 shows the pseudocode for the revised implementation of the *meet* operator. The algorithm works by iterating through the sorted input set P until reaching the end (lines 3–25). During each iteration, the next posting is read (u) and a temporary *meet* (v) is calculated. Whenever one of the rules applies to the temporary *meet* (v), v is added to the result set R. After adding v to the result set, we copy the value of u into v and the iteration continues. By computing *meet* in this way, we incrementally identify relevant elements, by traversing from the most specific elements (those returned by the inverted index) up to more generally elements (containing multiple occurrences of terms). Source code for the MEET-SCAN *meet* operator is given in Listing A.3 on page 94.

The time complexity of MEET-SCAN is $O(n)$ where n is the number of elements in the input set.

4.4 Summary

The *meet* operator is a graph function intended for finding the "most relevant" nodes within a given set of Dewey paths. In theory, it works by calculating the *lowest common ancestor*. In the context of XML, the lowest common ancestor between two XML elements is promoted as being more relevant than the two XML elements alone. Computing the *meet* is done using our proposed MEET-SCAN algorithm. In addition to finding the lowest common ancestors of a set of XML elements, ranking the search results according to the computed relevance with respect to the user-specified query has also been incorporated in the *meet* operator.

Chapter 5

System Architecture

This chapter primarily focuses on the design principles behind the KALCHAS architecture. First, we describe the overall system architecture of KALCHAS (Section 5.1). Second, we describe how to embed KALCHAS in applications (Section 5.2), and we describe two applications having KALCHAS embedded (Section 5.3). Finally, we describe how developers can create plugins to enable KALCHAS to support specific file formats (Section 5.4).

5.1 KALCHAS Architecture

The design of the KALCHAS architecture has laid stress on high modularity and extensibility. This has resulted in a layered system design as shown in Figure 5.1. This shows that the system is divided into three layers, each layer unaware of the layer on top of it. The layering is reflected directly in the source code as individual namespaces.

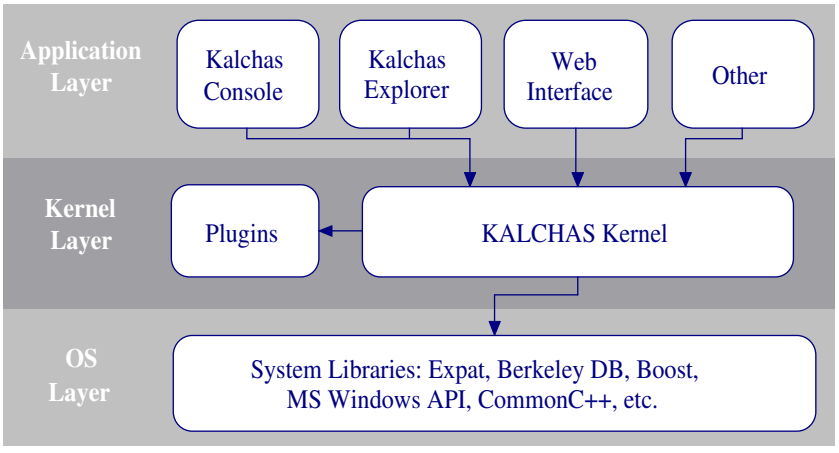

Figure 5.1 : The KALCHAS architecture

Starting from the bottom we find the Operating System Layer. In this layer,

all system libraries are contained, including Microsoft Windows API, Expat [37], Berkeley DB [38], Boost C++ libraries [39], and GNU CommonC++ [40] which are used directly in the KALCHAS kernel.

In the middle is the Kernel Layer containing the main KALCHAS kernel and auxiliary plugins. The KALCHAS kernel is the KALCHAS core system itself that can be embedded in external applications (see Section 5.2). The auxiliary plugins are introduced to allow third party developers to add their own file format handlers/shredders to the KALCHAS system (see Section 5.4). The implementation of the middle layer is separated into two namespaces, namely `kalchas_kernel` and `kalchas_plugins`.

At the top of the layered architecture is the Application Layer. In this layer we have all applications that embed the KALCHAS kernel. Examples of such applications are Kalchas Console, the web-based PHP interface, and Kalchas Explorer. In the implementation all the functionalities provided by KALCHAS to external applications are wrapped in the `kalchas_api` namespace.

Developing the KALCHAS kernel has taken place in object-oriented C++, with no strict coupling to the object-oriented analysis and design. We have utilized the object-oriented aspect of C++ to group functions and data structures by functionality rather than simulating real-life phenomena. This has been done by organizing classes in namespaces.

Building the system as modular as possible allows easy adjustments for new techniques without tampering with the functionality of the rest of the code. To do that we have additionally designed a set of generic interfaces (virtual abstract classes) to be implemented.

5.2 Embedding KALCHAS

The KALCHAS kernel has been designed and compiled as a dynamic link library `kalchas.dll` to allow an easy application integration. The `kalchas.dll` library file provides an API (as shown in Table 5.1) for third party application developers at the Application Layer level. For a detailed description of each function in the API, the interested reader is referred to Chapter 7 and/or the KALCHAS API Documentation in `http://kalchas.dk/`.

To demonstrate the ease of embedding KALCHAS into third party applications, a code example is given in Appendix A.1 on page 91. This code example illustrates how to (i) import KALCHAS into the application, (ii) add the file `bibliography.xml`, and (iii) execute a query for the terms “Engine” and “Sergey”. The `bibliography.xml` file in the example is the same file shown in Listing 2.1, and the query result after being processed by the *meet* operator is as explained in Example 2 on page 21.

Since the file is a standard XML document, the internal XML shredder will be used to produce the formatted query result as shown in Listing 5.1. The listing illustrates that meta information to the query results is added. Each individual result is packed into a `<kalchas_result>` XML element. Within this element are the generated meta information, including the `<filename>`, `<keywords>` and `<value>` tag. The

Functions	Description
`AddFile`	Add a file to the index.
`DeleteFile`	Delete a file from the index and make sure that any future queries will not return results in the given file until it is added again.
`UpdateFile`	Notify the index that a certain file is updated. If the file has not changed on disk the file is not re-indexed.
`QueryRetrieve`	Query the index for a sequence of search terms and return a list results in XML format.
`ForceCronjob`	Allow KALCHAS to perform maintenance jobs, such as index merges, index clean up, etc. This function should in general not be used, as KALCHAS is equipped with a mechanism to automatically detect when maintenance should be executed.
`ErrorCodeToString`	Translate one of the defined eKalchasErrorCode enumeration values into human readable text.

Table 5.1 : KALCHAS API

```
<kalchas_result>
  <filename>bibliography.xml</filename>
  <keywords>Engine Sergey</keywords>
  <value>
    <paper id="1">
      <title>The Anatomy of a Large-Scale Hypertextual
      Web Search Engine</title>
      <author>Sergey Brin, Lawrence Page</author>
      <year>2000</year>
    </paper>
  </value>
</kalchas_result>
```

Listing 5.1 : Portion of the original XML

content of the `<value>` tag in this case is generated by the internal XML shredder, and as such is a verbatim copy of the original XML element returned by the *meet* operator[1].

5.3 Applications Using KALCHAS

In this project, a set of applications embedding the KALCHAS kernel has been implemented, *e.g.* Kalchas Console and Kalchas Explorer. These applications demonstrate the majority of the functionality presented by the API. The most significant application is Kalchas Explorer which is a state-of-the-art desktop search engine comparable to prominent systems such as Copernic Desktop Search [41], Google Desktop [42], Apple Spotlight [43], MSN Search Toolbar [44], and Yahoo!

[1] External plugins for other file formats may choose to return other kinds of data in the value field

Desktop Search [45].

5.3.1 Kalchas Console

A simple example of employing KALCHAS is the Kalchas Console (see Figure 5.2). This is a small text-based query application that serves as a tool to get immediate access in order to test KALCHAS' functionality.

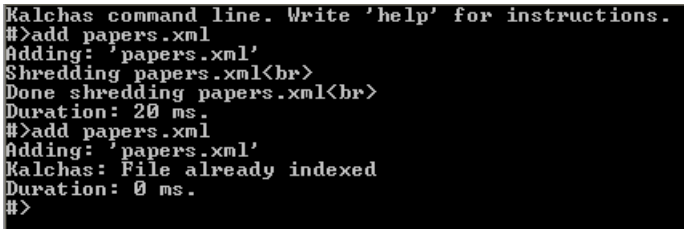

Figure 5.2 : Kalchas Console screenshot

5.3.2 Kalchas Explorer

In addition, another application is also provided to demonstrate an advanced scenario of employing KALCHAS. Kalchas Explorer is our desktop search engine for Microsoft Windows with real-time dynamic updates. This application facilitates all of the functionality provided by KALCHAS. If using the default search mechanism that is shipped with Microsoft Windows, one is often presented with inaccurate results and long search time. This is due to the lack of proper index management. Whenever one starts a new search for a given query, the MS Windows desktop search will traverse the whole file system while opening all files to test for relevance according to the query. Instead, using the KALCHAS kernel we have been able to build a system that manages to execute searches in reasonable time while providing accurate results ranked by relevance. Figure 5.3 shows a screenshot of Kalchas Explorer running for the first time.

The structure of Kalchas Explorer is shown in Figure 5.5, and each of the components shown in the figure is described below:

Kalchas Explorer: Once the system is installed the program Kalchas Explorer will have its own folder in the Start Menu and additionally start every time Windows is booted.

XML Explorer: We have placed a small query field within the graphical user interface. Once a query has been processed, the query result will be shown in a new window — the Kalchas Explorer XML (see Figure 5.6)

Disk Crawler: Disk Crawler is used to traverse the entire file system while indexing all known file formats. This function is intended to be executed at install time.

File System Integration: After the disk has been crawled using Disk Crawler, all new or added files will be indexed automatically. This is done by peeking Windows

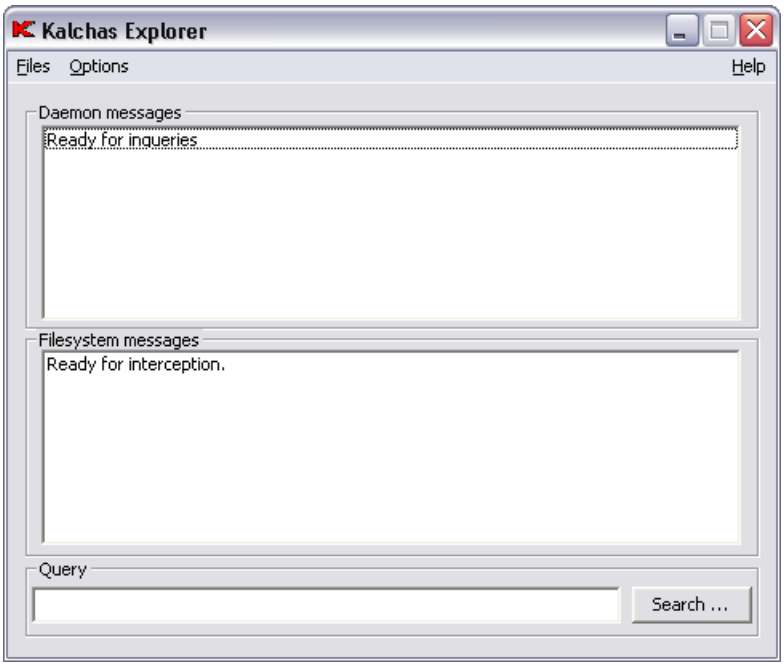

Figure 5.3 : Kalchas Explorer screenshot

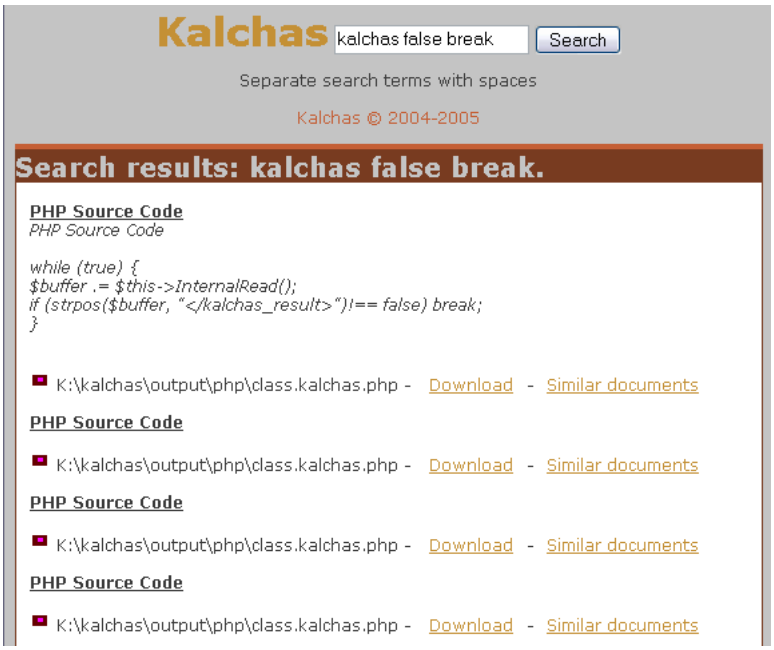

Figure 5.4 : Kalchas Web Interface screenshot

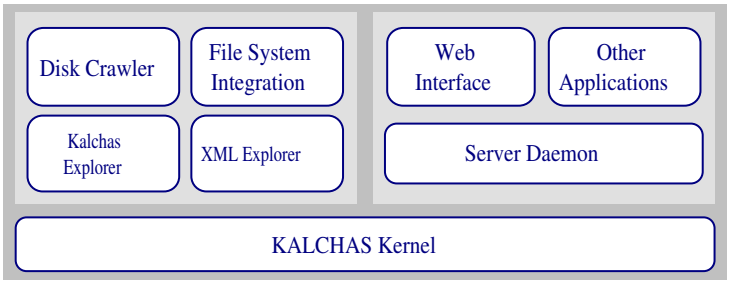

Figure 5.5 : Kalchas Explorer structure

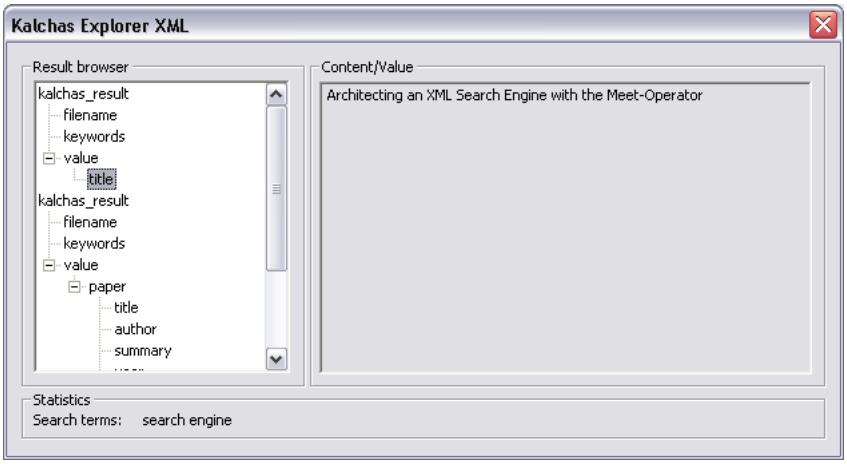

Figure 5.6 : XML Explorer screenshot

Shell Notifications. These notifications are sent out whenever a file has been added to the file system, deleted from the file system, moved from the file system, etc.

Server Daemon: In order to enable other applications to use the index generated by the Kalchas Explorer we have implemented a tiny TCP/IP server daemon using a home made textual protocol. The daemon is started on port 9999 every on every start up.

Web Interface: If the destination desktop system has a web server installed (such as Apache with PHP), one is able to install the provided web interface for querying the index. This web interface provides a cleaner and more intuitive way of browsing results of a query. The server-side PHP script is communicating with the server daemon in order to query for results. Figure 5.4 shows how the web interface is designed.

Other Applications: Using the textual protocol to communicate with the server daemon, external developers are able to interact with the Kalchas Explorer. This could be used to build advanced search mechanism such as Google Desktop which combines online web search with desktop search available *e.g.* from a web browser.

5.4 Extending KALCHAS

The basic functionality of KALCHAS is to implement a dynamic XML search index. Additionally, one may also want to index other files, especially in cases like Kalchas Explorer. In order to support as many file formats as possible in KALCHAS, we have developed a plugin framework, in which any third party developer is allowed to create shredders for any known file formats. Internal to KALCHAS is the XML shredder that handles all files identified. The data output from the XML shredder are tokens based on the actual text found in the document. However, other file formats may contain no indexable content and thus need to have meta data generated, *e.g.*, JPEG or MP3 files mainly consist of binary data unsuitable for indexing. Support for such files should be implemented as KALCHAS file support plugins. The idea of using external plugins, and thus allowing third party developers to extend the functionality of the search engine, is a fairly new turn in desktop search engine development and has only been officially announced in Apple Spotlight [43] (and KALCHAS).

5.4.1 File Support Interface

All file support plugins written for KALCHAS must implement the predefined `cKalchasFileSupport` interface located in the `kalchas_plugin` namespace. The interface is virtual and abstract and thus all functions listed must be implemented by the plugin developer. A description of the interface can be seen in Table 5.2 and a technical description can be found in the code documentation.

An example file plugin is shown in Listing A.2. The example uses all the functions mentioned in the interface shown in Table 5.2. In order to use the code, it should be compiled using Microsoft Visual Studio C++ with an appropriate work space. The

Functions	Description
`Initialize`	Initialize the processing of a given file.
`Process`	Start processing the initialized file. Postings should be buffered and be available until `Deinitialize` is called.
`GetNext`	Once the call to `Process` is done KALCHAS will iterate through the postings by calling this function and retrieve postings one by one.
`Deinitialize`	Release any resources allocated since the call to `Initialize`.
`GetNumExtensions`	Return the number of file format extensions supported by this plugin.
`GetExtension`	Retrieve a textual representation of the supported file format.
`RetrieveXML`	Retrieve the XML associated with a given input Dewey path. This function is called whenever a query return hits within a file format supported by an external plugin.
`DestroyInstance`	Release any resources allocated by the plugin. This is called once in the shut down phase of KALCHAS.

Table 5.2 : Kalchas File Support Interface

output `PLUGIN_PGP.DLL` should be placed in the same folder as the KALCHAS DLLs. When KALCHAS kernel invoked, it will scan the folder for any files matching the `PLUGIN_*.DLL` pattern and attempt to load them. For each plugin, KALCHAS will query the plugin for its supported extensions and keep track of those using a string map. If two plugins, `a` and `b`, are supporting the same file format/formats and `a` was loaded before `b` only the latter will be used for shredding. Using these plugins, a third party developer is able to rewrite our internal XML shredder, which uses Standard Template Library (STL) [46] and Expat [37], to something that may suit his purpose better than our general purpose shredder.

5.4.2 Example Extensions

To provide example on how to extend the file support in KALCHAS we have developed the following plugins:

Plain text The plugin `PLUGIN_TXT.DLL` is developed to be able to index simple plain text formats (.txt files). In order to index the tokens correctly, the plugin introduces meta data. This meta data is seen by the location of the postings outputted by the plain text plugin. All locations in the output postings are $/42/1$ `token` where 42 is the document ID from KALCHAS.

Source code Our plugin to handle C/C++ source code, C/C++ header files, PHP files, C# files is located in the `PLUGIN_SRC.DLL` plugin. Normally source code files would be indexed as plain text files, however our example plugin shows how to index source code files down to semantic structures. This fine granularity is

implemented by shredding the source code as XML files with `{`-signs indicating an XML start tag (`<tag>`) and `}`-signs indicating XML end tags. Searching for keywords within source code results in structural chunk of code, *i.e.*, if a search query returns hits within a given function the whole function source code is shown in the output.

Audio. Audio files, MP3 only, are indexed using their ID3v2 tags and generated meta data in the `PLUGIN_MP3.DLL`. The content of the ID3v2 tags is basically indexed as plain text and on top of that we extract meta data about the audio file (*i.e.*, stereo/mono, bit rate, etc).

Chapter 6

Index Structures

This chapter describes the structures used for storing the inverted indexes and the approaches used for maintaining these. Moreover, we present the techniques used for keeping the indexes up-to-date while distributing the cost of updates.

XML documents in the repository may be subject to frequent changes (*e.g.*, updating, addition and deletion of documents), thus the inverted index should simultaneously reflect these changes in order to provide users an up-to-date access to the information. This is, however, a challenging task since frequent updates on the inverted index cause high workload.

In this project we have employed a range of techniques to optimize index updating. The first technique is *index partitioning*. This is accomplished by having a cascade of three indexes — composed of an in-memory cached index, a small dynamic index, and a large static index — instead of a single one. Doing so, we are able to index newly arrived or changed documents in-place and, at the same time, avoid frequent full index reconstructions. In-place indexing means whenever new documents are added or existing documents are modified/deleted they will be indexed when re-building the entire index, thus providing up-to-date information access for users. When the cached index becomes full, a batch of old documents will be flushed into the dynamic index, and the dynamic index will occasionally be merged with the static index. The reason why we do this is mainly to reduce disk I/O accesses. An earlier experiment on three alternative strategies for index updates (in-place update, index merging, and complete re-build) conducted by Lester *et al.* [47] has also shown that merging is the fastest approach for large numbers of updates. A similar technique has also been proposed by Tomasic *et al.* [22]. To address the problem of incremental updates of inverted lists, they propose a new dual-structure index which is able to dynamically separate long and short inverted lists.

In addition to index partitioning, we also use *caching* and a number of strategies for moving posting from the cached index to disk to optimize index updates. Recently, Lim *et al.* [9] have proposed a new technique to update the inverted index for previously indexed documents whose contents have changed. Their technique uses the idea of using landmarks together with the diff algorithm to detect the differences between documents to reduce the number of postings in the inverted index that need to be updated. Further, to underpin this research direction, a number of algorithms to

detect changes in XML data has also been developed [48, 49]. However, a repository containing old versions of documents is a prerequisite for all of these techniques, and thus they do not apply for KALCHAS.

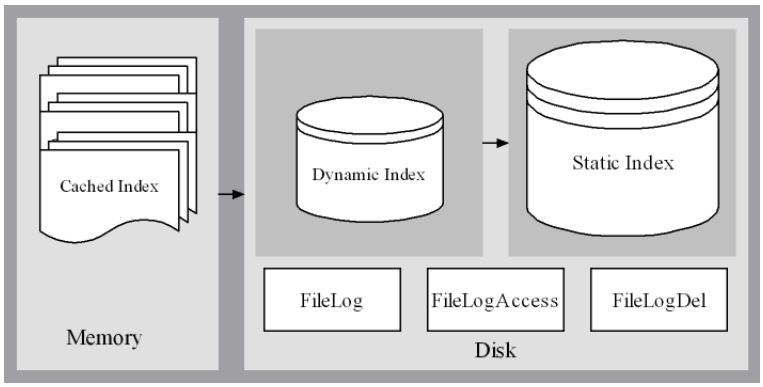

Figure 6.1 : Index structure

In general, the overall index structure of KALCHAS is designed as a series of cascading indexes, where each index has a specific purpose, providing distinct properties and trade-offs. Postings enter the system in one index and slowly seeps through to the next one, as illustrated in Figure 6.1. This approach allows flexible fine-tuning a various parameters which directly effects the performance of the system, especially update performance. The *cached index* (CI) is an in-memory inverted index, serving as a working set containing recently processed postings and thus yielding better index update performance on frequently modified documents. Unlike CI, the *dynamic index* (DI) and *static index* (SI) are rather large, therefore they are kept on disk. Both the DI and SI are used to support persistent storage of the inverted indexes. The former, however, has an additional special purpose, namely supporting incremental updates.

6.1 The Cached Index

The dual index structure proposed in the previous project [19, see pp.48] contained two indexes: a small dynamic index (*i.e.* DYNDIL) and a large static index (*i.e.* STATDIL). This scheme allowed incremental updates and outperformed the single index structure; however, while performing well on bulk insertion of documents, it did not handle frequently modified documents efficiently, which was problematic given the intended environment. Often, desktop user work on a relative small set of files at any given time, but this set is frequently modified. For instance, when working on spreadsheets users often save the contents periodically (or it is auto-saved). The problem, in the context of the dual index structure, was caused by only using disk based B-trees for storing postings, *i.e.* when a document had been processed the postings would immediately be written to disk. In the context of frequently modified documents, this approach has the disadvantage that data is likely to be modified shortly after it has been written to disk, thereby causing two problems: (i) the data initially written to disk is rendered stale (while still occupying space in the B-tree), and (ii) all data in the modified document must be written to disk again. Thus, the approach resulted in inefficient use of both

disk access and the B-tree. To handle these problems a *cached index* (CI) is introduced in KALCHAS.

CI is provided as an in-memory inverted index containing postings from the most recently created or modified documents. The advantage of a CI is twofold: (i) utilization of disk I/O and handling of the disk based B-tree is improved, and (ii) the general responsiveness of the system can be enhanced by postponing disk I/O operations. The imposed increase in performance is depending on the strategy used for writing the contents of the cache to disk. Having CI along with the other indexes, however, has raised a number of questions. For instance:

Organization. How should the generated postings be stored and organized in CI after being shredded?

Migration. How should the cached documents be moved from CI to DI in order to minimize disk I/O accesses?

Each of these questions will be discussed in the following subsections.

6.1.1 Data Organization

When a document has been shredded by Expat [37] and tokenized, we create an inverted index representing the postings extracted from the document. In order to achieve high performance, we store these postings according to their term in a search tree. Definition 6.1 formally defines the organization of postings shredded from a given document. A search tree could be implemented as a red-black search tree which guarantees efficient insertion, deletion and retrieval.

Definition 6.1 (Organization of postings in CI). The search tree T for a given document D is organized as follows:

- N is the set of nodes in T covering all unique terms found in D
- $IL_{term_i} = (loc_0, ..., loc_k)$ is the inverted list of locations associated with $term_i \in D$
- $N[term_i]$ is the node in T where $N[term_i] = IL_{term_i}$

sorted on term, so that

- $term_i \leq term_j$ iff $term_i$ is lexicographically less than or equal to $term_j$

In the C++ implementation of KALCHAS we have chosen to use the Standard Template Library's `map` data structure which is a sorted associative container. An *associative container* is a variable sized container that supports efficient retrieval of elements (values) based on keys. It supports insertion and removal of elements, but differs from a sequence in that it does not provide a mechanism for inserting an element at a specific position. Additionally, *sorted associative containers* use an ordering relation on their keys; two keys are considered to be equivalent if neither one is less than the other. This sorting order in the context of shredded terms is shown in Definition 6.1. Sorted associative containers guarantee that the complexity for most operations is never

worse than logarithmic, and they also guarantee that their elements are always sorted in ascending order by key [46]. The `map` container is implemented as a (red-black) search tree, thus common operations like insertion, deletion, and search are supported. Put shortly, a document stored in CI is represented as a search tree ordered by indexed terms where the nodes contain inverted lists for the associated term.

In order to store several documents in CI, we use a double linked list containing all the shredded data. This is done to allow fast insertion and removal of documents. When inserting a new document in the cache, we simply place the search tree containing the shredded data into the linked list (as seen in Figure 6.2). A consequence of this design is that our inverted lists are not interleaved in memory, but rather grouped by DocID. This has the advantage that cached documents can easily be moved from CI to DI.

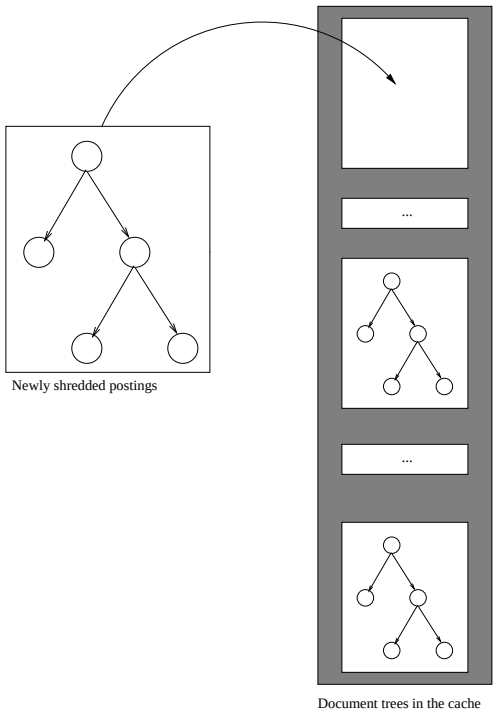

Figure 6.2 : Insertion of a new document in the cache

While a multitude of research projects concludes that compressing inverted lists can increase the overall performance of an inverted index, we have found that compressing the inverted lists stored in CI introduces an unnecessary overhead in terms of execution time. This is due to the fact that documents stored in the CI are subject to frequent updates and compression would only prolong the time of updating the overall index. Additionally, compressing data stored in CI would introduce a computation overhead when migrating data from CI to DI due to different storage schemes. However, we do apply compression on single postings stored in the CI, *i.e.* postings are compressed individually and not sequentially. Compressing single postings has two advantages: (i) compressed postings consume less RAM than uncompressed, and (ii) migration from CI to DI is faster. As described in Chapter 3 a customized VBL codec is used when compressing single postings.

6.1.2 CI-to-DI Migration Policy

As the main purpose of CI is to avoid writing unnecessarily to disk by postponing write operations, the memory allocated to CI should be able to hold a working set of files. By "working set" we mean the set of files which is likely to be modified multiple times within a certain time frame. We assume that of all the previously modified documents, the document most likely to be modified next is the one most recently modified, and based on this assumption a LRU (Least Recently Used) policy is used for moving documents from the cache. Although this is a somewhat simplified assumption, we believe it catches the essence of general desktop user behavior, where users normally work on only a few documents at a time. This assumption is supported by the widely used Zipf's law [50]. Figure 6.3 shows the relationship between specific documents and their frequency of update. Using Zipf's law we observe that only a subset of documents are subject to frequent updates which is the scenario we support through the CI.

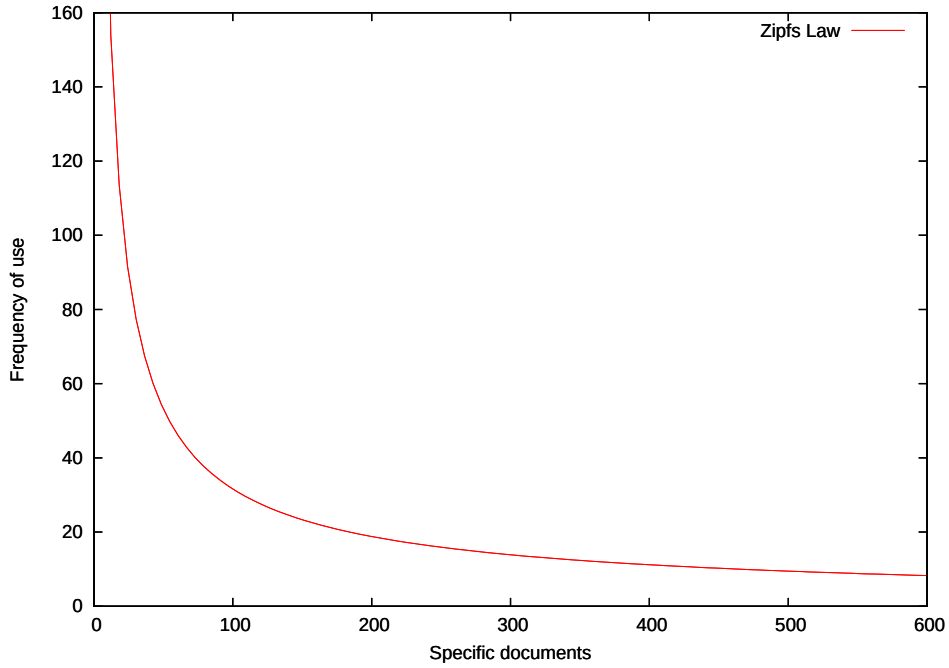

Figure 6.3 : Zipf's law applied to a document collection

As the size of the working set can vary from user to user and from day to day, it is difficult to point out an universal appropriate size. If the memory allocated to CI is too small to hold the working set, it will at some point be necessary to write data to disk, which is likely to be modified again within a certain time frame causing additional disk I/O. On the other hand, if too much memory is allocated to CI, memory would simply be wasted. It should be noted that this problem is not solved by dynamically allocating memory to CI, since such an approach simply ensures that the allocated memory contains data. However, from the drawback described above it should be clear that allocating more memory than necessary to CI is preferable as compared to allocating too little memory.

In KALCHAS the amount of memory allocated to CI is controlled manually. More specifically, it is possible to control the maximum number of posting which at any time can be in the cache. As both the term and location field of postings can be of variable length, it is not possible to control the exact amount of allocated memory.

When new documents are loaded into memory CI may become full, thus some cached documents must be moved from the volatile memory cache into a persistent storage (*i.e.* dynamic index). In this context, the LRU policy is employed, meaning that (some of) the documents which have been least recently modified are selected as candidates for being written to DI. As postings always are moved from CI to DI on a per-document basis, postings are clustered in memory on DocID.

When migrating the cached documents from CI into DI, the performance aspects must be taken into consideration. One way is to move one document at a time from CI to DI. By using such a strategy we can make good use of the cache by postponing write operations for as long as possible, but unfortunately writing only relative few postings to disk at a time does not make efficient use of disk I/O. However, if more than one document is moved at a time, chances are that a higher degree of disk I/O utilization can be achieved. Unfortunately, this can also affect cache performance negatively, since write operations are not postponed for as long as possible. This problem is solved by increasing the size of CI, so that the cache can store the working set and additionally a batch set as illustrated by Figure 6.4.

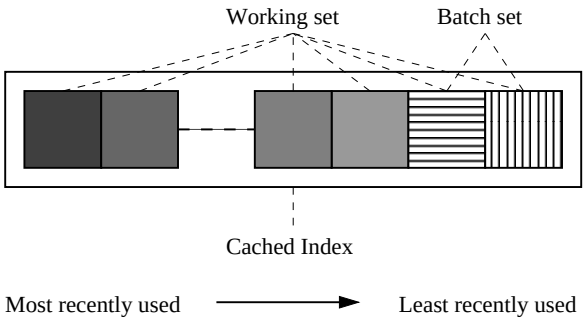

Figure 6.4 : The cached index contains a working set of documents. A subset of the documents in working set also constitute the batch set, which contains all documents that will be written to disk during the next migration.

The batch set contains documents that will be written to disk during the next migration, and as can be seen from Figure 6.4 it overlaps with the working set. While all documents in the batch set are candidates for being written to disk, it is still desirable to be able to modify them while they still are in memory. Batching documents to be migrated from CI to DI also has the advantage of increasing the possibility of common terms. This means that when moving several documents from CI to DI, the probability of the migrating documents sharing one or more terms is high; *i.e.* if two documents contain the word “the”, the inverted lists would be written sequentially to disk rather and thus increase performance. The probability of colliding terms in two documents is described using Zipf’s law, which is assumed to apply to most natural languages, *e.g.* the play “Hamlet” by Shakespeare follows the Zipfian distribution in terms of frequency of words (the word “the” appears 27921 times while “abuser” appears 1 time).

In Section 8.3 the performance of CI with respect to cache size is tested and evaluated.

6.1.3 Summary

CI is the in-memory index, intended for maintaining a working set of documents. Storing postings in CI, as opposed to storing them in a disk based index, is introduced in order to reflect frequent updates in-place. Additionally, CI allows migration of consecutive postings in local sort order, which improves the penalty of moving data from CI to DI.

6.2 The Dynamic Index

In this section we describe the *dynamic index*, DI, which is the first disk based inverted index in the index chain of KALCHAS. Being the first index after CI, DI is updated as documents are migrated from the former. As CI is only capable of storing relative few documents, DI must frequently be updated, hence update performance is essential. Thus, we are willing to trade in low disk space consumption, efficient storage and retrieval of inverted lists in order to perform these updates in an efficient manner.

6.2.1 Access Methods

Berkeley DB offers four access methods: Queue, Recno, Hash and, B-tree[1]. While the two primer methods use logical record number as primary key, the two latter support custom primary keys making both suitable access methods as it is necessary to perform lookups on terms in a full text index. However, we chose the B-tree access method for two reasons: first, according to [23] B-trees are likely to render better performance for applications using variable length records; second, as the records in a B-tree are logically ordered on the key, *i.e.* term in our case, the Berkeley DB B-tree implementation can apply prefix omission in order to reduce the amount of data needed to uniquely identify each key item.

A B^+-tree consists of internal pages containing `(key,pointer)` pairs pointing to other pages, and leaf pages containing `(key,value)` pairs, *i.e.* records (we will use the terms "record" and `(key,value)` pair interchangeably). It is generally shallow, thus only a few disk accesses are usually needed to retrieve a record and, furthermore, frequently used pages can easily be cached to reduce the number of disk accesses even more. Both internal and leaf pages are stored on disk blocks, but whereas the former stores pointers, the latter can also store data, making the B-tree not only an indexing method, but also a file organization of records. Thus, using Berkeley DB's B-tree it is possible to store both index term and associated locations. Records and leaf pages are ordered as described by Definitions 6.2 and 6.3 respectively and illustrated by Figure 6.5.

[1] Although the access method is labeled B-tree it is most likely a B^+-tree implementation (see [24]).

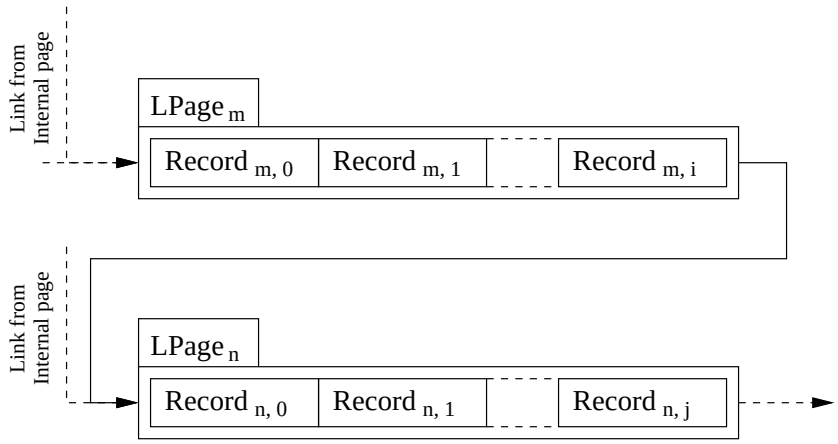

Figure 6.5 : Two leaf pages stored on disk blocks. Records within a leaf page are physically organized according to their key value and leaf pages are link in such a way that no record in one page has a key value smaller than any record in any previous page.

Definition 6.2 (Order of records in a leaf page). The set of records R_m in a leaf page p_m forms a list RL_m where:

- P is the set of all leaf pages
- $r_{m,i}$ is the i'th record in $p_m \in P$
- $key_{m,i}$ is the key of $r_{m,i}$
- $r_{m,i} \leq r_{n,j}$ iff $key_{m,i} \leq key_{n,j}$

so that

- $RL_m \equiv \{[r_{m,0}, r_{m,1} ... r_{m,y}] | r_{m,i} \leq r_{m,i+1} \forall m, i \in Z^+\}$

Definition 6.3 (Order of leaf pages in a B-tree). The set of leaf pages P in a B-tree forms a list LPL where, given p_n and $p_n \in P$:

- $p_m \leq p_n$ iff $r_{m,i} \leq r_{n,j} \forall r_{m,i} \in p_m \wedge r_{n,j} \in p_n, p_m \in P \wedge p_n \in P$

so that

- $LPL \equiv \{[p_0, p_1 \ldots p_m, p_n \ldots p_z] | p_m \leq p_n \forall m \leq n\}$.

Above definitions will serve as a basis for the B-tree related discussions in the remainder of this chapter. While both definitions describe the logical organization of records and leaf pages, the former definition also describes the physical organization of records on leaf pages; thus, if data is inserted into a B-tree in key order, records can simply be written sequentially to disk blocks, avoiding rearrangements on disk blocks. Inserting or deleting records will eventually lead to rearrangement of the internal pages of a B-tree, but since this generally would be localized to a few pages, *i.e.* disk blocks, the operations are fairly inexpensive (in terms of Big-Oh notation the operations might be expensive, but in the context of large data structures that do not fit into main memory avoiding disk I/O is often more important than general time

complexity considerations). In the case of massive incremental updates, however, the reorganization of both internal and leaf pages can become very disk I/O intensive due to pages becoming over- or underfull and potentially effecting other pages, thereby making further reorganization necessary. Finally, since the leaf pages are ordered and linked as defined in Definition 6.2 and Definition 6.3 and illustrated by Figure 6.5, both equality queries and range queries are supported.

6.2.2 Index Maintenance Strategies

Some systems which use inverted indexes to support full-text search apply a re-build strategy for maintaining the index. Using such a strategy, the index is not incrementally updated, rather it is frequently re-built from scratch by shredding the documents and extracting postings. The postings are then often sorted and stored in intermediate data structures, referred to as *sorted runs*, before they are finally processed into inverted lists [10]. Using a re-build strategy and sorted runs allows for great flexibility when storing the postings in inverted lists. As incremental updates are not applied to the index, the cost of modifying inverted lists can simply be ignored, making it possible to use a record storage scheme which is equivalent to the logical concept of an inverted list. Hence, an inverted list maps directly to a record; the term associated with the inverted list is stored in the key field and the sorted list of locations is stored in the data field. This has the advantage of rendering relative good disk space utilization as overhead introduced by the database system for keeping meta information for each record is kept relatively low, since the number of records is equivalent to the number of unique terms, cf. the equivalent mapping between inverted lists and records. As Berkeley DB uses a 5 bytes overhead to store a key or data item on a page, storing only 10 bytes (5 bytes for the key item and 5 bytes for the data item) per unique term rather than 10 bytes per term occurrence reduces the overhead significantly[2]. Unfortunately, keeping the index up-to-date by applying a re-build strategy is often very expensive in terms of CPU and disk usage due to the fact that the entire process of shredding, extracting, sorting, and storing must be repeated for each re-build. Thus, the strategy is seldom employed on a per-document-basis, rather the process is often repeated with time intervals. Finally, the re-build strategy has the disadvantage of requiring additionally disk space since a copy of the old index must be kept during the index re-build process in order to handle queries.

As opposed to the index re-build strategy, an index maintenance strategy based on incremental updates does not re-build the index from scratch for each update, rather only changes made to the indexed document collection between successive updates are "inserted" into the index. This includes newly created documents, modified documents, and deleted documents. Although this is likely to render index updates on a per-document-basis at a lower cost, than when using an index re-build strategy, applying an incremental update strategy is not as straightforward as aforementioned.

The overall principles for adding new documents are fairly simple: documents are shredded, postings extracted, sorted and inserted into inverted lists. However, when using an incremental update strategy the extracted postings are intended to augment the existing ones rather than replace these, and thus the existing inverted lists must be patched with the extracted postings and therefore the cost of modifying inverted

[2] http://www.sleepycat.com/docs/ref/am_misc/diskspace.html

lists must be considered. Especially, care must be taken when designing the storage scheme for the inverted lists as these are subject to frequent updates. In particular, the equivalent mapping between inverted lists and records, as described above, would not be optimal. If, for instance, a new occurrence of a term is to be reflected in the index, it would be necessary to retrieve the entire inverted list for this term from disk (causing disk I/O), insert the new posting in sort order into the inverted list (causing CPU usage), and finally the modified inverted list must be written to disk (again causing disk I/O). For very large inverted lists, which can be several kilobytes large and spanning multiple disk blocks, this can become prohibitively resource demanding, which is additionally offset by the fact that each new document easily contains several hundred distinct terms.

Handling modified and deleted documents is somewhat more difficult when using an incremental update strategy than when using a re-build strategy. In essence, the problem is how to update the inverted lists by removing locations associated with deleted occurrences of a term (due to documents being modified or deleted). While this is relatively simple in repository-based systems, such as most Web search engines, it is somewhat more challenging in a system which does not have access to previous versions of documents [9]. In such systems, identifying and updating inverted lists containing stale information (*i.e.*, locations) associated with modified or deleted documents requires a complete scan of all inverted lists which poses heavy disk I/O loads on the system and could not be employed on a per-document-basis. Alternatively, a forward index could be used to identify the relevant inverted lists, but unfortunately this also adds a significant performance overhead to the system, due to the fact that for each incremental update both the inverted and forward index must be updated, again causing heavy disk I/O loads.

6.2.3 Supporting Incremental Updates

In KALCHAS, we wish to make new or modified data available for querying as quickly as possible by keeping the inverted index fresh, and thus we need to employ an index maintenance strategy which allows us to apply updates on a per-document-basis while not seizing all system resources. Although the index re-build strategy has desirable benefits, such as straightforwardness and efficient disk space utilization, it is mainly applicable in contexts where index maintenance is not triggered by document modifications but rather time intervals, as is the case with Web search engines, due to the massive CPU and disk usage associated with each re-build. For this reason, the index re-build strategy is not used in KALCHAS, rather the employed maintenance strategy is based on incremental updates and index merges (the latter will be described in Section 6.3).

As described previously, applying incremental updates to an inverted index introduces challenges, which must be addressed in order to yield good performance. More specifically, the following topics must be addressed:

Storage scheme. Storage schemes for storing the inverted lists must be “reasonable”. However, a trade-off must often be made between disk space consumption and update performance.

Stale postings. Stale postings must be removed from the inverted index without

Term	Location
$term_1$	$location_2$
$term_1$	$location_7$
$\vdots$	$\vdots$
$term_1$	$location_{177}$
$term_2$	$location_4$
$term_2$	$location_5$
$\vdots$	$\vdots$
$term_2$	$location_{231}$
$\vdots$	$\vdots$
$term_n$	$location_i$

Figure 6.6 : The storage scheme used in DI

frequently seizing a substantial part of system resources.

Quality of service. Some quality of service assurance (QoS) should be applied with respect to the cost of incremental updates. As the performance of updates on a B-tree is inverse proportionally the size of the tree, the cost of inserting, deleting, or modifying a single record increases as the tree grows. While it is not possible to provide a strict upper bound on the cost of indexing a single document (due to the fact that the cost heavily depends on the number of terms stored in the document), we will instead employ an index maintenance strategy which allows us to control and limit the cost of updating a record in the tree (see Section 6.3.3).

As pointed out in Section 6.2.2 a trade-off often exists between general update performance and storage consumption with respect to inverted lists. On the one hand we do not want the system use excessive amounts of disk space to store the inverted lists, and on the other hand we want to optimize incremental updates of the inverted lists. Since we wish to perform index updates on a per-document-basis, it is reasonable to expect rather frequent updates, and thus DI has been designed with an emphasis on update performance, mapping a posting directly to a record as illustrated by Figure 6.6.

This storage scheme is very similar to the *single payload* scheme suggested by Melnik *et al.* [10] and in terms of advantages and disadvantages they are alike. However, for historical reasons a composite key containing both term and location was used in the latter scheme in order to support ordering of records on both term and location[3].

As illustrated by Figure 6.6 DI is likely to contain records with duplicate keys, which is handled by leveraging on Berkeley DB's built-in support for handling multiple data items for a single key item. By default duplicate items are not supported and successive store operations will overwrite previous data items for a given key item. However, Berkeley DB can be set up to provide basic support for duplicates. This is performed by Berkeley DB by only storing the key item once and providing a pointer to an off-tree duplicate page storing the data items ordered (by default) by insertion-order.

[3]Older version of Berkeley DB did not support custom ordering of multiple data items associated to a single key item. By using a composite key and supplying a custom key comparison function records could be ordered on both term and location.

In order to retrieve all data items for a given key it is necessary to use Berkeley DB's cursor interface and iterate through the data items, as the standard lookup method only retrieves the first data item for a key item. Thus, the concept of an inverted list is not directly supported by this storage scheme, rather it must be implemented at a higher level by using the approach described above. However, for the sake of simplicity we will in the remainder of this chapter refer to an single key item with an associated list of data items as an "inverted list". Finally, it should be noted that VBL encoding discussed in Section 3.2.1 is applied to value field of records in DI in order to reduce the size of these. Due to the small size of data stored in these fields applying more elaborate compression is generally not beneficial as the compression ratio would generally be to poor to justify the computational overhead.

Usually, the locations in inverted list are stored in ascending location order, but we have chosen to use the standard insertion-order in DI due to performance considerations: (i) Encoding schemes requiring location-ordered locations, *e.g.* numerical difference and prefix omission [11], are not applied to data items, and (ii) all locations must be sorted prior to being processed by the *meet* operator. As the locations may be associated with different terms, it would be necessary to perform a (merge) sort of the locations, even if every inverted list contained locations in location order. Thus, sorting the locations do not add any benefits. Rather, it might even add an unnecessary disk I/O overhead; when using location-ordered inverted list, inserting a single location can potentially push another location off-page, making it necessary to read and write one more page (this in turn can cause more locations to push off-page). Thus, for long lists inserting locations at the front can lead to many pages being modified. By using insertion-order rather than location-order it is possible append locations to the inverted list, making it possible only to modify the page storing the tail on the inverted list. It should be noted, this if real inverted lists were used (rather than linked lists of duplicates) this optimization would not be possible, as modifying a long inverted list spanning multiple disk block would require all blocks to read and written - not just the page containing the modification.

However, inserting records in insertion-order has the consequence that selective, partial retrieval of inverted lists is not supported in KALCHAS [4]. This can be explained by considering the SQL query below:

```
SELECT location
FROM InvertedLists
WHERE term = $TERM AND location < $MAX_LOCATION
```

In our case, the only index on table $InvertedList$ is on the attribute $term$ and no additional indexing or sorting is performed. The equality predicate in the `WHERE` clause can efficiently be handled due to the index on $term$, however the inequality predicate would require examination of every row satisfying the primer predicate, *i.e.* due to lack of additional indexing or sorting there is no way to exclude rows satisfying the equality predicate without examining them. If, however, rows were sorted additionally with respect to the attribute $location$, no further rows would have to examined once the inequality predicate was evaluated false.

[4] Some systems optimize query performance when a single (search) term is highly selective, *i.e.* has a short inverted list, by performing zig-zag joins.

While this would be a serious performance issue for systems handling high rates of queries, *e.g.* Web search engines, it is less important for a system where query processing performance is not paramount, *e.g.* in single user environments with standard desktop computers. Additionally, it must be taken into account that we trade in this feature in exchange of better update performance.

Figure 6.9 illustrate what aforementioned B^+-tree would look like after inserting a record with the term `wim` as key value, using the Berkeley DB implementation of a B^+-tree and a standard B^+-tree implementation respectively.

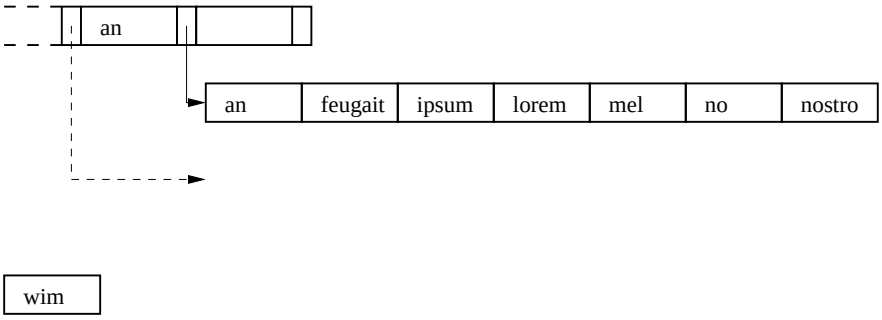

Figure 6.7 : A fragment of a B^+-tree, consisting of a single internal page and a single leaf page, prior to inserting a record with the term `wim` as key value.

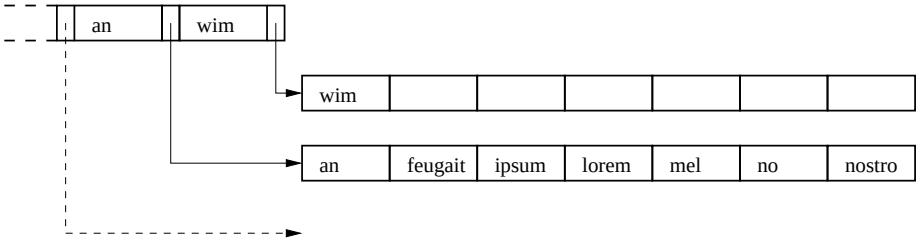

Figure 6.8 : A fragment of a Berkeley DB B^+-tree, consisting of a single internal page and two leaf pages, after having inserted a record with the term `wim` as key value.

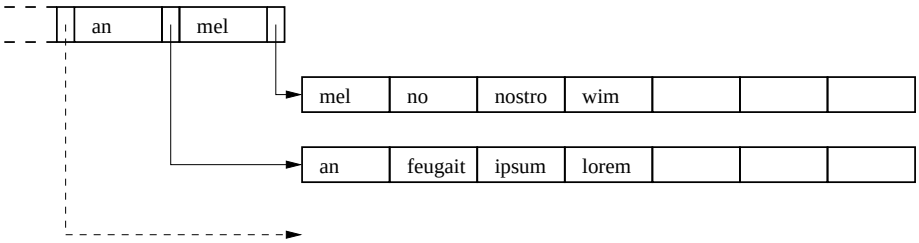

Figure 6.9 : A fragment of a standard B^+-tree, consisting of a single internal pages and two leaf pages, after having inserted a record with the term `wim` as key value.

In addition to considering the storage scheme of records, the order in which records

are inserted into the inverted index must also be taken into account. As mentioned previously, inserting records in key order makes it possible to write them sequentially to disk without having to rearrange disk blocks. However, this is a special property of the Berkeley DB implementation of a B-tree, as illustrated by Figure 6.7, Figure 6.8, and Figure 6.9.

Figure 6.7 illustrates a fragment of a B^+-tree, consisting of an internal page with a key value containing the term `an` and a pointer to a leaf page, which contains seven records[5]. When used as an file organization of records, the leaf pages of a B^+-tree store both `key` and `data` values, but for the sake of clarity, the data field of records have been omitted in all the figures. Figure 6.8 and

As can be seen from the two latter figures, the difference between the Berkeley DB and the standard B^+-tree implementation lies in how pages are split when becoming overfull. In the general case (as illustrated by Figure 6.9) approximately half of all records/pointers are moved to the subsequent page, but since records are inserted in key order, the gap left behind will never be filled. This renders poor page space utilization, thus increasing the height of the tree and thereby also the number of pages accessed in order to retrieve a record. In the Berkeley DB B-tree implementation, however, insertion in key order is considered "best case" and results in near-full pages; looking at Figure 6.8 subsequent records, which, due to key order insertion, all have a key with a lexicographical value of at least `wim` will be inserted to the "right" of the record with the currently highest key value, leaving no gaps behind.

Unfortunately, it is not possible to fully exploit the advantages related to key order insertion when inserting records into DI. As stated previously, the index is intended for handling incremental updates on a per-document-basis and thus posting are also extracted, sorted, and inserted on a per-document-basis, making it possible only to insert in locally sorted order rather than globally sorted order. Hence, incremental updates have an inherently negative effect on both page fill factor and well as disk access pattern. Although globally ordered key insertion-order cannot be achieved, records should still be pre-ordered prior to inserted into a B-tree:

- General time complexity of insertion into a B-tree is not favorable compared to other sorting algorithms.
- Expensive reorganisation of the B-tree may be needed. When inserting the i'th record it may become necessary to push one or more of the previously inserted i-1 records off-page, either directly or indirectly, resulting in an I/O penalty.

To lessen the negative effects of incremental updates, DI has been augmented with another disk based inverted index, SI, which is also based on a Berkeley DB B-tree. Splitting the disk based inverted index up into two parts, thereby making it possible to reduce the size of DI, has a number of advantages:

Reduced storage space overhead As explained previously Berkeley DB uses meta data for keeping track of `(key,pointer)` and `(key,value)` pairs. Hence, reducing the number of `(key,value)` pairs in DI also reduces the storage space needed to hold meta data.

[5] As illustrated by the dotted lines the internal page contains additional key values and pointers, which for purpose of illustration have been left out.

Improved B-tree update performance Since the performance of incremental updates performed on a B-tree deteriorates as the tree grows, reducing the number of `(key,value)` pairs improves performance.

Globally ordered key order insertion By applying a re-merge index maintenance strategy on SI, `(key,value)` pairs can be inserted in globally ordered key order into SI, making high page space utilization and roughly sequential disk access feasible.

SI will be covered in detail in Section 6.3, and for the moment we only note that postings at some point are moved from DI to SI, resulting in a merge of the two.

Finally, in addition to tuning DI for handling incremental updates, stale postings most also be removed without causing unnecessary heavy disk loads. As no information which makes it possible to locate stale postings is provided, identifying stale information requires a complete scan of all `(key,value)` pairs, which under normal circumstances will seize a substantial part of all disk I/O and effect general responsiveness. As a consequence, records are not explicitly deleted from the B-tree. Rather, during the merge process where postings are moved from DI to SI stale information can be filtered by only introducing additional CPU usage; as all postings and inverted lists must be read from disk in order to perform the merge operation, stale information can simple be filtered by examining the document ID of every location and comparing it to an in-memory list of invalid document IDs and any location containing an invalid document ID is simply discarded.

6.2.4 Summary

DI is a disk based inverted index which relies on a simple storage scheme for postings in order to efficiently provide support for incremental updates. In general, the main purpose of DI, with respect to the disk based part of KALCHAS' inverted index, is to distribute the cost of updates, mainly at the expense of efficient storage space utilization and sequential disk access.

To sum up DI provides distinct properties and trade-off as listed below:

Pros

- Postings can efficiently be added to the index without having to retrieve inverted lists, insert the locations, and finally write the modified inverted lists back to disk, potentially causing several blocks to be read and written back to disk for each inverted list.

Cons

- A fairly large storage space overhead is imposed by Berkeley DB due the storage scheme. As the meta data is added for each key and data item, storing each location as a single data item, rather then clustering locations in inverted lists, increases the overhead significantly.
- As records are inserted in locally sorted order, the page fill factor is effected negatively, making it difficult to achieve 100% page space utilization. This increases the height of the tree and thus the number of pages that must be read in order to retrieve a record. Additionally, as Berkeley DB uses 26 bytes of meta data per page, having a low page fill factor also effects the storage space overhead.

- The locally sorted order also effects the disk access pattern, causing more random access.

Although DI seems to provide more drawback than advantages, it is important to note that these drawbacks are inevitably linked to the support of incremental updates. In Chapter 8, a number of tests and evaluations describing the performance of KALCHAS with respect to migration policies are provided.

6.3 The Static Index

The second disk based inverted index in KALCHAS is the *static index* (SI). As indicated by the name the contents of SI is relative static as compared to the other inverted indexes, CI and DI. Additionally, the vast majority of data stored in KALCHAS resides inside SI.

The main purpose of SI is to provide for means of reducing the cost of performing incremental updates on DI by distributing the cost of these updates. As described previously the update performance of a B-tree is especially dependent on two factors: insertion order and size of the tree. By using only a single disk based inverted index it would not be possible to control these factors; postings from documents could only be inserted in locally sorted order and the tree would grow as more documents were indexed. However, once the inverted index is split into two disk based B-trees it is possible to reduce the size of one index, making it feasible to perform incremental updates at a reasonable cost, and insert in global order into the second.

In this section, we describe SI in detail. We present and discuss the data structures used for storing term and location related information and explain how SI is managed.

6.3.1 Data Organization

The index structure in KALCHAS is designed as a series of indexes (see Figure 6.1), where data enters the system in one index and slowly seeps through to the next one. If the documents, to which data is associated, remain unmodified, the data will eventually be inserted into SI which thus contains the least frequently modified data, hence the name. Assuming that the modification frequency of indexable documents on desktop computers at any given time follows a Zipf distribution [50], most documents would be indexed in SI. Based on this assumption, the storage scheme used for storing term and location related information in SI must be designed with particular emphasis on this property. Especially, the storage overhead of each posting should be kept as low. Additionally, as SI is maintained by applying a re-merge strategy (see Section 6.3.3) the cost of performing incrementally updates need not to be considered, thus the design should be quite different from that of DI.

Like DI, SI is also based on a Berkeley DB B-tree, which makes it possible for SI to contain variable length records with custom keys. Additionally, records can easily and relatively efficiently be accessed in key order, which is essential for the maintenance strategy used in SI.

Disk space consumption is minimized by applying a storage scheme identical to

Term	Locations
$term_1$	$location_2, location_7 \ldots location_{177}$
$term_2$	$location_4, location_5 \ldots location_{231}$
$\vdots$	$\vdots$
$term_n$	$location_i$

Figure 6.10 : The storage scheme used in SI

the *full list* scheme suggested by Melnik *et al.* [10] where a term and an associated inverted list maps directly to a `(key,value)` pair as illustrated by Figure 6.10. From the figure it can be seen that the index contains a single `(key,value)` pair for each unique term; the term is stored in the key field while the list of occurrences of the term is stored in the value field. As Berkeley DB uses 5 bytes of meta data to keep track of each key item and each data item and 26 bytes per page, the overall size of the inverted index can be heavily reduced by using this storage scheme as compared to the scheme used in DI, where the number of `(key,value)` pairs is equal to the number of postings.

As the concept of an inverted list is directly supported by this storage scheme, it is possible to retrieve an entire inverted list in a single Berkeley DB operation. It should be noted that although retrieval of inverted lists only takes a single operation, multiple disk blocks may be required to be read (apart from traversing the tree) due to variable length value fields,*i.e.* if the list is very long, it may span multiple pages. Managing very large key and data items in Berkeley DB is handled by introducing overflow pages, which are stored off-tree and require additional disk I/O to be accessed. By default, Berkeley DB stores at a minimum two `(key,value)` pairs per page. Therefore, if a key item or data item is more than one quarter of the page size, the pair is moved off-tree to an overflow page.

6.3.2 Reducing Storage Requirements

KALCHAS is designed to handle both text- and data-centric documents, which means that the lengths of the inverted lists can vary dramatically, assuming that the frequency distribution of terms in natural language text follows Zipf's law[6]. This is illustrated in Figure 6.11 which shows the distribution of the 200 most commonly used words in Shakespeare's *Hamlet*. As illustrated by the figure only a relative small number of words are very frequently used. *Hamlet* contains more than 30.000 words of which more than 4000 are unique, and the 20 most commonly used words occur nearly 10.000 time, *i.e.* roughly 0.5% of the (unique) words constitutes 33% of the entire text.

Assuming that (i) this type of distribution is commonly seen in text centric documents and (ii) documents often share a large part of their most commonly used words, it becomes evident that some inverted lists are prone to grow very long. In the context of a storage scheme mapping inverted lists directly to records, some records will grow so large, that they must be kept off-tree on overflow pages, which unfortunately both increases the size of the index as well a access time for these lists. This would be unacceptable in systems that spend a large percentage of processing time

[6] http://en.wikipedia.org/wiki/Zipf%27s_Law

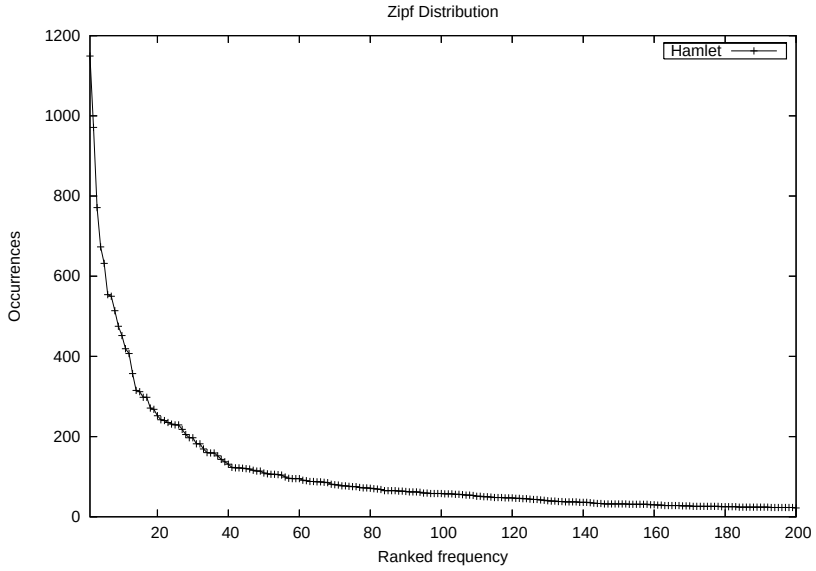

Figure 6.11 : Number of occurrences of the 200 most frequently used words in Shakespeare's *Hamlet*.

on handling queries, but as KALCHAS is mainly intended for desktop environments the cost of having to perform additional disk I/O for some queries is acceptable, although not desirable.

Nevertheless, since *stop words* are used in KALCHAS to prevent some terms from being indexed, the number of overflow pages is reduced. However, this is not done with an emphasis on reducing the number of overflow pages. Rather, some terms tend to be too general to aid in reducing the number of results for any keyword search. Hence, the indexing process can be optimized by omitting these without affecting the quality of results. This is illustrated in Table 6.1 which lists the 20 most commonly used words in *Hamlet*, *MacBeth*, and the *Old Testament*. Terms considered non-stop words in KALCHAS are in boldface. Looking at Table 6.1 it is worth noticing the considerable overlap of commonly used words in the three texts. With the exception of a few, most words are nouns, pronouns, verbs, and prepositions that are used in virtually any English, natural language text, and including them in the inverted index would rarely help produce more specific search results.

Rank	Document		
#	Hamlet	MacBeth	Old Testament
1	the	the	the
2	and	and	and
3	to	to	of
4	of	i	to
5	i	of	in
6	you	**macbeth**	that
7	a	a	shall
8	my	that	he

Rank	Document		
#	Hamlet	MacBeth	Old Testament
9	**hamlet**	in	**lord**
10	in	you	his
11	it	my	for
12	that	is	i
13	is	not	unto
14	not	with	a
15	**lord**	it	they
16	this	his	be
17	his	be	it
18	but	he	is
19	with	have	them
20	for	but	with

Table 6.1 : The most commonly used words in *Hamlet*, *MacBeth*, and *The Old Testament*. Terms in boldface are non-stop words.

Although overflow pages are not generally considered problematic in KALCHAS it is desirable to minimize the number, as this reduces the storage requirements. Apart from employing stop words, increasing the page size can also reduce the number, but unfortunately this approach has a number of drawbacks. As Berkeley DB provides locking of page as the finest level of granularity when using B-trees, increasing the page size can degrade performance with respect to concurrency as the risk of blocking is increased. The I/O efficiency is also likely to be affected since the minimum amount of data which can be read or written in a single access is increased. For instance, if most inverted lists can be stored on a single 4 KB page, retrieving one of these would require reading one disk block (assuming a disk block size of 4 KB)[7]. If the page size was increased to 8 KB to accommodate very long lists, the same scenario would require reading and writing the double amount of data, even if both the original and modified inverted list could easily be stored on a 4 KB page. However, if data generally is read and written in chunks larger than the standard disk block, increasing the page size can result in a performance gain, since (i) the chunks are less likely to be fragmented, (ii) most operating systems uses pre-fetch when accessing disk, and finally (iii) the fan-out is improved thereby reducing the height of the tree.

Since the level of concurrency in KALCHAS is low, increasing the page size is not likely to affect performance in this area. However, analyzing how disk I/O efficiency is affected is not as straightforward as the answer is dependent on a several factors. First, assuming that the frequency distribution of terms follows Zipf's law, so does the length of the inverted lists. The "steepness" of the distribution indicates the percentage of inverted lists which are considerably larger than the average length. The steeper distribution, the fewer lists will be affected by using a small page size. Second, the size of the indexed document collection also plays an important role, since the inverted lists inevitably will grow as the collection grows, hence the choice of page size should also be based on this parameter. Finally, it should be noted that using a page size smaller than the block size of the file system is generally not recommended, as the operating system must retrieve at least an entire disk block for every operation, regardless of the

[7] In the example we ignore the cost of traversing the tree to perform lookup.

page size.

In KALCHAS we are using the default Berkeley DB B-tree page size, which is based on the block size of the file system storing the B-tree. As this size is sufficient for storing the inverted lists of a medium sized document collection without having a high percentage of overflow pages, the result is a reasonable degree disk I/O efficiency

6.3.3 Index Maintenance Strategies

Since the task of handling modifications to a document collection on a per-document basis is mainly done by DI, many of the inherently associated problems need not be addressed in SI. This allows for greater flexibility when maintaining SI as random access modifications can be avoided altogether. In order to distribute the cost of expensive incremental updates, however, data stored in DI should periodically be migrated to SI, thereby limiting the growth of DI and reducing the amount of page reorganization due to updates.

```
MERGE(L, L')
 1  R ← ∅
 2  while (L ≠ ∅) ∧ (L' ≠ ∅) do
 3  if (L = ∅) then
 4    R ← R ∪ {L'_next}
 5    L' ← L' \ {L'_next}
 6  elseif (L' = ∅) then
 7        R ← R ∪ {L_next}
 8        L ← L \ {L_next}
 9      elseif (L_next = L'_next) then
10            R ← R ∪ {L_next} ∪ {L'_next}
11            L ← L \ {L_next}
12            L' ← L' \ {L'_next}
13          elseif (L_next < L'_next) then
14                while (L ≠ ∅) ∧ (L_next < L'_next) do
15                R ← R ∪ {L_next}
16                L ← L \ {L_next}
17              else while (L' ≠ ∅) ∧ (L_next ≥ L'_next) do
18                  R ← R ∪ {L'_next}
19                  L' ← L' \ {L'_next}
20  return R
```

Figure 6.12 : Merge algorithm: L, L' represent two inverted lists of DI or SI to be merged, respectively. R represents the merged result. The $<$ and $\geq$ operators indicate lexical comparison. $\cup$ indicates joins (preserving sort order) and $\setminus$ indicates removal of a subtree without reordering.

A simple, but obviously inefficient, approach of migrating postings from DI to SI would be to occasionally take postings from the former and insert them into the latter. However, not much could be gained from using such an approach as the postings would be inserted in locally sorted order, making the risk of having to perform expensive

reorganizations of the pages of SI very likely. Rather, we notice that both DI and SI contains similar records and the contents of both indexes are ordered and can be accessed relatively efficiently. The basic steps of merging DI and SI consist of scanning the leaf pages of both indexes, comparing records pair-wise and inserting the lesser into a new disk based B-tree SI′ yielding globally sorted insertion order. Once the merge operation terminates SI′ substitutes the old SI and DI is reset. Pseudocode describing the basic steeps of the merge operation is provided i Figure 6.12.

It should be noted that when generally assessing the efficiency of any algorithm in terms of Big-Oh notation, all operations are assumed to be equally expensive, *e.g.* any arithmetic operation is as expensive as accessing an element in an array and moreover accessing element i of an array is assumed to as expensive as accessing element j. However, this assumption is only true for accessing RAM (hence the "random access"), not disk. In addition to all disk access patterns not taking equal amounts of time, accessing data on disk is often several magnitudes more expensive than performing some instruction on the CPU. However, with this in mind merging DI and SI is still an attractive approach as compared to incrementally updating a large B-tree. Performing a merge of two disk based indexes is expensive in terms of disk I/O, since both indexes would have to read from disk and an (almost) equal amount of data would have to be written to disk to produce the merged index. Thus, the re-merge index maintenance strategy is only advantageous to apply when DI is sufficiently large. This can be described by a simple cost/benefit analysis. The benefit, or gain, can be thought of as being the amount of data which is added to SI and the cost is the amount of disk traffic needed to achieve the gain. We will illustrate this with two simplified examples in which we do not consider a realistic workload with live data that might become stale.

Example 3. Assume DI is 10 MB and SI is 100 MB. To merge the two indexes 110 MB would have to be read and 110 MB would have to be written. Thus, 220 MB of disk traffic would render a 10 MB increase of SI, meaning that for every 1 MB SI was increased, a total of 22 MB had to read and written.

Example 4. Assume DI is 50 MB and SI is 100 MB. To merge the two indexes 150 MB would have to be read and 150 MB would have to be written. Thus, 300 MB of disk traffic would render a 50 MB increase of SI, meaning that for every 1 MB SI was increased, a total 6 MB had to be read and written.

Comparing the two examples, it becomes clear that increasing the size of DI gives a better cost/benefit ration[8]. So, from the perspective of the re-merge index maintenance strategy, DI should be relative large when merging. Unfortunately, from the perspective of the incremental update index maintenance strategy, DI should be kept relative small. Hence, there is a trade-off between the two strategies, which was also indicated in [19], and by giving preference to one of them it is possible to either improve performance of incremental updates or overall indexing time. On the one hand, if the size limit of DI is kept static, incremental updates would take a fixed maximum amount of time, but the performance of the merge operation will at some point degrade as the document collection grows and become the dominant factor of the overall indexing time. On the other hand, if the size limit of DI is kept proportional to the size of SI, the cost/benefit

[8] In a more realistic scenario, the amount of data in DI which may potentially become stale increases with the size of DI. Additionally, due to different storage schemes and compression, migrating n bytes from DI to SI would unlikely result in an n bytes increase in SI file size. However, the amount of valid information moved from DI and inserted into SI is nevertheless equal

of merge operations would remain fixed, making it possible to achieve better overall indexing time, but the performance of incremental updates degrades as the document collection grows. Hence, the combination of maintenance strategies used in KALCHAS does not scale well for large document collections, but for the intended environment, *i.e.* desktop computers, this is unlikely to be an problem.

Apart from making index updates on a per-document basis feasible, the combination of incremental updates and re-merge also provides a number of other desirable properties. Since records and pages in both DI and SI are ordered as described in Definition 6.2 and Definition 6.3 records can be read in key order by scanning the leaf nodes. As SI always is written in "one go" its leaf pages are very likely to be written on sequential disk blocks, making the merge operation having a highly sequential disk access pattern. Additionally, most operating systems and hard disks also use some type of pre-fetch/buffering, the performance for this type of access is likely to be improved further.

In addition to the sequential disk access pattern, the performance of merge is further improved by having a more storage efficient representation of data. The compact representation of data is achieved in three ways: (i) by minimizing the number of pages, (ii) by applying a storage scheme which renders less meta data overhead, and (iii) by applying compression to inverted lists:

- In Figure 6.7, Figure 6.8, and Figure 6.9 we illustrated how a high page fill factor can be achieved by inserting in key order into a Berkeley DB B-tree. Since the merge operation basically consists of inserting records from DI and SI into SI′, SI′ will always be built in key order and have a high page fill factor.
- The storage scheme of SI maps a term and an associated list of locations to a record, hence a single record contains information from many postings. In contrast, the storage scheme of DI maps a single posting to a record. Therefore, SI has a significantly better average "meta data per posting" ratio than DI.
- The value field is compressed in both DI and SI. However, since large chunks of data generally renders better compression ratios than smaller chunks, the list of locations stored in the value field of records in SI yields better compression ratio than the single location stored in the value field of records in DI. For this reason, it make sense only to apply more CPU expensive types of compression to records in SI, explaining why only VBL encoding is used for DI and VBL encoding and Huffman for SI.

As the performance of the merge operation is directly dependent on the amount of disk I/O needed to read DI and SI and write SI′, improving storage space efficiency translates directly into a reduction of the amount of disk I/O needed to read and write the indexes.

6.3.4 Summary

SI is the final inverted index in the chain of indexes used in KALCHAS. As incremental updates are handled by DI, the associated problems of reorganization of pages and low page fill factor need not be addressed in SI. Rather, the main purpose of SI is

to make it possible to reduce the size of DI in order to perform incremental updates more efficiently by occasionally migrating postings to SI. This is done by applying a re-merge strategy. In essence, the approach of this strategy is merge the records of DI with those of SI, yielding a new index, SI′. Both of the old indexes are discarded after merging.

From the perspective of incremental updates, DI should be kept relatively small in order to apply incremental updates efficiently, but from the perspective of re-merging DI should be kept large due to cost/benefit ratio, hence a trad-off must be made between strategies; at one extreme indexing time of documents can be improved by keeping DI small, and at another extreme overall indexing time can be improved by maintaining some relationship between the size of DI and SI.

Since the majority of documents are likely to indexed in SI is desirable to improve its storage space efficiency. First, preventing the system from taking up considerable disk space is desirable on a standard desktop computer. Second and more importantly, having a high storage space efficiency also makes it less disk I/O expensive to perform merges, as the cost of these are directly dependent on the amount of disk I/O.

To sum up, SI provides the following advantages and disadvantages:

Pros

- Distributing the cost of incremental updates is made possible by occasionally migrating data from DI to SI.
- A high degree of storage space efficiency is achieved by (i) having a high page fill factor, (ii) using an efficient storage scheme, and (iii) compressing inverted lists.
- The index maintenance allows for high degree of sequential disk access.

Cons

- During merges disk space must be allocated to SI′. Since SI′ is the union of postings/inverted lists of DI and SI, the disk space allocated to SI′ is roughly the sum of disk space allocated to DI and SI.

In Chapter 8, we provide a number of tests and evaluations describing the performance of KALCHAS with respect to migration policies.

Chapter 7

Supported Operations

This chapter describes each of the operations provided by the KALCHAS API. We will explain how KALCHAS adds files (Section 7.2), deletes files (Section 7.3), updates indexes (Section 7.4), and performs keyword searches (Section 7.5), respectively.

7.1 Database Schema

In addition to the three inverted indexes introduced in Chapter 6, the operations featured in KALCHAS rely heavily on the following underlying database tables (see Figure 7.1). These tables are used to maintain information about the indexed documents: (a) *FileLog* is used to keep track of the indexed files; (b) *FileLogAccess* is a secondary index of *FileLog*; (c) *FileLogDel* is used to keep track of the deleted files. The attributes DocID, TimeStamp, and AuditTime are represented by integer values, and the URI attribute is represented by strings. All of them can be of variable length. However, the IsPersistent and IsFreeID attributes contain a Boolean value.

DocID	URI	TimeStamp	AuditTime	IsPersistent

(a) *FileLog* table

URI	DocID

(b) *FileLogAccess* table

DocID	IsFreeID

(c) *FileLogDel* table

Figure 7.1 : Storage scheme for records in the tables

In the following sections, we will describe how the operations in the KALCHAS API are designed, and explain how they interact with the above tables and the indexes described in Chapter 6.

7.2 Adding Files

In order to add files to the index, one simply invoke the `AddFile` function. In the following subsections, we will describe the working process of `AddFile` and discuss issues related to the implementation of this function.

7.2.1 Implementation

The working process of `AddFile` is illustrated by a flowchart in Figure 7.2.

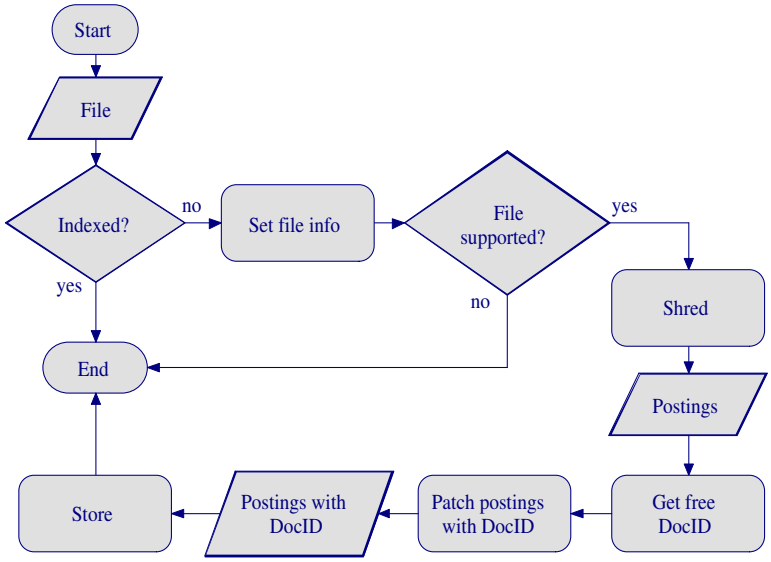

Figure 7.2 : The process of adding files

This operation takes a file name as input. First, the file is checked whether it is already indexed. If so, nothing is to be done and the add process terminates; otherwise, file information should be set, *e.g.*, TimeStamp is set to last update time of the file, AuditTime is set to the current system time, IsPersistent is set to false and IsFreeID is set to false. The purpose of having the attributes AuditTime and IsPersistent is to provide file consistency check support (described in Section 7.2.1.5) and IsFreeID is to support DocID reuse (see Section 7.2.1.3).

Afterwards, the file should be checked for file type support by KALCHAS. If the file type is not supported, nothing should be done and the add process terminates; otherwise, KALCHAS starts shredding the file (elaborated further in Section 7.2.1.1) and returns as output a set of postings. Next, we generate a DocID (elaborated further in Section 7.2.1.3). KALCHAS patches these postings with this DocID (elaborated further in Section 7.2.1.2). Finally, KALCHAS stores these DocID patched postings in CI (elaborated further in Section 7.2.1.4).

7.2.1.1 Shredding

The first step when building a full-text index is to process each document by removing all XML tags and extracting information relevant to the full-text index, as described in Section 2.2 on page 9. While some systems (*e.g.* [8]) collect various meta-information (such as positions of words within documents, encodings, font size, and date of updates, etc.), the information stored in the index should at a minimum contain postings with a term and a location computed by means of Dewey paths.

Shredding an XML document is done by extracting all terms from the source document and combining them with locations. This is done as follows: Read through the whole document while extracting terms and outputting `(term,location)` pairs. In the case of flat documents, *i.e.* HTML or TXT files, generating the `(term,location)` pairs are straightforward since the `location` part is simply the document itself. However, since we want to index XML documents at the granularity of document elements, we facilitate Dewey paths for addressing individual elements.

Computing a Dewey path is a simple task. As described in Section 3.1, the Dewey path of any given element within an XML document can be computed by recording all the Dewey numbers of the encountered elements (generated by the auxiliary function `nid`) found on the path from the document's root element (cf. Definition 3.1).

In practice, parsing XML files is conducted using the external Expat library [37]. Expat is a stream-oriented XML parser library written in C (though we use a C++ wrapper). Shredding is done by parsing the XML document with Expat. Communication from the parser to the shredded is conducted using callback functions, *i.e.* function handling start tags, end tags and character data.

The main idea of using Expat callback handlers to parse XML documents is illustrated in Figure 7.3. After reading the content of XML file into a buffer, Expat invokes the `OnStartElement` handler when it encounters an XML start tag, and the `OnEndElement` handler when it encounters an XML end tag. The start and end tag handlers are in our system used for calculating the Dewey path. In our C++ implementation, the stack data structure (implemented as a variable length vector of integers) is used to to keep track of the current Dewey path. Using a stack, we can push and pop Dewey numbers when the `OnStartElement` and `OnEndElement` handlers, respectively, are called.

Whenever the parser encounters a character data element, the `OnCharacterData` function is invoked, and this function will then extract all tokens from the element. Before a token can be represented in `(term,location)` posting format, a number of conditions should be checked: (i) if the read token is alphanumeric, all characters of the token is converted to lowercase; (ii) if the read token is not a stop word, the token is set to be a term associated with a Dewey path, *i.e.* as a `(term,location)` posting. After that, the posting will be inserted into a `map` container. This container is organized in the same way as a document in CI (as described in Section 6.1.1 on page 37), which means that for each term we associate an inverted list. Terms occurring more than once within the same document are therefore handled by appending to the associated inverted list. Table 7.2 shows an example of the output generated by the shredding process.

As can be seen from Table 7.2 all letters are converted into lowercase. The reason

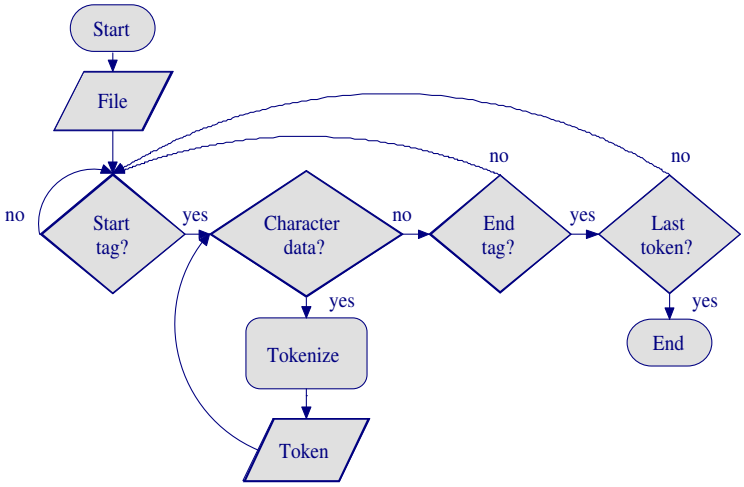

Figure 7.3 : Parsing process

Term	Dewey path
anatomy	/1/1/1
large	/1/1/1
scale	/1/1/1
hypertextual	/1/1/1
web	/1/1/1 /1/2/1
search	/1/1/1
engine	/1/1/1
sergey	/1/1/2 /1/2/2
brin	/1/1/2
lawrence	/1/1/2
page	/1/1/2
2000	/1/1/3
building	/1/2/1
distributed	/1/2/1
full	/1/2/1
text	/1/2/1
index	/1/2/1
melnik	/1/2/2
sriram	/1/2/2
raghavan	/1/2/2
beverly	/1/2/2
yang	/1/2/2
hector	/1/2/2
garcia	/1/2/2
molina	/1/2/2

Table 7.2 : Example output from the shred tool

for this is twofold. First, it makes the algorithms for retrieval much simpler and more efficient, since query terms are turned into lowercase and matched against the keys in the index. Second, storing everything in lowercase reduces the number of keys in the index, which both has an impact on the size of the index as well as retrieval time. Additionally, in order to reduce the size of the index we are using a stop word filter. Currently, the collection of stop words includes English pronouns and adverbs.

7.2.1.2 Patching

Shredding documents as described above introduces an immediate problem of distinguishing postings of one document from postings of another document. A quick example is, for instance, when `copy_file.xml` is an exact copy of `file.xml`. Therefore, shredding documents may result in a set of postings with the same values in both the `term` and `location` fields. In order to distinguish a set of postings of one document from another, we introduce a document identification, called DocID. To save space we have chosen to represent DocID in the `location`. A straightforward way to do this is, once a file has been shredded, we generate a unique DocID value (elaborated further in Section 7.2.1.3), and this value will then be pre-pended to all of the Dewey paths of this document (see Figure 7.4(b)).

However, in our implementation, a slight improvement has been made; we simply substitute the Dewey number of the document root with DocID. Recalling that the local ordering scheme (see Section 3.1) always assigns the document root Dewey number 1 after shredding. Instead of having this value replicated in all postings, we choose to substitute this value with DocID. In this way a number of advantages is achieved: (i) the length of location becomes shortere, (ii) less memory is needed to store, and (iii) response time will be slightly enhanced.

7.2.1.3 Getting DocID

DocID is an integer value, represented by a `32 bit unsigned int`. Whenever a document is added to the index, it is assigned a unique generated DocID value that is one larger than the previously assigned value. Assigning DocID to documents in this way, the DocID values may be exhausted at some point, because (i) we always need to add new documents, thus new DocID values must continously be generated, and (ii) the documents that have been deleted still occupy their DocID values.

To handle this situation, we need to reuse the DocID of the deleted documents. This is done by looking up in the *FileLogDel* table (see Figure 7.1(c) on page 58) and selecting the smallest DocID value, because whenever a document is deleted from the index, its DocID and URI is registered in the *FileLogDel* table (see Section 7.3). Note that whenever a DocID from *FileLogDel* is reused, this DocID entry must be deleted. If there is no free DocID in *FileLogDel*, a new DocID is generated. In ordinary SQL this would be expressed as follows:

```
SELECT DocID FROM FileLogDel
WHERE IsFree = TRUE
ORDER BY DocID ASC
LIMIT 1
```

DocID	Dewey path
44	/1/1/1
⋮	⋮
44	/1/1/1
44	/1/1/2
⋮	⋮
44	/1/1/2
44	/1/1/3
44	/1/2/1
⋮	⋮
44	/1/2/1
44	/1/2/2
⋮	⋮
44	/1/2/2

(a) Before patching

location
/44/1/1/1
⋮
/44/1/1/1
/44/1/1/2
⋮
/44/1/1/2
/44/1/1/3
/44/1/2/1
⋮
/44/1/2/1
/44/1/2/2
⋮
/44/1/2/2

(b) Prepending

location
/44/1/1
⋮
/44/1/1
/44/1/2
⋮
/44/1/2
/44/1/3
/44/2/1
⋮
/44/2/1
/44/2/2
⋮
/44/2/2

(c) Substitution

Figure 7.4 : Representation of the `location` field: (a) A set of Dewey paths along with a generated DocID value 44; (b) All Dewey paths are prepended with DocID; (c) Dewey number of the document root, *i.e.* the leading number in Dewey paths, are substituted with DocID.

7.2.1.4 Storing

Once the data has been processed, the extracted postings should be stored in an inverted index, more specific CI, where the value of `term` is stored in the `key` field of the database and `location` in the `data` field. This approach is chosen instead of a forward index, in which the Dewey path of a given element is used as key, while all terms are stored in the data fields (*i.e.* location order vs. term order as defined in Definition 7.2.1.4).

Shredding documents and constructing Dewey paths are done sequentially, which implies that the extracted postings are in location order. However, CI expects data to be in term order. Storing postings in CI in term order makes it faster to query the indexes by terms. Further, it is more space efficient in that the number of terms is normally less than that of locations.

Definition 7.1 (Ordering). Given an XML document there are two sort orders:

- **Location order:** Given a set of postings $(loc_l, term_l)$, we sort by location as follows:
  $\{(loc_0, term_0), (loc_1, term_1), ...\}$ and $\forall i, j \in Z^+$ with $i < j \Rightarrow loc_i < loc_j$.
- **Term order:** Given a set of postings $(loc_l, term_l)$, we sort by terms as follows:
  $\{< term_0, (loc_{d(0,1)}, ..) >, ...\}$ where $\forall i, j \in Z^+$ with $i < j \Rightarrow term_i < term_j$ and $d : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ is the auxiliary function that finds the locations associated with the given term.

7.2.1.5 File Consistency Check

Before storing postings into CI, file information is registered in the *FileLog* and *FileLogAccess* tables (see Figure 7.1 on page 58). The *FileLog* table is provided to keep track of the indexed files. In this table we store a number of attributes, *e.g.* DocID, URI, TimeStamp, AuditTime, IsPersistent, and IsFreeID, where DocID is the primary key (see Figure 7.1(a)).

- DocID is a unique identifier of a document.
- URI describes the physical location of a document (*e.g.* *file.xml* or `www.example.com/example.xml`)
- TimeStamp indicates the time of last update, as reported by the file system.
- AuditTime indicates the last system time where KALCHAS touched the file (*i.e.* time of indexing).
- IsPersistent is a Boolean value indicating if the postings associated with the document are stored in persistent storage (DI or SI) or in volatile storage (CI).
- IsFreeID is a Boolean value used to indicate if the DocID is occupied by any "live" documents (see Section 7.2.1.3).

When engineering a full-text index, one should carefully consider the possible issues of consistency. Having data stored in volatile indexes (*i.e.* CI) runs the risk of rendering invalid data in the index. This can be shown by indexing a document, storing it in CI, and then turn off the power to PC, all data residing in CI will get lost and must be reconstructed to turn the index to a valid state.

To accommodate such inconsistencies we have introduced the IsPersistent attribute in *FileLog*. By default, all documents are initially stored in CI and are thus prone to be inconsistent. To indicate that shredded documents are in this state, we set the IsPersistent to "false". As soon as shredded documents are migrated from CI to DI, the attribute will be set to "true".

7.2.2 Database Usage

Recapturing the above, we see that adding documents to the index consists of the following processes: (i) shredding documents to obtain inverted lists for the documents,

(ii) generating a unique DocID for associating postings with a specific document (and URI) and finally (iii) storing the inverted lists in the underlying indexes. Operations on the tables caused by `AddFile` are shown in Table 7.3.

	Read	Write	Delete
FileLog	Always	Occasionally	Never
FileLogAccess	Never	Occasionally	Never
FileLogDel	Never	Never	Occasionally

Table 7.3 : Database tables modified by the `AddFile` operation

7.3 Deleting Files

Working in a desktop setting, some files are prone to be deleted. This could happen when the user runs out of disk space or whenever a document gets physically deleted. To cope with the situation we have implemented the `DeleteFile` in the KALCHAS API.

7.3.1 Implementation

The working process of `DeleteFile` is illustrated by the flowchart in Figure 7.5. This operation takes a URI as input. First, it is verified if the document is indexed. If not, nothing will be done and the file deletion process terminates; otherwise, the file should be deleted from the index.

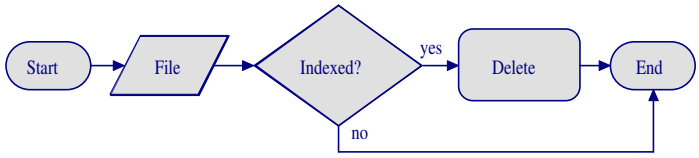

Figure 7.5 : The process of deleting files

Deleting a document from the index is done in two steps. The first step assures that postings from the document is omitted from query results, and the second step removes the actual postings from the indexes. Asserting that obsolete documents are omitted in queries is done by deleting the entries in *FileLog* and *FileLogAccess* associated with the DocID for the current document. Removing the postings from the indexes is postponed until the next index merge. This is done by inserting a new entry in the *FileLogDel*. Once a merge occurs the entry associated with the DocID is updated and IsFreeID is set to “true”.

We will elaborate on the document omission from queries in the description of `QueryRetrieve`. The description of the merge function is found in Figure 6.12 on page 54.

7.3.2 Database Usage

Deleting a single document from the index is reflected immediately in terms of query results and in terms of database entries internally in KALCHAS. The tables used by `DeleteFile` is shown in Table 7.4.

	Read	Write	Delete
FileLog	Always	Never	Occasionally
FileLogAccess	Never	Never	Occasionally
FileLogDel	Never	Occasionally	Never

Table 7.4 : Database tables modified by the `DeleteFile` operation

7.4 Updating Indexes

Utilizing the file system integration introduces the demand for an update functionality in the index. As an example, imagine a user writing a short novel in a word processor with auto-save enabled. Every time the auto-save writes to disk, the file system notifies KALCHAS about the change. This change is handled by the `UpdateFile` function.

7.4.1 Implementation

The inverted index needs to be updated in order to reflect real-time changes in the file system. In order to update files, KALCHAS provides the `UpdateFile` function. The working process of `UpdateFile` is illustrated by a flowchart in Figure 7.6.

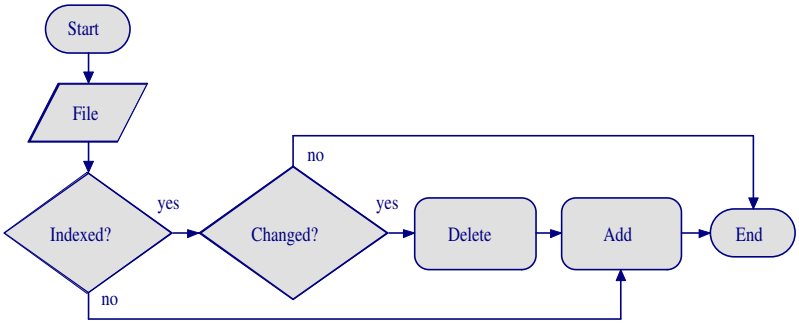

Figure 7.6 : The process of updating indexes

This operation takes a file name as input. First, the file is checked whether it has been already indexed. If the file has not been indexed, the file will be added using `AddFile` function. If the file is indexed and has changed (checked using the attribute TimeStamp of *FileLog* and the file system's time stamp), the file is deleted (Section 7.3) from the index and then re-inserted. If the file is not indexed, the file is simply added (Section 7.2). This process is illustrated in Figure 7.6.

This file update strategy is straightforward to implement, and it seems quite simple. However, it has a main drawback. Let us consider the following case. Suppose that only a few words (says, a single word) in an indexed file are now added/deleted/changed. To update this file, every postings associated with this file need to be deleted and added again in any cases, even though some of the postings need not to be updated at all. Updating files in this way may result in lots of redundant disk reads and rewrites if files are large. Therefore, this file update strategy is not disk I/O efficient. However, if a file is completely changed (*i.e.*, every `(term,location)` postings are changed) then using the delete-(re)add strategy seems to be fine.

7.4.2 Database Usage

The described update strategy has only few direct accesses to the database. However, the calls to `AddFile` and `DeleteFile` introduce more database use. Table 7.5 shows the tables involved in updating a file; indirect cases are marked with parentheses.

	Read	Write	Delete
FileLog	Always	(Occasionally)	(Occasionally)
FileLogAccess	Never	(Occasionally)	(Occasionally)
FileLogDel	Never	(Occasionally)	(Occasionally)

Table 7.5 : Database tables modified by the `UpdateFile` operation. Indirect cases are marked with parentheses.

7.5 Keyword Search

Having a full-text index makes no sense without a search functionality. The KALCHAS API search functionality is a keyword search, as described in Section 1.1.4. This section will describe the implementation of the search functionality (`QueryRetrieve`).

7.5.1 Implementation

When performing keyword search, we need to query the inverted indexes, retrieve relevant postings, and compute meets. To do so, we use the `QueryRetrieve` function. The working process of `QueryRetrieve` is illustrated by a flowchart in Figure 7.7.

Generally, keyword searches proceed as follows. First, users submit search term(s) into the system. There are provided three interaction options:

- Kalchas Console (see Figure 5.2),
- Kalchas Web Interface (see Figure 5.4), or
- Kalchas Explorer, a graphical user interface (see Figure 5.3).

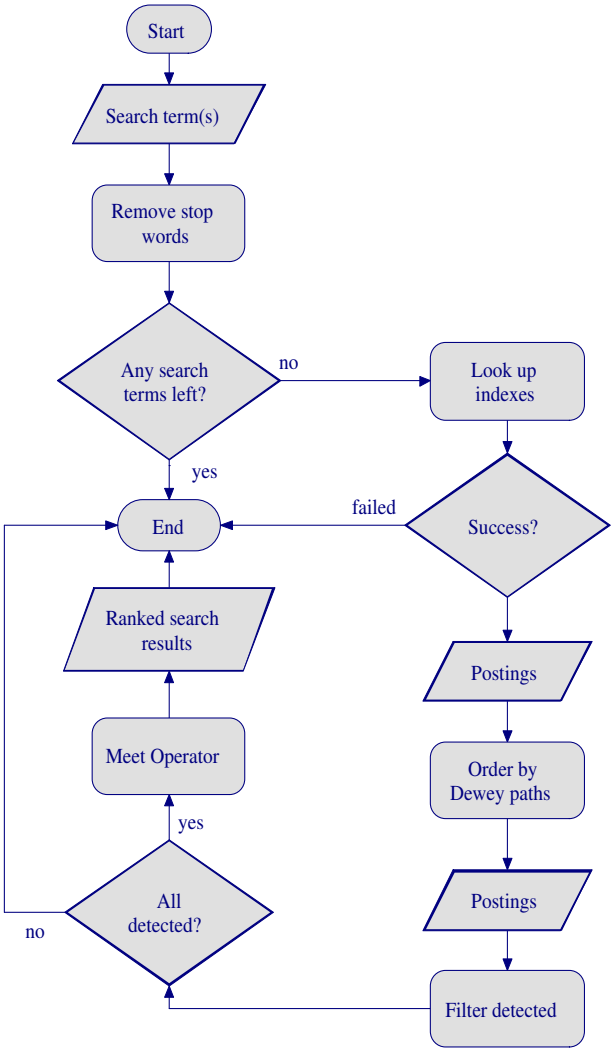

Figure 7.7 : The process of keyword search

When search terms are submitted, the system then checks for stop words and removes them if encounted. Afterwards, KALCHAS checks for the existence of search terms. If there are no valid search terms, the system issues a message to users and terminates; otherwise, KALCHAS looks up in the cached, dynamic, and static indexes to retrieve all the postings associated with the search term(s). As a result, a set of selected but unsorted postings is returned. These postings will then be merged and ordered by `location`. Next, these postings will then be passed to a filter to filter out the deleted files. Afterwards, the *meet* operator takes these sorted postings as input and computes (and at the same time ranks) meets between them. Finally, the search results will be returned and displayed on the chosen interface.

7.5.2 Database Usage

In general, the search functionality does not need to use any of the supplied databases. However, in order to filter out results originating from deleted documents and translate DocID's of the results into file names require lookups in the databases. The use of the database is shown in Table 7.6.

	Read	**Write**	**Delete**
FileLog	Never	Never	Never
FileLogAccess	Always	Never	Never
FileLogDel	Always	Never	Never

Table 7.6 : Database tables used by the `QueryRetrieve` operation

Chapter 8

Tests and Evaluation

8.1 Test Strategies

To ensure that the system works as expected several tests have been conducted. Our tests can be classified in two categories: functionality testing and performance testing. In functionality testing the aim is to test the operations supported by KALCHAS. In performance testing the focus has been to verify the behavior of the system by tuning different parameters, and to validate the relationship between different factors. In this context we focus on testing the performance of shredding documents, migrating data from CI into DI, and merging DI with SI. Furthermore, we also test the compression ratio of different codecs.

The test has been conducted on both real and synthetic data. Real data is taken from a collection of Shakespeare plays presented by Jon Bosak [51] while the synthetic data is generated using XMark [52] and xmlgenz[1]. We have used a desktop Intel Pentium 1,3 GHz CPU, with 512 MB RAM and a 40 GB harddisk (IDE, 5400 RPM, single partition) as test system. This is a 2005 substandard desktop system, chosen in order to generate realistic results in terms of machines available in the average household.

Each test is structured with an introduction to the problem at hand, followed by specific test cases and finally an evaluation is given. An outline of this chapter is given below:

File Adding. Profiling of the components involved in adding new documents to the index. This is done in Section 8.2 and relates to the discussion held in Section 7.2.1.1 on page 60.

CI-to-DI Migration. Supplementing the discussion in Section 6.2 on page 41, we evaluate different parameters involved in migrating postings from the volatile in-memory CI to the disk based Berkeley DB managed DI.

Merging DI and SI. Section Section 8.4 evaluates on the different strategies available when DI and SI should be merged. The test supports the arguments given in Section 6.3 on page 50.

[1] xmlgenz is a custom made tool for generating Zipfian distributed documents.

Compression Schemes. A wide range of compression schemes is profiled, in order to discus which codec would be more appropriate in DI and which would be more appropriate in SI. The test can be found in Section 8.5 and concludes on issues brought up in Section 3.2 on page 14.

Keyword Search. Implementing a full-text index embodying keyword search entails that the function also must be evaluated. Section Section 8.6 demonstrates the techniques described in Chapter 4.

8.2 File Adding

The process of adding a document or file to the index is done by sequentially shredding and indexing the content of the file. This section will test and evaluate on the techniques implemented in KALCHAS for adding files or documents to the index.

8.2.1 Test

Profiling the performance of shredding, in terms of execution times, is depending on a variety of parameters (*i.e.* the distribution of words, size of the documents, system work load, etc.). We have chosen to test the shredder using the following setup:

Test Data I. 600 small XML files in the range 5 KB - 45 KB.

Test Data II. 40 XML files in the range 45 KB - 500 KB.

Test Data III. Shakespeare plays in the range of 100 KB - 500 KB. All cases has an overall size of 10 MB.

Conducting the above tests gave the results shown in Figure 8.1.

Case	No. files	No. bytes	Execution time	KB/sec	Avg. unique terms
i	600	9.118.131	5.948 ms.	1497,05	31
ii	40	10.536.595	12.969 ms.	793,40	8362
iii	50	10.474.072	7.871 ms.	1299,53	3404

Figure 8.1 : Results of the shredder test

8.2.2 Evaluation

Looking at Figure 8.1 we see that the throughput of the shredder module seems to work faster on small files than on large files (*i.e.* the avg. throughput is remarkably low in case (ii) compared to (i)). This behaviour is due to the distribution of words within the documents: The fewer unique indexed term a document introduces, the faster the shredder will process the file. Looking at case (iii) we see that the average throughput, on human readable text, is acceptable whereas XMark generated data imposes poor execution times. The slow down occurs when a document has a low term utilization, *i.e.*

the same term rarely occurs more than once in the document, since we intermediately store all shredded data in a sorted tree that needs expensive re-balancing every now and then. Inserting duplicate occurrences of terms is cheap, since the tree does not need re-balancing (the node is already in the tree) and inserting in the linked list is done in O(1). An interesting observation is, that small files generated by XMark, *i.e.* files from case (i), consists of almost 80% structural information (start and end tags) and almost no textual content.

8.3 CI-to-DI Migration

As mentioned in Chapter 6, the cached index is temporary storage for shredded documents waiting either to be updated by the user or migrated from CI to the disk based DI. This section will underline the necessity of an in-memory storage and demonstrate the increased performance in terms of execution time. In the existing implementation of KALCHAS we have used multi-threading to gain performance when performing bulk insertions: KALCHAS spawns a worker thread for migrating postings from CI to DI while allowing KALCHAS to continue indexing in the main thread. This multi-threading has been disabled while testing the CI-to-DI migration performance, since it would obfuscate the results. Additionally, we have also disabled the DI-to-SI merge operation, which will be evaluated separately in Section 8.4.

8.3.1 Test: Migration

Once a file has been shredded, postings are moved from a temporary storage into the actual indexing structure. The following tests will measure the performance of the implemented index infrastructure, as discussed in Chapter 6.

Our tests is conducted with the Zipfian distribution law in mind, meaning that we perform the tests by simulating expected desktop user behavior, *i.e.* documents are modified in a way that follows a Zipfian distribution. Additionally, we also test how performance is influenced by a random pattern of modifications. The two types of behaviors are captured in the following test cases:

Test Case I. Loading the XML collection and iteratively updating files at random.

Test Case II. Loading the XML collection and iteratively updating files according to a Zipfian distribution.

Case II simulates a Zipfian distribution while case I simulates a uniform distribution. In order to implement the above test, we have used the following statistical functions:

Definition 8.1 (Uniform and Zipfian probability functions). The probability of choosing a specific file f_i from a given collection of files F where $F = \{f_0, ..., f_n\}$.

- Uniform: $P(i) = \frac{1}{n}$
- Zipfian: $P(i) = \frac{1}{i^\alpha}$ with $\alpha > 1$

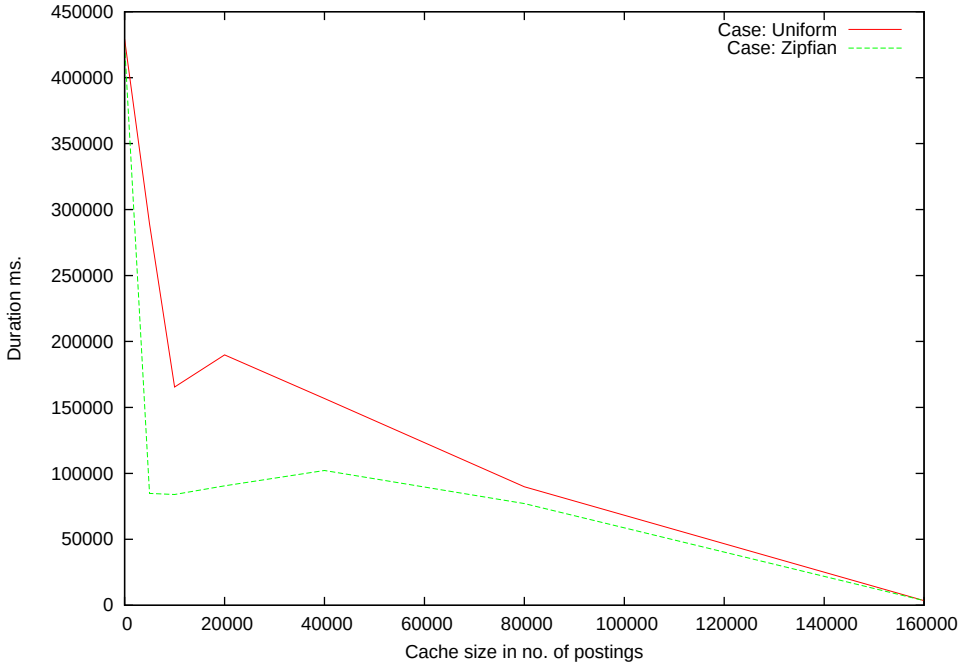

Figure 8.2 : CI-to-DI migration performance

8.3.2 Evaluation

Figure 8.2 shows a graph indicating the duration in ms. when executing the above mentioned test cases. From Figure 8.2 we observe that having a low cache size has crucial impact on indexing time, *e.g.* when having no cache (*i.e.* a cache size of 0 postings) it takes nearly 450.000 ms. to index the XML collection. The figure also shows that the chosen cache replacement policy (LRU) responds very well to the assumption that users follow Zip's law when updating files, *i.e.* updates some files regularly while others remain untouched. It should be noted that as the cache size increases, the importance of having a cache replacement policy with a high hit ratio decreases; when only having limited amount cache the importance of "guessing" right when replacing postings is essential. Increasing the cache size to comprehend all postings extracted from the collection does naturally result in equal performance of the two test cases. However, storing the whole index in memory is not a desired feature when implementing a full-text search engine for desktop use, as this scheme could cause unreasonable amount of main memory to allocated to KALCHAS thereby impacting general system responsiveness. Finally, choosing a cache size that fits all users is not possible as the number of files, users frequently modify, and the size of these vary from user to user. However KALCHAS is implemented with a default cache size of 20.000 postings.

8.4 Merging DI and SI

As mentioned in Chapter 6, SI is the static storage of KALCHAS and postings located in this index originates from files left untouched on the hard drive for a long period of time. As a consequence SI is the least frequently updated index. However, in order to test the performance of SI with respect to different merge policies, we make a number of simplifications making it possible to frequently migrate postings from DI to SI. In this context, a merge police is simply a trigger indicating when a merge of DI and SI should be performed, and based on experience from [19] a merge policy describes a maximum ratio between DI and SI. The tests are conducted by bulk loading XML files into KALCHAS, thereby causing both CI and DI to become full more often than in an every day use scenario. Furthermore, this synthetic workload differs from normal usage in that documents are not modified. However, for the purpose of evaluating different merge policies these simplifications are valid.

As discussed in Section 6.3.3 the performance of both the incremental update strategy and the re-merge strategy depend on the size of DI, but in contradicting ways. By using different merge policies it is possible to either favour the former or latter strategy, thereby affecting overall indexing time as well as indexing performance with respect to a single file.

8.4.1 Test: Merge

In order to test the performance of different merge policies a 260 MB large XML data set was generated with XMark. Five tests were conducted using merging policies with ratios of 100%, 50%, 33%, 25% and 20% respectively. Each test was conducted by initially resetting both DI and SI and then bulk loading the XML document collection. The CI was setup to hold a maximum of 20.000 postings, meaning that once this limit was exceeded a set of postings were migrated to DI. If then the number of postings in DI exceeded the merge policy ration, DI and SI would be merge. Figure 8.3 illustrates the overall indexing time as a function of the collection size for the five merge policies. Figure 8.4 illustrates the average merge duration as a function of the collection size.

8.4.2 Evaluation

Looking at figure Figure 8.3, we notice that with respect to the overall indexing time of a 260 MB document collection, there is less than a 1000 seconds difference between the duration of the best performing policy and the worst performing policy. While this could be interpreted as merge policies having little influence on performance, the figure only shows one property of the system, namely overall indexing time for a small/mid-sized document collect. As opposed to the overall indexing time, the average indexing time is often equally or even more important in a desktop environment, since often users are not concerned with overall efficiency, but rather system responsiveness. Hence, from the perspective of a desktop user it is often more desirable to have fast incremental updates than good cost/benefit ratios for merge operations. This property is illustrated by Figure 8.4, which illustrates the average merge duration as a function

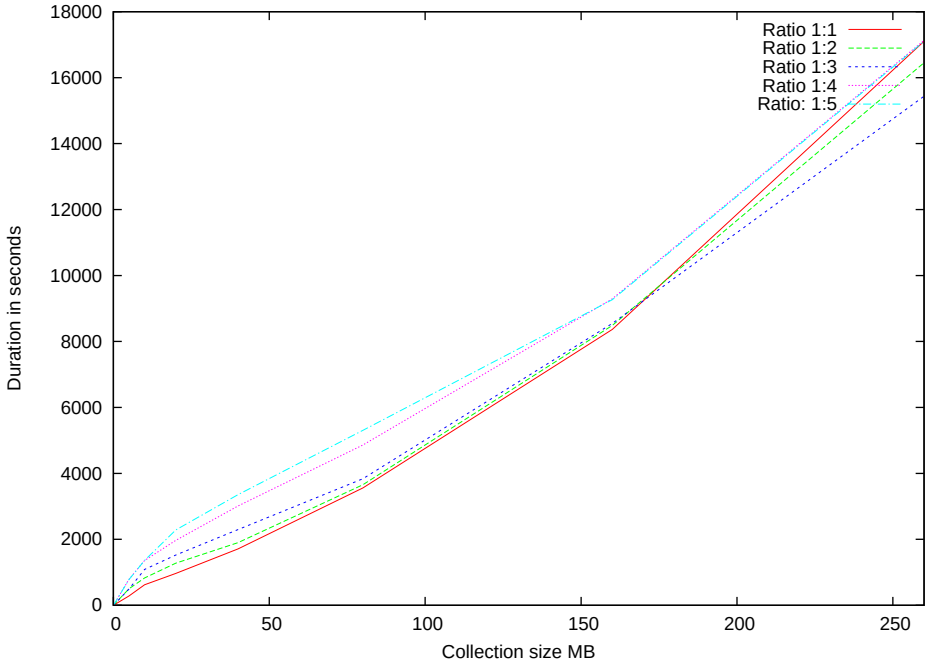

Figure 8.3 : Overall performance of indexing XML collections while modifying the size of DI

of the collection size[2]. Focusing on this property, the 1:5 ratio policy is by far more desirable than the 1:1 ratio policy, as the average time spent on incremental updates and resource demanding merge operations is significantly smaller - although the overall indexing time is the same. Thus, there is a trade-off between performing a few expensive operations and many reasonably cheap operations.

Returning to Figure 8.3 we notice that for small collection sizes the 1:1 ratio policy performs best, however as the collection exceeds 160 MB its performance degrades. This can be explained by the conflicting interests of the index maintenance strategies as discussed in Section 6.3.3. For collections size less than 160 MB and using the 1:1 ratio policy, the size of DI does not exceed the point, where incremental update become prohibitively expensive due to page reorganization. However, passing the 160 MB collection size, this point is exceeded and performing incremental updates becomes the dominant factor. In general, the problem is that the cost of performing merges is linear to the input size (*i.e.* DI + SI), while the cost of incremental updates grows in the size of the tree. Hence, regardless the merge policy ratio, the cost of incremental updates will at some point become more expensive than merging.

Using the low ratio merge policies, however, preference is given to the incremental update strategy. By using a 1:5 or 1:4 ratio policy the growth rate of DI is limited as postings are frequently migrated from DI to SI, thereby making page reorganizations

[2]Note, the duration covers two properties: 1) Performing many small merges results in better average merge duration than fewer larger merges - naturally. 2) Performing frequent merges results in a limited growth of DI which in turn renders better incremental update performance.

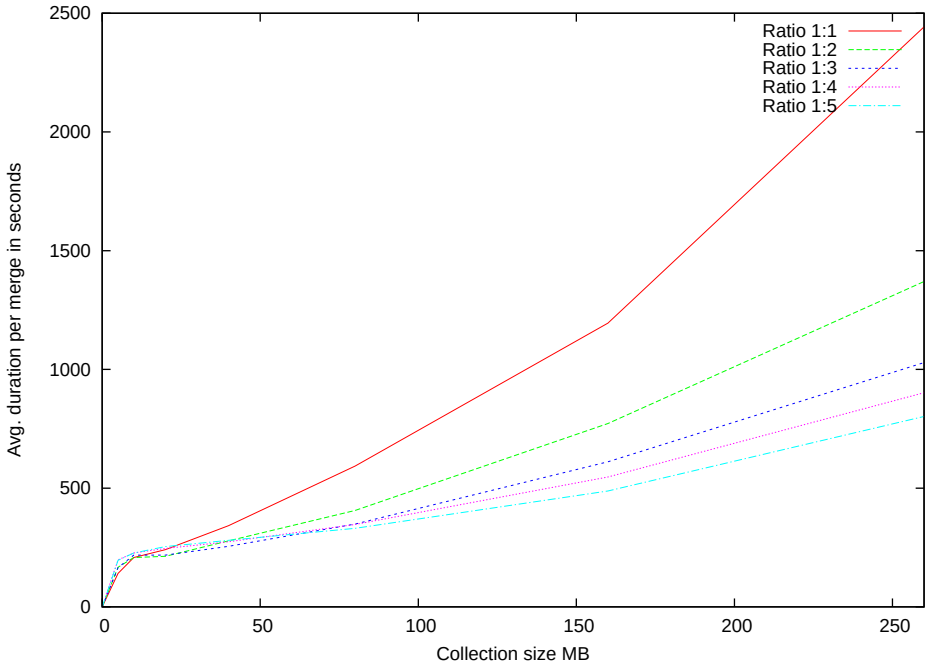

Figure 8.4 : Average time spent on indexing and merging XML collections measured as $\frac{Duration}{No.merges}$

less expensive when performing incremental updates. However, as illustrated by Figure 8.3, using such policies does not result in overall best indexing time. Rather, poor cost/benefit ratios now become the dominant factor.

The tests demonstrate that it is difficult to point out a single merge policy as being best. Rather, any policy is only "best" for a certain collection size and a preferred system behavior, *i.e.* good incremental update performance or good overall indexing time.

8.5 Compression Schemes

In this section we will look into the different aspects of data compression that have been utilized in our project. In order to reduce the amount of data stored in both DI and SI, we have tested a range of specialized data compression strategies. Our findings have been that we need to divide our compression schemes into two major groups: (i) compression of individual locations, and (ii) compression of lists of locations. Case (i) is a direct consequence of having DI which is updated frequently while (ii) is associated with SI which is updated rarely. In (i) we must be able to compress and decompress efficiently without adding a size overhead to the data while (ii) may favor compressed data size over compression time.

8.5.1 Test: Postings

A crucial property of single locations, as stored in DI, is the fact that the number of bytes used for the internal representation is relatively small. The size of the raw data is dependant on the depth of the XML document, *i.e.* if a given Dewey path is of length 5 the internal representation would require 20 bytes. Looking at the previously mentioned codecs (see Section 3.2) we see that model-based codecs, such as Huffman, Rice and LZ77 adds too much overhead to the compressed data and thus cannot be used.

The test was been conducted on both real and synthetic data. Real data was taken from a small collection of Shakespeare plays as presented by Jon Bosak [51] while the synthetic data was generated randomly and thus bring out a worst case scenario of postings.

Synthetic Data. Encoding of the synthetic data is shown in Figure 8.5. The figure is generated as a frequency graph where the property of produces many small compressed locations and only a few large compressed locations is preferable. As expected, writing raw data (in this case 4 byte per Dewey number) is the least efficient way to encode a Dewey path followed by the unary coding (no matter if RLE is applied or not). Using our customized VBL codec the coded data is around 40% of the original raw representation. Additionally, the VBL codec is the only compression scheme which clearly produces only few very large compressed locations.

Real Data. After having tested the selected codecs on synthetic data we will now demonstrate the efficiency of the VBL compared on a collection of real data.

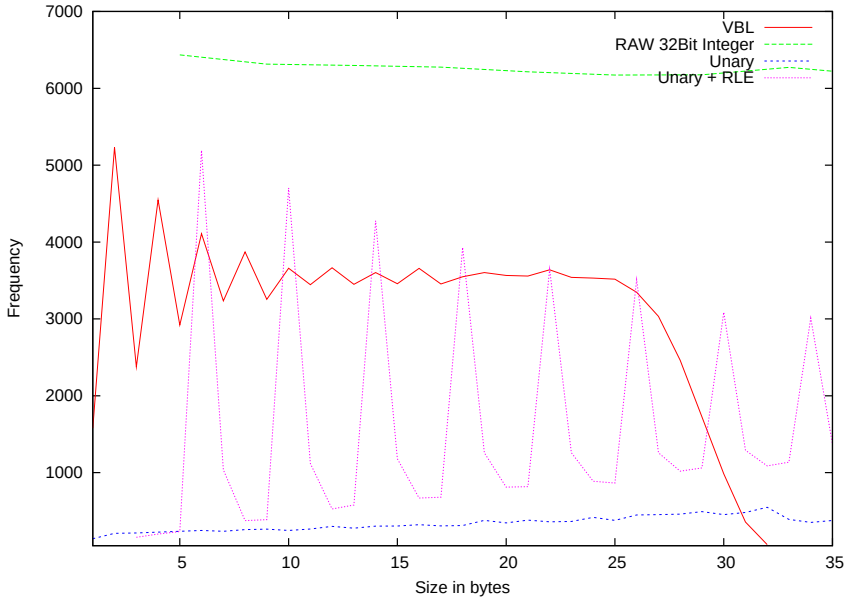

Figure 8.5 : Postings: Compression test of random generated data.

A frequency graph of the test is shown in Figure 8.6. Once again we observe that the RAW data is unusable without further compression applied. In contradiction with the results of the unary coding on synthetic data we observe that the unary notation appears to be useful on real data. However, one should be very careful when using unary coding, since it may perform very well on low values and very poorly on high values. Adding an RLE compression to the unary code reduces the output size remarkably and the overall compression ratio is comparable with the VBL compression, which again proves to be most efficient.

8.5.2 Test: Inverted Lists

Moving on to SI and compression of the data stored in SI, we observe that the data sizes are remarkably larger than those of DI. This is caused by the storage scheme of SI, which collapses all locations associated with a given term into one entry in the index. The increase in data size introduces the need and feasibility for more advanced compression techniques than those applied in Section 8.5.1. It is beyond the scope of this project to invent new compression codecs; however, we have tested a variety of different standard codecs (see Section 3.2.2). As with the compression test for DI, the compression test for SI is conducted on both synthetic and real data.

Synthetic Data. The results of encoding the synthetic data set using the chosen set of compression codecs is shown in Figure 8.7. The most efficient compression scheme in the test is the combination of the BWT and RLE codecs. The next best combination of codes is VBL and Huffman. As expected the raw data representation is unusable and VBL encoding alone is inefficient. Comparing

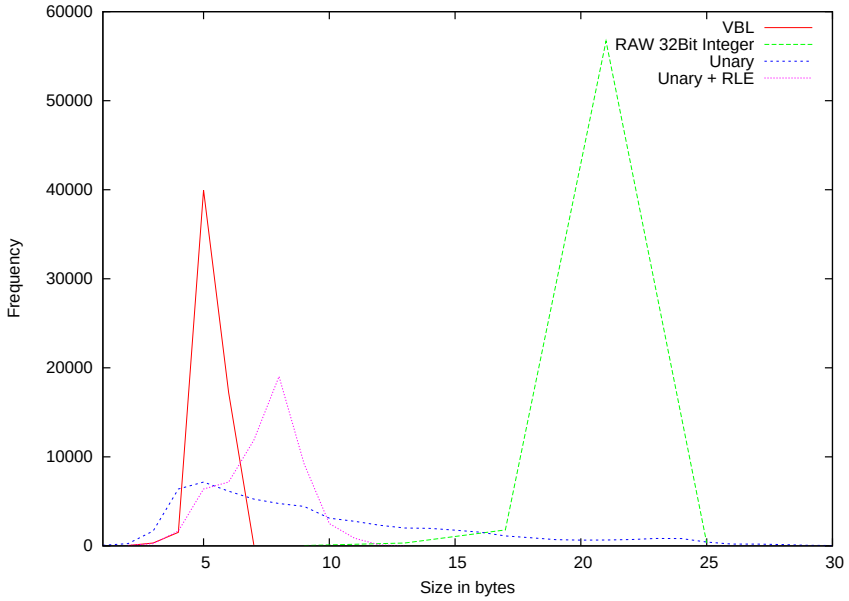

Figure 8.6 : Postings: Compression test of Shakespeare plays.

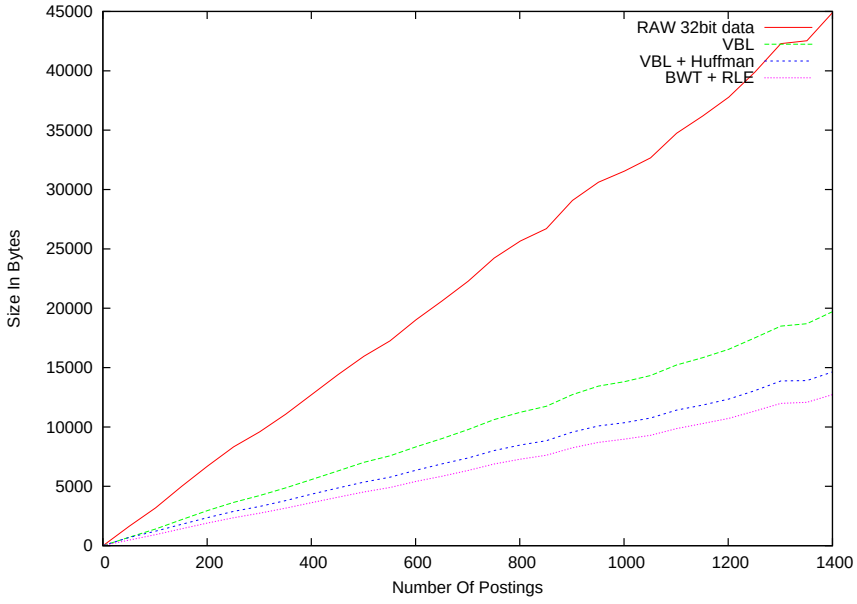

Figure 8.7 : Inverted lists: Compression test of synthetic data.

the combination of the BWT and RLE codecs with the raw data representation we see that we are able to compress data down to 28%.

Real Data. After having tested the codecs on synthetic, generated, data we will now see how the codecs perform when applied to real data. Once again the real data is provided by the Shakespeare plays Othello, Mac Beth and Hamlet. Output of the compression test is shown in Figure 8.8. In general the figure is as expected; however in this case the two best compression schemes has swapped places, meaning that the combination of VBL and Huffman codecs is to prefer over the combination of BWT and RLE. This is due to the nature of the XML representation of these Shakespeare plays, where the document tree is wide and shallow (*i.e.* the Dewey paths are short with very little overlap among them). The VBL codec encodes the lists of Dewey paths into very "noisy" data, which the Huffman codec is able to compress very efficient. Additionally the combination of BWT and RLE works best on longer Dewey paths with higher redundancy of Dewey numbers (*i.e.* XML documents with a narrow and high document tree). Looking at Figure 8.8 it is worth mentioning that the noise occurring in the bottom of the graphs is caused by the associated postings are deeply nested in the XML document, which makes the accompanying Dewey paths relatively long.

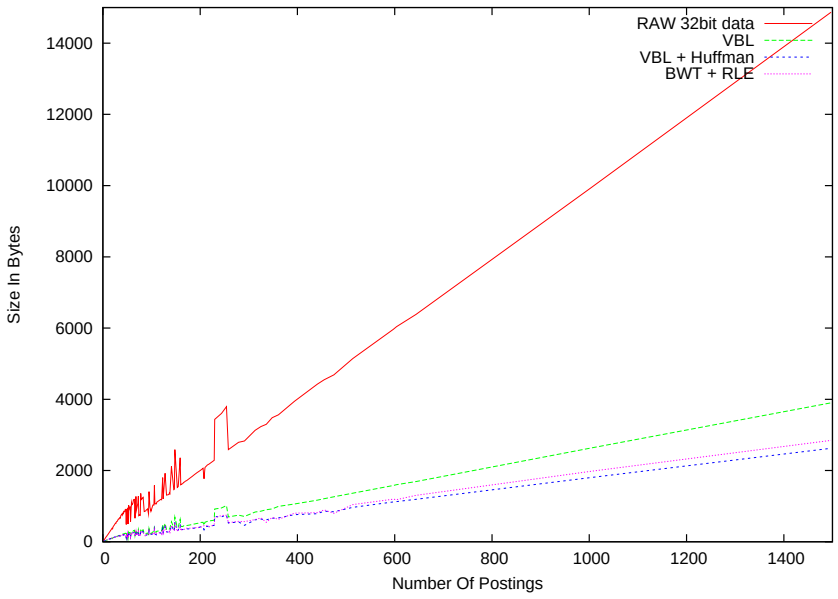

Figure 8.8 : Inverted Lists: Compression test of Shakespeare plays.

8.5.3 Evaluation

Based on the shown tests we are now in a position to evaluate on which compression codecs, or sequences of codecs, is preferable in SI and DI. In the case of compressing single postings (*i.e.* in DI); the Variable Byte Length codec proved its worth on

both synthetic and real data. However, there are certain situations where the unary codec may be preferred — *e.g.* in indexes with very few documents where all of the documents span over narrow and low document trees.

When compressing inverted lists, it is inefficient to encode only using the VBL codec. A combination of VBL and Huffman or BWT and RLE is more attractive, depending on the situation. In general, the combination of VBL and Huffman is preferred, however the BWT and RLE combination should be preferred in cases where one is certain that documents have high document trees with redundancy of Dewey numbers.

8.6 Keyword Search

Test and evaluation on the keyword search functionality in KALCHAS should be seen as an extension to the results found in [19]. These tests established the fact that the scan-based algorithm (previously mentioned as the line-based algorithm) is by far the most efficient implementation. However, while this algorithm was the most efficient one its initial implementation returned inaccurate results, *e.g.* querying a collection of Shakespeare plays for the terms "thunder" and "lightning" would return hits ranking multiple occurrences of either term higher than the hits were just the two terms were present. As a result of this the scan-based algorithm has evolved introducing new design objectives and improvements of results.

8.6.1 Test: Quality of Results

A small test setup consisting of a 10 MB collection of XML documents has been indexed by KALCHAS in order to test the keyword search functionality. In particular, we have indexed a XML file generated from the bibliography used in this report.

Testing the quality of results of search engines is always difficult, since quality of results is a matter of subjective opinions and thus no formal evaluation is possible [8, 13]. Utilizing the *meet* operator we have indirectly defined what KALCHAS perceives as plausible results (Definition 4.4 on page 23): (i) rank specific nodes over general nodes and (ii) rank by node proximity.

Query Results. Executing a query of *rasmus* and *dennis* returns the `<author>` node containing "Dennis Alexander Nørgaard and Rasmus Kaae" from the bibliography entry referring to the DAT5 report [19]. This is illustrated in Figure 8.9. Returning to the problem of ranking multiple occurrences of single terms higher than single occurrences of multiple terms; we see that Figure 8.9 takes care of this. Notice how the element containing both "rasmus" and "dennis" is ranked higher than elements containing only "rasmus" or "dennis".

Stop Word Filtering. Introducing stop word filtering, as described in Definition 2.3 on page 9, has caused new issues to arise. Searching for the sentence "to be or not to be" in Hamlet returns an empty set, as shown in Figure 8.10. This is caused by the fact that each individual term is a stop word. However, looking at Google we see similar problems. The Google search engine is

Figure 8.9 : The keyword search for “rasmus dennis” returns plausible results

confused by the query string, “to be or not to be”, and suggests executing a Boolean query with the OR-operator. In order to retrieve relevant results on Google one has to formulate a quoted query (which seems to be implicitly done when querying Yahoo! for the exact same un-quoted query). Instead of simply leaving the user with an empty result set, the web interface of KALCHAS suggests alternative search terms. This is done by looking up (potentially) misspelled words in the online Merriam-Webster[3] dictionary. The suggestions proposed by the dictionary is presented to the user as shown in Figure 8.11.

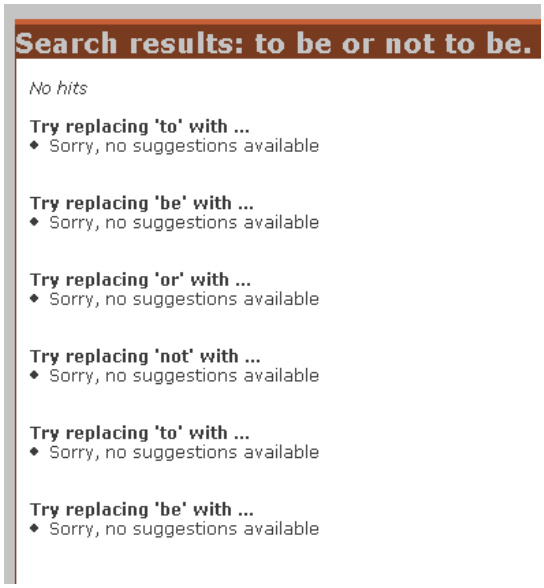

Figure 8.10 : Querying for “to be or not to be” in Hamlet yeilds an empty result set

[3] http://www.m-w.com

Figure 8.11 : If an empty result set is returned by KALCHAS the PHP script suggests alternative search terms

8.6.2 Test: Performance

Having established that the *meet* operator, in combination with the underlying inverted index, returns plausible results, we will now look into the performance of querying.

Query and Retrieve. Executing a query can be split up into three significant processes: (i) retrieving postings from the index, (ii) ranking nodes with *meet* and (iii) presenting the results in a nice presentable way. As explained in [19], (i) and (ii) is executed in less than a second on queries involving more than 80.000 postings, but the Kalchas Explorer and the Web Interface reacts somewhat slower. This behaviour is caused by the way we present results to the user. In terms of Kalchas Explorer and Web Interface we have chosen to present the results in a user friendly manner, heavily inspired by Google and other search engines. However, generating this output takes a significant amount of time, due to the fact that we do not index the actual contents of documents but only the search-able terms. We have previously suggested using a *sparse index* for presenting query results faster, however optimization of displaying results are not within the scope of this project. Instead we have chosen to generate the user friendly representation of the query results in real time, which is possible since we do not expect to handle high rates of queries in a desktop environment.

8.6.3 Evaluation

Even though our system suffers a significant time penalty as result of not having the document contents indexed, the implemented system is fast enough to respond

to queries within a reasonable amount of time on a sub-standard 1,3 GHz machine. Adding newer/better hardware will result in faster query evaluation. However, facilitating the *meet* operator enables us to refrain from having statistical data such as term frequency in a particular document (IDF) and term frequency for the indexed collection (TF) [13, 14, 27]. Adding such statistics could be done on the expense of either disk space or query time:

Term frequency – whole collection TF information is already available at query time, since the results returned from querying the indexes (CI, DI and SI) is an invert list containing all indexed locations of a term.

Term frequency – single element IDF information for individual indexed elements is available at the time of indexing, and could simply be stored together with the term and location in the index.

According to other projects [13, 14], refining the ranking function by involving TF, IDF and TF*IDF values, along with lowest common ancestor calculus, yields more precise query results. While this may be the case, we have decided to focus on the *meet* operator as the primary ranking function and save both disk space and time spent on executing queries.

Chapter 9

Conclusion

We will now evaluate and conclude the work conducted in this report. Moreover, we will conclude by looking at the initial objectives set up in the beginning of the report and evaluate the results found. Additionally, we will give a list of ideas for future work.

9.1 Conclusion

This project has addressed issues related to the development of a *dynamic full-text search engine* targeted at desktop PCs, with particular focus on implementing an efficient index structure. To sum up the work conducted in this project, we will now evaluate on the issues listed in Section 1.2.

1. **Architecture design.** In order to design and implement an efficient index structure, we have redesigned the code framework found in the previous project. This is caused by a natural refactoring, moving from a console based and imperative implementation to a modular object oriented framework, suitable for experimenting with a range of equivalent modules without loosing the overall functionality of the system. Additionally, the framework described in Chapter 5 complies with the need for integration with third-party applications, by integrating KALCHAS' functionality directly in the file system.

2. **Index structure.** To accommodate the Zipfian pattern assumed to be the default user behavior (*i.e.* frequency of file updates follows Zipf's law), we have implemented an efficient cascading inverted index structure, consisting of (i) an in-memory cached index holding the current working set, (ii) a disk based B-tree holding documents migrated from CI, and finally (iii) a disk based B-tree holding documents assumed to be static.

   Empirical tests have proved that having the in-memory CI is useful. This is seen as a consequence of the working set, which consists of the most recently used documents. Migrating documents from CI introduces a penalty in terms of in-responsiveness at the user end, since documents have to be moved onto disk based storage. However, migrating documents from CI to DI is also likely to improve performance as incremental updates become less expensive, resulting in

improved responsiveness. This behaviour is demonstrated through the empirical tests found in Section 8.3. Additionally, seeing that insertion of documents in DI is done via incremental updates, DI will over time become increasingly expensive to update. This is caused by the growth of the B-tree, since inserting data into a large tree is slower than inserting into a small tree. When incremental updates get too expensive, DI and SI can be merged.

Merging the two disk based B-trees (DI and SI) is I/O-bound and thus very time consuming (as illustrated by the tests found in Section 8.4). However, a merge yields a more efficient system in terms of both querying and updating the indexes. This is caused by a higher page fill factor and better storage utilization, as compared to DI, which was indicated in our previous report; *i.e.* working on minimizing the storage space requirements would make the indexes more efficient. Additionally, when merging, the content of DI is moved to SI rendering DI empty and thus incremental updates can be performed relatively fast.

3. **Index compression.** Postings in DI are stored individually, *i.e.* as `<term,location>` pairs. Compressing single postings using the customized VBL encoder has shown to be more efficient than any of the tested codecs, as can be seen from the tests shown in Section 8.5.1.

   Looking at the storage scheme for SI, Section 3.2 introduces the need for additional compression schemes. Compressing inverted lists stored as records in the B-tree using a combination of the VBL encoder and a generic Huffman codec entail the most efficient compression ratio, as shown by empirical tests in Section 8.5.2.

4. **Keyword search.** As with the previous project, this project embodies a full-text index endorsing keywords searches. These searches are supported by the aforementioned underlying inverted indexes. Executing a search for a given search string yields relevant results at the granularity of XML elements. This is facilitated by the *meet* operator, implemented as described in Section 4.3. The empirical tests conducted in Section 8.6 indicate that the search results are plausible for most cases; however, there are still room for improvement by adding statistical information to the ranking functionality.

9.2 Future Work

In this section we will evaluate our work and discuss topics to be investigated in the future work.

9.2.1 Refactoring the Code

Due to the emphasis on modularity and extensibility the current design and implementation of KALCHAS is not fully optimized with respect to performance issue such as efficient/fast code and memory usage and re-factoring and redesigning the code is assumed to increase the overall performance. Moving from the highly experimental, and low-level, implementation made earlier [19] to the highly structured and very flexible design of KALCHAS, has introduced an overhead caused by code abstractions.

All parts of KALCHAS is modular, and thus can be replaced with a single line of code; while this has proved useful while writing this thesis, it is not as efficient as a more strict and narrow design could be.

Using the experience gained in this project and the previous one will help identifying key design issues for future versions of KALCHAS.

9.2.2 Constructing the Index

Constructing the initial index for KALCHAS has not been addressed in this project, but tests conducted in Section 8.3 and sec:test-eval:Merging indicate that the initial creation of the indexes could benefit from applying other strategies, than those used in every day usage. For instance, it would be relevant to evaluate how the index building process employed by many search engines would perform when initially creating the indexes; using such strategies the document collection is crawled, but rather than immediately inserting postings directly into one of the ordinary indexes, postings could be stored in sorted runs. Once the entire collection has been processed, the sorted runs could be merged into SI by using a modified version of the merge algorithm discussed in Section 6.3.3.

9.2.3 Supporting Advanced Searches

Query handling is disjunctive with the current implementation of the *meet* operator. Although ranking gives preference to more specific results, *e.g.* results containing all search terms as described in Section 8.6, it would be useful to add support for explicit, conjunctive queries as well.

Most web search engines support quoted search. As this can be considered a special case of conjunctive search, it is currently not supported in KALCHAS. In order to do so, it would be necessary to extend the scheme used for representing locations with term offset within elements. Additionally, the *meet* operator would need to be extended to handle this types of query.

9.2.4 Refining the *meet* Operator

Querying the inverted index facilitating the *meet* operator to rank results by relevance is generally a good idea. It brings forth relevant, specific, results without adding additional overhead to the data stored in the index. As explained in Section 8.6.3, implementing TF, IDF and TF*IDF style ranking would be necessary to increase the level of relevance in search results. In previous projects [18, 19, 53], more simplistic approaches to the *meet* operator have been conducted, which all seem to entail reasonable results but yet not as accurate as those found in commercial search engines such as Google or Yahoo!. Adding statistical information, as described in Section 8.6.3, could be taken into consideration in a scenario similar to that of Guo *et al.* [13]. However, this method of ranking using statistical information stored in the database could be adopted in a revision of the *meet* operator. While doing so we would be able to present better query results and still maintain a decent size of the indexes stored on disk.

9.2.5 Auditing

In Section 7.1, we introduce the *FileLog* table which features the AuditTime attribute. However, this attribute is currently in-accessible from the KALCHAS API. This attribute can be used to indicate the time of indexing of a document associated with a DocID. This functionality could be used to retrieve useful information in the application layer, *i.e.* to answer the question "Which documents have been indexed since the last query?" In SQL this could be expressed as follows:

```
SELECT DocID
FROM FileLog
WHERE AuditTime > $LAST_QUERY_TIME
```

Adding support for AuditTime queries enables third party developers to implement caching mechanism of search results into their applications, *i.e.* if the mentioned SQL returns 0 rows, then the cached result is still relevant.

Additionally, it would also be able to verify that changes in a given file is updated in KALCHAS. This is illustrated in SQL as follows:

```
SELECT DocID
FROM FileLog
WHERE AuditTime > $LAST_AUDIT_TIME AND URI = $FILE_URI
```

As can be seen from the two SQL examples, this information can be extracted from the current system; however, we have not had the time or need to utilize these functions in the implemented applications described in Section 5.2.

9.2.6 Displaying Results

As described in Section 8.6.2, the current approach of displaying results is not very efficient. For each result, the document containing the result is re-shredded in order to extract the relevant element. Thus, increasing the displayed result set or displaying results located at the end of a large file is expensive in terms of disk I/O.

A more efficient way of displaying results would be to include an index, mapping locations to *e.g.* byte offsets in files. For every document added to the system, entries in the index could be created while shredding. As such an index also requires maintenance, it would be preferable to keep expenses low, hence the index could be sparse, containing only a few `(location,byteoffset)` entries for every file. By distributing these entries evenly across the document, it would be possible to display elements by only performing a limited amount of parsing, *e.g.* if a 10 MB document was indexed by only adding a entry to this index for 1 MB, it would be possible to extract an elements from this document by only parsing 0.5 MB data on average.

9.2.7 Internationalization

We have observed that our current keyword search is most suitable for *synthetic*[1] languages, as compared to *analytic*[2] languages. Most of the European[3] languages are synthetic (*e.g.* English, German, Danish), whereas most of the Asian languages are analytic (*e.g.* Vietnamese[4], Chinese[5]). In the European languages, most words are represented as a single token, and only few span more, *e.g.* "in addition", "for instance". In contrast, the number of words represented by two or more tokens in Chinese, Vietnamese, Korean and Japanese are considerably large (see Figure 9.1). Thus, the ability to efficiently search a sequence of terms using quotes is desirable for most Asian languages. Our current system does not support searching a sequence of terms using quotes. When entering more than one term into the system, the system will search for the occurrence of each in different documents, and return the element/document that contains one of these terms. Even though the current implementation does not directly support *analytic* languages, the current system will search for any random permutation of the tokens comprising a word from an *analytic* language.

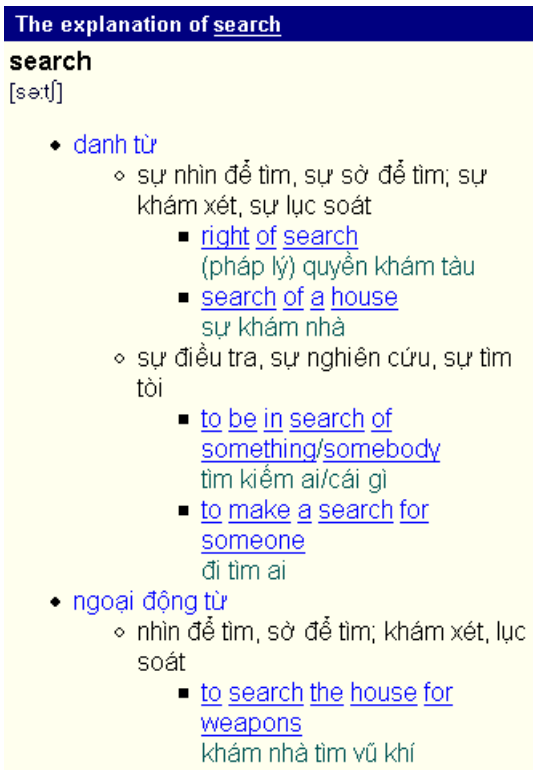

Figure 9.1 : An example of a Vietnamese text taken from an online dictionary

[1] http://en.wikipedia.org/wiki/Synthetic_language
[2] http://en.wikipedia.org/wiki/Analytic_language
[3] http://en.wikipedia.org/wiki/European_languages
[4] http://en.wikipedia.org/wiki/Vietnamese_language
[5] http://en.wikipedia.org/wiki/Chinese_language

The current stop word filter implemented in KALCHAS relates to the English language. Extending the project to include documents written in other native languages would introduce new challenges: (i) How to identify the language used within the document, (ii) How to construct reasonable lists of stop words for the filter, (iii) Is it feasible to construct a stop words for all languages?

The XML standard recommends that all documents include information about the character encoding used (*e.g.* UTF-8, UTF-16, ISO-8859-1, etc). While this determines the character encoding, it does not reveal the language used in the content. However, even if we were able to determine the language of a given document, not all languages are suited for stop word filtering; *e.g. analytic* languages as described in Section 9.2.3 contain words spanning multiple tokens in contrast to *synthetic* languages where a majority of words consists of a single token. This characteristic of *analytic* languages makes it impossible to have stop word filtering for all languages; however, the keyword search in KALCHAS does not support searching for composite terms, as observed in *analytic* languages.

Performing term frequent analysis on documents results in histograms usable for identifying important terms. An interesting topic for future work would be to experiment with a dynamic stop word filtering, based on the word frequencies obtained from individual documents, and only index the tokens occurring rarely within the indexed document.

9.2.8 Distributed Searches

Working in a small office settings, we imagine that KALCHAS could be used for searching the whole office for specific documents. This could be implemented by allowing Kalchas Explorer to perform searches on desktops connected to the LAN, in somewhat the same way as common peer-to-peer file sharing programs.

9.2.9 Handheld Devices

Following the idea of distributed searches, we could imagine porting KALCHAS to BlueTooth or W-LAN enabled handheld devices. A scenario could be that, entering a building would allow one to query other handheld devices in the same building.

Appendix A

Source Code

The following sections provide the source code for the most interesting operations and algorithms discussed earlier in the reprort.

A.1 Using KALCHAS API

The following source code demonstrates how to implement a simple application embedding the KALCHAS API.

Listing A.1 : An example of using KALCHAS API

```
// Include the Kalchas API header file
#include <kalchas.h>
#include <list>
#include <string>

// Open the Kalchas API namespace
using namespace kalchas_api;
using namespace std;

int main(int argc, char **argv) {
  // Instantiate Kalchas
  cKalchasAPI kalchasInstance;
  // An error value variable
  eKalchasErrorCode retVal;

  cout << "Adding:" << endl;
  // Add a file into the index
  retVal = kalchasInstance.AddFile("bibliography.xml");

  // Output the status message from Kalchas
  cerr << cKalchasAPI::ErrorCodeToString(retVal) << endl;

  cout << "Querying" << endl;

  // Create the query as a list of terms
```

```
  list<string> query;
  query.push_back("Engine");
  query.push_back("Sergey");

  // Declare a result pointer variable
  tKalchasQueryResult *result;

  // Query the index
  retVal = kalchasInstance.QueryRetrieve(query, &result);

  // Output the status message from Kalchas
  cerr << cKalchasAPI::ErrorCodeToString(retVal) << endl;
  return 0;
}
```

A.2 Example Plugin: PGP File Support

The following source code demonstrates how to implement a file support plugin for KALCHAS. The plugin is able to index PGP files by simply reporting that the file is a PGP file; this is the only indexable data in PGP which consists of checksums for PGP encryption.

Listing A.2 : Example plugin – PGP file support

```
// Include the Kalchas File Support Header
#include <KalchasFileSupport.h>
// Include the STL string library
#include <string>
// Include the STL list library
#include <list>
// Open the std namespace
using namespace std;
// Associate our plugin with the other plugins
namespace kalchas_plugins {
  // Define our class as an implementation of the
  // cKalchasFileSupport interface
  class cPluginPGP : public cKalchasFileSupport {
  protected:
    // Current file name
    string m_Filename;
    // List of tokens generated from the file
    list<string> m_Tokens;
    // Iterator in m_Tokens
    list<string>::iterator m_Iterator;
    // Last XML
    string m_LastXML;
    // Tokenizes a string and pushes all tokens
    // onto the m_Tokens list
    void TokenizeField(string field) {
      string buffer = "";
      for (unsigned int i=0; i<field.length(); i++) {
        char ch = field[i];
```

```
      if (KALCHAS_TOKEN_CHAR(ch))
        buffer += KALCHAS_FORMAT_CHAR(ch);
      else {
        if (buffer != "") m_Tokens.push_back(buffer);
        buffer = "";
      }
    }
    if (buffer != "") m_Tokens.push_back(buffer);
  }
public:
  // Empty constructor
  cPluginMP3() {}
  // Setup processing of a new file
  virtual bool Initialize(const char *p_Filename) {
    m_Filename = p_Filename;
    m_Tokens.clear();
  }
  // Extract tokens from the opened file
  virtual bool Process() {
    TokenizeField("Pretty Good Privacy PGP File");
    m_Iterator = m_Tokens.begin();
    return true;
  }
  // Retrieve postings
  virtual bool GetNext(char * p_DestToken,
                       unsigned int &p_DestTokenLength,
                       unsigned int * p_DestDewey,
                       unsigned int &p_DestDeweyLength) {
    // Return false if the end is reached
    if (m_Iterator == m_Tokens.end()) return false;
    else {
      // Copy the token string
      strcpy(p_DestToken, (*m_Iterator).c_str());
      p_DestTokenLength = (*m_Iterator).length();
    // Setup a dummy location
    p_DestDewey[0] = 1;
    p_DestDewey[1] = 1;
    p_DestDeweyLength = 2;
    // Advance the iterator
    m_Iterator ++;
    // Signal that we have not reached the end
    return true;
  }
}
// Release any intermediate memory allocated
virtual bool Deinitialize() {
  m_Tokens.clear();
  m_Filename = "";
}
// Return the number of extensions supported
virtual const int GetNumExtensions() {
  return 1;
}
// Return the string of the current supported extension
```

```
  virtual const char * GetExtension(const int p_Which) {
    return "PGP";
  }
  // Release all data (none)
  virtual void DestroyInstance() {}
  // Tranform a Dewey Path into readable XML, this is
  // stored in the "<value>" tag of the output
  virtual const char * RetrieveXML(const int *p_DeweyPath,
                                   const int p_DeweyPathLength,
                                   const char *p_Filename) {
    m_LastXML = "<filetype>Pretty Good Privacy File - PGP</filetype>";
    return m_LastXML.c_str();
  }
};
// Macro that takes care of the export functions for the plugin
KALCHAS_EXPORT_DLL(cPluginPGP);
};
```

A.3 *meet* Operator

KALCHAS' implementation of the MEET-SCAN *meet* operator, as proposed in Section 4.3 on page 22, is shown in the following source code. The code relies on a range of non-standard functions, such as `LongestCommonPrefix` and `LongestCommonPrefixLength` from the `kalchas_kernel` namespace. `cMeetOperator::Process` takes a linked list of postings as input and computes the results using a sorted container.

Listing A.3 : C++ implementation of the MEET-SCAN *meet* operator

```
#include <meetoperator.h>

namespace kalchas_kernel {
  /* Implementation of the Meet operator */
  void cMeetOperator::Process(tDeweyList &pInput,
                              tResultSet &pResultSet)
  {
    /*
      Check if the input set is empty
    */
    if (pInput.size() < 1) return;

    /*
      Check if the input set only has one element
    */
    if (pInput.size() == 1) {
      pResultSet.push_back(*(pInput.begin()));
      return;
    }

    /*
      Create an iterator
    */
```

```
tDeweyList::iterator i = pInput.begin();

/*
  Set the current working meet to the first element
*/
cDewey current_meet = *i;

/*
  Counter for measuring unions of multiple occurrences and
  multiple terms in the same meet
*/
int hitcounter = 0;

/*
  Should the last calculated meet be added to the result
*/
bool pushLast = false;


/*
  Iterate through the whole set
*/
while (i!=pInput.end()) {
  cDewey & current_dewey = *i;
  pushLast = false;

  // RULE 4
  if (hitcounter > 20) {
    pResultSet[hitcounter].push_back(current_meet);
    hitcounter = 0;
    current_meet = current_dewey;
    pushLast = true;
  } else {
    // RULE 1
    if (current_meet.GetDocumentID() !=
        current_dewey.GetDocumentID()) {
      pResultSet[hitcounter].push_back(current_meet);
      hitcounter = 0;
      current_meet = current_dewey;
      pushLast = true;
    } else {
      int lca_length =
        current_meet.
        LongestCommonPrefixLength(current_dewey);

      // RULE 2
      if (lca_length < 2) {
        pResultSet[hitcounter].push_back(current_meet);
        hitcounter = 0;
        current_meet = current_dewey;
        pushLast = true;
      } else {
        cDewey new_meet =
          current_meet.
```

```
              LongestCommonPrefix(current_dewey);

            new_meet.SetKeyword(current_meet.GetKeyword());

            // RULE 3
            if (new_meet.GetKeyword().
                find(current_dewey.GetKeyword()) ==
                string::npos) {
              new_meet.SetKeyword(new_meet.GetKeyword() + "␣" +
                                  current_dewey.GetKeyword());
              hitcounter += 10;
            } else {
              hitcounter ++;
            }
            current_meet = new_meet;
          }
        }
      }
      i++;
    }
    if (pushLast)
      pResultSet[hitcounter].push_back(current_meet);
  }
};
```

A.4 Shredding Using the Expat Parser

In the following we give the source code for the shredder class, inherited from a C++ LibExpat wrapper. The key functions are `OnStartElement`, `OnEndElement`, and `OnCharacterData`. The `cShredderExpat` class is used to create an Expat instance to shred a given XML document while tokenizing all terms and calculating `<term,location>` postings.

Listing A.4 : C++ implementation of the shredder function

```
namespace kalchas_kernel {
  class cShredderExpat : public cExpat<cShredderExpat> {
  protected:
    tOrderdedDeweyList *m_Postings;
    tDeweyPath m_WorkingDewey;
  public:
    /*
       Instantiate the shredded and setting default values.
     */
    cShredderExpat(tOrderdedDeweyList *pDestPostings)
      : m_Postings(pDestPostings) {
      // make room for Document ID / Document Root
      m_WorkingDewey.push_back(0);
    }

    /**
       Return the number of shredded postings from the current
```

```
    XML document
  */
int NumPostings() {
  return (int)m_Postings->size();
}

/**
    Retrieve all postings as a list of Dewey Paths
    tDeweyList a list of <term, location> postings sorted by
    location.
  */
tOrderdedDeweyList * GetPostings() {
  return m_Postings;
}

/**
    Setup the Start, End and Character handlers of Expat
  */
void OnPostCreate() {
  EnableStartElementHandler();
  EnableEndElementHandler();
  EnableCharacterDataHandler();
}

/**
    Process the start of an element e.g. <element>
  */
void OnStartElement(const XML_Char *pszName,
                    const XML_Char **papszAttrs){
  m_WorkingDewey.back()++;
  m_WorkingDewey.push_back(0);
}

/**
    Process the end of an element e.g. </element>
  */
void OnEndElement(const XML_Char *pszName) {
  m_WorkingDewey.pop_back();
}

/**
    Returns true if the given input is allowable
    (alphanumeric)
  */
bool inline TokenChar(const char &p_Input) {
  return    (p_Input >= 'a' && p_Input <= 'z') ||
    (p_Input >= 'A' && p_Input <= 'Z') ||
    (p_Input >= '0' && p_Input <= '9');
}

/**
    Process the character data within the current XML Element
  */
void OnCharacterData(const XML_Char *pszData,
```

```
                          int nLength)
    {
      if (nLength < 2) return;
      cDewey dewey(m_WorkingDewey);

      char token_str [512];
      int token_length = 0;

      int idx = 0;
      while (idx < nLength) {
        int startIdx = idx;

        token_length = 0;
        // If the token read is alphanumeric
        while (TokenChar(pszData[idx]) && idx < nLength) {
          // convert to lowercase
          char ch = pszData[idx];
          if (ch >= 'A' && ch <='Z') ch -='A' - 'a';
          token_str[token_length ++] = ch;
          idx++;
        }

        if (idx != startIdx) {
          string token(token_str, token_length);
          if (!StopWords->IsAStopWord(token)) {
            (*m_Postings)[token] = dewey;
          }
        }

        idx++;
      }
    }
  };
}
```

Bibliography

[1] "Overview of SGML Resources." http://www.w3.org/MarkUp/SGML/.

[2] "HyperText Markup Language (HTML)." http://www.w3.org/MarkUp/.

[3] "eXtensible Markup Language (XML)." http://w3.org/XML/.

[4] H. Jiang, *Efficient Structural Query Processing in XML Databases*. PhD thesis, The Hong Kong University of Science and Technology, May 13 2004.

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "eXtensible Markup Language (XML) 1.0," recommendation, W3C, February 10 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[6] R. W. Luk, H. V. Leong, T. S. Dillon, A. T. Chan, W. B. Croft, and J. Allan, "A Survey in Indexing and Searching XML Documents," *Journal of the American Society for Information Science and Technology (JASIST)*, vol. 53, no. 6, pp. 415–437, 2002.

[7] P. I. for Journalists, "Microsoft `Office 12` XML File Formats to Give Customers Improved Data Interoperability and Dramatically Smaller File Sizes." http://www.microsoft.com/presspass/press/2005/jun05/06-01OfficeXMLFormatPR.msp

[8] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks*, vol. 30, pp. 107–117, April 1 1998.

[9] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Agarwal, "Dynamic Maintenance of Web Indexes Using Landmarks," in *Proc.* 12^{th} *Int'l Conf. World Wide Web (WWW'03)*, pp. 102–111, ACM Press, May 20-24 2003.

[10] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina, "Building a Distributed Full-Text Index for the Web," *ACM Transactions on Information Systems*, vol. 19, no. 3, pp. 217–241, 2001.

[11] G. Linoff and C. Stanfill, "Compression of Indexes with Full Positional Information in Very Large Text Databases," in *Proc.* 16^{th} *Annual Int'l ACM-SIGIR Conf. Research and Development in Information Retrieval (SIGIR'93)* (R. Korfhage, E. M. Rasmussen, and P. Willett, eds.), pp. 88–95, ACM, June 27 - July 1 1993.

[12] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML Query Language," working draft, World Wide Web Consortium (W3C), February 11 2005.
http://www.w3.org/TR/xquery/.

[13] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," in *Proc. 2003 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD´03)*, pp. 16–27, ACM, June 9-12 2003.

[14] A. Theobald and G. Weikum, "The Index-Based XXL Search Engine for Querying XML Data With Relevance Ranking," in *Proc.* 8^{th} *Int'l Conf. Extending Database Technology (EDBT'02*, pp. 477–495, Springer-Verlag, June 2002.

[15] "XQL FAG."
http://www.ibiblio.org/xql/.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "eXtensible Markup Language (XML) 1.0," recommendation, W3C, February 04 2004.
http://www.w3.org/TR/REC-xml/.

[17] A. Schmidt, M. L. Kersten, and M. Windhouwer, "Querying XML Documents Made Easy: Nearest Concept Queries," in *Proc.* 17^{th} *Int'l Conf. Data Engineering (ICDE'01)*, pp. 321–329, IEEE Computer Society, April 2-6 2001.

[18] A. Schmidt, "Architecting an XML Search Engine with the Meet-Operator." Unpublished draft, September 21 2004.

[19] D. A. Nørgaard and R. C. Kaae, "Engineering an XML Full-Text Index," Master's thesis, Aalborg University, January 2005. First part of the Masters thesis.

[20] J. Boyer, "Canonical XML v1.0," recommendation, W3C, March 15 2001.
http://www.w3.org/TR/xml-c14n.

[21] M. Mealling and R. Denenberg, "RFC 3305 – URIs, URLs, and URNs," request for comments: 3305, W3C, August 2002.
http://www.ietf.org/rfc/rfc3305.txt.

[22] A. Tomasic, H. Garcia-Molina, and K. Shoens, "Incremental Updates of Inverted Lists for Text Document Retrieval," in *Proc. 1994 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD'94)*, pp. 289–300, ACM Press, May 24-27 1994.

[23] Sleepycat Software, Inc., *Berkeley DB Reference Guide*, 4.3.27 ed., 2004.
http://www.sleepycat.com/docs/ref/toc.html.

[24] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *Proc. 1999 USENIX Annual Technical Conf.*, pp. 183–192, June 6-11 1999.
See http://www.usenix.org/events/usenix99/olson.html.

[25] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, "Compression of inverted indexes for fast query evaluation.," in *SIGIR*, pp. 222–229, 2002.

[26] M. Dewey, "Dewey Decimal Classification," *Gutenberg.org*, 2004.
http://www.gutenberg.org/dirs/1/2/5/1/12513/.

[27] J. Zobel, A. Moffat, and R. Sacks-Davis, "An Efficient Indexing Technique for Full-Text Database Systems," in *Proc. 18^{th} Int'l Conf. Very Large Data Bases (VLDB'92)*, pp. 352–362, Morgan Kaufmann, August, 1992.

[28] B. B. Yao, M. T. Ozsu, and N. Khandelwal, "XBench Benchmark and Performance Testing of XML DBMSs," in *Proc. 20^{th} Int'l Conf. Data Engineering (ICDE'04)*, pp. 621–633, IEEE Computer Society, March-April, 2004.

[29] A. Trotman, "Compressing Inverted Files," *Information Retrieval*, vol. 6, pp. 5–19, January 2003.

[30] A. N. Vo and A. Moffat, "Compressed Inverted Files with Reduced Decoding Overheads," in *Proc. 21^{st} Annual Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR'98)*, pp. 290–297, ACM Press, 1998.

[31] J. Zobel and A. Moffat, "Adding Compression to a Full-text Retrieval System," *Softw., Pract. Exper.*, vol. 25, no. 8, pp. 891–903, 1995.

[32] M. Geelnard, "Basic Compression Library, v1.1." http://bcl.sourceforge.net/.

[33] M. Nelson, "Data Compression with the Burrows-Wheeler Transform," 1996.

[34] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications," in *Proc. 12^{th} Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, pp. 181–192, Springer-Verlag, July 1-4 2001.

[35] D. Kunkle, "Empirical Complexities of Longest Common Subsequence Algorithms," June 12, 2002.

[36] H. Chu and M. Rosenthal, "Search Engines For The World Wide Web: A Comparative Study And Evaluation Methodology," in *Proc. ASIS Annual Conference*, October 19-24 1996.

[37] J. Clark, "The Expat XML Parser, v1.95.8." http://expat.sourceforge.net/.

[38] "Berkeley DB v4.3.27." http://www.sleepycat.com/products/db.shtml.

[39] "C++ Boost Libraries, v1.32.0," 2004. http://boost.org.

[40] "GNU Common C++." http://cplusplus.sourceforge.net/.

[41] "Copernic Desktop Search." http://www.copernic.com.

[42] "Google Desktop." http://desktop.google.com.

[43] "Spotlight - Find Anything On Your Mac Instantly," 2005.
http://www.apple.com/macosx/features/spotlight/.

[44] "MSN Toolbar."
http://toolbar.msn.com.

[45] "Yahoo! Desktop Search."
http://desktop.yahoo.com.

[46] "Standard Template Library Programmer's Guide."
http://www.sgi.com/tech/stl/.

[47] N. Lester, J. Zobel, and H. E. Williams, "In-place versus Re-build versus Re-merge: Index Maintenance Strategies for Text Retrieval Systems," in *Proc. 27^{th} Australasian Computer Science Conf. (ACSC'04)*, pp. 15–23, Australian Computer Society, Inc., January 2004.

[48] G. Cobéna, S. Abiteboul, and A. Marian, "Detecting Changes in XML Documents," in *Proc. 18^{th} Int'l Conf. Data Engineering (ICDE'02)*, pp. 41–52, IEEE Computer Society, February 26–March 1 2002.

[49] Y. Wang, D. J. DeWitt, and J.-Y. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents," in *Proc. 19^{th} Int'l Conf. Data Engineering (ICDE'03)*, pp. 519–530, IEEE Computer Society, March 5-8 2003.

[50] G. K. Zipf, *Human Behaviour and the Principle of Least-Effort: An Introduction to Human Ecology*. Addison-Wesley, Cambridge, MA, 1949.
See also http://en.wikipedia.org/wiki/Zipf's_law and http://www.iridis.com/Zipf's_law.

[51] Jon Bosak, "The Plays of Shakespeare in XML."
http://www.oasis-open.org/cover/bosakShakespeare200.html.

[52] "XMark – An XML Benchmark Project," 2003.
www.xml-benchmark.org.

[53] C. Andersen, T. Boesen, and D. Pedersen, "Querying XML using the Meet operator," 2004.