EFFICIENCY AND EFFECTIVENESS OF XML KEYWORD SEARCH USING A FULL ELEMENT INDEX

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Duygu Atılgan

August, 2010

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Özgür Ulusoy (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Fazlı Can

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Ahmet Coşar

Approved for the Institute of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Institute

ABSTRACT

EFFICIENCY AND EFFECTIVENESS OF XML KEYWORD SEARCH USING A FULL ELEMENT INDEX

Duygu Atılgan
M.S. in Computer Engineering
Supervisor: Prof. Dr. Özgür Ulusoy
August, 2010

In the last decade, both the academia and industry proposed several techniques to allow keyword search on XML databases and document collections. A common data structure employed in most of these approaches is an inverted index, which is the state-of-the-art for conducting keyword search over large volumes of textual data, such as world wide web. In particular, a full element-index considers (and indexes) each XML element as a separate document, which is formed of the text directly contained in it and the textual content of all of its descendants. A major criticism for a full element-index is the high degree of redundancy in the index (due to the nested structure of XML documents), which diminishes its usage for large-scale XML retrieval scenarios.

As the first contribution of this thesis, we investigate the efficiency and effectiveness of using a full element-index for XML keyword search. First, we suggest that lossless index compression methods can significantly reduce the size of a full element-index so that query processing strategies, such as those employed in a typical search engine, can efficiently operate on it. We show that once the most essential problem of a full element-index, i.e., its size, is remedied, using such an index can improve both the result quality (effectiveness) and query execution performance (efficiency) in comparison to other recently proposed techniques in the literature. Moreover, using a full element-index also allows generating query results in different forms, such as a ranked list of documents (as expected by a search engine user) or a complete list of elements that include all of the query terms (as expected by a DBMS user), in a unified framework.

As a second contribution of this thesis, we propose to use a lossy approach, static index pruning, to further reduce the size of a full element-index. In this

way, we aim to eliminate the repetition of an element's terms at upper levels in an adaptive manner considering the element's textual content and search system's ranking function. That is, we attempt to remove the repetitions in the index only when we expect that removal of them would not reduce the result quality. We conduct a well-crafted set of experiments and show that pruned index files are comparable or even superior to the full element-index up to very high pruning levels for various ad hoc tasks in terms of retrieval effectiveness.

As a final contribution of this thesis, we propose to apply index pruning strategies to reduce the size of the document vectors in an XML collection to improve the clustering performance of the collection. Our experiments show that for certain cases, it is possible to prune up to 70% of the collection (or, more specifically, underlying document vectors) and still generate a clustering structure that yields the same quality with that of the original collection, in terms of a set of evaluation metrics.

Keywords: Information Retrieval, XML Keyword Search, Full Element-Index, LCA, SLCA, Static Pruning, Clustering.

ÖZET

TAM ELEMAN İNDEKSİ KULLANARAK XML ANAHTAR SÖZCÜK ARAMANIN VERİMLİLİK VE ETKİLİLİĞİ

Duygu Atılgan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Özgür Ulusoy
Ağustos, 2010

Son yıllarda akademide ve endüstride, XML veritabanları ve belge derlemlerinde anahtar sözcük aramak için çeşitli teknikler önerilmiştir. Bu tekniklerin pek çoğunda kullanılan veri yapısı, dünya çapında ağ (WWW) gibi büyük metin verileri üzerinde anahtar sözcük aramada en gelişmiş teknik olan ters indekstir. Bir tam eleman indeksi her bir XML elemanını, metni, kendisinin doğrudan içeriği ve torunlarının içeriklerinden oluşan ayrı bir belge olarak düşünür ve indeksler. Tam eleman indekse yöneltilen önemli bir eleştiri (XML belgelerinin iç içe yapısından dolayı) yüksek derecede fazlalık içermesidir. Bu durum tam eleman indeksin büyük ölçekli XML erişimi durumlarında kullanımını azaltır.

Bu tezde XML anahtar sözcük arama için tam eleman indeksinin kullanımının verimlilik ve etkiliği araştırılmaktadır. Öncelikle, kayıpsız indeks sıkıştırma tekniklerinin tam eleman indeksinin büyüklüğünü önemli ölçüde azaltabileceği, böylece tipik bir arama motorundaki sorgu işleme stratejilerinin böyle bir indeks üzerinde verimli bir şekilde çalışabileceği öne sürülmektedir. Bir tam eleman indexinin en önemli dezavantajı boyutunun büyüklüğüdür. Bu sorun çözüldüğü takdirde bu tip indeks kullanımının, sonuç kalitesi (etkililik) ve sorgu işleme performansını (verimlilik) son zamanlarda önerilen diğer tekniklere kıyasla geliştirebileceği gösterilmektedir. Ayrıca tam eleman indeksi kullanmak, birleşik bir taslakta sorgu sonuçlarını, sıralı belge listesi (bir arama motorunun kullanıcısının beklediği şekilde) ya da sorgu sözcüklerinin tümünü içeren eleman listesi (bir veritabanı sistemi kullanıcısının beklediği şekilde) gibi farklı formlarda oluşturmaya olanak sağlar.

Bu tezin ikinci bir katkısı olarak, tam eleman indeksin büyüklüğünü daha

da azaltmak için kayıplı bir yaklaşım olan statik budama tekniğinin kullanılması önerilmektedir. Bu şekilde, bir elemanın sözcüklerinin yukarı seviyelerdeki tekrarının, elemanın metinsel içeriği ve arama motorunun sıralama işlevi dikkate alınarak, uyarlanabilir bir şekilde azaltılması amaçlanmaktadır. Yani indeksteki tekrarlamaların, çıkarılmaları sonuç kalitesini azaltmadığı takdirde, ortadan kaldırılmasına çalışılmaktadır. Deneysel çalışmalarla, budanmış indeks dosyalarının çok yüksek budama seviyelerine kadar, erişim etkililiği açısından, tam eleman indeksiyle karşılaştırabilir, hatta ondan daha iyi olduğu gösterilmektedir.

Son olarak, indeks budama stratejilerinin, bir XML derleminin belge vektörlerinin büyüklüklerinin azaltılarak gruplama performansının geliştirilmesinde kullanılması önerilmektedir. Deneyler, belli durumlar için, koleksiyonun %70 kadarı budanarak, bir grup değerlendirme metriğine göre, orijinal koleksiyonla aynı kaliteyi sağlayan bir gruplama yapısı oluşturulabildiğini göstermektedir.

Anahtar sözcükler: Bilgiyi Geri Alma, XML Anahtar Sözcük Arama, Tam Eleman İndeksi, LCA, SLCA, Statik Budama, Gruplandırma.

Acknowledgement

I would like to express my sincere gratitude to my supervisor Prof. Dr. Özgür Ulusoy for his invaluable support and guidance during this thesis.

I am also thankful to Prof. Dr. Fazlı Can and Assoc. Prof. Dr. Ahmet Coşar for kindly accepting to be in the committee and spending their time to read and review my thesis. I am indepted to Dr. Sengör Altıngövde not only for his endless help and support in this research but also for his friendship. I also want to thank my officemates Rıfat and Şadiye for sharing the office with me.

I am grateful for the financial support of The Scientific and Technological Research Council of Turkey (TÜBİTAK-BİDEB) for two years during this thesis.

I would like to thank my friends Nil, Emre, Aslı, Nilgün, Funda, Özlem, Eda and Büşra for their valuable friendship and understanding. Special thanks go to Kamer for his existence.

And last but most of the my gratitude goes to my dearest family. Nothing makes sense without their love. To them, I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been an abundance of Extensible Markup Language (XML) data on the World Wide Web (WWW) and elsewhere. In addition to being used as a storage format for WWW, XML is also used as an encoding format for data in several domains such as digital libraries, databases, scientific data repositories and web services. This increasing adoption of XML has brought the need to retrieve XML data efficiently and effectively. As XML documents have a logical structure, retrieval of XML data is different from classic 'flat' document retrieval in some ways. While most of the previous works in the Information Retrieval (IR) field presume a document as the typical unit of retrieval, XML documents allow a finer-grain retrieval at the level of elements. Such an approach is expected to provide further gains for the end users in locating the specific relevant information, however, it also requires the development of systems to effectively access XML documents.

Although a large amount of research has been going on to retrieve XML documents, a consensus hasn't been reached yet about the retrieval strategy for many reasons. The issues that are being researched by XML retrieval community could be listed as querying, indexing, ranking, presenting and evaluating [23]. In

this thesis, we focus on querying, indexing and ranking of XML documents for an efficient and effective keyword search.

1.2 Contributions

In the last decade, especially under the INitiative for the Evaluation of XML retrieval (INEX) [18] campaigns, a variety of indexing, ranking and presentation strategies for XML collections have been proposed and evaluated. Given the freshness of this area, there exist a number of issues that are still under debate. One such fundamental problem is indexing XML documents. The focused XML retrieval aims to identify the most relevant parts of an XML document to a query, rather than retrieving the entire document. This requires constructing an index at a lower granularity, say, at the level of elements, which is not a trivial issue given the nested structure of XML documents.

Element-indexing is a crucial mechanism for supporting content-only (CO) queries over XML collections. It creates a full element-index by indexing each XML element as a separate document. With this method, each element is formed of the text directly contained in it and the textual content of all of its descendants. However, this results in a considerable amount of repetition in the index as the textual content occurring at level n of the XML logical structure is indexed n times. Due to this redundancy in the index, element indexing is criticized for yielding efficiency problems and its promises are rarely explored. In this thesis, we investigate the effectiveness and efficiency of using a full element-index for XML keyword search over XML databases and document collections.

Following a brief discussion of the related work in the next chapter, in Chapter 3, we propose to use state-of-the-art IR query processing techniques that operate on top of a full element-index (with some slight modifications). We show that such an index can be simple yet efficient enough for supporting keyword searches on XML data to satisfy the requirements of both DB and IR communities. We build an XML keyword search framework which uses document ordered processing and

apply different query result definition techniques. Query result definition, one of the biggest challanges in XML keyword search, aims to find the 'closely related' nodes that are 'collectively relevant' to the query [37]. Smallest Lowest Common Ancestor (SLCA) method is one of the query result definition methods widely used for XML keyword search. The notion of SLCA is first proposed in XKSearch system [36] and afterwards employed in other result definition techniques such as Valuable Lowest Common Ancestor (VLCA) [24], Meaningful Lowest Common Ancestor (MLCA) [25] and MaxMatch [26]. As SLCA is a widely used technique, it is crucial to implement it efficiently. Within our framework we implement a novel query processing method to find SLCA nodes and evaluate our method through a comprehensive set of experiments. We compare the performance of our query processing strategy using a full element-index to that of the strategies which use a Dewey-encoded index [36]. The experiments show that the full element-index with document ordered query processing could improve both the result quality (effectiveness) and query execution performance (efficiency) in comparison to XKSearch system.

In Chapter 4, we aim to increase the efficiency further by reducing the index size. For this purpose, we propose using static index pruning techniques for obtaining more compact index files that can still result in comparable retrieval performance to that of an unpruned full index. We also compare our approach with some other strategies which make use of another common indexing technique, leaf-only indexing. Leaf-only indexing creates a 'direct index' which only indexes the content that is directly under each element and disregards the descendants. This results in a smaller index, but possibly in return to some reduction in system effectiveness. Our experiments conducted along with the lines of INEX evaluation framework reveal that pruned index files yield comparable to or even better retrieval performance than the full index and direct index, for several tasks in the ad hoc track of INEX.

In Chapter 5, we investigate the usage of index pruning techniques on another aspect of XML retrieval which is clustering XML collections. First, we employ the well known Cover-Coefficient Based Clustering Methodology (C3M) for clustering XML documents and evaluate its effectiveness. Then, we apply the index pruning

techniques from the literature to reduce the size of the document vectors of the XML collection. Our experiments show that, for certain cases, it is possible to prune up to 70% of the collection (or, more specifically, underlying document vectors) and still generate a clustering structure that yields the same quality with that of the original collection, in terms of a set of evaluation metrics.

Finally, we conclude and point to future work directions in Chapter 6.

Chapter 2

Related Work

In this chapter, we briefly present some of the research literature related to keyword search in unstructured and structured documents. We also review compression methods which are employed in this thesis to reduce the sizes of the indexes.

2.1 Keyword Search for Unstructured Documents

Keyword search, being used by millions of people all over the world now, is an effective, user friendly way for querying HTML documents. As it does not require any knowledge of the collection, the user can create queries intuitively and fulfill his information need. Such a popular method should be supported with efficient retrieval strategies to meet the needs of the users.

In terms of the retrieval strategies, keyword search in a collection could be done by linear scanning in the most simple and naive way. However, to be able to process the queries in a reasonable amount of time, an index structure is needed for any information retrieval strategy. With the help of such a structure, one could determine the list of documents that contain a term and make a boolean

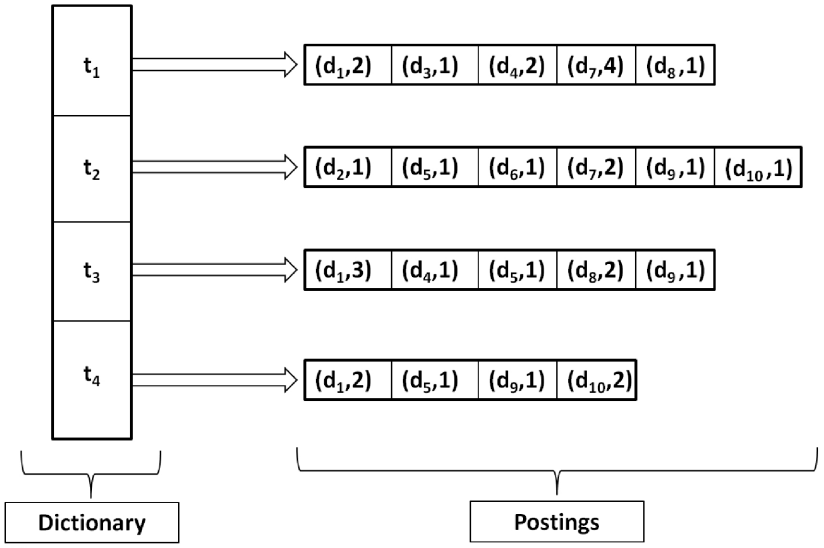


Figure 2.1: Structure of an inverted index

search. However, if query ranking should also be supported, an inverted index which also stores the frequency of each occurrence of a term in a document would be the optimal data structure. Currently, the inverted index is the state-of-the-art data structure of the search engines for ranked retrieval of the documents. The basic structure of an inverted index is shown in Figure 2.1, which is comprised of dictionary and postings. For each term in the dictionary, there is a posting list which lists the documents that the corresponding term occurs in. Each item in the posting list is called a posting and contains the document id and frequency of the term (Term positions could also be included, if phrase or proximity queries should be supported as well). The posting list is sorted in the order of the document id of the postings which is useful for the compression of the inverted list.

Query processing over an inverted index could be classified into two as document ordered processing and term ordered processing according to the processing order of the postings:

- Term Ordered Processing (TO): Term ordered processing, also called term-at-a-time evaluation in [34], processes the posting lists sequentially. For

a conjunctive query consisting of one or more keywords, the method finds the documents containing all of the query terms by intersecting the posting lists in a sequential manner. Once the postings of a term are completely handled, then the postings of the next term could be processed.

- Document Ordered Processing (DO): The disadvantage of term ordered processing is that one should wait for all the posting lists to be processed to obtain a complete score. However, if posting lists could be processed in parallel instead of sequentially, then query results could be returned at the time of processing. In document ordered processing, the posting lists are treated in parallel by making use of the fact that once a document id is seen in a posting, there can not be a smaller document id in one of the succeeding postings in that list since the postings of a term are stored in increasing order of document ids.

2.2 Keyword Search for XML

Keyword search is a user-friendly way for accessing structured data. It is easy to use and it does not require the knowledge of complex schemas or query languages. Also more meaningful results could be returned by exploiting structural information instead of returning a list of unranked results as in query languages. However, there are challenges of accessing structured data as it requires different strategies in retrieval process. Different than the flat documents, XML documents are modeled as labeled trees with a hierarchical semantic structure. Due to this hierarchical structure, researchers are faced with various challenges regarding the retrieval tasks such as indexing, query processing, etc. In this section, we review the literature in terms of the different aspects of XML keyword search each of which corresponds to a stage in the retrieval process.

2.2.1 Node Labeling Schemes

Node labeling is a way to identify the nodes of an XML tree. While there are many ways to identify a node, the recently developed techniques aim to find a matching between the label of a node and its structural relationships with the other nodes. A number of labeling schemes are proposed to represent the nodes of XML trees and to support structural queries. In this thesis, we investigate the usage of Dewey encoding [31] and tree traversal order encoding [11] which are labeling types of prefix based labeling and subtree labeling, respectively.

First XML numbering scheme based on tree traversal order is introduced by Dietz [11]. In this scheme, a node v is labeled with a pair of unique integers $pre(v)$ and $post(v)$ which correspond to the preorder and postorder traversal ids of v. In other words, $pre(v)$ is the id assigned to the node v when it is visited for the first time and $post(v)$ is the id assigned to v when it is visited for the last time. Tree traversal labeling of a sample XML tree is shown in Figure 2.2. Note that for two given nodes u and v of a tree T, the following are true:

- $pre(v) < post(v)$ for each node v of T
- $pre(u) < pre(v)$ and $post(u) > post(v)$ if node u is an ancestor of v
- $post(u) < pre(v)$ if node u is a left sibling of v
- u is an ancestor of v, if and only if u occurs before v in the preorder traversal of T and after v in the postorder traversal.

By using tree traversal order encoding, we can determine ancestor-descendant relationships easily. Nevertheless, parent-child relationship could not be determined directly. This kind of encoding has the advantage of being easy to implement and efficient to use, however, it is inefficient for dynamic XML documents for which node insertions and deletions occur frequently.

On the other hand, most of the recent systems for XML keyword search use Dewey ID labelling scheme which is based on Dewey Decimal Classification

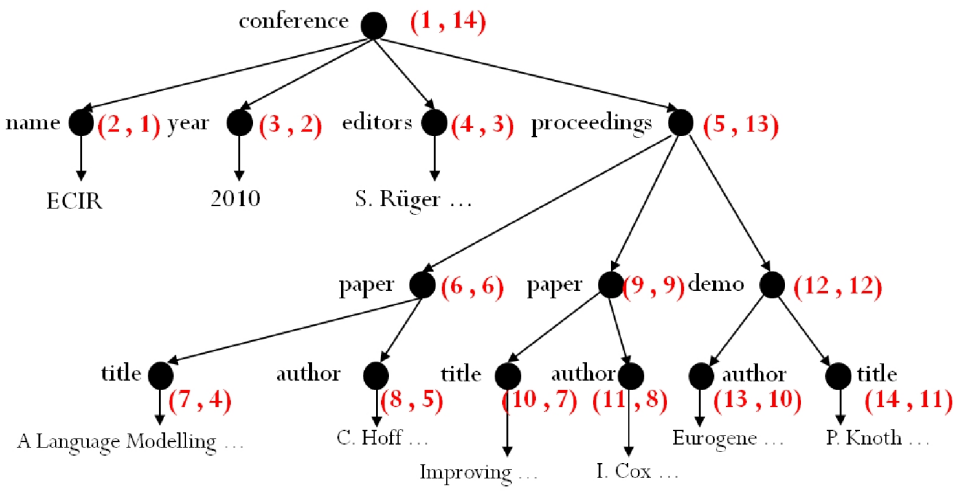


Figure 2.2: Tree Traversal Labeling of an XML Tree

System. In this scheme, the label of a given node encodes the path from the document root down to the node so that the ancestor-descendant relationships between the nodes could be determined directly. According to this, given the nodes u and v, u is an ancestor of v if $label(u)$ is a prefix of $label(v)$. However, in this labeling scheme, the disadvantage is that the size of the label grows with the length of the encoded path which is in the order of the depth of the XML tree in the worst case [17]. In Chapter 3, we compare the sizes of the indexes built using Dewey encoding and tree traversal order encoding both formally and experimentally. Dewey ID labeling of a sample XML tree is shown in Figure 2.3.

2.2.2 Indexing Techniques

In the literature, several techniques are proposed for indexing the XML collections and for query processing on top of these indexes. In a recent study, Lalmas [23] provides an exhaustive survey of indexing techniques -essentially from the perspective of IR discipline- that we briefly summarize in the rest of this section.

The most straightforward approach for XML indexing is creating a full element-index, in which each element is considered along with the content of

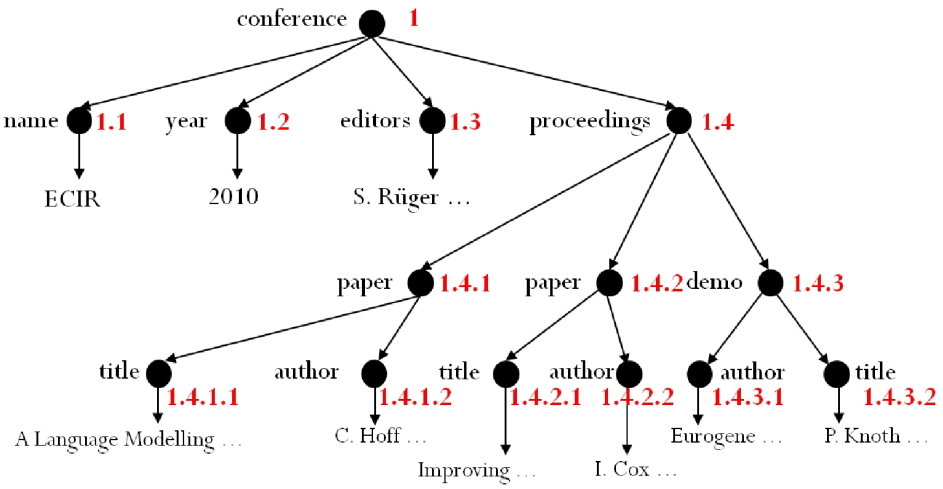


Figure 2.3: Dewey ID Labeling of an XML Tree

its descendants. In this case, how to compute inverse document frequency (IDF), a major component used in many similarity metrics, is an open question. Basically, IDF can be computed across all elements, which also happens to be the approach taken in our work. As a more crucial problem [23], a full element-index is highly redundant because the terms are repeated for each nested element and the number of elements is typically far larger than the number of documents. To cope with the latter problem, an indexing strategy can only consider the direct textual content of each element, so that redundancy due to nesting of the elements could be totally removed. In [13, 14], only leaf nodes are indexed, and the scores of the leaf elements are propagated upwards to contribute to the scores of the interior (ancestor) elements. In a follow-up work [15], the direct content of each element (either leaf or interior) is indexed, and again a similar propagation mechanism is employed. Another alternative is propagating the representations of elements, e.g., term statistics, instead of the scores. However, the propagation stage, which has to be executed during the query processing time, can also degrade the overall system efficiency. The comparison of inverted indexes for full and direct indexing techniques is given in Figure 2.4. As it could be observed from the figure, occurrence of a term t in an element e at depth d is repeated d times in the full index. However, in the direct index, only the elements directly

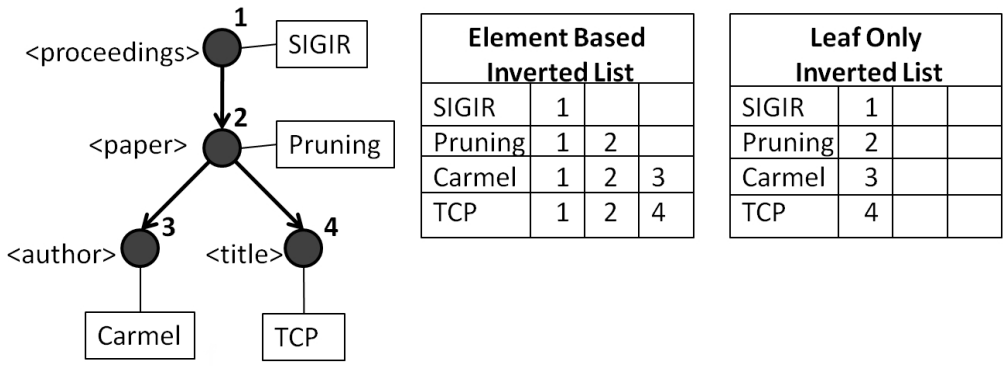


Figure 2.4: Structure of full and direct inverted index

containing the term occur in the posting list.

In the database field, where XML is essentially considered from a data-centric rather than a document-centric point of view, a number of labeling schemes are proposed especially to support structural queries (see [17] for a survey). In XRANK system [16], postings are again only for the textual content directly under an element, however, document identifiers are encoded using Dewey IDs so that the scores for the ancestor elements can also be computed without a propagation mechanism. This indexing strategy allows computing the same scores as a full index while the size of the index can be in the order of a direct index. However, this scheme may suffer from other problems, such as the excessive Dewey ID length for very deeply located elements. An in-between approach to remedy the redundancy in a full element-index is indexing only certain elements of the documents in the collection. Element selection can be based upon several heuristics (see [23] for details). For instance, shorter elements (i.e., with only few terms) can be discarded. Another possibility is selecting elements based on their popularity of being assessed as relevant in the corresponding framework. The semantics of the elements can also be considered while deciding which elements to index by a system designer. Yet another indexing technique that is also related is distributed indexing, which proposes to create separate indexes for each element type, possibly selected with one of the heuristics discussed above. This latter technique may be especially useful for computing the term statistics in a

specific manner for each element type.

2.2.3 Query Processing Techniques

In traditional information retrieval, the typical unit of retrieval is the whole document. However, in XML retrieval only some part of the document could be returned in response to a user query by exploiting the structure of a document. While such a focused strategy helps the user to access the desired data more quickly, it requires more complex strategies to locate the relevant parts of the documents in terms of the retrieval systems. If there is relevant information scattered among different nodes, focused retrieval should assemble these relevant nodes into a single result node. Such challenges of XML keyword search has attracted the researchers to develop more complex query processing and result definition techniques. In the literature, the result nodes are determined either according to the tree structure of the retrieved document or tags of the elements or peer node comparisons [37]. Below we explain these different approaches for determining result nodes:

Result definition according to tree structure:

- ELCA: Exclusive Lowest Common Ancestor method, proposed in [16], finds the lowest common ancestors which include all the keywords after excluding the occurrences of the keywords in sub-elements that already contain all of the query keywords.
- SLCA: Smallest Lowest Common Ancestor method, proposed in [36], finds the smallest lowest common ancestor nodes which contain all the keywords and is not ancestor of any other node which also contains all the keywords. According to this method, the smallest result nodes are considered the most relevant nodes.
- MLCA: Meaningful Lowest Common Ancestor method, proposed in [25], finds the lowest common ancestor of nodes which are meaningfully related.

Meaningfully relatedness concept is defined according to the structural relationships between the nodes containing query terms.

Result definition according to labels/tags:

- XSEarch: In XSEarch system [8], LCA of the interconnected nodes are defined as the result nodes where the interconnection relationship determines whether the nodes are meaningfully related. According to this method, two nodes are interconnected, thereby meaningfully related, if there is no two nodes with the same label on their path.
- VLCA: In this work [24], the notions of Valuable LCA and Compact LCA are proposed to efficiently answer XML keyword queries. Valuable LCA is the LCA of a set of nodes which are homogenous. Homogeneity concept is similar to interconnection relationship between the nodes. The nodes u and v are homogenous if there are no nodes of the same type on u and v's path to root.

Result definition according to peer node comparisons:

- MaxMatch: In this work, XML keyword search is inspected from a formal perspective. The method first finds the SLCA nodes of the query. Afterwards, the relevant matches are chosen from the subtrees rooted at SLCA nodes according to whether they satisfy the two proporties, monotonicity and consistency. Monotonicity states that data insertion (query keyword insertion) causes the number of query results to non-strictly monotonically increase (decrease). Consistency states that after data (query keyword) insertion, if an XML subtree becomes valid to be part of new query results, then it must contain the new data node (a match to the new query keyword) [26]. With this method, the SLCA nodes which have stronger siblings are pruned.

2.2.3.1 Algorithms for Finding LCA and SLCA

In focused retrieval, the most focused results consisting of elements are returned as the answer of a query. The most basic and intuitive method for finding focused results is the lowest common ancestor (LCA) method many extensions of which are developed afterwards. One of these extensions is the smallest lowest common ancestor (SLCA) proposed in XKSearch system by Xu et al. [36]. In this thesis, we implement an efficient method for finding SLCA nodes. Below, we give the notation and methods for finding LCA and SLCA nodes from the literature.

An XML document is modeled as a rooted, ordered, and labeled tree. Nodes in this rooted tree correspond to elements in the XML document. For each node v of a tree, $\lambda(v)$ denotes the label/tag of node v. $u \prec v$ ($u \succ v$) denotes that u is an ancestor (descendant) of node v. Given a query q, containing k terms listed as $w_1, w_2, ..., w_k$, the posting list of each query term w_i can be denoted as S_i. According to this, each node in S_i contains the keyword w_i in its direct text content. The basic motivation for LCA is that if a node v' is an LCA of $(v_1, v_2, ..., v_k)$, where v_i belongs to S_i, then v' contains all the keywords and should be an answer for the query q.

Definition 2.1: Given k nodes $v_1, v_2, ..., v_k$, w is called LCA of these k nodes, iff, $\forall 1 \leq i \leq k$, w is an ancestor of v_i and $\nexists u, w \prec u$, u is also an ancestor of each v_i.

Definition 2.2: Given a query $M = m_1, m_2, ..., m_k$ and an XML document D, the sets of LCAs of M on D is, $LCASet = LCA(S_1, S_2, ..., S_k) = \{v | v = LCA(v_1, v_2, ..., v_k), v_i \in S_i\}$.

Most of the retrieval systems finding common ancestor nodes employ Dewey IDs to identify the nodes. Dewey IDs provide a straightforward solution for locating the LCA of two nodes. Given two nodes, v_1 and v_2, and their Dewey IDs, p_1 and p_2, the LCA of two nodes is the node v having the Dewey ID p such that p is the longest prefix of p_1 and p_2. Finding the LCA of two nodes is $O(d)$ where d is the maximum length of a Dewey ID, and at the same time the

maximum depth of the XML tree. While LCA is the most intuitive method for finding common ancestors, it suffers from false positive and false negative result problems. Some of the LCA nodes could be irrelevant to the query since the keywords are scattered in different nodes which are not meaningfully related or some of the nodes that are not LCA could be more relevant and complete. The approaches following LCA have focused on the problem of meaningfulness and completeness. SLCA is one of these methods which we study in detail below.

An SLCA node contains all the keywords of a query and is not an ancestor of any other node which also contains all the keywords. As a straightforward approach, SLCA nodes could be found by finding all of the possible LCAs and then eliminating the nodes which are ancestors of the other LCA nodes. Finding all of the lowest common ancestors of a given query requires to compute LCA of each possible node combination $v_1, v_2, ..., v_k$ where $v_i \in S_i$. However, this method is very expensive as $(|S_1| * |S_2| * ... * |S_n|)$ number of LCA computations should be done. Instead of this straightforward approach, in [36] Xu et al. avoid redundant LCA computations by making use of the Scan Eager and Indexed Lookup Eager algorithms that they propose. With Indexed Lookup Eager algorithm, the number of LCA computations is reduced by using the notion of left and right match of a node v with respect to a set S. Below we first give the formal definitions of left and right match and show how to compute the SLCAs with the help of these definitions in Algorithm 1 (adapted from Indexed Lookup Eager algorithm given in [36]).

Definition 2.3: A node v belongs to the $SLCASet(S_1, S_2, ..., S_k)$ if $v \in LCASet(S_1, S_2, ..., S_k)$ and $\forall u \in LCASet(S_1, S_2, ..., S_k)$ $v \not\prec u$.

Definition 2.4: Right match of v in a set S ($rm(v, S)$) is the node of S that has the smallest preorder id that is greater than or equal to $pre(v)$.

Definition 2.5: Left match of v in a set S ($lm(v, S)$) is the node of S that has the biggest postorder id that is less than or equal to $post(v)$.

Definition 2.6: $slca(\{v\}, S) = descendant(lca(v, lm(v, S)), lca(v, rm(v, S)))$ where descendant function returns the descendant node of its arguments.

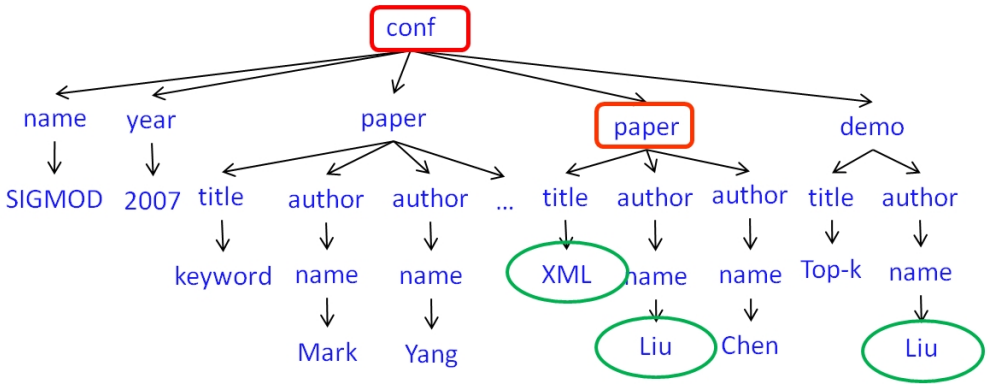


Figure 2.5: LCA nodes of the query 'XML, Liu'

Algorithm 1 Indexed Lookup Algorithm

1: k = number of keywords in the query
2: $B = S_1$
3: **for** $i = 1$ to n **do**
4: B = getSLCA(B,S_i)
5: **end for**
6: output B

function getSLCA(S_1, S_2)
Result={}
$u = 0$
for each node $v \epsilon S_1$ **do**
$x = descendant(lca(v, lm(v, S_2)), lca(v, rm(v, S_2)))$
if $pre(u) <= pre(x)$ **then**
if $u \not\preceq x$ **then**
Result = Result $\cup \{x\}$
end if
$u = x$
end if
end for
return Result $\cup\, u$

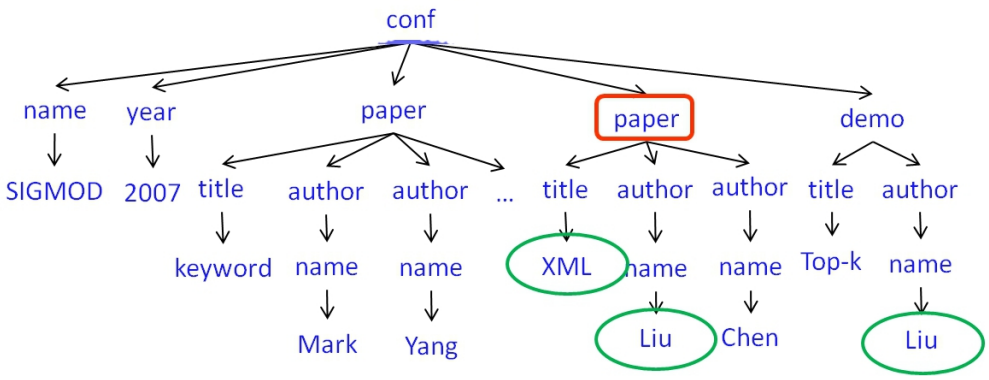


Figure 2.6: SLCA nodes of the query 'XML, Liu'

In Figures 2.5 and 2.6, LCA and SLCA nodes of a query are shown respectively for a sample XML tree. In Figure 2.6, 'conf' node is not an SLCA as it already contains an LCA, 'paper' node in its subtree.

2.3 Compression of Indexes

The inverted indexes could be very large since the collections that the current search engines use contain billions of documents. For such large collections, index compression techniques become essential to provide an efficient retrieval. With the help of compression, disk and memory requirements of the index are reduced. Furthermore, with a smaller usage of disk space, transfer of data from disk to memory becomes much faster. Index compression techniques are divided into two as lossless and lossy compression. While lossless approaches do not lose any information, lossy compression techniques discard certain information. However these two approaches are complementary as an index could be compressed by applying lossy and lossless compression techniques sequentially. In this thesis, we experiment both techniques on XML full element-indexes.

2.3.1 Lossless Compression

Lossless compression is a compression technique which allows the exact data to be recreated from the compressed data. In large scale search engines, due to the need of more efficient data structures, lossless compression techniques are applied inevitably. Most of the techniques for inverted indexes employ integer compression algorithms on document id gaps (d-gap). A d-gap is the difference between two consecutive document ids in a posting list of a term. As the posting lists are stored in the increasing order of document ids of the postings, a list of d-gaps following an initial document id could be encoded instead of document ids themselves. By this way, the values to be compressed become smaller and require less space with variable encoding methods. Variable encoding methods are lossless compression tehcniques which could be applied in either bit or byte level. Variable byte encoding uses an integral number of bytes to encode a gap and it is quite simple to implement. In [31], Tatarinov et al. propose to use UTF-8 for encoding Dewey IDs, which is a variable length character encoding method and is widely used to represent text. However, if disk space is a scarce resource, even better compression ratios could be achieved by using bit-level encodings, particularly Elias-γ and Elias-δ codes.

In this thesis, we investigate the effect of these compression techniques on XML collections. We try UTF-8, Elias-γ and Elias-δ encodings on the full and direct indexes. The bit-aligned code, Elias-γ requires $2 * \lg k + 1$ bits to encode a number k. Elias-δ, on the other hand requires about $2 \lg \lg k + \lg k$ bits to encode a number k. UTF-8 requires different number of bytes for different ranges of numbers as in Table 2.1.

Decimal range	**Encoded # of bytes**
0-127	1
128-2047	2
2048-65535	3
65536-2097152	4

Table 2.1: Storage Requirement of UTF-8 Encoding

2.3.2 Lossy Compression

Lossy compression techniques discard certain part of an index and attain a smaller index size while aiming to lose the least information as possible. Latent semantic indexing and stopword omission are lossy approaches where the complete posting list of a term is discarded from the index. Static pruning, on the other hand, removes certain postings from a posting list and promises a more effective compression strategy. In this thesis, we employ static pruning strategies, which aim to reduce the file size and query processing time, while keeping the effectiveness of the system unaffected, or only slightly affected. In the last decade, a number of different approaches have been proposed for the static index pruning. In this thesis, as in [5], we use the expressions term-centric and document-centric to indicate whether the pruning process iterates over the terms (or, equivalently, the posting lists) or the documents at the first place, respectively.

In one of the earliest works in this field, Carmel et al. proposed term-centric approaches with uniform and adaptive versions [30]. Roughly, the adaptive top-k algorithm sorts the posting list of each term according to some scoring function (Smart's TF-IDF in [30]) and removes those postings that have scores under a threshold determined for that particular term. The algorithm is reported to provide substantial pruning of the index and exhibit excellent performance at keeping the top-ranked results intact in comparison to the original index. In our study, this algorithm (which is referred to as TCP strategy hereafter) is employed for pruning full element-index files, and it is further discussed in Section 4.2.

As an alternative to term-centric pruning, Büttcher et al. proposed a document-centric pruning (referred to as DCP hereafter) approach with uniform and adaptive versions [5]. In the DCP approach, only the most important terms are left in a document, and the rest are discarded. The importance of a term for a document is determined by its contribution to the document's Kullback-Leibler divergence (KLD) from the entire collection. However, the experimental setup in this latter work is significantly different than that of [5]. In a more recent study [3], a comparison of TCP and DCP for pruning the entire index is provided in a uniform framework. It is reported that, for disjunctive query processing, TCP

essentially outperforms DCP for various parameter selections. In this thesis, we use the DCP strategy as well to prune the full element-index, and further discuss DCP in Section 4.2.

There are several other proposals for static index pruning in the literature. A locality based approach is proposed in [9] for the purpose of supporting conjunctive and phrase queries. In a number of other works, search engine query logs are exploited to guide the static index pruning [2, 12, 27, 29].

Chapter 3

XML Keyword Search with Full Element Index

3.1 Introduction

Keyword search is a popular way to search data in several domains such as web documents, digital libraries and databases. Firstly, as it does not require any knowledge of query languages or complex data schemas, it increases the usability. Furthermore, it enables data interconnection by collecting the data pieces that are relevant to the query. The methods for keyword search in structured documents aim to find focused results by exploiting the structural relationships between the nodes. However, how to infer these structural relationships and how to determine the result nodes is still a big challenge in this area. Upon studying the literature, it could be easily observed that most of the proposed methods promise good effectiveness values with quite impractical frameworks.

In this chapter of the thesis, we propose to build a common framework for retrieving XML documents together with unstructured documents. For this, we employ indexing and query processing methods similar to traditional IR methods. We use regular document-oriented keyword search methods on a full element-index built for the corresponding database or collection. In the literature, this

approach is claimed to have some problems such as space overhead, spurious query results and inaccurate ranking. As mentioned in Section 2.2.2, in a full element-index, each element e in an XML document is considered as a separate indexable unit and includes all terms at the subtree rooted at e. Since the full element index is assumed to yield efficiency problems, its promises are rarely explored in the literature. In this chapter of the thesis, we list the major criticisms (e.g., see [16]) against using a full element-index and discuss how we handle each case. We support our arguments by the experiments conducted on our framework.

3.2 Document Ordered Query Processing Using Full Element Index

In this section, we first give the details of our approach that employs a full element-index for keyword search on XML databases and collections. The techniques used to accomplish different tasks of our retrieval system are as given below:

Query Processing: For the basic query processing task, we make use of document ordered processing so that query results can be obtained before the processing of all the posting lists is finished. The query processing is conjunctive in accordance with the result definition method that is used.

Node Labeling: As mentioned in Section 2.2.1 there are many node labeling techniques proposed for XML trees. In this work, we label the nodes according to their preorder and postorder traversal ids. This labeling scheme is simple to implement and useful for deducing ancestor-descendant relationships and thereby finding focused results such as SLCAs.

Indexing: In XML keyword search, each element of an XML document is indexed separately. There are many indexing techniques proposed for XML documents as mentioned in Section 2.2.2. In this work we make use of a full element index which is the most suitable index for traditional query processing techniques. With this

index, there is no need for a propagation mechanism or a special query processing algorithm as in XRank [16] or SLCA [36].

Result Definition: Resulting elements obtained by document ordered processing could be overlapping with each other which is an undesired case according to user studies. To prevent overlapping and to find the elements at the best granularity, the result list should be eliminated further. Among various result definition methods, we focus on SLCA method, as it is one of the most basic and intuitive methods in XML keyword search. This method finds the smallest lowest common ancestor nodes which contain all the keywords and is not an ancestor of any other node which also contains all the keywords. After a temporary result list, R, is obtained by applying document ordered processing to a query, the nodes which are not SLCAs are eliminated from R. For this, we propose a method which is in the order of the length of the result list, $O(|R|)$, and depends on the following lemma.

Lemma: Given a temporary result list, R, sorted according to postorder traversal ids of the result nodes, a node n is an SLCA if the previous node n' in the result list is not an SLCA and is not a descendant of n.

Proof: Consider a collection which consists of a single XML document. Assume that nodes n' and n are two adjacent nodes in R which is sorted according to postorder traversal ids. According to this, it could easily be deduced that $post(n') < post(n)$. This implies that either n' is a descendant of n or n' does not have an ancestor-descendant relationship with n. If n' is a descendant of n and n' is an SLCA, then n can not be an SLCA according to the definition. However, if n' is not a descendant of n, then n is an SLCA since there can not be any other node n'' where n'' is a descendant of n and $n'' < n' < n$.

In our algorithm, we make use of this lemma and check whether a result node is an SLCA in $O(1)$ time. The pseudocode for the algorithm is given in Algorithm 2.

Given the techniques and the algorithms used in this work, we list the criticisms against using a full element index (e.g., [16]) and state our solutions below:

Algorithm 2 Finding SLCAs with Document Ordered Processing q_i : i^{th} query term I_i : The inverted list associated with t_i

```
Result={}
for each query term q_i do
  h[i] = 1 {Current head of I_i points to the first posting I_i^1}
end for
min = index of the query term having minimum posting list size
I_finished = false
while (¬I_finished && h[min] > size(I_min)) do
  p = (I_min^h[min])
  current_id = p.docid
  for i = 0 to query_size do
    if i == min then
      continue
    end if
    repeat
      p = I_i^h[i]
      h[i] = h[i] + 1
    until p.did ≥ current_id || h[i] > size(I_i)
    if h[i] > size(I_i) then
      I_finished = true
      break {End of the posting list I_i, processing finished}
    else if p.docid == current_id then
      num_updated = num_updated + 1
      break {t_i is in p, continue processing with t_{i+1}}
    else if p.docid > current_id then
      h[min_term] = h[min_term] + 1
      break {t_i is not in p, continue processing with the next posting}
    end if
  end for
  if num_updated == query_size then
    if isSLCA(current_id, pre_id) then
      Result = Result ∪ {current_id}
      pre_id = current_id
    end if
  end if
end while
```

```
function isSLCA(id_1, id_2)
  p_1 = preorder id of node with id_1
  p_2 = preorder id of node with id_2
  if p_1 < p_2 then
    return true
  else
    return false
  end if
```

Criticism#1: The full index causes significant amount of redundancy since a term that is indexed at a particular node has to be indexed for all ancestors of that node, as well.

Discussion#1: It is obvious that the raw (uncompressed) full element-index is inefficient in terms of storage space in comparison to the most widely used indexing approach in the literature, namely, Dewey-encoded index. However, as we discuss in the experiments section, both index files are comparable in size when compressed using state-of-the-art index compression methods. Indeed, our empirical findings can also be supported with the formal discussion given in Section 3.3.

Criticism#2: The graph-based relationships (e.g., ancestor-descendant relationship) cannot be captured in the full index (without significantly increasing its size). Such relationships are crucial to determine LCA, SLCA, VLCA, etc. that are typically used to define the result of search in data-centric usages of XML.

Discussion#2: In this work, we defend that, such a relationship of element ids can be kept separately instead of being coupled with the index. More specifically, let's assume that each element is assigned a post-order traversal id in the index. Furthermore, in an in-memory mapping, we store the preorder traversal id of each element. Then, given two elements e_1 and e_2 (such that $post(e_1) < post(e_2)$), testing their ancestor-descendant relationship simply means that testing whether $pre(e_1) < pre(e_2)$. Such a mapping can be kept in the main memory, and accessed during query processing. Note that, such an auxiliary structure (which can be used by all query processing threads in case of the existence of several parallel

QP threads) would be reasonable in size in comparison to the other components of the search system.

Criticism#3: The query processing on the full element-index would yield spurious results since for an answer node that includes all query terms, all ancestors of that node would also be listed in the result.

Discussion#3: The full element-index would clearly include all ancestors of an SLCA node. However, assuming that the index is sorted in element id (postorder traversal id as in our framework) order, it is guaranteed that if the previous node in the result list is not a descendant of the current node, then the current node would be an SLCA. According to this, whether a node is an SLCA could be found out in $O(1)$ time.

3.3 Full Element Index versus Direct Dewey Index

Assume we have a complete *k-ary* tree of depth d.

Case 1 - Direct index with Dewey ID (I_D)

Direct Dewey index is a widely used indexing method especially in focused retrieval on databases. With this method, each node of an XML tree is labeled with Dewey ID labeling scheme and includes only the direct text content.

Dewey ID of a node at level m consists of m integers, where $1 \leq m \leq d$. A node at level m with Dewey ID $a = a_1.a_2.a_3.\ldots.a_m$ has the following constraints:

In the worst case, each node could be a leaf node and $m = d$. Since, each a_i is smaller than k, by using Elias-γ compression, such a Dewey ID can be represented by at most $d(2\lg k + 1)$ bits. If the posting list of term t in the direct index I_D consists of e number of elements, then the size of the posting list of t would be $e \times d(2\lg k + 1)$ bits. Hence, the size of a direct index with Dewey ID is $O(ed \lg k)$.

Case 2 - Full index with postorder traversal id (I_F)

With tree traversal labeling scheme, the nodes of an XML tree T are labeled with respect to the postorder traversal of T. If T is a complete *k-ary* tree, these labels are smaller than the number of nodes in T which is

$$K = 1 + k + k^2 + ... + k^{d-1} = (k^d - 1)/(k - 1)$$

Assume that there are e elements in a posting list of direct index I_D as in Case 1, and e' elements in a posting list of full index I_F. We also assume that all of the elements in a posting list of I_D are leaf elements at depth d. To compare the sizes of direct and full indexes, we try to estimate e' by analyzing two extreme cases:

1. If none of the leaf elements has a common ancestor except the root node, then they would have $e(d-1)+1$ distinct ancestors. In this case, the corresponding posting list in I_F would have $e(d-1)+1+e = ed+1$ elements.

2. All of the ancestors of these leaves could be common. In this case, leaf elements would have $d-1$ ancestors and the corresponding posting list in I_F would have $e+d-1$ elements.

However, both of these cases are quite rare. Therefore, we try to estimate a decay factor, α, which symbolizes the proportion of decrease in number of nodes in consecutive levels. Assume that there are e_ℓ number of nodes at depth ℓ which contain term t directly and these elements have $e_{\ell-1} = \alpha e_\ell$ number of ancestors at depth $\ell - 1$. Note that $\alpha \leq 1$ and hence, $e_{\ell-1} \leq e_\ell$.

For example, if $\alpha \leq 1/2$ since the number of nodes in I_F is less than $e + e/2 + e/4 + ... + e/2^d \leq 2e$, e' is in the order of e. Note that the element id gaps could be indexed instead of element ids themselves for a smaller index size. Since the ids are between 1 and K and there are e' elements, the average element id gap

would be K/e'. Since $e' \leq 2e$, the size of the Elias-γ compressed posting list consisting of e' elements is

$$\begin{aligned} e' \lg \frac{K}{e'} &= e' \lg \frac{(k^d - 1)/(k-1)}{e'} \\ &< 2e \lg \frac{(k^d - 1)}{2e} \\ &< 2ed \lg k \\ &= O(ed \lg k). \end{aligned}$$

Hence, when $\alpha \leq 1/2$, the space complexity of full and direct indexes are both $O(ed \lg k)$.

3.4 Experiments

3.4.1 Experimental Setup

Collection: In this work, we use real datasets which are obtained from DBLP [32] and Wikipedia [10] collections. DBLP dataset contains a single XML document of size 207 MB retrieved from the DBLP Computer Science Bibliography website. English Wikipedia XML collection, on the other hand, consists of multiple XML files (659,388 articles of total size 4,5 GB) and has been employed in INEX campaigns between 2006 and 2008.

Queries: The queries used for DBLP dataset are randomly generated from the word lists of the datasets. We have 8 of these synthetic query sets each consisting of 1000 queries with different number of keywords which have different frequencies. For Wikipedia dataset, we use the query set provided in INEX 2008 which contains 70 queries with relevance assessments (see [21] p. 8 for the exact list of the queries).

Evaluation: In these experiments, we compare the performance of our DocOrdered algorithm with Scan Eager and Indexed Lookup Eager algorithms proposed in [36]. We make a comparison based on efficiency and effectiveness of these

methods. For evaluating the time performance of the algorithms, we measure the time for each query to be processed in milliseconds. For comparing effectiveness values, we give interpolated precision at 1% recall and mean average interpolated precision. We use BM25 function to rank the elements in the result list. While we employ full element-index for our algorithm, direct Dewey index is used for Scan Eager and Indexed Lookup Eager algorithms. To provide a fair evaluation, we also compare the sizes of full element index and direct Dewey index both theoretically and experimentally.

3.4.2 SLCA Retrieval Efficiency

In the experiments below, we provide the comparison of DocOrdered, Scan Eager and Indexed Lookup Eager algorithms in terms of query processing time for finding SLCAs. The complexity of Scan Eager and Indexed Lookup Eager algorithms are given in Table 3.1 (adapted from [36]) while that of our algorithm is given in Table 3.2. The main memory complexity of the algorithms depends on several variables such as the number of query terms, the length of the longest and shortest posting lists in the indexes and maximum depth of XML tree. As stated in Section 2.2.2, the full element index (I_F) is known to have longer posting lists than that of direct Dewey index (I_D). Therefore, more postings should be processed to find all SLCAs. An advantage of full index, however, is that the ancestor-descendant relationships between the nodes could be found out in $O(1)$ time while the cost of comparing two Dewey IDs is $O(d)$ in direct Dewey index. The DocOrdered algorithm finds the set of nodes that contain all keywords in $O(kI_F^{max})$ number of operations. This temporary result set, say R, should be eliminated to find the nodes that are SLCAs. The length of R could be at most equal to the length of the longest posting list in I_F, denoted as I_F^{max}. Since it costs $O(1)$ to check whether a node is an SLCA, the number of total SLCA operations could be at most $O(I_F^{max})$. In total, memory time complexity of our algorithm is $O(kI_F^{max})$. The disk I/O time complexity, on the other hand, is equal to $O(T_F)$ where T_F is the number of blocks that posting lists reside on disk.

Scan Eager algorithm processes the shortest posting list and finds a left and

Algorithm	**Disk I/O**	**#LCA operations**	**#Dewey comparisons**	**Memory Complexity**
Scan Eager	$O(T_D)$	$O(kI_D^{min})$	$O(kI_D^{max})$	$O(kdI_D^{max})$
IL Eager	$O(kI_D^{min})$	$O(kI_D^{min})$	$O(kI_D^{min} \log I_D^{max})$	$O(kdI_D^{min} \log I_D^{max})$

Table 3.1: Complexity Analysis for Indexed Lookup Eager and Scan Eager Algorithms, where I_D is the Dewey index, I_D^{min} (I_D^{max}) is the length of the shortest (longest) posting list in I_D, k is the number of query terms, T_D is the total number of blocks that I_D occupies on disk and d is the maximum depth of the tree.

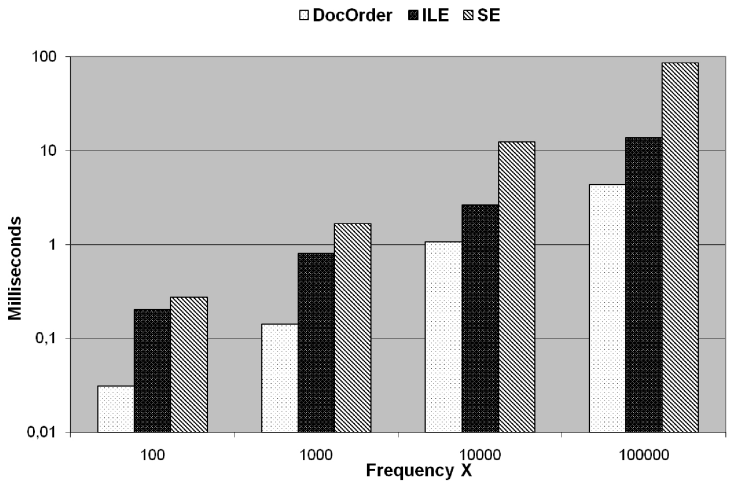
right match in each one of the other posting lists. Therefore the number of left and right match operations is $O(kI_D^{max})$. Then, for each posting in the shortest posting list, the LCAs with left and right matches are found, which is in the order of $O(kI_D^{min})$. Each of the LCA and left and right match operations costs $O(d)$ where d is the length of a Dewey ID which is at most equal to the depth of the XML tree. In total, memory complexity of Scan Eager algorithm is $O(kdI_D^{max})$. Indexed Lookup Eager algorithm differs from Scan Eager in one aspect that while Indexed Lookup Eager uses binary search to find left and right matches of a node, Scan Eager scans the posting lists. As the complexities of the three algorithms depend on the index sizes, we also make an analysis of the sizes in Section 3.4.4 to provide a better insight for the comparison of three methods and to give an idea of disk access times.

Algorithm	**Disk I/O**	**#Postorder Id Comparison**	**#SLCA Comparison**	**Memory Complexity**
DocOrdered	$O(T_F)$	$O(kI_F^{max})$	$O(I_F^{max})$	$O(kI_F^{max})$

Table 3.2: Complexity Analysis for DocOrdered Processing Algorithm, where I_F is the full index, I_F^{min} is the length of the shortest posting list in I_F, k is the number of query terms, and T_F is the total number of blocks that I_F occupies on disk.

In Figure 3.1, each query contains two keywords with smaller frequency 100, and the bigger frequency variable X. In Figure 3.2, each query contains two keywords with smaller frequency 1000, and the bigger frequency variable X. In

Figure 3.1: Processing Time of 2-Keyword Query with Frequency 100-X



these experiments, we observe the effect of the length of the posting list with bigger frequency on query processing time. While the Scan Eager algorithm performs better than the Indexed Lookup Eager algorithm as in [36], our algorithm computes SLCA results significantly (i.e., around five times) faster than both algorithms.

In Figure 3.3, each query contains two keywords with smaller frequency variable X, and the bigger frequency 100000. In these experiments, we evaluate the effect of the size of the smaller posting list on the performance by varying the smaller frequency and keeping the bigger frequency constant. DocOrdered and Scan Eager algorithms' performance does not vary much since their memory complexity depends on the length of the longest posting list. Similarly, DocOrdered algorithm has a much better performance than that of the Scan Eager and Indexed Lookup Eager algorithms.

In Figure 3.4, we give the processing time of the queries with different number of keywords. For each query with k number of keywords, the keyword with the smallest posting list has a frequency of 100, while the remaining (k-1) keywords posting lists' frequency is 100000.

Figure 3.2: Processing Time of 2-Keyword Query with Frequency 1000-X

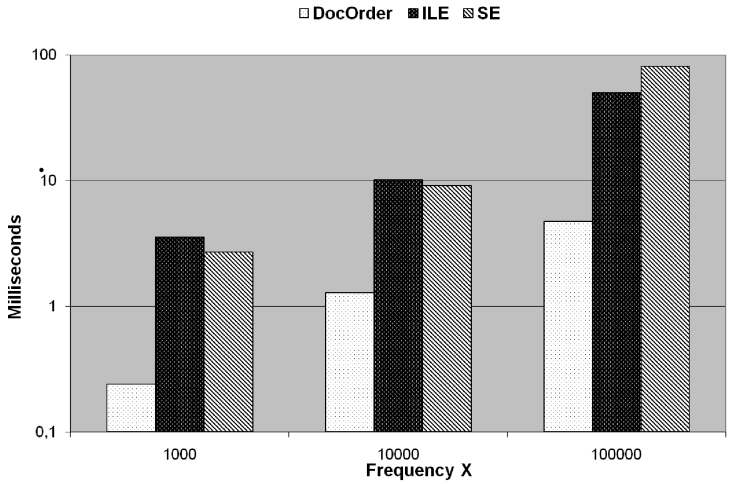


Figure 3.3: Processing Time of 2-Keyword Query with Frequency 100000-X

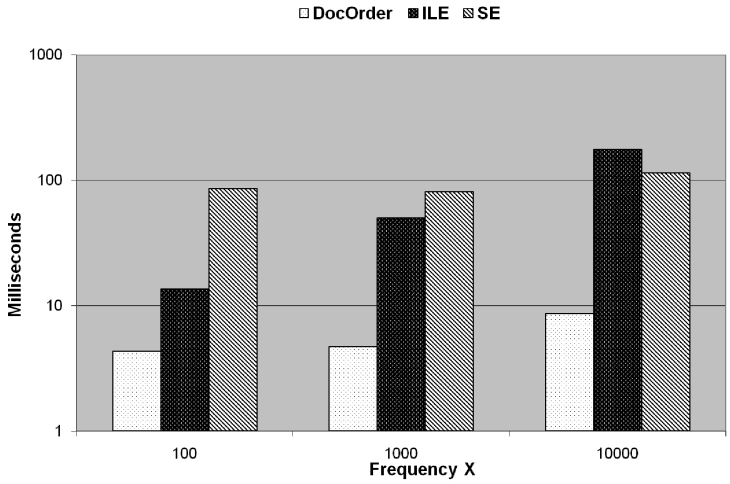
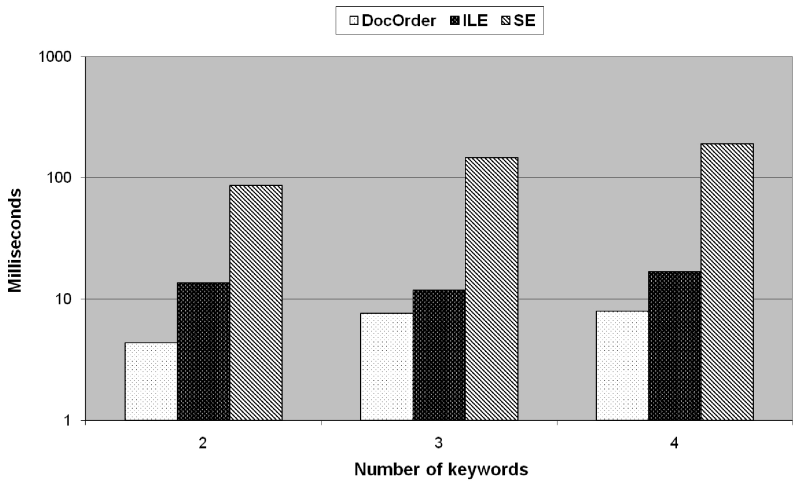


Figure 3.4: Processing Time of Queries with Varying Number of Keywords (Frequency 100-100000)



3.4.3 SLCA Retrieval Effectiveness

In this section, we evaluate the effectiveness of the SLCA method. The basic idea behind this method is that, if a node contains all the keywords in a query, then it will be more relevant than its ancestors. However, since the SLCA method returns an unranked list of results as database query languages, a ranking mechanism is required to improve the effectiveness. We implement a ranking mechanism by adapting BM25 ranking function to XML retrieval. The term statistics for the traditional BM25 function are within-document term frequency, tf, inverse document frequency, idf, document length, and average document length. In XML retrieval, these traditional measures could be calculated at element level. However, because of the nested structure of XML, the interpretation of these statistics could vary depending on the indexing mechanism. We adapt the term statistics according to the full index and the direct Dewey index. In full index, each element is indexed with its full content and term statistics are calculated accordingly. In direct Dewey index, on the other hand, since each element is indexed with its direct content, term statistics are calculated at query execution time. In Index Eager algorithm, depending on the way that SLCAs are found, tf of an SLCA node, v, is the sum of the tfs of children nodes whose $slca$ is v.

Document length of v is also calculated similarly.

Another common method in XML keyword search is to calculate the score of each element containing all the keywords and eliminate the overlapping results. Overlap elimination is achieved by choosing the highest scoring element on a path in the XML document. In Table 3.3, we compare the effectiveness of this traditional IR method, named as Ranked Top-1000, to that of Ranked and Unranked SLCA methods. As the effectiveness evaluation measure, we use mean average interpolated precison(MAiP) and interpolated precision at 1% recall level(iP[0.01]). While Unranked SLCA results give the worst iP[0.01] and MAiP, the best effectiveness values are achieved by Ranked SLCA DocOrdered method. The difference in ranking of DocOrdered and Index Eager methods results from the values of term statistics. The way that term statistics are calculated in Index Eager method possibly causes inaccurate ranking of the results. Ranked Top-1000 method is also less effective then Ranked SLCA DocOrdered Method. This may result from the fact that Ranked SLCA favors smaller nodes and most of the content in the datasets occur in leaf nodes. We also provide the efficiency results in Table 3.3 to give an idea about the query execution time of each algorithm. The results reveal that ranking operation increases the query execution times slightly.

	Unranked SLCA DocOrdered	**Unranked SLCA Index Eager**	**Ranked SLCA DocOrdered**	**Ranked SLCA Index Eager**	**Ranked Top-1000**
iP[0.01]	0.103	0.103	0.326	0.202	0.256
MAiP	0.016	0.016	0.086	0.028	0.086
Time(mSec)	5.941	39.824	6.118	40.662	6.559

Table 3.3: Effectiveness and Efficiency Comparison of SLCA and Top-1000 Methods

3.4.4 Size Comparison of Full Element Index and Direct Dewey Index

As mentioned in Section 2.2.2, several techniques are proposed for indexing XML collections. In this thesis, we employ the element indexing which is the closest technique to traditional IR. This technique indexes each XML element considering the content of the element itself and its descendants. However, with this approach, each term occurring at nth level of the XML tree is repeated n times in the index which yields a considerable amount of redundancy in terms of the index size. Another technique for indexing an XML collection is leaf-only indexing. With this technique, a direct index is obtained where each element contains only the direct text content. With direct index, Dewey encoding is used so that the ancestor-descendant relationships could be deduced from the ids of the elements.

Below, we compare the sizes of full element index and direct Dewey index built from DBLP and Wikipedia collections. While the DBLP collection consists of a single XML document, Wikipedia collection consists of multiple XML documents. Both the full element index and direct Dewey index consist of the posting lists of the terms. Each posting list is made up of the postings which include the element identifier and the frequency of the term. As we use tree traversal based ordering in full index and Dewey ID based ordering in direct index, the element identifiers have different structure in these indexes. In full index, the element identifier is the postorder traversal id of the element. In direct Dewey index, the element identifier consists of the Dewey ID of the element, and the depth of the Dewey ID. In Tables 3.4, 3.5 and 3.6, sizes of the indexes are compared with different compression methods, namely Elias-γ, Elias-δ and UTF-8 encoding, respectively. Here we observe that Elias-γ is the best method to compress a Dewey index. Although UTF-8 encoding is proposed for Dewey index in [31], it results in the biggest index size. Postorder traversal ids are single integers ,therefore, it is possible to use id gaps instead of ids themselves. However, since Dewey IDs consist of d number of integers where d is equal to the depth of the element in the XML tree it is not possible to use gap encoding with Dewey IDs.

	Element Id	**Depth**	**TF**	**Total**
$I_{DirectDBLP}$	68.744	5.172	1.787	75.703
$I_{FullDBLP}$	32.190	0	2.991	35.181
$I_{DirectINEX}$	441.599	67.246	32.346	541.191
$I_{FullINEX}$	389.834	0	103.407	493.241

Table 3.4: Size Comparison of Full Element Index and Direct Dewey Index with Elias-γ compression

	Element Id	**Depth**	**TF**	**Total**
$I_{DirectDBLP}$	53.561	6.877	1.800	62.238
$I_{FullDBLP}$	28.489	0	3.029	31.518
$I_{DirectINEX}$	463.107	83.139	35.705	581.951
$I_{FullINEX}$	351.204	0	114.612	465.816

Table 3.5: Size Comparison of Full Element Index and Direct Dewey Index with Elias-δ compression

	Element Id	**Depth**	**TF**	**Total**
$I_{DirectDBLP}$	68.129	14.097	14.097	96.323
$I_{FullDBLP}$	32.384	0	23.325	55.709
$I_{DirectINEX}$	701.762	174.898	176.528	1053.188
$I_{FullINEX}$	590.558	0	482.799	1073.357

Table 3.6: Size Comparison of Full Element Index and Direct Dewey Index with UTF-8 compression

3.5 Conclusion

In this chapter, we compare the performance of our query processing strategy using a full element-index to that of the strategies that use a Dewey-encoded index [36]. Our findings are as follows:

- When the datasets are compressed using state-of-the-art techniques, full element-index files take less disk space than a direct Dewey index.
- Computation of SLCAs on both datasets are significantly (i.e., around five times) faster than using the SLCA computation algorithms [36] that operate on direct Dewey index.
- Our experiments on Wikipedia dataset show that IR-based ranking of elements may yield better results than producing SLCAs for document-centric datasets. The efficiency promises are still kept.

Chapter 4

XML Retrieval using Pruned Index Files

4.1 Introduction

In this chapter, we essentially focus on the strategies for constructing space-efficient element-index files to support content-only (CO) queries. In the literature, the most straight-forward element-indexing method considers each XML element as a separate document, which is formed of the text directly contained in it and the textual content of all of its descendants. As mentioned in Section 2.2.2 this structure is called a full element-index. Clearly, this approach yields significant redundancy in terms of the index size, as elements in the XML documents are highly nested. To remedy this, a number of approaches are proposed in the literature. One such method is restricting the set of elements indexed, based on the size [28] or type of the elements. Such an approach may still involve redundancy for the elements that are selected to be indexed. Furthermore, determining what elements to index would be collection and scenario dependent. There are also other indexing strategies that can fully eliminate the redundancy. For instance, a direct element-index is constructed by only considering the text that is directly

contained under an element (i.e., disregarding the content of the element's descendents). In the literature, propagation based mechanisms, in which the score of each element is propagated upwards in the XML structure, are coupled with the direct index for effective retrieval (e.g., [15]). In this case, the redundancy in the index is somewhat minimized, but query processing efficiency would be degraded.

For an IR system, it is crucial to optimize the underlying inverted index size for the efficiency purposes. A lossless method for reducing the index size is using compression methods some of which are experimented with full element index in Chapter 3. On the other hand, many lossy static index pruning methods have also been proposed in the last decade. All of these methods aim to reduce the storage space for the index and, subsequently, query execution time, while keeping the quality of the search results unaffected. While it is straight-forward to apply index compression methods to (most of) the indexing methods proposed for XML, it is still unexplored how those pruning techniques serve for XML collections, and how they compare to the XML-specific indexing methods proposed in the literature.

In this chapter of the thesis, we propose to employ static index pruning techniques for XML indexing. We envision that these techniques may serve as a compromise between a full element-index and a direct element-index. In particular, we first model each element as the concatenation of the textual content in its descendants, as typical in a full index. Then, the redundancy in the index is eliminated by pruning this initial index. In this way, an element is allowed to contain some terms, say, the most important ones, belonging to its descendants; and this decision is given based on the full content of the element in an adaptive manner.

For the purposes of index pruning, we apply two major methods from the IR literature, namely, term-centric [30] and document-centric pruning [5] on the full element-index. We evaluate the performance for various retrieval tasks as described in the latest INEX campaigns. More specifically, we show that retrieval using pruned index files is comparable or even superior to that of the full-index up to very high levels of pruning. Furthermore, we compare these pruning-based

approaches to a retrieval strategy coupled with a direct index (as in [15]) and show that pruning-based approaches are also superior to that strategy. As another advantage, the pruning-based approaches are more flexible and can reduce an index to a required level of storage space.

In Section 2.3.2, we have summarized some of the static index pruning strategies that are proposed for large-scale IR systems and search engines. In the remaining part of this chapter, we describe the pruning techniques that are adapted for reducing the size of a 'full' element-index, in which all descendants of an element are considered during indexing. Next, we give the experimental evaluations and point to future work directions on XML index pruning.

4.2 Pruning the Element-Index

The size of the full element-index for an XML collection may be prohibitively large due to the nested structure of the documents. A large index file does not only consume storage space but also degrades the performance of the actual query processing (as longer posting lists should be traversed). The large index size would also undermine the other optimization mechanisms, such as the list caching (as longer list should be stored in the main memory).

In this chapter of the thesis, our main concern is reducing the index size to essentially support content-only queries. Thus, we attempt to make an estimation of how the index sizes for the above approaches can be ordered. Of course, $size(I_{full})$, i.e., size of full element-index, would have the largest size. Selective (and/or distributed) index, denoted as $size(I_{sel})$, would possibly be smaller; but they can still have some degree of redundancy for those elements that are selected for indexing. Assuming that I_{sel} would typically index all leaf level elements, a direct index ($size(I_{direct})$) that indexes only the text under each element would be smaller than the former. Finally, the lower bound for the index size can be obtained by discarding all the structuring in an XML document and creating an index only on the document basis (i.e., I_{doc}). Thus, a rough ordering can be like

$size(I_{full}) < size(I_{sel}) < size(I_{direct}) < size(I_{doc})$. In this work, we employ some pruning methods that can yield indexes of sizes comparable to $size(I_{direct})$ or $size(I_{sel})$. We envision that such methods can prune an index up to a given level in a robust an adaptive way, without requiring a priori knowledge on the collection (e.g., semantics or popularity of elements). Furthermore, the redundancy that remains in the index can even help improving the retrieval performance.

Previous techniques in the literature attempt to reduce the size of a full element-index by either totally discarding the overlapping content, or only indexing a subset of the elements in a collection. In contrast, we envision that some of the terms that appear in an element's descendants may be crucial for the retrieval performance and should be repeated at the upper levels; whereas some other terms can be safely discarded. Thus, instead of a crude mechanism, for each element, the decision for indexing the terms from the element's descendants should be given adaptively, considering the element's textual content and search system's ranking function. To this end, we employ two major static index pruning techniques, namely term-centric pruning (TCP) [30] and document-centric pruning (DCP) [5] for indexing the XML collections:

- $TCP(I, k, \varepsilon)$: As mentioned in the related work, TCP, the adaptive version of the top-k algorithm proposed in [30], is reported to be very successful in static pruning. In this strategy, for each term t in the index I, first the postings in t's posting list are sorted by a scoring function (e.g, TF-IDF). Next, the kth highest score, z_t, is determined and all postings that have scores less than $z_t * \varepsilon$ are removed, where ε is a user defined parameter to govern the pruning level. Following the practice in [4], we disregard any theoretical guarantees and determine ε values according to the desired pruning level. A recent study shows that the performance of the TCP strategy can be further boosted by carefully selecting and tuning the scoring function used in the pruning stage [4]. Following the recommendations of that work, we employ BM25 as the scoring function for TCP.

- $DCP(D, \lambda)$: We apply the DCP strategy for the entire index, which is slightly different from pruning only the most frequent terms as originally

proposed by [5]. For each document d in the collection D, its terms are sorted by the scoring function. Next, the top $|d| * \lambda$ terms are kept in the document and the rest are discarded, where λ specifies the pruning level. Then, the inverted index is created over these pruned documents. KLD has been employed as the scoring function in [5]. However, in a more recent work [3], it is reported that BM25 performs better when it is used during both pruning and retrieval. Thus, we also use BM25 with DCP algorithm.

4.3 Experiments

4.3.1 Experimental Setup

Collection and queries: In this work, we use English Wikipedia XML collection [10] employed in INEX campaigns since 2006. The dataset includes 659,388 articles obtained from Wikipedia. After conversion to XML the collection includes 52 million elements. The textual content is 1.6 GB whereas the entire collection (i.e., with element tags) takes 4.5 GB. Our main focus is content-only (CO) queries whereas content-and-structure queries (CAS) are left as a future work. In the majority of the experiments reported below, we use 70 query topics with relevance assessments provided for the Wikipedia collection in INEX 2008 (see [21] p. 8 for the exact list of the queries). The actual query set is obtained from the title field of these topics after eliminating the negated terms and stopwords. No stemming is applied.

Indexing: As we essentially focus on CO queries, the index files are built upon only using the textual content of the documents in the collection; i.e., tag names and/or paths are not indexed. In the best performing system in all three tasks of INEX 2008 ad hoc retrieval track, only a subset of elements in the collection are used for scoring [19]. Following the same practice, we only index the following elements: <p>, <section>, <normallist>, <article>, <body>, <td>, <numberlist>, <tr>, <table>, <definitionlist>, <th>, <blockquote>, , <li>, <u>. Each of these elements in an XML document is treated as a separate

document and assigned a unique global identifier. Thus, the number of elements to be indexed is found to be 7.4 million out of 52 million elements in Wikipedia collection. During indexing, we use the open-source Zettair open-source search engine [38] to parse the documents in the collection and obtain a list of terms per element. Then, an element-level index is constructed by using each of the strategies described in Section 2.2.2. We compare the retrieval performance of four different XML indexing approaches which are I_{full} (full element-index as described before), I_{direct} (an index created by using only the text directly under each element), I_{TCP} (index files created from I_{full} by using TCP algorithm at a pruning level) and I_{DCP} (index files created from I_{full} by using DCP algorithm at a pruning level). The posting lists in the resulting index files include <element-id, frequency> pairs for each term in the collection, as this is adequate to support the CO queries. Of course, the index can be extended to include additional information, say, term positions, if the system is asked to support phrase or proximity queries, as well. Posting lists are typically stored in a binary file format where each posting takes 8 bytes (i.e., a 4 byte integer is used per each field). In the below discussions, all index sizes are considered in terms of their raw (uncompressed) sizes.

Retrieval tasks and evaluation: We concentrate on three ad-hoc retrieval tasks, namely, Focused, Relevant-in-Context (RiC) and Best-in-Context (BiC), as described in recent INEX campaigns (e.g., see [18, 21]). In short, the Focused task is designed to retrieve the most focused results for a query without returning overlapping elements. The underlying motivation for this task is retrieving the relevant information at the correct granularity. Relevant-in-Context task requires returning a ranked list of documents and a set of relevant (non-overlapping) elements listed for each article. Finally, Best-in-Context task is designed to find the best-entry-point for starting to read the relevant articles. Thus, the retrieval system should return a ranked list of documents along with a (presumably) best entry point for each document. We evaluate the performance of different XML indexing strategies for all these three tasks along with the lines of INEX 2008 framework. That is, we use INEXeval software provided in [18] which computes a number of measures for each task, which is essentially based on the amount of

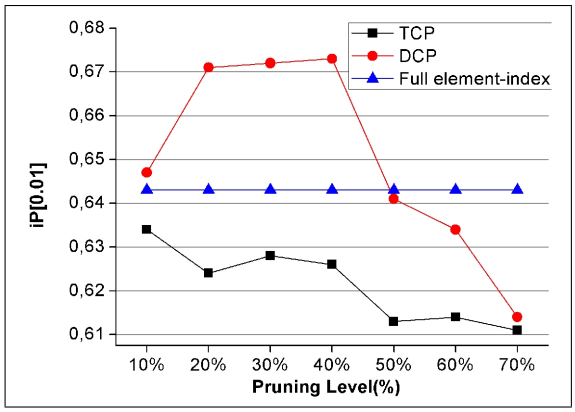


Figure 4.1: Effectiveness comparison of I_{full}, I_{TCP} and I_{DCP} in terms of iP[0.01]

retrieved text that overlaps with the relevant text in assessments. In all experiments, we return up to 1500 highest scoring results.

4.3.2 Performance Comparison of Indexing Strategies: Focused Task

For the focused retrieval task, we return the highest scoring 1500 elements after eliminating the overlaps. The overlap elimination is simply achieved by choosing the highest scoring element on a path in the XML document.

In Figure 4.1, we plot the performance of TCP and DCP based indexing strategies with respect to the full element-index, I_{full}. The evaluation measure is interpolated precision at 1% recall level, iP[0.01], which happens to be the official measure of INEX 2008. For this experiment, we use BM25 function as described in [5] to rank the elements using each of the index files. For the pruned index files, the element length, i.e., number of terms in an element, reduces after pruning. In earlier studies [3, 4], it is reported that using the updated element lengths results better in terms of effectiveness. We observed the same situation also for XML retrieval case, and thus, used the updated element lengths for each pruning level of TCP and DCP.

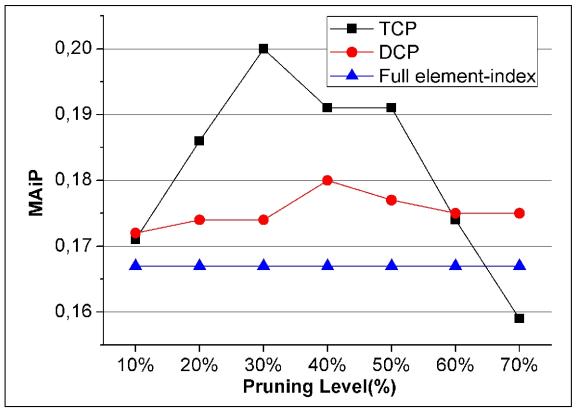


Figure 4.2: Effectiveness comparison of I_{full}, I_{TCP} and I_{DCP} in terms of MAiP[0.01]

To start with, we emphasize that the system performance with I_{full} is reasonable in comparison to INEX 2008 results. That is, focused retrieval based on I_{full} yields an iP figure of 0.643 at 1% recall level. The best result in INEX for this task, as we discussed in the experimental setup section, yielded 0.687 and our result is within the top-10 results of this task (see Table 6 in [21]). Note that, this is also the case for RiC and BiC results that will be discussed in the upcoming sections, proving that we have a reasonable baseline for drawing results in our experimental framework.

Figure 4.1 reveals that DCP based indexing is as effective as I_{full} up to 50% pruning and indeed, at some pruning levels, it can even outperform I_{full}. In other words, it is possible to halve the index and still obtain the same or even better effectiveness than the full index. TCP is also comparable to I_{full} up to 40%. For this setup, DCP seems to be better than TCP, an interesting finding given that just the reverse is observed for typical document retrieval in previous works [2, 3]. However, the situation changes for higher levels of recall (as shown in Table 4.1), and, say, for iP[.10] TCP performs better than DCP up to 70% pruning.

In Table 4.1, we report the interpolated precision at different recall levels and mean average interpolated precision (MAiP) computed for 101 recall levels (from 0.00 to 1.00). For these experiments, we show three pruning levels (30%, 50%

Indexing Strategy	**Prune(%)**	**iP[.00]**	**iP[.01]**	**iP[.05]**	**iP[.10]**	**MAiP**
I_{full}	0%	0.725	0.643	0.507	0.446	0.167
$I_{TCP,0.3}$	30%	0.700	0.628	**0.560**	0.511	**0.200**
$I_{DCP,0.3}$	30%	**0.750**	**0.672**	0.529	0.469	0.174
$I_{TCP,0.5}$	50%	0.666	0.613	0.549	**0.518**	0.191
$I_{DCP,0.5}$	50%	0.708	0.641	0.518	0.473	0.177
$I_{TCP,0.7}$	70%	0.680	0.611	0.511	0.446	0.159
$I_{DCP,0.7}$	70%	0.681	0.614	0.534	0.477	0.175
I_{direct}	65%	0.731	0.611	0.448	0.362	0.126
$I_{direct+PROPSCORE}$	65%	0.519	0.473	0.341	0.302	0.110
$I_{direct+PROPBM25}$	65%	0.450	0.435	0.384	0.302	0.116

Table 4.1: Effectiveness comparison of indexing strategies for Focused task. Prune (%) field denotes the percentage of pruning with respect to full element-index (I_{full}). Shaded measures are official evaluation measure of INEX 2008. Best results for each measure are shown in bold.

and 70%) for both TCP and DCP. The results reveal that, up to 70% pruning, both indexing approaches lead higher MAiP figures than I_{full} (also see Figure 4.2). The same trend also applies for iP at higher recall levels, namely, 5% and 10%.

In Table 4.1, we further compare the pruning based approaches to other retrieval strategies using I_{direct}. Recall that, as discussed in Section 4.2, I_{direct} is constructed by considering only the textual content immediately under each element, disregarding the element's descendants. For this collection, the size of the index turns out to be almost 35% of the I_{full}, i.e., corresponding to 65% pruning level. In our first experiment, we evaluate focused retrieval using I_{direct} and BM25 as in the above. In this case, I_{direct} also performs well and yields 0.611 for iP[.01] measure, almost the same effectiveness for slightly smaller indexes created by TCP and DCP (see the case for 70% pruning for TCP and DCP in Table 4.1). However, in terms of iP at higher levels and MAiP, I_{direct} is clearly inferior to the pruning based approaches, as shown in Table 4.1.

As another experiment, we implemented the propagation mechanism used in the GPX system that participated in INEX between 2004 and 2006 [13, 14, 15].

In these campaigns, GPX is shown to yield very competitive results and ranked among the top systems for various retrieval tasks. Furthermore, GPX is designed to work with an index as I_{direct}, i.e., without indexing the content of descendants for the elements. In this system, first, the score of every element is computed as in the typical case. However, before obtaining the final query output, the scores of all elements in an XML document are propagated upwards so that they can also contribute to their ancestors' scores. We implement the propagation mechanism (denoted as PROP) of GPX and calculate the relevance score of leaf and inner nodes by using the equations 4.1 and 4.2 proposed in [15].

$$L = K^{n-1} \sum_{i=1}^{n} \frac{t_i}{f_i} \tag{4.1}$$

$$R = L_0 + D(n) \sum_{i=1}^{n} L_i \tag{4.2}$$

L_0 = The score of the current node
n = the number of children elements
$D(n) = N1$ if $n = 1$
$= N2$ otherwise
L_i = The relevance score of the i_{th} child element

In accordance with the INEX official run setup described in GPX work, we set $K = 5$, $N1 = 0.11$ and $N2 = 0.31$ in our implementation. Their work reports that the scoring function defined in Equation 4.1 (denoted as SCORE here) also performs quite well when coupled with the propagation. Thus, we obtained results for our implementation using both scoring functions, namely, BM25 and SCORE. In Table 4.1, corresponding experiments are denoted as $I_{direct+PROPBM25}$ and $I_{direct+PROPSCORE}$, respectively. The results reveal that, SCORE function performs better at early recall levels, but for both cases the effectiveness figures are considerably lower than the corresponding cases (i.e., 70% pruning level) based on TCP and DCP. We attribute the lower performance of PROP mechanism to the following observation. For the Wikipedia dataset, 78% of the data assessed as relevant resides in the leaf nodes. This means that, returning leafs in the result

set would improve effectiveness, and vice versa. In contrast, PROP propagates element scores to the upper levels in the document, which may increase the number of interior nodes in the final result and thus reduce the effectiveness.

Note that, we also attempted to verify the reliability of our implementation of propagation mechanism by using INEX 2006 topics and evaluation software, and to see how our results compare to the GPX results reported in [15]. We observed that the results slightly differ at early ranks but then match for higher rank percentages.

We can summarize our findings as follows: In terms of the official INEX measure, which considers the performance at the first results most, the index files constructed by the static pruning techniques lead to comparable to or even superior results than I_{full} up to 70% pruning level. A direct element-index also takes almost 35% of the full index. Its performance is as good as the pruned index files for iP[.01], but it falls rapidly at higher recall levels. Score propagating retrieval systems, similar to GPX, perform even worse with I_{direct} and do not seem to be a strong competitor. Another advantage of the index files created by the static pruning techniques is that they can be processed by typical IR systems without requiring any modifications in the query processing, indexing and compression modules.

4.3.3 Performance Comparison of Indexing Strategies: Relevant-in-Context Task

For the Relevant-in-Context task, after scoring the elements, we again eliminate the overlaps and determine the top-1500 results as in the Focused task. Then, those elements from the same document are grouped together. The result is a ranked list of documents along with a set of elements. While ranking the documents, we use the score of the highest scoring retrieved element per document as the score of this particular document. We also experimented with another technique in the literature; i.e. using the average score of elements in a document for ranking the documents, which performs worse than the former approach and

is not investigated further.

In Table 4.2, we present the results in terms of the generalized precision (gP) metric at early ranks and mean average generalized precision (MAgP); i.e., the official measure of INEX 2008 for both RiC and BiC tasks. The results show that approaches using I_{direct} are inferior to those using the pruned index files based on either TCP or DCP at 70% pruning level; however with a relatively smaller margin with respect to the previous task. In comparison of TCP and DCP based approaches to I_{full}, we observe that the former cases still yield comparable or better performance, however up to 50% pruning, again a more conservative result than that reported for the previous task.

Indexing Strategy	**Prune(%)**	**gP[5]**	**gP[10]**	**gP[25]**	**gP[50]**	**MAgP**
I_{full}	0%	0.364	**0.321**	0.246	0.198	0.190
$I_{TCP,0.3}$	30%	0.380	0.321	0.248	0.199	0.193
$I_{DCP,0.3}$	30%	0.381	0.321	**0.256**	**0.202**	**0.196**
$I_{TCP,0.5}$	50%	**0.385**	0.321	0.247	0.196	0.185
$I_{DCP,0.5}$	50%	0.366	0.321	0.252	0.196	0.185
$I_{TCP,0.7}$	70%	0.355	0.297	0.223	0.170	0.152
$I_{DCP,0.7}$	70%	0.352	0.305	0.232	0.172	0.157
I_{direct}	65%	0.312	0.281	0.210	0.168	0.140
$I_{direct+PROPSCORE}$	65%	0.275	0.242	0.201	0.158	0.142
$I_{direct+PROPBM25}$	65%	0.223	0.199	0.184	0.145	0.108

Table 4.2: Effectiveness comparison of indexing strategies for Relevant-in-Context task. Prune (%) field denotes the percentage of pruning with respect to full element-index (I_{full}). Best results for each measure are shown in bold.

4.3.4 Performance Comparison of Indexing Strategies: Best-in-Context Task

For the Best-in-Context task, we obtain the relevant documents exactly in the same way as in RiC. However, while ranking the documents, if the article node of the document is within these retrieved elements, we use its score as the document score. Otherwise, we use the score of the highest scoring retrieved element as the

score of this particular document. Then, we identify a best-entry-point (BEP) per document. In INEX 2008, a simple approach of setting BEP as 1 is found to be very effective and ranked second among all participants [21]. Note that, this suggests starting to read each ranked document from the beginning. For our work, we also experimented with providing the offset of the highest scoring element per document as BEP [21], which yielded inferior results to the former approach. Thus, we only report the results where BEP is set to 1.

In Table 4.3, we compare indexing strategies in terms of the same evaluation metrics used in RiC task. As in RiC case, the performance obtained by using pruned index files with TCP and DCP is comparable to that of using the full element-index up to 50% pruning. At the 70% pruning level, both pruning approaches have losses in effectiveness with respect to I_{full}, but they are still considerably better than using I_{direct} with the (approximately) same index size. For instance, while MAgP for DCP is 0.143, the retrieval strategies using I_{direct} (with BM25), $I_{direct+PROPSCORE}$ and $I_{direct+PROPBM25}$ yield the MAgP figures of 0.087, 0.086 and 0.074, respectively. Again, basic retrieval using I_{direct} outperforms propagation based approaches, especially at the earlier ranks for generalized precision metric.

Indexing Strategy	**Prune(%)**	**gP[5]**	**gP[10]**	**gP[25]**	**gP[50]**	**MAgP**
I_{full}	0%	0.367	0.314	0.237	0.186	0.178
$I_{TCP,0.3}$	30%	0.369	0.318	0.237	**0.187**	0.178
$I_{DCP,0.3}$	30%	**0.388**	**0.332**	**0.246**	0.187	**0.184**
$I_{TCP,0.5}$	50%	0.364	0.319	0.232	0.179	0.165
$I_{DCP,0.5}$	50%	0.363	0.310	0.234	0.178	0.166
$I_{TCP,0.7}$	70%	0.335	0.287	0.198	0.154	0.138
$I_{DCP,0.7}$	70%	0.340	0.280	0.214	0.157	0.143
I_{direct}	65%	0.215	0.183	0.141	0.116	0.087
$I_{direct+PROPSCORE}$	65%	0.127	0.132	0.120	0.100	0.086
$I_{direct+PROPBM25}$	65%	0.156	0.151	0.136	0.115	0.074

Table 4.3: Effectiveness comparison of indexing strategies for Best-in-Context task. Prune (%) field denotes the percentage of pruning with respect to full element-index (I_{full}). Best results for each measure are shown in bold.

4.4 Conclusion

Previous experiences with XML collections suggest that element indexing is important for high performance in ad hoc retrieval tasks. In our work, we propose to use static index pruning techniques for reducing the size of a full element-index, which would otherwise be very large due to the nested structure of XML documents. We also compare the performance of term and document based pruning strategies to those approaches that use a direct element index that avoids indexing nested content more than once. Our experiments are conducted along the lines of previous INEX campaigns, and the results reveal that pruned index files are comparable or even superior to the full element-index up to very high pruning levels for various ad hoc tasks (e.g., up to 70% pruning for and 50% for RiC and BiC tasks) in terms of retrieval performance. Furthermore, the performance of pruned index files is also better than that of the approaches using the direct index file at the same index size.

Chapter 5

Using Pruning Methods for Clustering XML

5.1 Introduction

As the number and size of XML collections increase rapidly, there occurs the need to manage these collections efficiently and effectively. While there is still an ongoing research in this area, INEX XML Mining Track fulfills the need for an evaluation platform to compare the performance of several clustering methods on the same set of data. Within the Clustering task of XML Mining Track of INEX campaign, clustering methods are evaluated according to cluster quality measures on a real-world Wikipedia collection.

To this end, in the last few workshops, many different approaches have been proposed which use structural, content-based and link-based features of XML documents. In INEX 2008, Kutty et al. [22] use both structure and content to cluster XML documents. They reduce the dimensionality of the content features by using only the content in frequent subtrees of an XML document. In another work, Zhang et al. [39] make use of the hyperlink structure between XML documents through an extension of a machine learning method based on the Self Organizing Maps for graphs. De Vries et al. [35] use K-Trees to cluster XML

documents so that they can obtain clusters in good quality with a low complexity method. Lastly, Tran et al. [33] construct a latent semantic kernel to measure the similarity between content of the XML documents. However, before constructing the kernel, they apply a dimension reduction method based on the common structural information of XML documents to make the construction process less expensive. In all of these works mentioned above, not only the quality of the clusters but the efficiency of the clustering process is also taken into account.

In this chapter of the thesis, we propose an approach which reduces the dimension of the underlying document vectors without change or with a slight change in the quality of the output clustering structure. More specifically, we use a partitioning type clustering algorithm, so-called Cover-Coefficient Based Clustering Methodology (C^3M) [7], along with some index pruning techniques for clustering XML documents.

5.2 Baseline Clustering with C^3M

In this thesis, we use the well-known Cover-Coefficient Based Clustering Methodology (C^3M) [7] to cluster the XML documents. C^3M is a single-pass partitioning type clustering algorithm which is shown to have good information retrieval performance with flat documents (e.g., see [6]). The algorithm operates on documents represented by vector space model. Using this model, a document collection can be abstracted by a document-term matrix, D; of size m by n whose individual entries, d_{ij} $(1 < i < m;\ 1 < j < n)$, indicate the number of occurrences of term j (t_j) in document i (d_i). In C^3M, the document-term matrix D is mapped into an m by m cover-coefficient matrix, C, which captures the relationships among the documents of a database. The diagonal entries of C are used to find the number of clusters, denoted as n_c; and to select the cluster seeds. During the construction of clusters, the relationships between a nonseed document (d_i) and a seed document (d_j) are determined by calculating the c_{ij} entry of C; where c_{ij} indicates the extent to which d_i is covered by d_j. A major strength of C^3M is

that, for a given dataset, the algorithm itself can determine the number of clusters; i.e., there is no need for specifying the number of clusters, as in some other algorithms. For the purposes of our work, we cluster the XML documents into a given number of clusters (for several values like 1000, 10000, etc.) using C^3M method and we simply use the content of XML documents for clustering.

5.3 Employing Pruning Strategies for Clustering

From the previous works, it is known that static index pruning techniques can reduce the size of an index (and the underlying collection) while providing comparative effectiveness performance with that of the unpruned case [2, 3, 4, 5, 30]. In Chapter 4, we have shown that such pruning techniques can also be adapted for pruning the element-index for an XML collection [1]. Here, with the aim of both improving the quality of clusters and reducing the dataset dimensions for clustering, we apply static pruning techniques on XML documents. We adapt the term-centric [30] and document-centric pruning [5] techniques mentioned in Section 2.3.2 to obtain more compact representations of the documents. Then, we cluster documents with these reduced representations for various pruning levels, again using C^3M algorithm.

5.4 Experiments

In this chapter of the thesis, we essentially use a subset of the INEX 2009 XML Wikipedia collection. This subset, so-called small collection, contains 54575 documents. On the other hand, the large collection contains around 2.7 million documents and takes 60 GB. The large collection is used only in the baseline experiments for various number of clusters.

As the baseline, we form clusters by applying C^3M algorithm to XML documents represented with the bag of words representation of terms. For several different number of output clusters, namely 100, 500, 1000, 2500, 5000 and 10000, we obtain the clusters and evaluate them at the online evaluation website of INEX 2009 XML Mining Track. The website reports the standard evaluation criteria for clustering such as micro purity, macro purity, micro entropy, macro entropy, normalized mutual information (NMI), micro F1 score and macro F1 score for a given clustering structure. However, only purity measures are used as the official evaluation criteria for this task. In Tables 5.1 and 5.2, we report those results for clustering small and large collections, respectively. For the latter case we experimented with three different numbers of clusters such as 100, 1000 and 10000. A quick comparison of the results in Tables 5.1 and 5.2 for corresponding cases implies that purity scores are better for the smaller dataset than that of the larger dataset, especially for large number of clusters. We anticipate that better purity scores for the large collection can be obtained by using a higher number of clusters. Next, we experiment with the clusters produced by the pruning-based approaches. For each pruning technique, namely, TCP and DCP, we obtain the document vectors at four different pruning levels; i.e., 30%, 50%, 70% and 90%. Note that, a document vector includes term id and number of occurrences for each term in a document, stored in the binary format (i.e., as a transpose of an inverted index). In Table 5.3, we provide results for the small collection and 10000 clusters. Our findings reveal that up to 70% pruning with DCP, quality of the clusters is still comparable to or even superior than the corresponding baseline case, in terms of the evaluation measures.

Regarding the comparison of pruning strategies, clusters obtained with DCP yield better results than those obtained with TCP up to 70% pruning for both micro and macro purity measures. For the pruning levels higher than 70%, DCP and TCP give better results interchangeably for these measures. In the experiments presented in Chapter 4, we observed a similar behavior regarding the retrieval effectiveness of indexes pruned with TCP and DCP.

From Table 5.3, we also deduce that DCP-based clustering at 30% pruning level produces the best results for both of the evaluation measures in comparison

No. of clusters	Micro Purity	Macro Purity
100	0.1152	0.1343
500	0.1528	0.1777
1000	0.1861	0.2147
2500	0.2487	0.3031
5000	0.3265	0.4160
10000	0.4004	0.5416

Table 5.1: Micro and macro purity values for the baseline C^3M clustering for different number of clusters using the small collection.

No. of clusters	Micro Purity	Macro Purity
100	0.1566	0.1234
1000	0.1617	0.1669
10000	0.1942	0.2408

Table 5.2: Micro and macro purity values for the baseline C^3M clustering for different number of clusters using the large collection.

Pruning Strategy	Prune(%)	Micro Purity	Macro Purity
No Prune	0%	0.4004	**0.5416**
DCP	30%	**0.4028**	0.5400
TCP	30%	0.3914	0.5229
DCP	50%	0.4019	0.5375
TCP	50%	0.3870	0.5141
DCP	70%	0.4016	0.5302
TCP	70%	0.3776	0.5042
DCP	90%	0.3783	0.4768
TCP	90%	0.3639	0.5073

Table 5.3: Comparison of the purity scores for clustering structures based on TCP and DCP at various pruning levels using the small collection. Number of clusters is 10000. Prune (%) field denotes the percentage of pruning. Best results for each measure are shown in bold.

to the other pruning-based clusters. For this best-performing case, namely DCP at 30% pruning, we also provide performance findings with varying number of clusters (see Table 5.4).

No. of clusters	**Micro Purity**	**Macro Purity**
100	0.1021	0.1265
500	0.1347	0.1539
1000	0.1641	0.1917
2500	0.2234	0.2737
5000	0.2986	0.3854
10000	0.4028	0.5400

Table 5.4: Micro and macro purity values for DCP at 30% pruning for different number of clusters.

The comparison of the results in Tables 5.1 and 5.4 shows that the DCP-based clusters are inferior to the corresponding baseline clustering up to 10000 clusters, but they provide almost the same performance for the 10000 clusters case. Other than the standard evaluation criteria INEX 2009 XML Mining Track, we also investigate the quality of the clusters relative to the optimal collection selection goal. To this end, a set of queries with manual query assessments from the INEX Ad Hoc track are used and each set of clusters obtained is scored according to the result set of each query. According to the clustering hypothesis [20], the documents that cluster together have similar relevance to a given query. Therefore, it is expected that the relevant documents for ad-hoc queries will be in the same cluster in a good clustering solution. In particular, mean Normalised Cluster Cumulative Gain (nCCG) score is used to evaluate the clusters according to the given queries.

In Table 5.5, we provide the mean and the standard deviation of nCCG values for our baseline C^3M clustering on the small data collection. Regarding the pruning-based approaches, the mean nCCG values obtained from the clusters produced by TCP and DCP for various pruning levels are provided in Table 5.6. In parallel with the findings obtained by the purity criteria, mean nCCG values of the clusters obtained by DCP are still better than or comparable to the ones

obtained by the baseline approach up to 70% pruning level. On the other hand, TCP approach yields better mean nCCG values even at 90% pruning level. In Table 5.7, we provide the mean nCCG values obtained from different number of clusters formed by the DCP approach at 30% pruning level. A quick comparison of the results in Table 5.7 with those in Table 5.5 reveals that clusters obtained after DCP pruning are more effective than the clusters obtained by the baseline strategy for various number of clusters.

No. of clusters	**Mean nCCG**	**Std. Dev. CCG**
100	0.7344	0.2124
500	0.6258	0.2482
1000	0.5986	0.2790
2500	0.5786	0.2352
5000	0.5918	0.2395
10000	0.4799	0.2507

Table 5.5: Mean and standard deviation of nCCG values for the baseline C3M clustering for different number of clusters using the small collection.

Pruning Strategy	**Prune(%)**	**Mean nCCG**	**Std. Dev. CCG**
No Prune	0%	0.4799	0.2507
DCP	30%	0.4950	0.2549
TCP	30%	0.4950	0.2549
DCP	50%	0.4828	0.2467
TCP	50%	0.4618	0.2236
DCP	70%	0.4601	0.2207
TCP	70%	0.5075	0.2176
DCP	90%	0.4613	0.2343
TCP	90%	0.5132	0.2804

Table 5.6: Comparison of the mean and standard deviation of nCCG values for clustering structures based on TCP and DCP at various pruning levels using the small collection. Number of clusters is 10000. Prune (%) field denotes the percentage of pruning.

Finally, in Figure 5.1 we compare the performance of the C^3M clustering with the other runs submitted to INEX 2009 in terms of the mean nCCG. For each

No. of clusters	Mean nCCG	Std. Dev. CCG
100	0.7426	0.1978
500	0.6424	0.2326
1000	0.5834	0.2804
2500	0.5965	0.2504
5000	0.5929	0.2468
10000	0.4950	0.2549

Table 5.7: Mean and standard deviation of nCCG values for DCP at 30% pruning for different number of clusters.

case (i.e., number of clusters), we plot the highest scoring clustering approaches from each group. Note that, for cluster numbers of 100, 500, 2500, and 5000, our strategy (denoted as C3M) corresponds to DCP based clustering at 30%; and for the cluster number of 10000 we report the score of TCP based clustering at 90%. For one last case where the cluster number is set to 1000, we report the baseline C^3M score, which turns out to be the highest.

5.5 Conclusion

In this chapter, we employ the well-known C^3M algorithm for content based clustering of XML documents. Furthermore, we use index pruning techniques from the literature to reduce the size of the document vectors on which C^3M operates. Our findings reveal that, for a high number of clusters, the quality of the clusters produced by the C^3M algorithm does not degrade when up to 70% of the index (and, equivalently, the document vectors) is pruned.

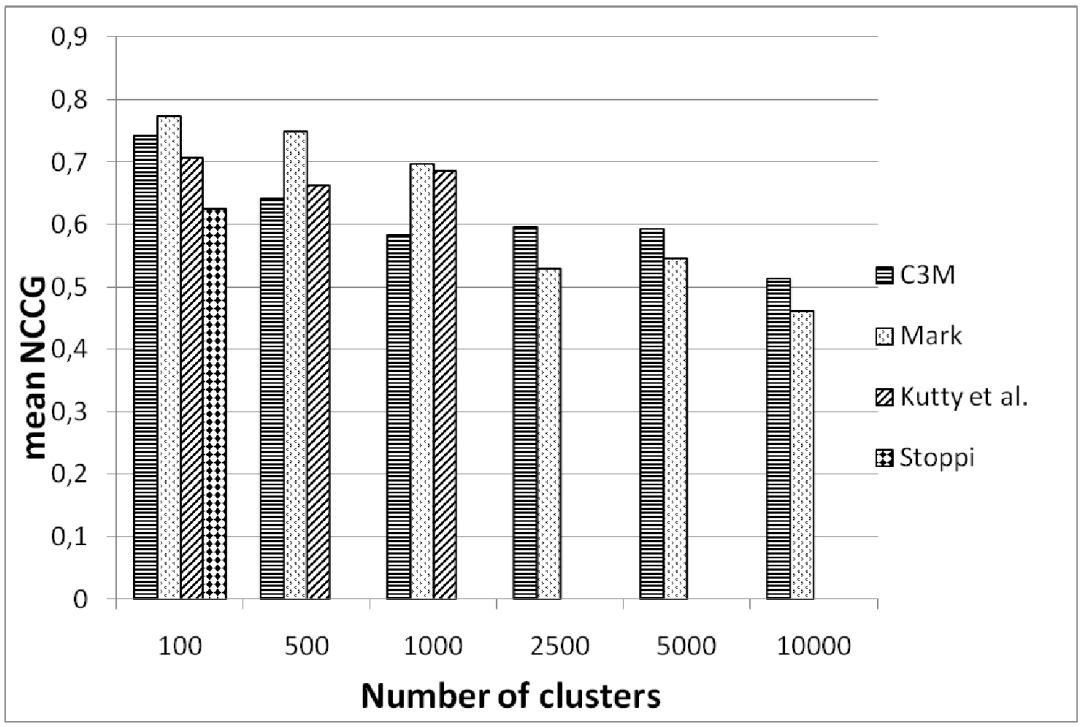


Figure 5.1: Comparison of the highest scoring runs submitted to INEX for varying number of clusters on the small collection.

Chapter 6

Conclusions and Future Work

XML keyword search is a challenging and promising research topic. It would satisfy the user by retrieving specific and relevant information at the right granularity, provided that efficient and effective XML retrieval systems could be developed. In this thesis, we have dealt with the querying, indexing and ranking aspects of XML keyword search using a full element-index. We have shown that such an index, together with state-of-the-art IR query processing techniques, could allow efficient and effective keyword search over XML databases and document collections in a unified manner.

More specifically, we first implemented two recent result definition techniques, SLCA and TOP-K using full element-index and document ordered query processing. We compared our approach to the well known techniques finding SLCA which are Scan Eager and Index Eager algorithms. The experimental results have revealed that, contrary to the current belief, using full element-index could be simple yet efficient for XML keyword search. In addition to the efficiency results, it has been shown that our approach can retrieve SLCA nodes with greater effectiveness compared to Index Eager and Scan Eager algorithms. Since the full element-index allows the term statistics to be calculated according to the full content of an element, more accurate ranking is possible in comparison to a direct dewey index.

In XML retrieval, when the documents are deeply nested, sizes of the indexes could be problematic for large scale systems. To handle that situation, we applied a lossy compression technique, static pruning, on full element index to further reduce its size. We managed to prune 50% of the index without losing effectiveness. With a comprehensive set of experiments, several tasks of XML keyword search were evaluated and static pruning techniques were proven to be useful not only on classic flat document indexes but also on XML element indexes.

Lastly, we investigated another aspect of XML retrieval, which is clustering. We employed the well known C^3M clustering algorithm to group XML documents. The document vectors used in clustering were obtained from the full element index. To observe the performance impact of static pruning on XML clustering, both unpruned and pruned indexes were employed. The experimental results revealed that even 70% pruned document vectors yield a clustering structure in the same quality with unpruned ones with C^3M algorithm.

The whole set of experiments in this thesis support that full element-index, when used with appropriate query processing and clustering algorithms, could be an optimal solution for XML retrieval. Future work directions involve extending our framework as a unified retrieval system for both XML and flat documents. Such a system could allow generating query results in different forms (such as SLCA or TOP-K) according to user's preferences and retrieving the relevant data from different types of collections. With regard to compression issues, we intend to experiment some other static index pruning techniques for query processing and clustering tasks.

Bibliography

[1] I. S. Altingövde, D. Atilgan, and Ö. Ulusoy. Xml retrieval using pruned element-index files. In *Proceedings of the 32nd European Conference on Information Retrieval*, pages 306–318, 2010.

[2] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy. Exploiting query views for static index pruning in web search engines. In *Proceedings of the 18th ACM International Conference on Information Knowledge and Management*, pages 1951–1954, 2009.

[3] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy. A practitioner's guide for static index pruning. In *Proceedings of the 31st European Conference on Information Retrieval*, pages 675–679, 2009.

[4] R. Blanco and A. Barreiro. Boosting static pruning of inverted files. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 777–778, 2007.

[5] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 2006 ACM International Conference on Information Knowledge and Management*, pages 182–189, 2006.

[6] F. Can, I. S. Altingövde, and E. Demir. Efficiency and effectiveness of query processing in cluster-based retrieval. *Information Systems*, 29(8):697–717, 2004.

[7] F. Can and E. A. Ozkarahan. Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases. *ACM Transactions on Database Systems*, 15(4):483–517, 1990.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 45–56, 2003.

[9] E. S. de Moura, C. F. dos Santos, B. D. de Araujo, A. S. da Silva, P. Calado, and M. A. Nascimento. Locality-based pruning methods for web search. *ACM Transactions on Information Systems*, 26(2), 2008.

[10] L. Denoyer and P. Gallinari. The wikipedia xml corpus. In *Comparative Evaluation of XML Information Retrieval Systems, 5th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2006, Revised and Selected Papers*, pages 12–19, 2007.

[11] P. F. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[12] S. Garcia. *Search Engine Optimization Using Past Queries*. PhD thesis, RMIT, 2007.

[13] S. Geva. Gpx - gardens point xml information retrieval at inex 2004. In *Advances in XML Information Retrieval, Third International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2004, Revised Selected Papers*, pages 211–223, 2005.

[14] S. Geva. Gpx - gardens point xml ir at inex 2005. In *Advances in XML Information Retrieval and Evaluation, 4th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2005, Revised Selected Papers*, pages 240–253, 2006.

[15] S. Geva. Gpx - gardens point xml ir at inex 2006. In *Comparative Evaluation of XML Information Retrieval Systems, 5th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2006, Revised and Selected Papers*, pages 137–150, 2007.

[16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 16–27, 2003.

[17] L. C.-S. H. Su-Cheng. Node labeling schemes in xml query optimization: A survey and trends. *IETE Technical Review*, 26(2):88–100, 2009.

[18] Initiative for the evaluation of xml retrieval, 2009. http://www.inex.otago.ac.nz/.

[19] K. Y. Itakura and C. L. A. Clarke. University of waterloo at inex 2008: Adhoc, book, and link-the-wiki tracks. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 132–139, 2009.

[20] N. Jardine and C. J. van Rijsbergen. The use of hierarchic clustering in information retrieval. *Information Storage and Retrieval*, 7(5):217–240, 1971.

[21] J. Kamps, S. Geva, A. Trotman, A. Woodley, and M. Koolen. Overview of the inex 2008 ad hoc track. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 1–28, 2009.

[22] S. Kutty, T. Tran, R. Nayak, and Y. Li. Clustering xml documents using frequent subtrees. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 436–445, 2009.

[23] M. Lalmas. *XML Retrieval*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2009.

[24] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *Proceedings of the 2007 ACM International Conference on Information Knowledge and Management*, pages 31–40, 2007.

[25] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 72–83, 2004.

[26] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *Proceedings of the VLDB Endowment*, 1(1):921–932, 2008.

[27] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 191–198, 2007.

[28] B. Sigurbjörnsson and J. Kamps. The effect of structured queries and selective indexing on xml retrieval. In *Advances in XML Information Retrieval and Evaluation, 4th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2005, Revised Selected Papers*, pages 104–118, 2006.

[29] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. A. Baeza-Yates. Resin: a combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 131–138, 2008.

[30] A. Soffer, D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, and Y. S. Maarek. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–50, 2001.

[31] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

[32] The dblp computer science bibliography, 2010. http://www.informatik.uni-trier.de/ ley/db/.

[33] T. Tran, S. Kutty, and R. Nayak. Utilizing the structure and content information for xml document clustering. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 460–468, 2009.

[34] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[35] C. M. D. Vries and S. Geva. Document clustering with k-tree. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 420–431, 2009.

[36] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 537–538, 2005.

[37] C. Yi, W. Wei, L. Ziyang, and L. Xuemin. Keyword search on structured and semi-structured data. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 1005–1010, New York, NY, USA, 2009. ACM.

[38] Zettair search engine, 2009. http://www.seg.rmit.edu.au/zettair/.

[39] S. Zhang, M. Hagenbuchner, A. C. Tsoi, and A. Sperduti. Self organizing maps for the clustering of large sets of labeled graphs. In *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Revised and Selected Papers*, pages 469–481, 2009.