Efficient XML Query Processing and 
Full-Text Search

submitted by
Mohammed AbuJarour
on December 20, 2007

Supervisor
Prof. Dr. Gerhard Weikum

Advisors
Dr. Ralf Schenkel
Dr. Martin Theobald

Reviewers
Prof. Dr. Gerhard Weikum
Prof. Dr. Felix Naumann
Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, December 20, 2007

< Mohammed >

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, December 20, 2007

< Mohammed >
Abstract

XML query processing is an essential building block in ranked XML information retrieval. To rank documents according to a user’s query, we need to maintain pre-computed inverted index lists about the data set we consider. The way we handle these lists is the key to evaluate XML queries efficiently.

In this work, we introduce a new approach to arrange such index lists about XML documents in a compact form by removing redundant pieces from the index as much as possible. Our approach optimizes IO throughput and hence optimizes the overall throughput. The main focus of our work is to optimize storage requirements for this kind of ranked retrieval.

The way in which the new approach stores indexes could be viewed as a pre-computed memory image that is loaded incrementally according to the block-based query processing approach that we use. Our experiments show a rough gain of a factor of more than 60 in speedup and a compression rate of about 3 with respect to the previous version of our system that used Oracle as a back-end.
To those who taught me to live, to learn and to succeed... To my parents: Fatima and Ahmed, I wish to dedicate this work as a kind of love and respect...

To the one who supported me through each step of my master study, as well as my life, to my wife: Safa’a, I wish also to show my endless love...

To my family: my sisters, my brothers and their families...

To all of them I dedicate this work.
Acknowledgments

With the unlimited support that I’ve been getting from my supervisor and advisors, I wish to acknowledge their support. My friends, my colleagues and my advisors: With these simple words, let me thank you all. Special thanks to Prof. Dr. Felix Naumann, from Hasso-Plattner-Institute, for his revision of my thesis. I want also to thank everybody who helped me produce this work, especially staff members in Max-Planck-Institut für Informatik.
# Contents

Abstract iii  
Dedication iv  
Acknowledgments v  
Contents vi  
List of Tables ix  
List of Figures xi  

## 1 Introduction 1  
1.1 Motivation 1  
1.2 Our Approach and Contributions 4  
1.3 Overview of the Thesis 5  

## 2 Foundations 7  
2.1 XML Information Retrieval 7  
2.2 Narrowed Extended XPath I Queries 9  
2.3 Scoring Model 13  
2.3.1 Scoring Model - Content Conditions 13  
2.3.2 Scoring Model - Structural Conditions 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>Inverted Index Lists</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Query Processing</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Related Work</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>TopX V 1.0</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>The Intersect Package</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Index Compression vs. Retrieval Time of Inverted Files for XML Documents</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Our Approach</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>System Overview</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Block-Index List Structure</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Document Blocks</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Element Blocks - Content Information</td>
<td>38</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Element Blocks - Structural Information</td>
<td>41</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Histograms</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Data Compression - Example</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>Implementation Details</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Internal Query Representation</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Encoding Scheme</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Content SA</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Content and Structural RA</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Structural Constraints and Cost Model for Random Accesses</td>
<td>57</td>
</tr>
<tr>
<td>5.6</td>
<td>Merge Joins</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>Probabilistic Pruning</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>Experiments and Results</td>
<td>63</td>
</tr>
</tbody>
</table>
6.1 Data Collection ........................................... 63
6.2 Experiments Setup ...................................... 63
6.3 Results Evaluation ...................................... 64
  6.3.1 Performance Evaluation .............................. 66
  6.3.2 Storage Evaluation .................................. 67
6.4 Discussion .................................................. 67
6.5 Probabilistic Pruning - Results ......................... 70
6.6 Varied $C_r/C_s$ values - Results ........................ 72

7 Conclusion & Future Work .................................. 74
  7.1 Conclusion ................................................. 74
  7.2 Future Work .............................................. 74

Bibliography ...................................................... 76
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Schema of content information in the Oracle Database</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Schema of structure information in the Oracle Database</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>A Sample data piece from the database about tag-term section=internationalization</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>The same data in Table 4.1 according to the new structure.</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>The main attributes that characterize the query in our system.</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>The main attributes that characterize our example query: (//article[about (//author, John)]//sec[about(//p, Top-k) efficient method])).</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>The same data in Table 4.1 according to the new structure and its binary representation according to our encoding scheme.</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>System parameters we used in our experiments.</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>CO Test Cases: Fourth column shows time (in milliseconds) required by v 1.0 to execute the query. Fifth column shows time required by the new version (v 2.0) to execute the query.</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>CAS Test Cases: Fourth column shows time (in milliseconds) required by v 1.0 to execute the query. Fifth column shows time required by the new version (v 2.0) to execute the query.</td>
<td>65</td>
</tr>
</tbody>
</table>
6.4 Lengths of CO index lists measured in document blocks. . . 66
6.5 Lengths of CAS index lists measured in document blocks. . 66
6.6 Storage requirements by Oracle and our new approach for
   CO queries. .................................................. 67
6.7 Storage requirements by Oracle and our new approach for
   CAS queries. .................................................. 68
6.8 Probabilistic Pruning Experiment: ............................. 71
6.9 Experiments on CO queries with different values of \( \frac{C_r}{C_s} \) . . 72
6.10 Experiments on CAS queries with different values of \( \frac{C_r}{C_s} \) . . 73
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Google Top-5 results for terms <em>Saarland University</em>.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Yahoo! Top-5 results for terms <em>Saarland University</em>.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>A simple XML document.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>An article XML document (A) modeled as a tree.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>An article XML document (B) modeled as a tree.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Our example NEXI query modeled as a tree.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Our example NEXI query transformed into a DAG</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Example: The concepts and statistical measures used to assign elements content scores.</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>a) Inverted index lists for tag-term dimensions $p=Top-k$ (left), $p=efficient$ (middle) and $p=method$ (right). Documents in each index list are sorted on their scores descending. b) Inverted Block-index lists for the same tag-terms in part (a). The previous index lists are partitioned into blocks that could fit in the memory; block-size equals 2 in this example. The blocks are also resorted on document ids ascending.</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Example: Elements’ and Document’s scores with respect to a tag-term condition.</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Inverted Block-Index Lists used in the example shown in Figure 2.10.</td>
<td>21</td>
</tr>
</tbody>
</table>
2.10 TA example using the query \( \text{//p\[Top-k efficient method\]} \)
on the index lists shown in Figure 2.9. .......................... 22

4.1 Documents Bookkeeping [Adapted from [16]]. ......... 31
4.2 A high-level description of file structure including index lists,
document and element blocks. ................................. 34
4.3 A binary file containing three index lists. ................. 35
4.4 An example \textit{Keys_sa} file for the index lists shown in Figure 4.3. 36
4.5 An index list containing two document blocks. ............ 37
4.6 Document and Element blocks for content information. ... 39
4.7 Element Blocks and a \textit{Keys_ra} file for content information. 41
4.8 Document Blocks and element blocks for structural infor-
mentation. .......................................................... 42
4.9 A sample Histogram. .......................................... 44

5.1 Query DAG for the query: \( \text{//article[about (\text{//author, John})]/\text{//sec[about(\text{//p,}
\text{Top-k efficient method})]})} \). ................................. 50
5.2 Example: Fetching a content document block using RA. ... 56
5.3 Example: Fetching a structural document block using RA. 57
Chapter 1

Introduction

The amount of information is increasing all the time everywhere. We don’t face the problem of lack of information, but we are in most cases stuck when we try to find the required piece of information in time. Search engines play the basic role in the solution of this problem. But, because types and structures of the information we have are versatile, some specific fields or sub-fields of Information Retrieval need special care. In this work, we deal with XML documents, trying to rank them according to user-defined criteria.

We have already investigated this field in the work of TopX version 1.0 \[16\]. In this thesis, we introduce a new approach to arrange the index lists so that IO-throughput is optimized. This special arrangement of indexes helps also to execute user’s queries faster and report the results earlier.

1.1 Motivation

Almost everyone uses web-based search engines, like Google or Yahoo!, all the time. In most cases, our queries are answered within the first 10 or 20 results reported by the search engine. We get the results ordered according to their relevance to the queried terms. The first \( k \) such results are called
CHAPTER 1. INTRODUCTION

top-

k

results (where

k

is an integer). Top-k results are usually ranked according to some scoring model [16].

Suppose our search terms are Saarland University. These terms occur in many pages on the web. These include the official web site of Saarland University, departments’ web pages, Wikipedia article about Saarland University, personal web pages for people studied at Saarland University and added this information to their résumés (for example), and more. . . Among this collection, it’s obvious that Saarland University’s official web page is the most relevant page, and personal pages are the least relevant ones (for normal people). The classification of the remaining pages is not strict. Google classifies departments pages before the Wikipedia article, whereas Yahoo! reports the Wikipedia article before departments pages. This classification is determined by the scoring model [16] used by the search engine.

Figure 1.1 and Figure 1.2 show the results of the previous query in Google and Yahoo! respectively. Notice that Google classifies the Wikipedia article as the 4th result, whereas Yahoo! classifies it as the 2nd result.

In order to answer such queries, the system maintains a sort of meta-data about the items. The arrangement of this meta-data is crucial and could make life easier for the search engine. In this work, we introduce a special arrangement, of such meta-data about XML documents in the form of index lists, that optimizes IO throughput and helps produce the results faster. We organize information about documents and elements in a block-based structure that divides index lists into block-index lists and organizes blocks in each list carefully. We make sure that we remove as much as possible from the redundant pieces of information in the index structure; this gives a compression rate of about 3 in the index. This structure allows also a block-based execution of the query. The execution of the query is based on the family of threshold algorithms such as Fagin’s Threshold
1.1. MOTIVATION

Figure 1.1: Google Top-5 results for terms Saarland University.

Figure 1.2: Yahoo! Top-5 results for terms Saarland University.
algorithm [5]. The basic idea here is to terminate the evaluation process as early as we get enough information about documents from the index lists.

1.2 Our Approach and Contributions

The problem statement of this work is the following: We have a collection of XML documents. Each XML document has its own structure (no DTD or XSD). The user inputs his/her query, which is a multidimensional query with content and structural conditions. The output should be a ranked list of documents that satisfy the query. This list is output in iterations; each iteration contains $k$ documents. The main challenge is to manage IO operations to speed up the evaluation of the first iteration that produces the top-$k$ documents.

We use binary files as a storage medium. All content information is stored in one binary file on disk. This binary file is partitioned logically into index lists. Each index list contains content information about one tag-term. An index list is divided into document blocks that could be fetched one by one sequentially or one at a time randomly. Each document block is divided into element blocks where the information about one document with respect to a content condition is stored.

The structural information is encoded in the same way, but an index list contains information about one element type (tag); no terms is considered. The main difference here, is that structural constraints have static scores that we don’t store in the element blocks.

This approach allows both sorted accesses and random accesses. In sorted accesses, we read an index list sequentially; this is cheap. In random accesses, a specific part of the index list is read as needed; this is more expensive. Special schedulers are used to manage both types of accesses.
to speedup the whole process and guarantee that we don’t miss any top-k document.

The contributions of this work are:

1. A new approach that optimizes storage requirements for ranked XML IR using NEXI query language.

2. Better IO throughput achieved using sequential accesses to the block-index lists in the binary file, or using random accesses to resolve expensive predicates and uncertainty about some candidates.

3. Better overall performance since

   - the block structure could be considered as a pre-computed memory image that is loaded incrementally.

   - we can use merge joins instead of hash joins, where we order elements using their in-memory binary representations.

   - most operations boil down to pointer operations because we load the pre-computed memory image incrementally.

1.3 Overview of the Thesis

After this introductory part, the remaining parts of this thesis are organized like this:

1. Chapter 2 contains the necessary terminology and concepts to proceed with the thesis.

2. Chapter 3 shows some related works, to which we compare our approach, or which we use in this work.
3. Chapter 4 describes in details our approach and the efficient data structures that we introduce.

4. Chapter 5 contains more technical and implementation details about our approach.

5. Chapter 6 shows our experiments and the results we achieve.

6. Chapter 7 sums up the work and shows our future plans.
Chapter 2

Foundations

This chapter includes the necessary concepts and terminology to proceed with the remaining chapters of this thesis. We introduce XML IR using a simple example in section 2.1. Then, we define the query language that we support, which is the NEXI query language. After that, we show how to assign elements and documents their scores, and we refer to the original work. The inverted index lists are introduced in section 2.4. The last section shows how we process the query.

2.1 XML Information Retrieval

The Extensible Markup Language (XML) is a data storage toolkit, a configurable vehicle for any kind of information, an evolving and open standard embraced by everyone from bankers to webmasters [14]. XML is a metmarkup language for text documents. Data is included in XML documents as strings of text. The data is surrounded by text markup that describes the data. XML’s basic unit of data and markup is called an element. The XML specification defines the exact syntax this markup must follow: how elements are delimited by tags, what a tag looks like, what names are acceptable for elements, where attributes are placed and so forth [8].

In Figure 2.1, tags are shown in brown and content data (string) is shown
in black. XML documents are usually modeled as trees, where elements represent nodes and the edges between them represent the relationships between elements in the document.

```
- <article>
  <author>John</author>
  <id>24</id>
  <title>IR</title>
- <sec>
  <p>IR is ...</p>
  <p>New IR ...</p>
</sec>
</article>
```

Figure 2.1: A simple XML document.

Figure 2.2 and Figure 2.3 show two XML documents that model two articles. The contents (or part of them) of each content tag are shown in blue below each tag respectively. Suppose that we are interested in articles written by John about XML IR. This query is expressed in XPath like this:
```
//article[//author[contains(text(), "John")]] and
//section[contains(text(), "XML IR")]
```

Document B in Figure 2.3 satisfies the three content conditions; the author is John and it has both terms XML and IR among its sections, whereas document A Figure 2.2 satisfies only two content conditions among the three because it doesn't include the term XML among its sections. Still, in XML IR both documents could be considered as valid results for the above query, but document B is more relevant to this query and should be reported before document A. We say that document B has a higher score and has a higher rank than document A, with respect to the above query.

Modeling XML documents as trees is not only a conceptual representation that is useful to understand and deal with XML documents, but
it is also useful to check some constraints, like parent-child or ancestor-
descendant relationships among elements. A simple and an efficient tech-
nique to deal with XML elements is to assign each element a unique id 
using pre-, in- or post-order traversals [18]. In this work, we use pre- and 
post-orders of elements to test many constraints and relationships. For 
example, to test whether two elements have a common parent, to test 
whether two elements satisfy the descendant relationship and more ...

2.2 Narrowed Extended XPath I Queries

Our system supports the Narrowed Extended XPath I [17] query lan-
guage; NEXI for short. NEXI is based on XPath 1.0 [3], where the descendant(//) and the self (.) axes are the only allowed XPath axes and the 
IR-style about operator is introduced instead of the contains operator in
XPath.

The user could simply issue Content-Only (CO) queries where she doesn’t care about the elements where the terms occur or simply doesn’t know the structure of the queried files. For example, the query “Top-k efficient method” doesn’t impose any structural constraints on the elements where the terms occur, nor it requires a specific sequence or order of elements, and hence, it could be rewritten in NEXI like this

`//*[about(., Top-k efficient method)]`. The wildcard character (*) allows any type of elements to occur here.

If the user knows the structure of the queried files and she cares about it, then she could issue more narrowed queries including structural constraints called Content-And-Structure (CAS) queries. Again, the descendant(//) and the self(.) axes are the only allowed axes in NEXI. Consider this example query:

`//article[about(.//author, John)]//sec
//p[about(., Top-k efficient method)]`

In this query, we are interested only in `p` (paragraph) elements containing the terms “Top-k efficient method” among their contents and satisfying the descendant relationship with `sec` (section) elements in an `article` written by `John`. So, if a paragraph element doesn’t include any of these terms, is not a descendant of a section element or doesn’t occur in an `article` written by `John`, then it’s excluded (generally speaking).

We classify the nodes denoted by a NEXI query into two categories:

- **Target Elements**: the right-most top level node test in the query is called the target element. In the previous CAS query, `p` is the target element.
- **Support Elements**: all other nodes denoted by the query are called
support elements. In the previous CAS query, article, author and sec elements represent support elements.

The result of such a query includes documents satisfying at least one target condition. A document containing the Top-k term in a p (paragraph) element that doesn’t include the other two terms (efficient method) is considered a valid result for the above CAS query. Of course, other documents satisfying more conditions are better matches. This is called the “Andish” mode. On the other hand, a document is a valid result in the “conjunctive” mode if it satisfies all conditions in the query. In this work, we consider the “Andish” mode, though it could be expanded to cover the conjunctive case easily.

The user has the choice to specify either a document granularity or an element granularity. In the first case, the system reports documents where target elements occur. In the latter case, the system reports the target elements themselves.

We model NEXI queries as trees also, where terms represent leaf nodes, see Figure 2.4. Those terms (leaf nodes) are merged with their directly preceding nodes in the tree, see Figure 2.5. Our example CAS query, that asks for paragraph elements containing terms (Top-k efficient method) and satisfying the descendant relationship with section elements in an article written by John, is modeled as a tree like the one shown in Figure 2.4. Terms John, Top-k, efficient and method are shown as leaf nodes in the tree.

We merge each of these terms with its directly preceding node in the tree. John is merged with author to get the content condition or tag-term author=John. The terms Top-k, efficient and method are also merged with their directly preceding node in the tree to get the content
conditions or tag-terms \( p=\text{Top-k} \), \( p=\text{efficient} \) and \( p=\text{method} \), respectively. To reflect that \( p=\text{Top-k} \), \( p=\text{efficient} \) and \( p=\text{method} \) should occur in the same element, they are connected through the self constraint, see Figure 2.5. These steps translate the query into a DAG. We are translating the input NEXI query into a structure that could be handled by the system. More details about the internal query representation are found in section 5.1.

Our system deals with the query and its nodes using their corresponding numbers in the root-to-leaf paths. Words in our example CAS query are numbered from 0 to 5 as shown in red in Figure 2.5. All root-to-leaf paths of this query are:
Bold-faced paths represent target conditions. We say that this query has a dimensionality of 6 because it has in total 6 (structural and content) conditions.

2.3 Scoring Model

The score of an item reflects the importance (relevance) of that item with respect to the query; the more relevant the item is, the higher score it gets. A paragraph, that contains many occurrences of the terms Top-k, efficient and method, is more relevant and more important than a paragraph containing only one occurrence of the term method. To specify the score of each content element with respect to a term, we use a scoring model \([16]\) as we’ll see in the sequel. The case of structural constraints is simple because structural constraints are assigned static score masses. If an item satisfies the article structural constraint or the sec structural constraint -for example-, then it gets one structural score mass in either case.

2.3.1 Scoring Model - Content Conditions

Content scores are usually normalized between 0.0 and 1.0. If an item has a score of 0.0 with respect to a term, then it’s not that relevant or important, in contrast to an item that is highly relevant to a term with score 1.0. The scoring model we use is taken from \([16]\).
Term ordering in IR queries is often assumed to be irrelevant, so, in this scoring model, the content of node $n$ is viewed as a bag-of-words. In this interpretation, a query is an unordered set of search terms (and phrases) [17]. It ignores the order of words as well as any punctuation or structural information, but retains the number of times each word appears [19, 13]. The full-content of element $e$ is defined as the content of $e$ plus the contents of all its descendant elements. In Figure 2.6, the contents of each content element are shown in blue below the element respectively. The author element has the value (John) ... etc. The full-content of the sec element is the concatenation of the contents of its two descendant elements (sub-nodes), $p_6$ and $p_7$. This is shown in red in Figure 2.6. The full-content of the article element, is the concatenation of the contents of all content elements in the document, this is shown in red, right to the article element.

This scoring model relies on statistical measures. These are:

- **Full-content Term Frequency $ftf(t,e)$**: The full-content term frequency of term $t$ in element $e$ is defined as the absolute number of $t$ occurrences in the full-content of element $e$. For example, the full-content
term frequency of term query in the article element \( \text{ftf}(\text{"query"}, \text{article}) \) in Figure 2.6 is 3, because term query occurs 3 times (underlined) in the full-content of the article element.

- Tag Frequency \( N_A \): Tag frequency of tag \( A \) is defined as the absolute number of elements with tag \( A \) in the entire collection. For example, tag frequency of tag \( p \) (\( N_p \)) in Figure 2.6 is 2, because there are only 2 elements with tag \( p \) in this collection. Tag frequency of tag author (\( N_{author} \)) is 1, and tag frequency of tag figure (\( N_{publisher} \)) is 0 in Figure 2.6.

- Element Frequency \( ef_A(t) \): element frequency of tag \( A \) with respect to term \( t \) is defined as the absolute number of elements with tag \( A \) containing term \( t \) among their full contents in the entire collection. Element frequency of tag \( p \) with respect to term “query” \( ef_p(\text{"query"}) \) in Figure 2.6 is 2, because there are 2 elements with tag \( p \) containing term “query” among their full contents in this collection. Element frequency of tag sec with respect to term “query” \( ef_{sec}(\text{"query"}) \) is 1 in Figure 2.6.

Each element is assigned a score with respect to each term in the query. The score of an element \( e \) with respect to such a tag-term condition \( A= t \), where \( A \) is the tag and \( t \) is the term that should occur in the full-content of the element, should reflect the occurrence statistics of that term and the specificity of the tag-term, as well as the size or the compactness of the sub-tree rooted at \( e \). These measures are combined together in Equation 2.1 to derive such a score. In Equation 2.1, occurrence captures \( \text{ftf}(t, e) \), specificity is derived from tag frequency and term frequency, and size considers the subtree or element size for length normalization.

\[
\text{score} (e, A = t) = \frac{\text{occurrence} \cdot \text{specificity}}{\text{size} (e)}
\] (2.1)
The score of element $e$ with respect to content conditions $A//[t_1, \ldots, t_m]$ is the aggregation of element’s scores for each term $t_i$. This is formalized in Equation 2.2.

$$score(e, //A[t_1\ldots t_m]) = \sum_{i=1}^{m} score(e, A = t_i)$$ (2.2)

For more details, please refer to [16].

2.3.2 Scoring Model - Structural Conditions

Each structural constraint is assigned a static score mass $c$ which is a system parameter, see Table 6.1. If a document satisfies a structural constraint, then it gains a structural score of $c$. Moreover, if a document satisfies $\rho$ structural constraints, then it gains a structural score ($S_s$) of $(\rho \times c)$.

The value of $c$ reflects the importance of structural constraints versus content conditions. If $c$ has a large value, then structural constraints are more important than content conditions. If content conditions are more important than structural constraints, then $c$ gets a smaller value. In the current settings, we set $c$ to 1.0.

Our example query, shown in Figure 2.5, has a dimensionality of 6, among which two are structural constraints. All constraints in this query are:

0: //article
1: //article//author=John
2: //article//sec
3: //article//sec//p=Top-k
4: //article//sec//p=efficient
5: //article//sec//p=method

Structural constraints are shown in red; these are the first (article) and the third (sec) constraints. A document that satisfies the article structural constraint gains a structural score of $c$. It gains another structural
score of $c$ if it satisfies the other structural constraint; the sec constraint. Any document could gain at most $(2 \times c)$ structural score with respect to this query.

The score of the document combines both content conditions and structural constraints based on the scoring model used, [15].

### 2.4 Inverted Index Lists

Among the collection of documents that we consider, not all documents are relevant to each query. Moreover, the relevant documents don’t have the same level of importance. This means that we should arrange the documents in a smart way that enables the system to answer user’s queries efficiently. This issue is handled using the Inverted Index-Lists.

An inverted index list [21] is a sequence of documents and their scores, where documents in the same index list are ordered on the score in a descending manner. For each content and structural condition in the query, we create such an inverted index list.

The left part of Figure 2.7 shows some examples of such inverted index lists. The left-most index list shows information about documents whose $p$ elements contain the term Top-$k$ among their contents. The first part of the index list holds document id and the second part holds the score of the document with respect to the respective term (Top-$k$ in this case). The middle index list shows information about documents whose $p$ elements contain the term efficient among their contents. The third index list shows information about documents whose $p$ elements contain the term method among their contents. Notice that documents in each index list are sorted on score in a descending manner.

We need to read these index lists in order to answer the query, but in
some cases, these lists could grow very large so that an index list couldn’t
fit in the memory. Here, the concept of inverted block-index lists shines. As
the name suggests, each index list is partitioned into (document-) blocks
that could fit in the memory. Documents in the same document block are
re-sorted on document id in an ascending manner. This step allows us to
join the new document with the already-read documents using merge join.

The right part of Figure 2.7 shows this concept. Each inverted index list
is transformed into an inverted block-index list by partitioning the list into
document blocks, with the same size; size equals 2 in our example. Notice
that documents in the same document block are re-sorted according to
document id’s. For example, the index list ($p=Top-k$) is divided into three
blocks, each of which contains two documents. The first block contains
documents (12 [0.9], 5[0.8]). In the inverted index list, document 12 is
listed before document 5 because document 12 has a higher score (0.9) than
document 5 (0.8). In the inverted block-index list (right part), document 5
is listed before document 12 because document 5 has a smaller document
id.

Assume that we are interested in paragraph elements containing terms
($Top-k$ efficient method). This query is expressed in NEXI like this:
//p[about(., Top-k efficient method)]. Each $p$ element in each
document in the collection has a score with respect to each term in the
query. Figure 2.8 shows an example document with 9 $p$ elements. Each
$p$ element has a score with respect to each term in the query. Tag $p_{11}$
(highlighted with a red rectangle) has a score of 0.7 with respect to the
$Top-k$ term, has a score of 0.3 with respect to the efficient term and has a
score of 0.4 with respect to the method term. A zero score means that the
element is not that relevant to the term, e.g. $p_{15}$ is not that relevant to
any of the three terms in the query. Each document has also a score with
2.5 Query Processing

We model the query internally in a form of DAG. For each tag-term (content condition) and each structural condition we create an inverted block-index list. In order to answer the query, we have to read the related index lists. We have two choices: Sorted Accesses or Random Accesses. Sorted Accesses (SA) are cheaper than Random Accesses (RA), because in SA we fetch document blocks sequentially from disk, whereas in RA, we fetch document blocks in a random way from disk. Both types are used to read index lists,
but according to some heuristics that we introduce in the sequel.

When we read an index list using SA, we keep the current position of the cursor in the index list and we track the lowest score seen so far in the index list. This score is called high\textsubscript{i}, and serves as an upper bound for the unseen documents in the index list. High\textsubscript{i} is initialized with 1.0 at the beginning, because no document could have a score greater than 1.0. As we scan the index list, this value decreases monotonically until it reaches 0.0 at the end of the list. For example, in Figure 2.7, in the first index list \((p=\text{Top-}k)\), high\textsubscript{i} equals 0.8 after we read the first block from that index list because it is the lowest score seen so far and so no document in the remaining part of the index list could have a higher score. After we read the second block from the same index list, high\textsubscript{i} for that index list decreases to 0.4. The same is done for other index lists, but with different values.

The collection of relevant documents could be large or very large, and it is not wise to test all of them; i.e. we should use a smart algorithm to
get the mission complete. This is resolved using the Threshold-Algorithm family such as Fagin's Threshold-Algorithm [5]. The basic idea here is to stop the evaluation process as early as we gather enough information to judge which items are top-k items and which items are not.

The intuition behind threshold algorithms is simple. Because Index Lists could grow large and we want to find the top-k relevant documents as fast as possible, we stop reading the index lists as soon as the top-k results are found and we are sure that no more documents in the index lists could qualify as top-k documents. Let's show an example. Suppose our query is //p[Top-k efficient method] and our index lists are shown in Figure 2.9, and the user wants to find the top-3 documents. At the beginning, high \_i of each index list is initialized to 1.0. This means that no document could gather a final score more than (1.0+1.0+1.0= 3.0). Figure 2.10 shows document queue and other maintained information.

We read the index lists iteratively. In each iteration, we fetch a document block from each content index list sequentially. We keep track of the seen documents, their scores and their ranks in a queue. At the beginning, our queue is empty. Each index list has a high \_i value of 1.0. This is the upper bound for unseen documents in each list.

<table>
<thead>
<tr>
<th>ID</th>
<th>score</th>
<th>ID</th>
<th>score</th>
<th>ID</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.8</td>
<td>2</td>
<td>0.6</td>
<td>2</td>
<td>0.8</td>
</tr>
<tr>
<td>12</td>
<td>0.9</td>
<td>23</td>
<td>0.7</td>
<td>12</td>
<td>0.4</td>
</tr>
<tr>
<td>17</td>
<td>0.4</td>
<td>8</td>
<td>0.2</td>
<td>5</td>
<td>0.3</td>
</tr>
<tr>
<td>23</td>
<td>0.7</td>
<td>12</td>
<td>0.5</td>
<td>20</td>
<td>0.4</td>
</tr>
<tr>
<td>2</td>
<td>0.3</td>
<td>5</td>
<td>0.1</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>11</td>
<td>0.1</td>
<td>10</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.9: Inverted Block-Index Lists used in the example shown in Figure 2.10.

In the first iteration, we fetch the first document block from the first
index list ($p=\text{Top-k}$). This document block has two documents in this case. Our queue is empty and so these two documents are among the top-3 documents seen so far. The value of $\text{high}_i$ for this list is updated to (0.8). Then we fetch the first document block from the second index list ($p=\text{efficient}$). It has also two documents, they are new documents. We update our queue accordingly. Our queue has now four documents, the first three documents are Top-k items (bold ones) and the fourth document is a candidate document; we can’t remove it because it could gather more score from other index lists. The value of $\text{high}_i$ for the second index list is updated also to 0.6. When we fetch the first document block from the third index list ($p=\text{method}$), we don’t get new documents because documents 2 and 12 have already been seen in other dimensions. In this case, their scores are updated by aggregating their scores from different index lists. Document 2 had a score of 0.6 in the queue, its updated score now is
0.6 + 0.8 = 1.4. The same is done with document 12. \( \text{High}_i \) for this index list is updated to 0.4. Our queue is rebuilt to reflect the new scores. The first iteration is finished.

The second iteration starts by fetching the second document block from the first index list. New documents are added to the queue and the scores of the already-read documents are updated. The queue is rebuilt and \( \text{high}_i \) for this index list is updated. Then, we fetch the second index list from the second index list, the queue is rebuilt and \( \text{high}_i \) for this index list is updated. At this point, we can safely terminate the evaluation process because we have all the necessary information to judge our top-3 documents. The first three documents in our queue (bold faced) are the top-3 and the remaining documents could not qualify as top-k. The lowest sore in the top-k queue is 1.4, we call this score \( \text{min}_k \). Document 5, which is a candidate document, has been seen in the first index list only. It could be seen in the second and the third lists also, but it could gather at most the sum of its score and \( \text{high}_i \)'s of index lists two and three; this means \( 0.8 + 0.2 + 0.4 = 1.4 \). It couldn’t gather more than \( \text{mink} \). The same could be said about documents 17 and 8.
Chapter 3

Related Work

The first part of this chapter describes the first version of our XML ranking system, coined TopX [15, 16]. The second part introduces the work of Helmer et al.: “A Robust Scheme for Multilevel Extendible Hashing” [9]. We refer to this work as the intersect package throughout this thesis. The last part of this chapter briefly describes the work of Fuhr et al. about “Index Compression vs. Retrieval Time of Inverted Files for XML Documents” [6].

3.1 TopX V 1.0

TopX is a search engine for ranked retrieval of XML (and plain-text) data, developed at the Max-Planck Institute for Computer Science [12]. TopX supports a probabilistic-IR scoring model for full-text content conditions and tag-term combinations, path conditions for all XPath axes as exact or relaxable constraints, and ontology-based relaxation of terms and tag names as similarity conditions for ranked retrieval. For speeding up top-k queries, various techniques are employed: probabilistic models as efficient score predictors for a variant of the threshold algorithm, judicious scheduling of sequential accesses for scanning index lists and random accesses to compute full scores, incremental merging of index lists for on-demand, self-tuning query expansion, and a suite of specifically designed, precomputed indexes.
3.2. THE INTERSECT PACKAGE

TopX has been stress-tested and experimentally evaluated on a variety of datasets including the TREC Terabyte benchmark, the INEX XML information retrieval benchmark, and an XML version of the Wikipedia encyclopedia. TopX has also served as a reference engine for the INEX 2006 benchmarking initiative. It can be accessed for interactive queries on various datasets.

TopX v 1.0 is java-based and uses Oracle as a back-end. Content information is stored in the database according to the schema shown in Table 3.1. Structural information is stored in the database according to the schema shown in Table 3.2.

<table>
<thead>
<tr>
<th>DID</th>
<th>TAG</th>
<th>TERM</th>
<th>PRE</th>
<th>POST</th>
<th>LOCALSCORE</th>
<th>MAXSCORE</th>
</tr>
</thead>
</table>

Table 3.1: Schema of content information in the Oracle Database

<table>
<thead>
<tr>
<th>DID</th>
<th>TAG</th>
<th>PRE</th>
<th>POST</th>
<th>CPTS</th>
<th>MAXCPTS</th>
</tr>
</thead>
</table>

Table 3.2: Schema of structure information in the Oracle Database.

to evaluate structural path conditions.

3.2 The Intersect Package

The Intersect Package is an implementation of the work of “robust scheme for multilevel extendible hashing” proposed in [9]. This work handles the case of skewed hash keys efficiently, also it works for uniformly distributed keys very well.

The directory of the hash table is divided hierarchically into several subdirectories. The main motivation here is to arrange subdirectories on lower levels such that they share pages in an elegant way. This allows the package to save space without introducing a large overhead or comprising retrieval performance.

We use this package to retrieve offsets for document blocks in case we
need to issue a random access. Keys and offsets are organized in the keys.ra file in the form (key ‘	’ offset), see chapter 4. A key is composed of document id and tag( and term in case of content conditions).

3.3 Index Compression vs. Retrieval Time of Inverted Files for XML Documents

In [6], they try to minimize the size of the inverted files by compressing the index. To do so, they use two different approaches for reducing index space: the first approach compresses the entries in the index file and in the second approach they introduce a new data structure (called XS tree) that contains the description of the structure of one document in a compact form, so that it could be kept in the main memory.

According to their evaluation experiments, this approach can achieve very high compression rates. Unfortunately, this has the penalty of increasing query processing time because the compressed data needs to be decompressed so that they could process the query. Using this approach, we have to compromise between index compression rates and query evaluation time.

In the current work, which we introduce in this thesis, our compression techniques - both for content and structural information - help evaluate queries more efficiently. In the query evaluation step, we don’t need to decode all encoded entries in the index; rather most operations are carried out on the encoded representation of the entries. This also allows us to perform most operations using pointer operations, which are efficient in C++.
Chapter 4

Our Approach

The previous index used in TopX v1.0 uses Oracle as a back-end. Relational database schemas entail redundancy in the data. This motivated us to try to get rid of these redundant pieces of data as much as possible, this was in the first place. In the second place, we want to allow both SA and RA to our new index and evaluate the query in a block-based manner.

Our index is based on the concept *inverted index lists*. These index lists are pre-computed and stored on disk in a binary file according to some defined structure that we’ll introduce in this chapter. Some of these index lists could grow large or very large that they don’t fit in the main memory, that’s why an index list is divided into document blocks that could fit in the main memory, and hence the name Inverted Block-Index Lists. Documents in the same document block are re-sorted on their id’s ascending to allow applying merge joins, in the evaluation process. An index list contains other information about the elements of each document in the list. This information includes, for example, pre- and post-orders of each element in the document.

In this chapter, we show an overview of our system and show how we evaluate the query. Then, we show our new structure of inverted lists and their building blocks. After that, we show also an example to illustrate
the benefits that we gain. The example is followed by a discussion of our approach.

4.1 System Overview

Our system takes a NEXI query as input. The query is transformed into a special internal representation so that the system could handle it. The internal representation is similar to the one used in [16] and explained in section 5.1. For each tag-term and each structural constraint in the query, we create an index list. The structure of these index lists is described in the sequel. We maintain some state information about documents, their elements and scores; these are updated as we get more information from the index lists.

Content index lists are read mainly using sorted accesses. For each such an index list, we keep track of the least score seen so far in the index list. This score is called high_i. High_i serves as an upper limit for scores of unseen documents in the index list, as discussed in section 2.5. The values of high_i for all content index lists are used (with other information) to formulate the threshold condition that terminates the whole evaluation process whenever it’s satisfied. Fagin’s threshold algorithm [5] is the baseline of the evaluation process.

The set of content conditions in the query are referred to as E. Each document has to satisfy as much as it could from those conditions in E. We aggregate the scores of each document from each dimension that the document satisfies to get document’s score.

An element appears in the index list of dimension dim if it satisfies that condition. For content conditions, the element has to satisfy the tag-term condition, and hence it has a local score with respect to that tag-term. The
element could satisfy many conditions in the query, and so it’s assigned a local score for each dimension. Element’s score is the aggregation of its scores from different dimensions, as discusses in section 2.3.1.

Each document, that satisfies some content conditions $E$ and still has to satisfy the remaining content conditions $\overline{E}$, has two types of scores:

- **Worst score**: The worst score of a document $d$, $ws(d)$, is the score that has already been accumulated by the document. This is calculated from the score of the document in dimensions $E$ according to Equation 4.1. $E$ refers to the set of content conditions satisfied by the document. The worst score of a document is referred to as the score of the document through out this work.

$$ws(d) = \sum_{i \in E} s_i$$ (4.1)

- **Best score**: the best score that the document $d$ could gather, $bs(d)$, is the upper limit for the final score of the document. A document’s best score is the worst score of that document plus $high_i$ values of index lists that the document hasn’t satisfied yet; $\overline{E}$. Equation 4.2 formalizes this.

$$bs(d) = ws(d) + \sum_{i \in E} high_i$$ (4.2)

Our system keeps documents in two queues. Top-k queue holds the current top-k documents ordered by their scores in a descending manner. Other documents, that have some missing dimensions and could gather more score to qualify as top-k documents, are kept in the candidate queue, ordered by document id in an ascending manner. The score of the $k^{th}$ document in the top-k queue is called $min-k$. 
Min-\(k\) classifies documents, that satisfy at least one target condition, into three classes (see Figure 4.1):

1. If a document’s worst score is greater than \(min-k\), then this document is classified as a top-\(k\) document.
   Document \(d\) is top-\(k\) \(\iff ws(d) > min-k\).

2. If a document’s best score is greater than \(min-k\), but its worst score is less than or equal \(min-k\), then this document is kept in the candidate queue because it could gather enough score to qualify as a top-\(k\) document.
   Document \(d\) is candidate \(\iff (bs(d) > min-k\) and \(ws(d) \leq min-k\).

3. If a document’s best score is not greater than \(min-k\), then this document has no chance to qualify as a top-\(k\) document, and therefore it is dropped from the queue.
   Document \(d\) could be removed safely \(\iff bs(d) \leq min-k\).

Other documents that don’t satisfy any target condition yet, but have already gathered more than \(min-k\) i.e. \((ws(d) > min-k)\) are kept in the candidate queue.

The difference between \(min-k\) and a document’s (d) worst score is called \(\delta(d)\), see Figure 4.1. This is the amount of score needed for the document to move to the top-\(k\) queue. In probabilistic pruning (section 5.7), we need to estimate the probability that a candidate document could gather this \(\delta\) to qualify as a top-\(k\) item.

CAS queries contain structural constraints, in addition to the content conditions. An element satisfies a structural constraint if it has the same type in the structural condition. If the element satisfies a structural constraint, then the element appears in the index list for that dimension. A
document satisfies a structural constraint, if it has at least one element that satisfies that structural constraint.

Each structural constraint has a static score mass $c$. If a document satisfies some structural constraints $O$, and needs to satisfy the remaining structural constraints $\bar{O}$ in the query, it gets a structural score $S_s$ according to Equation 4.3.

$$S_s(d) = \sum_{i \in O} c$$  \hspace{0.5cm} (4.3)

The structural score mass that the document $d$ could gain by satisfying more structural constraints is called $\text{gap}(d)$. This value is calculated according to Equation 4.4. We use $\text{gap}(d)$ to decide whether to issue structural RAs for the document or not; this is called the Min-Probing principle [2]. We issue a structural RA for document $d$ if:

$$\text{score}(d) + \text{gap}(d) > \text{min-k}.$$
When a document satisfies some content conditions $E$ and some structural constraints $O$, then its worst score is calculated using the “Incremental Path Tests” [16]. Whenever we get more information from the index lists, we update the top-k queue, rebuild and prune the candidate queue. It’s necessary to remove a candidate document as soon as we’re sure that it won’t qualify as a top-k document because this makes the join process easier (section 5.6).

Algorithm 1 Main Algorithm(Query)

1: Initialize all index lists.
2: Issue SA in a round-robin manner against each content index list.
3: Merge new documents with old ones.
4: Rebuild Topk and Candidate Queues.
5: Check if any structural RA is needed.
6: Update documents and queues accordingly.
7: if the threshold condition is true then
6: Terminate
7: go to Line 19
8: end if
9: Check if any content RA is needed.
10: Update documents and the queues accordingly.
11: if the threshold condition is true then
12: Terminate
13: go to Line 19
14: else
15: goto Line 2
16: end if
17: Report Topk documents.

The main block-based query processing steps are shown in Algorithm 1. In the initialization step, the index lists are prepared to be scanned using SA. This includes moving the reading cursor to the beginning of the list and initializing the high\textsubscript{i} value of the index list to 1.0, and other state variables. In the next step, we start issuing SA to the content index lists in a round-robin manner. In each SA (or RA later) we fetch one document block from the index list. Each of these accesses is followed by a merge step that merges new fetched documents with old ones in the queue. According to the new information that we got from the SA, topk and candidate queues
are updated and irrelevant documents are dropped. In the next step, we check if we need to perform structural RA for some documents in the candidate queue according to the Min-Probe principle. In each RA we fetch one document block from the respective index list for structural information. As we get more information from structural RA, we update documents and queues according to this information. At this step, it could be possible to terminate the whole process, so we check the termination threshold condition. If we are not done yet, we check if any content RA is needed according to the Ben-Probing principle. Each content RA reads a content document block from the related index list. If any new information is fetched, we update state information according to it. Before we repeat the whole process, we check the termination threshold condition. This holds if no candidate document has a best score better than the current mink-k or the candidate queue is empty, and no unseen document could qualify as a top-k document. Of course, we are done if all content index lists are exhausted. Then, we stop and report the documents in the topk queue.

A high-level description of the main algorithm shows that we fetch a document block from the index, merge the new documents with the old ones, update our state information and queues, and test the termination condition. This shows another advantage of our block-based index structure.

4.2 Block-Index List Structure

The basic building block in the inverted block-index lists is the document block. An index list is composed of one or more document blocks. This is depicted in Figure 4.2. The first index list contains two document blocks, whereas the second index list contains only one document block. A list
separator marks the borders of these index lists, so that they don’t overlap. A document block is divided into element blocks, as shown in the figure. Each element block contains information about elements in one document. The first document block in the first index list in Figure 4.2 contains \( n \) element blocks. They are ordered by document id ascending.

![Figure 4.2: A high-level description of file structure including index lists, document and element blocks.](image)

The structure introduced in this work, allows both sorted accesses and random accesses to the index lists. By sorted accesses we mean fetching a document block from the index lists at a time sequentially. This document block contains information about many documents. The information about one document, in an index list, is written in one element block. To ensure that we don’t read partial information about any document, we arrange element blocks in document blocks carefully. If the remaining free space in a document block isn’t enough to hold the element block, the whole element block is written to the next document block and the free space
in the previous document block is filled with separators. Random accesses don’t fetch document blocks sequentially, but allow a flexible fetch of any document block as needed.

All index lists are pre-computed and stored in one binary file. We use list separators to mark their borders. A list separator is a constant character like ‘\n’. Figure 4.3 shows a binary file containing three index lists for tag-terms \( p=\text{Top-k} \), \( p=\text{efficient} \) and \( p=\text{method} \) respectively. List separators are shown in green. We use the combination of \textit{tag$term$} to distinguish each index list. This key is shown in blue in Figure 4.3. Each index list starts at its corresponding position in the binary file. The first index list \( (p$\text{Top-k}) \) starts at position/offset (0) in the binary, the second index list \( (p$\text{efficient}) \) starts at position /offset (460,000) in the binary file and the third index list \( (p$\text{method}) \) starts at position/offset (1,202,000) in the binary file. These positions or offsets are shown in red in the figure.

These offsets are necessary to issue sorted accesses to the index lists. We store these offsets in a \textit{keys$sa$} file at index time. Each entry in the \textit{keys$sa$} file is composed of the key for the index list and the offset where that index list starts in the binary file. Figure 4.4 shows a sample \textit{keys$sa$}
file for the index lists shown in Figure 4.3. Keys are shown in blue and offsets are shown in red. For example; the first index list \((p^\text{Top-k})\) starts at position (0) in the binary file and so on. To lookup the offset of a key from the \textit{keys\_sa} file, we use the intersect package explained in section 3.2.

![Example.keys\_sa](image)

Figure 4.4: An example \textit{Keys\_sa} file for the index lists shown in Figure 4.3.

For long index lists, spanning over more than one document block, we create a Histogram [11] that captures the distribution of scores in the index list. We sample scores in the index lists over discrete intervals or buckets and store their frequencies in the histogram. The histogram of an index list is pre-appended to that index list. This statistical information is useful to perform probabilistic pruning (section 5.7). For short index lists, containing only one document block, this statistical information is not useful because we have to scan that document block anyways; this entails a kind of overhead.

4.2.1 Document Blocks

An inverted block-index list is composed of one or more document blocks. Again, we use separators to mark their borders. This separator is a special constant character like ‘\(\texttt{\textbackslash t}’\). A document block is a container for element blocks that contain the actual information. The set of documents in the inverted list are first ordered by score in a descending manner. Then, they are partitioned into document blocks. Elements in the same document block are re-sorted on document id in an ascending manner, this allows us to use merge join to join new documents with old ones, please refer to
section 5.6 for more details. Figure 4.5 shows the upper part of the binary file shown in Figure 4.3. This part includes the index list for tag-term ($p_{Top-k}$). In this example, this index list is composed of two document blocks. The separator is shown in orange in Figure 4.5. Notice that the end of the second document block is the end of the list, and there we need a list separator (green). Each document block starts at a position or offset in the binary file, these are shown in red in the figure.

Document block size is a system parameter, see Table 6.1. For example, in the current settings we set block size to 256 KB. In Figure 4.5, the first document block starts at position 0 in the binary file and the second starts at position 256,000 in the binary file. The free space in a document block that is not enough to include the next element block is filled with separators to align document blocks. This space is usually small. If the last document block in an index list has a size less than a document block, we don’t need to fill in the remaining space with separators to align it, but we add a list separator to tell that this is the end of this list. For example, in Figure 4.5, the first document block has a size of 256 KB, whereas the last document block has a size of (460 - 256 = 204 KB). In this case, we don’t fill the remaining space (256-204=52 KB) with separators; instead we add a list separator.

![Figure 4.5: An index list containing two document blocks.](image)

When we scan an index list using sorted accesses, we maintain the current
position of the cursor in the index list, so that we read always the correct data. At the beginning this cursor has a value of the offset for that index list from the keys_sa file. This means that the next block to scan is the one at the corresponding position in the file. After we scan the first document block from the index list, the cursor is set to the offset where the second document block starts or to null if the list doesn’t contain more document blocks, where we close the index list because we find an index separator or the end-of-file.

4.2.2 Element Blocks - Content Information

A document block is further divided into element blocks where the actual information is stored. Document blocks are just containers for element blocks. Figure 4.6 shows this relationship between document and element blocks. The right-most top part is the index list for tag-term $p\$Top-k$, which is the same shown in Figure 4.5. If we take the first document block from this index list and investigate it, we find the element blocks that form the document block. An element block contains information about one document with respect to a tag-term. We use the combination of $docid$\$tag$\$term$ as a key to distinguish each element. Consider the document shown in the left-upper part of the same figure. This document contains 9 $p$ elements, among which 7 elements contain the term $Top-k$ among their contents. The scores of these $p$ elements are highlighted with red rectangles. Each $p$ element among these 7 elements has an entry in the element block that describes this document against the tag-term $p\$Top-k$. The first piece of information we find in the element block is the id of that document. In this example, it is 24. Then, we find the entries for each $p$ element in the document that satisfies the content condition ($Top-k$, in this case). An entry for such an element contains pre- and post- order of
that element and its score with respect to that term. For example, the first element has pre-order of 11, post-order of 9 and a score of 0.7 with respect to the Top-k term. Elements in the same element block are ordered on score in a descending manner. The element with the best score with respect to the term is the first element. We defined the score of a document with respect to a tag-term as the score of the element with the best score with respect to that term. According to the above arrangement, document’s score with respect to a tag-term is the score of the first element in its element block. In this example, document’s score is 0.7.

![Diagram of Document and Element blocks for content information.](image)

Our binary encoding scheme, explained in section 5.2, allows a flexible size for each entry in the element block as a kind of a simple compression technique. A document id has a maximum size of doc_bytes, which is a system parameter. If a document id could be encoded with less bytes, the remaining space is saved. The same is done with pre, post and score.
In the current setting, we set \textit{doc\_bytes} to 4 bytes, \textit{pre\_post\_bytes} to 3 and \textit{score\_bytes} to 2, see Table 6.1. When the value is encoded with less bytes, we add a separator byte to show that this value has a smaller length. According to this setting, the above element block has a maximum size of 60 bytes. Document id has a maximum length of \textit{doc\_bytes}, which equals 4. This element block contains also 7 \( p \) elements that satisfy the content condition \((p=\text{Topk})\). Each of these 7 elements has per, post and score values stored in the element block. Each per and post value has a maximum length of \textit{per\_post\_bytes} which equals 3. Each score has a maximum length of \textit{score\_bytes}, which equals 2. We’ve encoded this element block using the same values as in the above example, and we got an actual size of 45 bytes only.

\[
\text{Size} \leq 4 + 7 \times 3 + 7 \times 3 + 7 \times 2 \\
\leq 60 \text{ bytes}
\]

Actual size = 45 bytes

Each element block is associated with the document block which contains it. This association is done at index time and stored in the \textit{keys\_ra} file. In the \textit{keys\_ra} file, we store the key for each element block associated with the offset of the document block containing that element block in the binary file. Element blocks keys for content information is formed from document id \((\text{docid})\), tag and term in the form \((\text{docid}\$\text{tag}\$\text{term})\). What we store in the \textit{keys\_ra} file is the key of each element block and the offset of the document block containing that element block in the binary file. For example, the first document block taken from the first index list for tag-term \((p\$\text{Top-}k)\) has the structure as shown in Figure 4.7. The left part is the document block containing some element blocks. This document block starts at position 0 in the binary file and ends at position 256,000.

The \textit{key} for each element block is shown in blue underlined. Notice that
4.2. BLOCK-INDEX LIST STRUCTURE

Figure 4.7: Element Blocks and a Keys_ra file for content information.

element blocks in the same document block are re-ordered on document id’s in an ascending manner; this is useful to apply merge joins instead of hash joins, for example. The right part shows the structure of the keys_ra file. It is split into two parts, the first part contains the key (blue) of each element block and the second part contains the offset (red) of the document block containing that element block in the binary file. For example, the element block containing content information about document 24, with respect to tag-term pair ($p$Top-k), is contained in the first document block that starts at position 0 in the binary file. The offset value of this element block is hence 0. The same is done with other element blocks in all document blocks.

4.2.3 Element Blocks - Structural Information

We need also to store information about the structure of the documents in case our query has some structural constraints. Our example query shown in Figure 2.5 contains two structural constraints; these are the article and the sec constraints. So, for each document we store information about article and sec elements.
The case of structural information is a little bit different because structural constraints have static scores, and these need not be stored in the index list. Structural information are fetched only through RA, and since RA’s are expensive, they should be minimized. This is controlled using the Min-Probe principle [2].

The structure of an element block is similar to the one shown in Figure 4.6, but without entries for scores. An element block, for one or more structural elements having the same types in a document, includes the pre- and post-order of each such structural element only. Document id is also included in the element block to show to which document it belongs. When we issue a RA to a structural index list, we fetch the whole document block that contains the needed element block. This limits the number of needed RA’s to the index lists of structural information. All fetched documents are marked as read, so that they are not read again.

In the case of structural information, document blocks have keys of the
form \((tag)\) like \((sec)\) and element blocks have keys of the form \(docid$tag\) like \((24$sec)\). One document block with respect to a structural constraint, contains many element blocks, each of which describes the element(s) of that type in each document.

Figure 4.8 shows the index list for element sec and the structure of its first document block. The index list is shown in the right upper part. This index list has three document blocks separated by separator bytes. The first document block is investigated as shown in the left lower part of the figure. The document block is further divided into element blocks. Each element block describes elements with a sec tag in one document. The first element block describes sec elements in document 10. The second element block contains information about sec elements in document 24, which is shown in the left upper part of the figure. Document 24 has three sec elements. Its element block contains document id, then it’s followed by pre and post order of each sec element in the document, as shown in Figure 4.8. Notice that element block’s key is composed from document id and the element type \((tag)\) it describes. Again, element blocks in the same document block are re-ordered on document id ascending, to allow the use of merge join.

4.2.4 Histograms

A long index list, that contains more than one document block, includes a statistical measure in the form of a histogram. For short index lists, that contain only one document block, we don’t need such kind of statistics because these lists will be scanned definitely.

A histogram samples all scores in the index list over discrete intervals or buckets. The number of these buckets or intervals is a system parameter and could be configured as needed, see Table 6.1. In each bucket, we
calculate number of documents whose scores (as defined in section 2.4) fall in this interval. Figure 4.9 shows such a histogram taken from the index list for the tag-term pair (p$Top-k$). In this example, number of bucket or intervals is 10. The numbers shown in brown represent the score range for each interval. The first interval samples score fall in the interval [0.0, 0.1). In this index list, we have 78 documents whose score fall in this interval. The second interval samples documents whose scores fall in the interval [0.1, 0.2) which has 60 documents, and so on.

![Histogram Diagram](image)

Frequencies in the histogram are stored in the same way document id’s are encoded. This means that a frequency has a maximum size of `doc_bytes`. According to the previous settings, a histogram has a maximum size of 

\[
(buckets \times doc\_bytes) \text{ bytes}
\]

We need these histograms in the probabilistic pruning process, see section 5.7. In probabilistic pruning, we use these frequencies to estimate the probability that a candidate document could gather enough score to qualify as a top-k document.
4.3 Data Compression - Example

Suppose that our query is:

```
//section[intercontinental internationalization
telecommunications] In this query, we ask for section elements containing terms intercontinental, internationalization and telecommunications among their contents. Table 4.1 shows a piece of the data, taken from the second index list about one document. The data is stored in the database in the same way shown in the table. Because this is a relational database, each table has to follow a fixed schema. This entails a lot of redundancy. In our example, document (548218) has 6 section elements containing the term internationalization. The entry of each element contains the same DID, TAG, TERM and MAXSCORE redundantly.
```

Each field has a size in the database, this is shown in the last row of Table 4.1. DID has a size of 4 bytes in the database, TAG has a size of 7 bytes in the database etc. This piece has a size of 267 bytes in the database. The same data is stored according to the new structure in the binary file as shown in Table 4.2. This element block has a size of 50 bytes in the binary file. The total size of three index lists in the database is ≈ 133 KB; whereas the total size of the three index lists in the binary file including histograms is ≈ 34 KB. The size of each field is computed using Oracle’s VSIZE function. “VSIZE(expr) returns the number of bytes in the internal representation of expr” [1]. For other examples on data compression, please refer to sub-section 6.3.2 in the results part.

4.4 Discussion

Document blocks don’t contain information themselves; rather they are containers for element blocks which in turn contain the actual information.
### Table 4.1: A Sample data piece from the database about tag-term $\text{section}=\text{internationalization}$.

<table>
<thead>
<tr>
<th>DID</th>
<th>TAG</th>
<th>TERM</th>
<th>PRE</th>
<th>POST</th>
<th>SCORE</th>
<th>MAXSCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>260</td>
<td>380</td>
<td>0.544853</td>
<td>0.544853</td>
</tr>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>178</td>
<td>193</td>
<td>0.468877</td>
<td>0.544853</td>
</tr>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>66</td>
<td>149</td>
<td>0.458116</td>
<td>0.544853</td>
</tr>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>196</td>
<td>208</td>
<td>0.447342</td>
<td>0.544853</td>
</tr>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>152</td>
<td>170</td>
<td>0.432795</td>
<td>0.544853</td>
</tr>
<tr>
<td>548218</td>
<td>section</td>
<td>internationalization</td>
<td>23</td>
<td>63</td>
<td>0.359378</td>
<td>0.544853</td>
</tr>
</tbody>
</table>

| size | 4 | 7 | 20 | 2-3 | 2-3 | 4 | 4 |

Table 4.2: The same data in Table 4.1 according to the new structure.
This double-nested structure is also suitable for both sorted accesses and random accesses since we can either fetch a document block at a time sequentially or we can fetch a specific document block randomly.

Documents in the same document block are resorted on their document id ascending, and since we fetch the block and put it in the memory, it is possible to do in-memory merge joins that are efficient and cheaper than hash-joins. Moreover, we can apply merge joins without decoding docid’s because it is possible to decide which docid is less than or greater or equal to another docid using their in-memory encoded representations, see section 5.6. In the current implementation of the core engine, we need to decode document id’s for documents in the results only.

Since most operations could be carried out on the binary representations of items, our encoded structure could be viewed as a pre-computed memory image that we load as needed. This is a big gain because memory management is very important to make computations more efficient.

Content information is fetched mainly using SA. In order to fetch all document blocks of a file sequentially, it is enough to know the beginning of the index list; this information is stored in the keys_sa file during indexing process. Keys have the form tag$term and offsets point to the beginning of the index list in the binary file.

At later stages in the evaluation process, it would be better to switch to RA instead of SA to fetch more content information (according to the Ben-Probe principle). To read a document block using random accesses, we need to know where this document block starts in the binary file, this offset for each key of the form docid$tag$term is stored in the keys_ra file at index time also. The main difference here is that we don’t scan the index list sequentially, but, we jump back and forth to the appropriate offset in a random way as needed.
The same could be said about structural information where we use RA’s to fetch them as needed according to the *Min-Probe* principle. Elements’ keys have the form \((docid$tag)\) and offsets refer to the positions where their container document blocks start in the binary file. The keys and offsets are stored in a keys_ra file.

We need another package to fetch the offset for a document block for both content and structural information, we use the intersect package, section 3.2.
Chapter 5

Implementation Details

The input NEXI query is transformed into a special representation that could be handled by the system. This step is done manually in the current work because the NEXI parser is not yet implemented. More details about the internal query representation is explained in section 5.1.

The binary representation of index entries is explained in section 5.2 where we show our encoding algorithms for both integer and double values. Also, we show the decoding algorithms used to extract the original values from their binary representations.

More details about performing content SA’s, and content and structural RA’s are shown in sections 5.3, 5.4. Section 5.5 shows the heuristics that we use to control both content and structural RA’s.

In section 5.6, we show more implementation details about applying merge joins on the binary representations of document id’s. Probabilistic pruning is shown in section 5.7.

5.1 Internal Query Representation

We transform the input NEXI query into an internal representation that the system could handle. At the beginning, we model the query as a tree of nodes (elements) and edges (relationships among elements). Please, refer
to our introductory example in section 2.2. In the final step, we model the query as a DAG. Figure 5.1 shows the DAG model of the NEXI query:

```
//article[about (//author, John)]
//sec[about(//p, Top-k efficient method)]
```

![Query DAG for the query: (//article[about (//author, John)]//sec[about(//p, Top-k efficient method)])](image)

The main attributes that characterize the query in our system are shown in Table 5.1. The internal representation of the query shown in Figure 5.1 is shown in Table 5.2.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality</td>
<td>Number of content and structural conditions in the query.</td>
</tr>
<tr>
<td>Tag_terms</td>
<td>Tag and tag-term conditions in the query.</td>
</tr>
<tr>
<td>Targets</td>
<td>Target elements in the query.</td>
</tr>
<tr>
<td>Selfs</td>
<td>Self constraints that connects elements with the same type.</td>
</tr>
<tr>
<td>Descendants</td>
<td>Descendant nodes of each node in the query.</td>
</tr>
<tr>
<td>Parents</td>
<td>The parent of each node in the query.</td>
</tr>
</tbody>
</table>

Table 5.1: The main attributes that characterize the query in our system.

This example query has a dimensionality of 6 which means 6 conditions; both content and structural conditions. The first condition is a structural constraint, which is the “article” structural constraint. The second condition is a content or tag-term condition, which is author=John. Both types of conditions are included in the tag-terms attribute. In this query, target elements are “p$Top-k”, “p$efficient”, “p$method”. These target
elements are contained in the query in the targets attribute where their numbers in the DAG are used to refer to them. Each node is connected to itself though the self constraint. Also, the self constraints are used to connect content conditions that should occur in the same element, like the three target conditions in our example query. This is necessary to aggregate the best score of a document. The descendant and parent nodes are also included for each node in the query. A leaf node doesn’t have any descendant nodes. The root node doesn’t have a parent (reflected in the special value -1).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality</td>
<td>6</td>
</tr>
<tr>
<td>Tag_terms</td>
<td>{“article”, “authorJohn”, “sec”, “pTop - k”, “pefficient”, “pmethod”}</td>
</tr>
<tr>
<td>Targets</td>
<td>{3, 4, 5}</td>
</tr>
<tr>
<td>Selfs</td>
<td>{{0}, {1}, {2}, {3, 4, 5}, {3, 4, 5}, {3, 4, 5}}</td>
</tr>
<tr>
<td>Descendants</td>
<td>{{1, 2}, {}, {3, 4, 5}, {}, {}, {}}</td>
</tr>
<tr>
<td>Parents</td>
<td>{-1, 0, 0, 2, 2, 2}</td>
</tr>
</tbody>
</table>

Table 5.2: The main attributes that characterize our example query: (//article[about (//author, John)]//sec[about([//p, Top-k efficient method])]).

5.2 Encoding Scheme

Content and Structural information are encoded in the binary file according to our encoding scheme. All values that we need to encode are numbers; both integers and doubles. These include document ids, pre, post and scores of elements. Each document id is represented using a character array. Elements’ pre, post and score entries are also represented by character arrays. By fetching a document from the binary file, we update the pointers to arrays in its representation. The system could handle these encoded values to perform most operations, such as joining documents. We don’t need to decode all of these values. The only value that we need to decode
is the score of each element. Still, this is not a problem because this is done only once, whereas other frequent operations, such as join operation and path matching in CAS, are performed on the encoded values, see Algorithm 6.

Integer numbers like pre, post and document id's are encoded according to Algorithm 2. Scores, which are double values, are encoded according to Algorithm 3. This encoding scheme allows us also to retrieve the original numbers, in case they are needed. Algorithm 4 shows how to extract integers from their binary encoded representations. This is be needed at the end of the evaluation process to display the results to the user. On the other hand, scores need to be decoded because we need the concrete values to aggregate scores. Algorithm 5 shows how to extract the original double values from their binary representations.

#### Algorithm 2 Encode Integers(number, byteLength)

1. globalOffset=48
2. base=78
3. chars=char[byteLength]
4. r=number
5. i=0
6. while r > 0 and i < byteLength do
7.   chars[i] = globalOffset + (r % base)
8.   r = r/base
9.   i=i+1
10. end while
11. if i < byteLength then
12.   chars[i]=SEPARATOR
13. end if
14. return chars

Table 5.3 shows the element block shown in Table 4.2. The left part shows the actual information in integers and doubles. The right part shows the encoded representation of this element block according to our encoding scheme. The red dot (●) in the binary representation shows the separator byte. The first 4 bytes in the binary representation (R8<1) encodes document id (548218). Next two bytes (J3) encode the first pre value (260);
5.2. ENCODING SCHEME

Algorithm 3 Encode Doubles \((number, byteLength)\)

1: \(\text{globalOffset}=48\)
2: \(\text{base}=78\)
3: \(\text{chars}=[\text{byteLength}]\)
4: \(r=number\)
5: \(i=0\)
6: \(\text{while } r > 0 \text{ and } i < byteLength \text{ do}\)
7:     \(\text{chars}[i] = \text{globalOffset} + (r \times \text{base})\)
8:     \(r = (r \times \text{base}) - (\text{int})(r \times \text{base})\)
9:     \(i=i+1\)
10: \(\text{end while}\)
11: \(\text{if } i < byteLength \text{ then}\)
12:     \(\text{chars}[i]=\text{SEPARATOR}\)
13: \(\text{end if}\)
14: \(\text{return chars}\)

Algorithm 4 Decode Integers \((bytes, byteLength)\)

1: \(\text{globalOffset}=48\)
2: \(\text{base}=78\)
3: \(\text{value}=0\)
4: \(p=1\)
5: \(i=0\)
6: \(\text{while } i < byteLength \text{ and } bytes[i] \neq \text{SEPARATOR} \text{ and } bytes[i] \neq \text{LIST_SEPARATOR} \text{ do}\)
7:     \(\text{value} = \text{value} + (((\text{int})bytes[i] - \text{globalOffset}) \times p)\)
8:     \(p = p \times \text{base}\)
9:     \(i=i+1\)
10: \(\text{end while}\)
11: \(\text{return value}\)

Algorithm 5 Decode Doubles \((bytes, byteLength)\)

1: \(\text{globalOffset}=48\)
2: \(\text{base}=78\)
3: \(\text{score}=0\)
4: \(p=1\)
5: \(i=0\)
6: \(\text{while } i < byteLength \text{ and } bytes[i] \neq \text{SEPARATOR} \text{ and } bytes[i] \neq \text{LIST_SEPARATOR} \text{ do}\)
7:     \(\text{score} = \text{score} + (\text{double}) (((\text{int})bytes[i] - \text{globalOffset}) / p)\)
8:     \(p = p \times \text{base}\)
9:     \(i=i+1\)
10: \(\text{end while}\)
11: \(\text{return score}\)
we add the separator byte to show that this value has less length (its maximum length is 3). Next post value (380) is encoded with (t4). For the same reason, the separator is added. (ZV) encodes the first score(0.54485). Next two bytes (F2) encode pre of the second element (178). This value is encoded with 2 bytes only; a separator is added and one byte is saved.

<table>
<thead>
<tr>
<th>548218</th>
<th>260</th>
<th>380</th>
<th>0.544853</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>193</td>
<td></td>
<td>0.468877</td>
</tr>
<tr>
<td>66</td>
<td>149</td>
<td></td>
<td>0.458116</td>
</tr>
<tr>
<td>96</td>
<td>208</td>
<td></td>
<td>0.447342</td>
</tr>
<tr>
<td>152</td>
<td>170</td>
<td></td>
<td>0.432795</td>
</tr>
<tr>
<td>23</td>
<td>63</td>
<td></td>
<td>0.359378</td>
</tr>
</tbody>
</table>

Table 5.3: The same data in Table 4.1 according to the new structure and its binary representation according to our encoding scheme.

5.3 Content SA

Index lists for content information are fetched mainly using sorted accesses because they are cheap. By sorted accesses we mean fetching document blocks from the index lists sequentially. We start by fetching the first document block, then the second ... etc. Because documents in the index list are sorted by score in a descending manner, we fetch documents with high scores first, then documents with smaller scores and so on. For each index list, we keep the current location of the cursor to show the current scan position in the index list. At the beginning, we let the cursor point to the starting position of the index list in the binary file. This position or offset is stored in a keys_sa file at index time as described in sub-sections 4.2.2, 4.2.3.
5.4 Content and Structural RA

Random accesses are used to fetch some document blocks from content index lists to resolve uncertainty about candidate items. The case of expensive predicates, such as structural constraints, is handled using random accesses also. By random accesses we mean fetching document blocks from the index lists as needed. We jump back and forth to reach the necessary document block; we don’t scan them sequentially.

In order to fetch a document block using random accesses, we need to know the position where the document block starts in the binary file. After we get the appropriate offset from the keys_ra file using the Intersect package, we move to that position in the binary file, fetch that document block and merge the new documents with the already-read documents.

Consider Figure 5.2. The right-most part (DB p$Top-k) shows a piece of a document block from the index list for tag-term (p$Top-k). The figure shows only 4 element blocks (EB) from this document block for documents 2,5,9 and 12. The key for each element block is shown in bold. The document block starts at position 60 in the binary file. This position/offset is shown in red to the right of the document block. These offsets are stored in the keys_ra file at index time as shown in the left part (contents.keys_ra) of the same figure. When we need to fetch a document block using RA, we go to the keys_ra file, fetch the appropriate offset using the Intersect package, navigate to that position in the binary file and fetch that document block.

Suppose that we want to fetch the document block containing the element block for document 9. We go to the contents.key_ra in the example figure, find the offset (using the Intersect package) for the document block containing the element block for document 9 with respect to tag-term
(\textit{p$Top-k$}), which is 60, navigate to position 60 in the binary file and fetch that document block. We make sure that each document block is fetched only once by marking all documents we read. In the previous example, when we fetched document 9, we mark documents 2,5,12 ... as read for this dimension. This is necessary to avoid overhead because this way, we get many new documents merged "for free".

The case of structural constraints is more important because it’s the only way to fetch structural information. When we need to check a document for a structural constraint, we fetch the document block from the index list for that element from the binary file. We need first to know the offset in the binary file where that document block starts. This information we find in the \textit{keys_ra} file. After we know the necessary offset using the \textit{Intersect} package, we navigate to that position in the binary file, fetch that document block and merge the new documents with the ones we already have. Notice that we mark all documents that we fetch from the document block as read with respect to that dimension.

Consider Figure 5.3. The left part (\textit{DB sec}) shows a piece of a document block from the index list for element (\textit{sec}). The figure shows element blocks for documents 5,9 and 12. This document block starts at position 240 in
the binary file. This offset is stored in the keys_ra file as shown in the right part of the figure. This part of the keys_ra file shows offsets for the document block containing element blocks in the tail of the index list the article element (first two entries) and offsets for the document block containing element blocks in the head of the index list for sec elements (last two entries). If we want to check document 9 for the structural constraint sec, we go to the keys_ra file, find the offset for the document block containing that element block, 240 in this case, navigate to position 240 in the binary file, read that document block and merge new documents with the documents we already have. We mark documents 5 and 12 as read also with respect to this dimension.

![Figure 5.3: Example: Fetching a structural document block using RA.](image)

### 5.5 Structural Constraints and Cost Model for Random Accesses

Random accesses are expensive whereas sorted accesses are cheaper. Nevertheless, random accesses are necessary to resolve expensive predicates such as structural constraints. They are also important to resolve uncertainty about candidate items which have some missing dimensions.

Structural information is fetched through random accesses to the binary file; this means that they are expensive and should be minimized. Structural
constraints are tested by a minimal probing scheduler, for more details, please refer to [2]. Again the intersect package is used to lookup keys from the key ra file to get the appropriate offset value for the appropriate document block. In this case, the key is composed from docid and tag (docid$tag).

The uncertainty about some candidates is handled using the Beneficial Probing principle [16], where we assess the benefit of making random accesses vs. continuing the usual sorted accesses to the content index lists, using a cost model. The better of the two is chosen. Th idea here is to postpone RA’s to content index list as much as possible. In this principle, we count the number of issued sorted accesses and the number of remaining necessary random accesses. If the cost of issuing random accesses is less than the cost of continuing usual sorted accesses, we stop sorted accesses and switch to random accesses. We measure the cost of random accesses ($C_r$) and sorted accesses ($C_s$) using a cost ratio $\frac{C_r}{C_s}$ which shows the cost of random accesses with respect to sorted accesses. This cost ratio is a system parameter that could be tuned according to the case under consideration.

5.6 Merge Joins

All read documents are kept in the memory in an array whose size equals DOCUMENT BUFFER, see Table 6.1. They are arranged in an ordered linked-list using document id as a key. That’s why element blocks in each document block are ordered by document id ascending. This allows us to use merge joins instead of hash-joins used in TopX V 1.0 (section 3.1).

The key point here is that we don’t need to decode document id’s to determine their order; instead we can do this using their in-memory binary representation. This is a big advantage of our binary encoding scheme.
We mentioned that we allow a flexible size for each document id. If two document id’s have different lengths, then the shorter is smaller and the longer is greater. When both id’s have the same length then we compare their representations byte by byte. If each byte in the first document id matches its corresponding byte in the other document id, then they are equal. Otherwise, the document id, having the first byte which is less than its corresponding byte in the other document id, is smaller than the other.

This is described in Algorithm 6.

Algorithm 6 Compare Fast \((\text{bytes1, bytes2, maxLength})\)

1: \(p = \text{maxLength}\)
2: \textbf{for} \(i = 0\) to \(\text{maxLength}\) \textbf{do}
3: \quad \text{if} \ \text{bytes1}[i] == \text{SEPARATOR} \text{ and } \text{bytes2}[i] \neq \text{SEPARATOR} \text{ then}
4: \quad \quad \text{return} -2
5: \quad \text{else if} \ \text{bytes2}[i] == \text{SEPARATOR} \text{ and } \text{bytes1}[i] \neq \text{SEPARATOR} \text{ then}
6: \quad \quad \text{return} 2
7: \quad \text{else if} \ \text{bytes2}[i] == \text{SEPARATOR} \text{ and } \text{bytes1}[i] == \text{SEPARATOR} \text{ then}
8: \quad \quad \text{p} = i
9: \quad \quad \text{break}
10: \quad \textbf{end if}
11: \textbf{end for}
12: \textbf{for} \(i = 0\) to \(p\) \textbf{do}
13: \quad \text{if} \ \text{bytes1}[p - 1 - i] < \text{bytes2}[p - 1 - i] \text{ then}
14: \quad \quad \text{return} -1
15: \quad \text{else if} \ \text{bytes1}[p - 1 - i] > \text{bytes2}[p - 1 - i] \text{ then}
16: \quad \quad \text{return} 1
17: \quad \textbf{end if}
18: \textbf{end for}
19: \text{return} 0

Theorem 1 (Correctness of the encoding scheme) Let \(a, b\) be two integers \(\in [0, \infty)\) or two doubles \(\in [0, 1]\), then:
\[
\text{code}(a) \circ \text{code}(b) \leftrightarrow a \circ b
\]
where \(\circ \in \{<, \leq, =, >, \geq\}\) and \(\text{code}(x)\) is the binary encoding of \(x\).

Proof: We show the proof of one case only; the remaining cases are similar.

Let \(a, b\) be two integers \(\in [0, \infty)\) and \(\circ =\), then:
\[
\text{code}(a) = \text{code}(b) \leftrightarrow a = b.
\]
• →

\[ \text{code}(a) = \text{code}(b) \rightarrow a = b \]

If \( \text{code}(a) = \text{code}(b) \), then

\[ |\text{code}(a)| = |\text{code}(b)| = l \quad \text{and} \]

\[ \text{code}(a)[i] = \text{code}(b)[i] \quad \forall i \in [0, l) \]

For \( i = 0 : \)

\[ \text{globalOffset} + a\%\text{base} = \text{globalOffset} + b\%\text{base} \]

\[ a\%\text{base} = b + \%\text{base} \quad (1) \]

\[ a/\text{base} = b/\text{base} = c, \quad \text{where } c \text{ is an integer} \]

\[ a = c \times \text{base} + a\%\text{base} \]

\[ b = c \times \text{base} + b\%\text{base} \]

but, \( a\%\text{base} = b\%\text{base} \) from (1)

\[ \rightarrow a = b \]

• ←:

\[ a = b \rightarrow \text{code}(a) = \text{code}(b) \]

If \( a = b = 0 \), then \( \text{code}(a) = \text{code}(b) = \text{separator} \)

Otherwise:

let \( l \) be number of iterations needed to encode each number.

\[ \text{code}(a)[i] = \text{globalOffset} + (a/\text{base}^i)\%\text{base}, \quad \text{where} \quad i \in [0, l) \]

(According to Algorithm 2)

\[ \text{code}(b)[i] = \text{globalOffset} + (b/\text{base}^i)\%\text{base}, \quad \text{where} \quad i \in [0, l) \]

(According to Algorithm 2)

This means that both codes have the same length and each byte in the binary representation of each number equals its counterpart byte in the byte representation of the other number.
According to the compare \textit{fast} algorithm, this is the definition of equality.
\[ \therefore \text{code}(a) = \text{code}(b). \]

Algorithm 6 is used also to determine orders between elements using their pre and post values. Again, these orders are tested using their in-memory binary representations; no need to decode them. We need to determine these orders among elements in order to test self-constraints and descendant structural constraints. These tests are used so often; so this is a great gain also. To test the self constraint between two elements \((u, v)\), both elements must have the same pre values. To test the descendant constraint, we use Equation 5.1. For more details, please refer to [7]

\[ u \text{ is a descendant of } v \iff \text{pre}(v) < \text{pre}(u) \land \text{post}(u) < \text{post}(v) \]  
(5.1)

5.7 Probabilistic Pruning

Usually the user is not strict about the top-k documents; rather she wants her question to get answered. In our introductory example about \textit{Saarland University}, it could happen that the user forgot the web site of Saarland University, so she'll be satisfied by the first reported result.

We could make use of this observation to speed-up the whole evaluation process for the penalty of getting less accurate results. The degree of accuracy is controlled by a system parameter called \(\epsilon\). If \(\epsilon\) has a low value, the results are better and the processing time is higher, and vice versa.

Here comes in the Histograms that are precomputed at index time and pre-appended to long index lists only. Short index lists don’t need such
histograms because we have to scan them anyways. This heuristic is useful to reduce number of histograms. In the considered data set, this saves 99% of all histograms.

When we get more information about documents from the index list, either using SA or RA, we update the candidate queue where we keep all candidate documents that could qualify as top-k documents because they have some missing dimensions. If a candidate document could not qualify as a top-k, it’s removed and not considered anymore. In the case of probabilistic pruning, the document is removed and not considered if the probability that the document could qualify as top-k is small enough; namely less that $\epsilon$. This probability reflects the probability that the document ($d$) could gather a score amount greater than $\delta$ (where $\delta = min - k - ws(d)$), please see Figure 4.1. To estimate this probability, we use the same approach in [16].
Chapter 6

Experiments and Results

This chapter shows our experimental results. We mainly compare our new approach with the previous version of TopX v 1.0 (section 3.1), which is java-based and uses Oracle as a back end. In the following sections, we describe the data collection that we use to test our approach. Then, we show experimental settings. After that we evaluate the results and discuss them. Then, we show the effect of changing some system parameters; namely $\frac{C_r}{C_s}$ and $\epsilon$ for probabilistic pruning.

6.1 Data Collection

We tested out approach against the INEX Wikipedia collection [4]. Our data set is a collection of English Wikipedia XML documents. It has more than 600,000 articles, has a size of more than 4.6 GB (raw data), and has more than 30 million elements. On average, an article contains 161.35 XML nodes [10].

6.2 Experiments Setup

This approach was implemented in C++ using Microsoft Visual Studio 2005. System parameters are shown in Table 6.1.
All experiments were carried out under Microsoft Windows XP. The machine is Intel Pentium 4. It has a CPU of 3 GHz and 1 GB RAM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCUMENT BUFFER</td>
<td>1000000</td>
<td>Reserved space for documents to hold document information.</td>
</tr>
<tr>
<td>ELEMENT BUFFER</td>
<td>50000000</td>
<td>Reserved space for elements to hold Element Blocks.</td>
</tr>
<tr>
<td>histo buckets</td>
<td>12</td>
<td>Number of buckets (intervals) in the Histogram.</td>
</tr>
<tr>
<td>doc-bytes</td>
<td>4</td>
<td>Maximum allowed length for document id in the binary encoding.</td>
</tr>
<tr>
<td>pre-post-bytes</td>
<td>3</td>
<td>Maximum allowed length for pre and post in the binary encoding.</td>
</tr>
<tr>
<td>score-bytes</td>
<td>2</td>
<td>Maximum allowed length for score in the binary encoding.</td>
</tr>
<tr>
<td>base</td>
<td>78</td>
<td>Encoding base.</td>
</tr>
<tr>
<td>globalOffset</td>
<td>48</td>
<td>Encoding offset.</td>
</tr>
<tr>
<td>block-size</td>
<td>256 KB</td>
<td>Document block size in the index list.</td>
</tr>
<tr>
<td>separator</td>
<td>‘\t’</td>
<td>Separator character.</td>
</tr>
<tr>
<td>list-separator</td>
<td>‘\n’</td>
<td>List separator character.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>16 MB</td>
<td>Size of the database buffer (Bulkloader) used by the Intersect package.</td>
</tr>
<tr>
<td>C</td>
<td>1.0</td>
<td>Structural score.</td>
</tr>
<tr>
<td>CR-CS</td>
<td>1400</td>
<td>RA vs. SA cost ratio (\frac{C_r}{C_s}).</td>
</tr>
<tr>
<td>epsilon</td>
<td>0.0</td>
<td>Probabilistic pruning Threshold.</td>
</tr>
</tbody>
</table>

Table 6.1: System parameters we used in our experiments.

6.3 Results Evaluation

Table 6.2 and Table 6.3 show some INEX CO and CAS topics 2007, respectively. The first column shows query’s number. The second column shows the dimensionality of the query. The query itself (in NEXI) is shown in the third column. The fourth column shows time needed by the first version of TopX, v 1.0, to execute the query (on the same machine). The fifth column shows time needed by the new approach to execute the same query. Time is shown in milliseconds.
### 6.3. RESULTS EVALUATION

<table>
<thead>
<tr>
<th>#</th>
<th>DIM</th>
<th>Query</th>
<th>V 1.0 (ms)</th>
<th>V 2.0 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>/*[about(., hip hop beat)]</td>
<td>1,250</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>/*[about(., space history astronaut cosmonaut engineer)]</td>
<td>4,469</td>
<td>156</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>/*[about(., therapeutic breathing)]</td>
<td>297</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>/*[about(., french president fifth republic)]</td>
<td>7,969</td>
<td>344</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>/*[about(., difference American British English)]</td>
<td>20,609</td>
<td>688</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>/*[about(., pacific sea navigators australia explorers)]</td>
<td>8,578</td>
<td>188</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>/*[about(., operating system page replacement policy)]</td>
<td>11,093</td>
<td>281</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>/*[about(., died killed flight plane airplane accident crash)]</td>
<td>6,594</td>
<td>422</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>/*[about(., April 19th revolution peaceful revolution velvet revolution quiet revolution)]</td>
<td>10,063</td>
<td>156</td>
</tr>
</tbody>
</table>

**Table 6.2:** CO Test Cases: Fourth column shows time (in milliseconds) required by v 1.0 to execute the query. Fifth column shows time required by the new version (v 2.0) to execute the query.

<table>
<thead>
<tr>
<th>#</th>
<th>DIM</th>
<th>Query</th>
<th>V 1.0 (ms)</th>
<th>V 2.0 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>//article[about(., Neil Gaiman novels)]//section[about(., plot details)]</td>
<td>1,047</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>//article[about(., Ajax Asynchronous JavaScript and XML programming technologies applications)]</td>
<td>610</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>//article[about(., healthy diet)]//section[about(., diet features)]</td>
<td>907</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>//section[about(., operating system)]//p[about(., page replacement policy)]</td>
<td>2,281</td>
<td>109</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>//section[about(.,/p, motor car)]</td>
<td>812</td>
<td>63</td>
</tr>
</tbody>
</table>

**Table 6.3:** CAS Test Cases: Fourth column shows time (in milliseconds) required by v 1.0 to execute the query. Fifth column shows time required by the new version (v 2.0) to execute the query.
6.3.1 Performance Evaluation

Table 6.4 shows the total number of document blocks in all index lists created for the previous Content-Only queries. The first column refers to query’s number in Table 6.2. The second column shows number of document blocks in all index lists created for the query.

The size of all index lists (both content and structural) created for each of the previous Content-And-Structure queries is shown in Table 6.5. The first column refers to query’s number in Table 6.3. The second column shows number of document blocks in all index lists created for the query.
6.3.2 Storage Evaluation

Table 6.6 shows storage requirements by Oracle and the new structure to store the relevant *content* information. The first column shows the number of the query (in Table 6.2). The second column shows the total size required by Oracle to store the content index for the query. The third column shows the size required by our new approach to store the same data according to the new structure in the binary file.

<table>
<thead>
<tr>
<th>Query #</th>
<th>Oracle Size (MB)</th>
<th>File Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.205</td>
<td>0.830</td>
</tr>
<tr>
<td>2</td>
<td>19.728</td>
<td>6.489</td>
</tr>
<tr>
<td>3</td>
<td>0.341</td>
<td>0.101</td>
</tr>
<tr>
<td>4</td>
<td>17.943</td>
<td>5.697</td>
</tr>
<tr>
<td>5</td>
<td>35.753</td>
<td>11.273</td>
</tr>
<tr>
<td>6</td>
<td>15.290</td>
<td>4.997</td>
</tr>
<tr>
<td>7</td>
<td>16.839</td>
<td>5.758</td>
</tr>
<tr>
<td>8</td>
<td>11.985</td>
<td>4.299</td>
</tr>
<tr>
<td>9</td>
<td>14.560</td>
<td>4.930</td>
</tr>
</tbody>
</table>

Table 6.6: Storage requirements by Oracle and our new approach for CO queries.

Storage requirements by Oracle and by our new structure to store indexes (content and structure) for the previous Content-And-Structure queries are shown in Table 6.7. The first column shows the number of the query. The second column shows the total size required by Oracle to store the structure index for the query. The third column shows the size required by our new approach to store the same data according to the new structure.

<table>
<thead>
<tr>
<th>Query #</th>
<th>Oracle Size (MB)</th>
<th>File Size (MB)</th>
</tr>
</thead>
</table>

6.4 Discussion

Performance experiments showed a speedup gain of a factor 14-64 in CO queries and a factor of 13-60 in CAS queries. This depends on the query
CHAPTER 6. EXPERIMENTS AND RESULTS

<table>
<thead>
<tr>
<th>Query #</th>
<th>Oracle Size (MB)</th>
<th>File Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.104</td>
<td>0.427</td>
</tr>
<tr>
<td>2</td>
<td>0.783</td>
<td>0.266</td>
</tr>
<tr>
<td>3</td>
<td>0.989</td>
<td>0.349</td>
</tr>
<tr>
<td>4</td>
<td>2.970</td>
<td>1.084</td>
</tr>
<tr>
<td>5</td>
<td>2.340</td>
<td>0.739</td>
</tr>
</tbody>
</table>

Table 6.7: Storage requirements by Oracle and our new approach for CAS queries.

and the index lists. Usually, the more structural constraints the query has, the more time it needs to get executed. The length of the index lists also is an important factor, but is not the main factor in content conditions. Lengths of both content and structural index lists are shown in Table 6.6 and Table 6.7.

The first CO query, whose dimensionality is 3, asks for any element containing terms *hip hop beat*. Version 1.0 of TopX required about 1,250 ms to execute this query. According to the new structure and the new implementation, we can execute this query in 78 ms only. This query doesn’t have structural constraints and hence it is mainly evaluated using sorted accesses which are cheap.

The second CO query has a dimensionality of 5. We can evaluate this query faster than version 1.0 with a factor of $\sim 28$. The index of this query has 28 document blocks.

The third CO query, whose dimensionality is 2, has short index lists; each index list has only one document block. This explains the needed amount of time to execute this query; about 0 ms. Both document blocks are fetched using SA, that are cheap, and merge join is also cheap and it’s mainly carried out using pointer operations that are efficient in C++.

CO query number five has the longest index lists; it has a total of 46 document blocks. This explains the needed time by version 1.0 to execute
6.4. DISCUSSION

This query. It required more than 20 seconds. Our new approach can execute this query in less than one second. It needs only to fetch about 39 document blocks to execute the query.

CAS queries have -relatively- short index lists and this explains why they require less time to get executed, although they have structural constraints that are expensive.

Our binary encoding scheme and the new structure compresses the information in the binary file. Table 6.6 shows the storage requirements by Oracle and the new approach to store content index list for the previous CO queries. The storage requirements by Oracle and the new approach to store the structure index lists for CAS queries are shown in Table 6.7.

Content index is stored in the Oracle database (table FeaturesTagView) in this format (DID, TAG, TERM, PRE, POST, SCORE, MAXSCORE). For each tag-term, we store document id, the tag, the term and the maximum score. This is a kind of redundancy. According to the structure shown in section 4.2.2, we achieve a gain of 2.7-3.4 in storage requirements. The worst case shown in Table 6.6 is the case of query 1. In this query, we have short index lists and short terms (hip, hop, beat), still, we achieve a gain of factor $\sim 2.7$.

The case of structural information is similar. The structural index in Oracle (table ElementsTagView) follow this schema (DID, TAG, PRE, POST, CPTS, MAXCPTS). Again, document id, tag, CPTS and maximum CPTS are redundant. The new structure of structural information in the binary file (section 4.2.3), doesn’t include redundancy. In Table 6.7, the new approach achieves a gain of a factor of 2.6-3.1.

The size of the Oracle index is computed using Oracle’s VSIZE function. “VSIZE(expr) returns the number of bytes in the internal representation of expr” [1]. For example, to compute the size of the content index for
the 5th CAS query, we use this query:

\[
\text{SELECT sum (vsize(tag)+ vsize(term)+ vsize(did) +vsize(pre) +vsize(post) + vsize(localscore) + vsize(maxscore)) AS index\_size}
\]

\[
\text{FROM FeaturesTagView}
\]

\[
\text{WHERE (tag='p' and (term ='motor' or term='car'))}
\]

And the query used to compute the size of the structure index for the same query is:

\[
\text{SELECT sum(vsize(did)+ vsize(tag)+ vsize(pre)+ vsize(post)+ vsize(cpts)+ vsize(maxcpts)) AS index2\_size}
\]

\[
\text{FROM ElementsTagView e}
\]

\[
\text{WHERE e.tag='section' and e.DID IN (SELECT DISTINCT did}
\]

\[
\text{FROM FeaturesTagView f}
\]

\[
\text{WHERE (f.tag='p' and (f.term =‘motor’ or f.term=’car’))}
\]

6.5 Probabilistic Pruning - Results

Probabilistic pruning helps produce the results faster, but for the penalty of getting less accurate results. Table 6.5 shows the results of executing the fifth CO query with and without probabilistic pruning. The left part of the table shows the results of executing the query without probabilistic pruning, i.e. \(\epsilon = 0\%\). The right part of the table shows the results of executing the same query with \(\epsilon = 1\%\).
The value of $\epsilon$ should range from 0% to 10%. The higher $\epsilon$ is, the less accurate the results are and the faster the evaluation is. In this example, we’re able to execute the query in about 281 ms; with a gain of a factor of about 2 with respect to the one executed without probabilistic pruning. The accuracy is not bad and could be enough for some applications or some cases. Two documents that are ranked as top-10 in the case of $\epsilon = 1\%$ are also ranked as top-10 in the case of exact results. These are shown in red (630149, 5448). The remaining documents that we rank as top-10 in the probabilistic pruning case are not top-10 in the exact case, but they have good scores. The 10th document in the exact results has a score of (0.690828) and the 10th document in the probabilistic pruning case has a score of (0.618836).

<table>
<thead>
<tr>
<th>Rank</th>
<th>DOCID</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>630149</td>
<td>0.731098</td>
</tr>
<tr>
<td>2</td>
<td>617161</td>
<td>0.724852</td>
</tr>
<tr>
<td>3</td>
<td>654209</td>
<td>0.717784</td>
</tr>
<tr>
<td>4</td>
<td>5448</td>
<td>0.707758</td>
</tr>
<tr>
<td>5</td>
<td>254219</td>
<td>0.702991</td>
</tr>
<tr>
<td>6</td>
<td>254216</td>
<td>0.702991</td>
</tr>
<tr>
<td>7</td>
<td>451315</td>
<td>0.701183</td>
</tr>
<tr>
<td>8</td>
<td>20500</td>
<td>0.695759</td>
</tr>
<tr>
<td>9</td>
<td>654210</td>
<td>0.690828</td>
</tr>
<tr>
<td>10</td>
<td>654211</td>
<td>0.690828</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>DOCID</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>630149</td>
<td>0.731098</td>
</tr>
<tr>
<td>2</td>
<td>5448</td>
<td>0.707758</td>
</tr>
<tr>
<td>3</td>
<td>315475</td>
<td>0.676693</td>
</tr>
<tr>
<td>4</td>
<td>174175</td>
<td>0.671433</td>
</tr>
<tr>
<td>5</td>
<td>18041</td>
<td>0.651216</td>
</tr>
<tr>
<td>6</td>
<td>24439</td>
<td>0.632314</td>
</tr>
<tr>
<td>7</td>
<td>656499</td>
<td>0.632314</td>
</tr>
<tr>
<td>8</td>
<td>305463</td>
<td>0.621631</td>
</tr>
<tr>
<td>9</td>
<td>4783</td>
<td>0.62048</td>
</tr>
<tr>
<td>10</td>
<td>434897</td>
<td>0.618836</td>
</tr>
</tbody>
</table>

Table 6.8: Probabilistic Pruning Experiment:
6.6 Varied $\frac{C_r}{C_s}$ values - Results

Content information is fetched mainly through SA’s. RA’s to the content information index are also possible, and controlled by ($\frac{C_r}{C_s}$) which is a system parameter. The value of $\frac{C_r}{C_s}$ shows how cheap RA (to content information) are with respect to SA. If $\frac{C_r}{C_s}$ is small, then the number of RA’s is large. If $\frac{C_r}{C_s}$ is large, then RA’s are expensive and so there are fewer RA’s. Table 6.9 shows the CO queries’ experiments, and Table 6.10 shows CAS queries’ experiments.

The first CO query is processed using SA only because it has short index lists. This explains the same amount of time it needs to get executed regardless of the values of $\frac{C_r}{C_s}$, see Table 6.9. The second query, on the other hand, has longer index lists, and the value of $\frac{C_r}{C_s}$ matters. In the first case, where $\frac{C_r}{C_s} = 200$, many RA’s are issued and this is reflected in the larger amount of time it requires to get executed. In the second case, where $\frac{C_r}{C_s} = 800$, number of RA’s is decreased and hence the performance is better. The other two cases, $\frac{C_r}{C_s} = 1400$ and $\frac{C_r}{C_s} = 200$, give the best results for this query because they minimize the number of RA’s as possible.

<table>
<thead>
<tr>
<th>Query#</th>
<th>$\frac{C_r}{C_s} = 100$</th>
<th>$\frac{C_r}{C_s} = 800$</th>
<th>$\frac{C_r}{C_s} = 1400$</th>
<th>$\frac{C_r}{C_s} = 2000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>78</td>
<td>78</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>828</td>
<td>218</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>375</td>
<td>344</td>
<td>344</td>
<td>344</td>
</tr>
<tr>
<td>5</td>
<td>2266</td>
<td>688</td>
<td>688</td>
<td>688</td>
</tr>
<tr>
<td>6</td>
<td>188</td>
<td>188</td>
<td>188</td>
<td>188</td>
</tr>
<tr>
<td>7</td>
<td>281</td>
<td>281</td>
<td>281</td>
<td>281</td>
</tr>
<tr>
<td>8</td>
<td>1265</td>
<td>453</td>
<td>422</td>
<td>422</td>
</tr>
<tr>
<td>9</td>
<td>328</td>
<td>156</td>
<td>156</td>
<td>156</td>
</tr>
</tbody>
</table>

Table 6.9: Experiments on CO queries with different values of $\frac{C_r}{C_s}$. 
In the case of CAS queries, see Table 6.10, the impact of changing $\frac{C_r}{C_s}$ is not very clear because content index lists are short. We can, still, notice its effect in the second CAS query. The first case, where $\frac{C_r}{C_s} = 200$, allows more RA’s which requires much time. The remaining cases limit the number of RA’s, the thing that gives better performance results.

<table>
<thead>
<tr>
<th>Query#</th>
<th>$\frac{C_r}{C_s} = 100$</th>
<th>$\frac{C_r}{C_s} = 800$</th>
<th>$\frac{C_r}{C_s} = 1400$</th>
<th>$\frac{C_r}{C_s} = 2000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>234</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>5</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 6.10: Experiments on CAS queries with different values of $\frac{C_r}{C_s}$. 

Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this work we introduced a new approach to optimize storage requirements for ranked XML IR using the NEXI query language.

In most cases, we achieve better IO throughput using sequential accesses to the block-index lists in the binary file, or using random accesses to resolve expensive predicates and uncertainty about some candidates.

We achieve also a better overall performance because the block structure could be considered as a pre-computed memory image that is loaded incrementally. This pre-computed image is also compressed because of the flexible sizes allowed in the binary encoding scheme we use. We can also use merge joins instead of hash joins, where we can decide the relations between elements using their in-memory binary representations. Because we load the pre-computed memory image incrementally, most operations boil down to pointers operations which are efficient and fast in C++.

7.2 Future Work

Our future plan includes:

- Some optimizations to get more compressed index.
7.2. FUTURE WORK

- Some Optimization techniques to enhance the performance further.
- Extending our basic XML ranking engine with more features and functionality, such as phrases, mandatory/optional content conditions, and mandatory/optional path conditions.
- More work on the probabilistic pruning part.
- INEX query parser.
- And more ...
Bibliography


