Efficient Query Evaluation using Hybrid Index Organization

by

Zhou, Yingjie

Master Degree of Software Engineering

Faculty of Science and Technology
University of Macau
Efficient Query Evaluation using Hybrid Index Organization

by

Zhou, Yingjie

A thesis submitted in partial fulfillment of the requirements for the degree of

Master Degree of Software Engineering

Faculty of Science and Technology
University of Macau

2011

Approved by __________________________________________________

Supervisor

__________________________________________________

__________________________________________________

__________________________________________________

Date __________________________________________________________
In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Macau, I agree that the Library and the Faculty of Science and Technology shall make its copies freely available for inspection. However, reproduction of this thesis for any purposes or by any means shall not be allowed without my written permission. Authorization is sought by contacting the author at

Address:

Telephone:
Fax:
E-mail:

Signature ______________________
Date __________________________
Recently, a large amount of attention has been paid to efficient document retrieval from a gigantic data collection, such as in web search areas. For a keyword search, the top-K computation scans the documents from multiple involved inverted lists and the computation should be stopped when there is no other unseen documents better than the top-K documents being seen so far, instead of traversing the whole lists. In this thesis, we give a comprehensive study and analyze the pros and cons of the state-of-the-art indexing structures and the top-K document retrieval techniques. Furthermore, we propose a new top-K evaluation framework from the following aspects.

1. Reorganizing inverted index— we study a new indexing structure such that more promising documents appear to the beginning of an inverted index.

2. Refining execution strategies – we study a new processing strategy such that a faster evaluation can be achieved.

The superior of our proposed techniques has been demonstrated by a thorough experimental evaluation which compares our proposed techniques to the state-of-the-art approaches in the final section. It turns out that, on average, the
response time of our methodologies is about 80% less than that of the reviewed approaches.
## TABLE OF CONTENTS

List of Figures ........................................................................................................ iii  
List of Tables ........................................................................................................ vi  
LIST of Abbreviations ........................................................................................ vii  
PREFACE ................................................................................................................ ix  

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Top-K Document Retrieval</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.2 Challenges and Solutions</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1.3 Overview of Contents</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Basic Concepts</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Ranking Functions</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Literature Review</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Merge-sort on Document-sorted Index Organization</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
| 3.2 Evaluation Approaches on Score-sorted Index Organization | 12  
| 3.2.1 Threshold Algorithm | 13  
| 3.2.2 No Random Access Algorithm | 15  
| 3.2.3 Lattice-Based Rank Aggregation | 18  
| 3.2.4 Impact-based Pruning Method | 23  
| 3.3 Revisiting Globally Sorted Index | 28  

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Hybrid Method</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Problem analysis</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
| 4.2 our approaches | 33  
| 4.2.1 New Inverted Index Construction | 33  
| ➢ Equal Partition | 33  
| ➢ Deviation-based Partition | 34  
| 4.2.2 Retrieval Strategies | 37  
| ➢ Hybrid-I Method | 37  
| ➢ Hybrid-II Method | 43  

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Experimental Results</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Experimental Setup</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>
5.2 Experimental Results .......................................................................................... 48
  ➢ Tuning Parameters of Partition Methods ...................................................... 48
  ➢ Partition Method Comparison ................................................................... 49
  ➢ Retrieval Approaches Comparison .............................................................. 50

Chapter 6 Conclusions and Future Work .............................................................. 56

Bibliography ........................................................................................................... 58
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: Inverted Index Structure</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2: Compressed inverted index structure with blocks for caching. <em>docIDs</em> are shown after taking the differences to the preceding values</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3: Three inverted lists sorted by <em>docID</em></td>
<td>12</td>
</tr>
<tr>
<td>Figure 4: Three inverted lists sorted by score</td>
<td>14</td>
</tr>
<tr>
<td>Figure 5: The LARA algorithm</td>
<td>23</td>
</tr>
<tr>
<td>Figure 6: The relationship between modes conversion and the size of accumulator set</td>
<td>26</td>
</tr>
<tr>
<td>Figure 7: Impact-based Pruning Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8: The retrieval strategy</td>
<td>31</td>
</tr>
<tr>
<td>Figure 9: Equal Partition Method</td>
<td>34</td>
</tr>
<tr>
<td>Figure 10: Equal Partition Illustration (<em>P</em> = 2)</td>
<td>34</td>
</tr>
<tr>
<td>Figure 11: The distribution diagram of some inverted list samples</td>
<td>35</td>
</tr>
<tr>
<td>Figure 12: The deviation-based partition method</td>
<td>37</td>
</tr>
<tr>
<td>Figure 13: Lattice structure illustration</td>
<td>39</td>
</tr>
<tr>
<td>Figure 14: The hybrid-I method</td>
<td>42</td>
</tr>
<tr>
<td>Figure 15: The hybrid-II method</td>
<td>45</td>
</tr>
<tr>
<td>Figure 16: Turning parameter to get the best index partition of both equal and deviation-based partition for each method (hybrid-I &amp; hybrid-II)</td>
<td>49</td>
</tr>
<tr>
<td>Figure 17: Partition Method Comparison in hybrid-I and hybrid-II evaluation method</td>
<td>50</td>
</tr>
<tr>
<td>Figure 18: Overall Comparison of our IR approaches with the reviewed ones</td>
<td>52</td>
</tr>
<tr>
<td>Figure 19: Overall comparison of the hybrid-I method and the hybrid-II method</td>
<td>53</td>
</tr>
<tr>
<td>Figure 20: Comparison of the hybrid-I and the hybrid-II method for different correlated query sets</td>
<td>54</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Combinations of index organization and processing mode</td>
<td>8</td>
</tr>
<tr>
<td>Table 2: Average execution time of the three algorithms</td>
<td>32</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

**TA.** Threshold Algorithm

**NRA.** No Random Access

**LARA.** Lattice-based Rank Aggregation

**RGSI.** Revisiting Globally Sorted Index

**docID.** Document Id

**DAAT.** Document-At-A-Time

**TAAT.** Term-At-A-Time

**SAAT.** Score-At-A-Time
Efficient document information retrieval is a popular research topic in web search area. Many studies have been devoted to the organization structures of the inverted index as well as the top-$K$ document evaluation or retrieval approaches on those index structures.

For the researchers who are interested in this topic, this paper presents a new index organization method and two efficient evaluation approaches on the new index structure. Compared to the state-of-the-art methodologies, our new method turns out to be far more quickly in responding to a query.

What’s more, for those who are new to the information retrieval area, they’ll start to understand the concepts of this topic, in that this paper gives a detailed review of both the classic and the state-of-the-art methodologies and carefully implemented all the approaches that will be compared with our new method.
ACKNOWLEDGMENTS

It is an honor for me to express my appreciation to all those who helped me during the writing of this thesis.

My deepest gratitude goes first and foremost to Professor Gong Zhiguo, my supervisor, for his constant encouragement and guidance. He guided me through all the stages of the writing of this thesis.

Second, I would like to express my heartfelt gratitude to Dr. U Leong Hou, who has offered me many valuable suggestions in the academic studies as well as much technical support in gathering data.

I am also indebted to the professors and teachers at the Department of Science and Technology, who have instructed and helped me a lot in the past two years. My gratitude also extends to my beloved family and friends who always stand at my side.

Last but not least, I would like to thank members of the examination committee for their patiently reviewing this thesis.
CHAPTER 1  
INTRODUCTION

1.1 Top-K Document Retrieval

Top-K document retrieval is the process of obtaining a number (K) of highest ranking documents to a query, which is composed of several key terms, with monotone ranking functions. A ranking function combines observed statistical properties of a document (in the context of a collection) and a query to compute a numeric similarity score indicating the likelihood that the document is an answer to the query.

To better understand efficient top-K document retrieval, we first introduce the basic index structure: inverted index, which is composed of many inverted lists. Figure 1 illustrates inverted index concept. An inverted list can be understood as a mapping from a term to the documents in which this term appears. Each inverted list is organized as a sequence of postings, each of which contains a document ID (docID), and additional information such as the document-term relevant score $S_{d,t}$, the exact positions of the term’s occurrences in the document, and their context (e.g., title, URLs, etc.). The simplest similarity score $S_{d,q}$ of a document to a query can be calculated as the sum of its document-term relevant scores: $S_{d,q} = \sum_{t \in q} S_{d,t}$.

![Inverted Index Structure](image.png)

Figure 1: Inverted Index Structure
Thus, to process a query, a search engine has to traverse all the relevant inverted lists to calculate relevant ranking scores for all involved documents, and finally return the $K$ documents with the highest scores.

### 1.2 Challenges and Solutions

Most practical and commercially operated Internet search engines have to answer thousands of queries per second with prompt response times. However, due to the rapid growth of the web, even the size of the involved inverted lists becomes incredibly large, and the time spent on traversing those lists will not be acceptable. Thus, one of the main challenges for a web search engine is how to obtain the most relevant top-$K$ documents from the huge amount of document collection within a limited response time.

Many techniques have been proposed to solve this problem, such as massive parallelism, caching, inverted index compression and top-$K$ evaluation strategy. We focus on top-$K$ evaluation strategy, which has been widely used in carrier-class search engines.

To avoid traversing the whole inverted lists, top-$K$ retrieval strategies come into view. The basic idea of this technique is to first sort the postings of each inverted list in such a way that most promising documents appear to the beginning of that inverted list. Then appropriate evaluation strategies are proposed on the re-sorted lists such that the documents with the highest ranking scores will be obtained without traversing the entire inverted lists, when so called “early termination condition” is satisfied. The early termination condition can be described as the following: the minimum score in the current top-$K$ document list is greater than the maximal possible score of all documents to be processed, which indicates that none of the unseen documents will be
more likely to be the answer to the query than the $K$ candidate documents that have been seen up to now, so that we can stop here.

### 1.3 Overview of Contents

Chapter 2 will introduce some basic concepts. Some state-of-the-art methodologies will be reviewed in chapter 3. A top-$K$ document retrieval strategy typically involves two steps:

1. **Organizing inverted index** – design an indexing structure such that more promising documents appear to the beginning of an inverted index.

2. **Execution strategies** – design a processing strategy on the indexing structure such that a faster evaluation can be achieved.

The merge-sort method introduced in section 3.1 sorts the inverted index by document ID, which is a widely used indexing structure. This method can’t avoid traversing the whole inverted index but it can greatly speed up the evaluation process. Section 3.2 introduces four early termination methodologies based solely on term-dependent information: threshold algorithm, no random access method, lattice-based rank aggregation and impact-based pruning method. Section 3.3 introduces the revisiting globally sorted index algorithm, which takes global information into account.

Based on the studied algorithms, we propose our approach in chapter 4. Our index structure and query evaluation are introduced in this chapter.

Chapter 5 gives out the experimental results of our approach and we will compare our results to the state-of-the-art methodologies to evaluate the contributions of our work. It turns out that our solution outperforms all the reviewed methodologies.

Finally we’ll conclude the thesis in chapter 6 and derive our future work motivations for this research work.
2.1 Ranking Functions

Suppose that every document has a relevant score $S_{d,t}$ representing how relevant the document $d$ is to a term $t$. This kind of score is term-dependent information. For now, the overall numeric score $S_{d,q}$, which indicates the likelihood that a document $d$ will be an answer to a query $q$, can be calculated by the ranking function:

$$S_{d,q} = \sum_{t \in q} S_{d,t} \tag{2.1}$$

Many functions can be used in calculating $S_{d,q}$ and here we use BM25 formula, which has been widely used in search engine areas, shown as follows:

$$S_{d,t} = \omega_t \cdot \frac{(1+k_1) tf}{k_1 + \frac{b \cdot dl}{avdl} + tf} \tag{2.2}$$

Where $tf$ is the term frequency (i.e., the number of occurrences) in the document, $dl$ is the document length, $avdl$ is the average length of all documents in the collection, $k_1 (\geq 0)$ and $b (\in [0,1])$ are two constant parameters, and $\omega_t$ is the inverse document frequency weight of the term $t$ and can be computed as follows:

$$\omega_t = \log \frac{N - n_t + 0.5}{n_t + 0.5} \tag{2.3}$$

Where $N$ is the total number of all documents in the collection while $n_t$ is the number of documents containing the term $t$. From the above formula, we can see that $S_{d,t}$ can be pre-calculated when setting up the inverted index such that it can be directly used in document evaluation process. Besides, formula 2.2 and 2.3 indicate
that the rare terms in the collection will have higher $\omega_t$ values than the common terms, while the frequently-used terms in a document tend to have higher scores for the second factor in the formula 2.2 than those who appear in the document only a few times. Thus, a document will be assigned a high $S_{d,t}$ score if a rare term frequently appears in it.

In addition to term-dependent information, another kind of information used in evaluation strategies is term-independent information, denoted as global score $GS_e$, which will be described in detail in the revisiting global score method (RGS). For simplicity, before introducing RGS approach, we only take term-dependent score into account.
The common document retrieval strategies can be roughly divided into three categories: document-at-a-time (DAAT), term-at-a-time (TAAT) and score-at-a-time (SAAT). DAAT [3] evaluates a document considering the contributions of all query terms before it deals with the next document; TAAT evaluates all documents in the inverted list of a term before it does so for the next term; SAAT [4] first splits an inverted list into a few segments such that all documents of a segment have the same quantized term scores (called impacts). It sorts the segments by their impacts and then evaluates all documents in the first segments of all terms before it deals with the next segment. The impact-based pruning [4] essentially is an implementation of NRA in an SAAT evaluation way.

Table 1 summarizes these various options for index organization and processing mode, and gives some example approaches for each combination. The details of these approaches are described later.

<table>
<thead>
<tr>
<th>Processing mode</th>
<th>Document-sorted index organization</th>
<th>Score-sorted index organization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DAAT</strong></td>
<td>Reasonably fast unless query has many terms. E.g., merge sort.</td>
<td>Reasonably fast. E.g., TA, RGS.</td>
</tr>
<tr>
<td><strong>TAAT</strong></td>
<td>Fast. Accumulator structure required. Exhaustive searching (naive).</td>
<td>Fast. Accumulator structure required. Pruning possible by dropping tail of any list, or by dropping whole lists. E.g., NRA, LARA.</td>
</tr>
<tr>
<td><strong>SAAT</strong></td>
<td>Not practical.</td>
<td>Fast. Accumulator structure required. Pruning possible by</td>
</tr>
</tbody>
</table>
Table 1: Combinations of index organization and processing mode

Now we look into these state-of-the-art methodologies according to the two index structures. We first review the retrieval methodologies on document-sorted index organization. From table 1, it is shown that only the merge-sort algorithm (DAAT) and the naïve algorithm (TAAT) are practical for this index structure. The naïve algorithm just records the document-term scores of a document for each term and sums them up after traversing the whole inverted index. The logic is simple and the is bad so that we don’t discuss it in detail. Instead, we talk about merge-algorithm since it greatly saves the response time although it still can’t avoid traversing the whole index structure. For score-sorted index organization, all retrieval strategy options (DAAT, TAAT and SAAT) contain practical retrieval strategies and we study them one by one (DAAT: threshold algorithm; TAAT: no random access method and lattice-based rank algorithm; SAAT: impact-based pruning method).

3.1 Merge-sort on Document-sorted Index Organization

Merge sort algorithm sorts all the postings in each inverted list by *docIDs*. This kind of index structure enables speeding up searching process by three factors:

1. Data compression can be carried out on the inverted index.
2. The ranking evaluation can be done by merging.
3. Considerable memory of accumulator can be saved since only the highest $K$ ranking document information need be maintained.

Here we explain the three factors in detail.
Data Compression

Due to the large web data size, even the inverted lists of common query terms may contain millions of postings. Web search engines typically have to store the inverted index structure on disk. To accelerate fetching the lists from disk, search engines use sophisticated compression techniques that significantly reduce the size of each inverted list. The main idea is that docIDs can be compressed by storing not the raw docID but the difference between the docIDs in the current and the preceding posting (which is a much smaller number, particularly for very long lists with many postings).

In the inverted index sorted by docIDs, this difference value becomes very small, and it leads to faster fetching the lists from disk.

The overall structure we used is shown in Figure 2.

![Compressed inverted index structure with blocks for caching.](image)

**Figure 2:** Compressed inverted index structure with blocks for caching. docIDs are shown after taking the differences to the preceding values.

As shown in Figure 2, the inverted index is partitioned into a large number of blocks, each of the same size, say 64KB. An inverted list in the index will often stretch across multiple blocks, starting somewhere in one block and ending somewhere in another
block. Blocks are the basic unit for fetching index data from disk, and for caching index data in main memory. Thus, each block contains a large number of postings from one or more inverted lists. These postings are again divided into chunks. For example, the postings of an inverted list may be divided into chunks, each containing 128 postings. A block then consists of some offset data at the beginning telling how many inverted lists this block contains and where they start, followed by a few hundred or thousand such chunks. In each chunk of 128 postings, it is often beneficial to separately store the 128 docIDs, then the 128 document-term relevant scores.

Chunks are our basic unit for decompressing inverted index data, and decompression code is tuned to decompress a chunk in fractions of a microsecond.

Thus, data compression on the inverted index sorted by docIDs greatly saves the time spent on fetching the relevant inverted lists.

**Merge-sort Evaluation**

There is obvious naïve algorithm for obtaining the top-K answers to a query. It looks at each posting in each of the involved inverted lists, computes the overall grade of every object, and returns the top-K answers. This algorithm needs an accumulator structure to hold the ranking scores of the documents that have been seen. Every time a document is met, we have to traverse the accumulator to see whether the current document has been seen. If so, its overall ranking score should be added with the last $S_{d,r}$; otherwise, the new document should be added to the accumulator. Because of the enormous memory size occupied by the accumulator, determining whether the document has been seen will spend a lot of time, and thus the naïve algorithm will cost a lot of time.

Sorting the inverted index by docIDs avoid holding a large accumulator, as well as the determining step. In the merge-sort algorithm, we keep a result heap of $K$ pairs $<docID, S_{d,q}>$. We first access the top member of each of the lists, calculate the score of each document we have met, and we have a pointer pointing to the posting that has the smallest docID. It is sure that the document of the smallest docID will never
appear in the rest of the inverted lists, thus we can determine whether this document should be put into the result heap or it should be dropped. If the size of the result heap is smaller than $K$, it should be added to the result heap; or if the document ranking score is greater than the worst score in the current result heap, the worst document should be replaced by this document. Then we access the posting next to the current pointer. This process is continued until all the inverted lists have been traversed. The merge-sort algorithm will be very fast.

Memory Save

In the previous explanation, we can see that there the large-size accumulator can be avoided, and only $K$ candidate documents’ ranking scores need to be maintained. This saves considerable size of memory.

Merge sort is the simplest way to speed up response time. In this method, postings of a list are sorted by $docIDs$, so that data compression can be used, which will improve the searching efficiency. More importantly, $S_{d,q}$ can be calculated using merge, and this is very fast. Only the top-$K$ documents with the highest scores seen so far need to be kept, instead of using a very large accumulator to keep all the scores of all documents. However, this method still can’t avoid scanning the entire inverted index to give out the final result.

We’ll give an example of this algorithm in Figure 3. Assume the ranking function $S_{d,q} = \sum_{e \in q} S_{d,e}$ and $K=1$. The three input inverted lists are sorted by $docID$ alphabetically.
In the first round, merge-sort algorithm retrieves document $a$ and $b$ from $IL_x \quad IL_z$ and $IL_y \quad S_{a,q} = 1.2$ and $S_{b,q} = 0.8$. Then $< a, 1.2 >$ is pushed to the result set and we’ll go to the next round. We get $S_{b,q} = 0.8$ and $S_{c,q} = 1.2$, and $b$ can be thrown away in that we can make sure that it’ll never be met later and it’s no better than the worst candidate (the document $a$ at this moment). In iteration 3, $S_{d,q} = 1.9$ and $< a, 1.2 >$ should be replaced by $< d, 1.9 >$. This way, the document with the smallest document ID in each round will be compared to the worst candidate in the result set, if its score is better, it’ll replace the worst candidate, and otherwise it’ll be ignored. To the end of the inverted lists, we get the top-1 result $\{< d, 1.9 >\}$.

### 3.2 Evaluation Approaches on Score-sorted Index Organization

Then the idea of early termination comes into view. Researchers thought of sorting the inverted index by relevant scores $S_{d,z}$ such that more important documents skew to the beginning of the inverted lists. Then many appropriate evaluation strategies have been proposed to identify top-$K$ result without scanning the entire relevant lists, e.g., the Fagin’s Algorithm (FA), Threshold Algorithm (TA) and No Random-access Algorithm (NRA) strategies [2]. FA was first introduced while the TA and NRA algorithms are proposed later and attract more research attention.

Before further introducing these algorithms, we should understand two modes of access to data. The first mode of access is sorted (or sequential) access. Here the search engine obtains the information ($docID$, $S_{d,z}$) of a posting through the list sequentially from the top. Thus, if a document $d$ has the $l$-th highest score $S_{a,z}$ in list, then $l$ sorted accesses to the list are required to see this score under sorted
access. The second mode of access is random access. Given the above condition, here the search engine obtains the $l$-th score in one random access. If there are $s$ sorted accesses and $r$ random accesses, then the access cost is $sc_s + rc_R$, where $c_s$ and $c_R$ are positive constants. Typically $c_R$ is much greater than $c_s$.

The main difference between TA and NRA is that the former allows random access on the inverted list while the latter one only allows sequential access. Furthermore, Lattice-based Rank Aggregation (LARA) is an improved version of NRA. We now present the threshold algorithm (TA), no random-access algorithm (NRA) and lattice-based rank aggregation algorithm (LARA).

### 3.2.1 Threshold Algorithm

The threshold algorithm can be described by the following steps:

1. Do sorted access in parallel to each of the $m$ sorted lists $IL_i$. As a document $d$ is seen under sorted access in some list, do random access to the other lists to find the grade $x_i$ of the document $d$ in every list $IL_i$. Then compute the grade $S(d) = t(x_1, ..., x_m)$ of the document $d$. If this grade is one of the $k$ highest we have seen, then remember the document $d$ and its grade $S(d)$ (ties are broken arbitrarily, so that only $k$ objects and their grades need to be remembered at any time).

2. For each list $IL_i$, let $x_i$ be the grade of the last document seen under sorted access. Define the threshold value $\tau$ to be $t(\sum_i x_i)$. As soon as at least $k$ documents have been seen whose grade is at least equal to $\tau$, then halt.

3. Let $Y$ be a set containing the $k$ documents that have been seen with the highest grades. The output is then the graded set $\{(d, S(d)) | d \in Y\}$. 
To better understand TA, we consider an example of Figure 4. The data is the same with that of Figure 3, but the inverted lists are reordered and now they are sorted by document-term relevant scores. The ranking function is still \( S_{d,q} = \sum_{t \in q} S_{d,t} \) and \( K=1 \).

<table>
<thead>
<tr>
<th>( IL_1 )</th>
<th>( IL_2 )</th>
<th>( IL_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>h, 0.9</td>
<td>c, 0.6</td>
<td>a, 0.9</td>
</tr>
<tr>
<td>d, 0.7</td>
<td>b, 0.6</td>
<td>e, 0.9</td>
</tr>
<tr>
<td>c, 0.6</td>
<td>d, 0.5</td>
<td>f, 0.9</td>
</tr>
<tr>
<td>e, 0.6</td>
<td>g, 0.2</td>
<td>d, 0.7</td>
</tr>
<tr>
<td>g, 0.3</td>
<td>h, 0.1</td>
<td>b, 0.2</td>
</tr>
</tbody>
</table>

Figure 4: Three inverted lists sorted by score

TA retrieves document h (from \( IL_1 \)), c (from \( IL_2 \)) and a (from \( IL_2 \)). Since h has not been seen before and its ranking score is incomplete, two random accesses are performed to \( IL_2 \) and \( IL_3 \) and we get \( S_{h,q} = 1.0 \). Similarly, random accesses are performed to compute \( S_{c,q} = 1.2 \) and \( S_{a,q} = 1.2 \). After the first round, c is the best document found so far, but \( S_{c,q} \) is lower than \( \tau = \text{Sum}(0.9, 0.6, 0.9) = 2.4 \). Thus it is likely that a better solution can be found and the algorithm proceeds to the next round. There, documents d, b, e are accessed, and the overall ranking scores of them are computed by random access. \( S_{d,q} = 1.9 \), \( S_{b,q} = 0.8 \) and \( S_{e,q} = 1.5 \). The document d is found to be the top object but the algorithm goes to the next round because \( \tau = 2.2 \), which is higher than \( S_{d,q} \). The algorithm stops at the fourth round where \( \tau = 1.5 \leq S_{d,q} \). Thus, TA terminates returning d.

TA is only suitable for the situations where random access is allowed and we have not yet concerned about the number of random access. Suppose there are \( m \) relevant inverted lists for a query \( q \), in TA, for every sorted access, up to \( m - 1 \) random
accesses take place. Recall that if $s$ is the number of sorted accesses and $r$ is the number of random accesses, then the algorithm costs $sc_s + rc_r$, for some positive constants $c_s$ and $c_r$.

There are two scenarios where the number of random accesses must not be ignored. The first scenario is where random access is forbidden, which means $c_r = \infty$. Another scenario is where random access is possible but simply expensive, compared to sorted access.

This paper talks about textual search engines, where random access is impossible. Thus TA is not practically implemented, but just illustrates the basic idea of how to avoid exhaustive searching.

### 3.2.2 **No Random Access Algorithm**

To deal with the situations where random access is impossible or is too expensive, we show in this section how to modify TA to obtain an algorithm NRA (no random access) that does not use random access.

NRA accesses the information in a natural way, and intuitively halts when it determines that no improvement can be made. In general, at each point in an execution of a document retrieval algorithm where a number of sorted and random accesses have taken place, for each document $d$ there is a subset $S(d) = \{i_1, i_2, \ldots, i_l\} \subseteq \{1, \ldots, m\}$ of the $m$ inverted lists where the relevant scores $x_{i_1}, x_{i_2}, \ldots, x_{i_l}$ of the document $d$ with these terms have been obtained. The algorithm proceeds until there are no more candidates whose current upper bound is better than the current $k$-th largest lower bound. The definitions of “lower bound” and “upper bound” are given below:

**Lower Bound:** Given a document $d$ and subset $S(d) = \{i_1, i_2, \ldots, i_l\} \subseteq \{1, \ldots, m\}$ of the $m$ inverted lists where the relevant scores $x_{i_1}, x_{i_2}, \ldots, x_{i_l}$ of the document $d$ with these
terms have been obtained, we define $W_S(d)$ (or $W(d)$ when the subset $S = S(d)$ is clear) as the minimum (or worst) score the ranking function $t$ can attain for the document $d$. When $t$ is monotone, this minimum value is obtained by assigning 0 to each unknown score of missing inverted list $i \in \{1, \cdots, m\} \setminus S$, and applying $t$ to the result. Thus $W_S(d)$ can be obtained by the function $W_S(d) = t(x_{i_1}, x_{i_2}, \cdots, x_{i_t}, 0, \cdots, 0)$.

In another word, $W(d)$ represents a lower bound on $t(d)$. In general, as the algorithm progresses and we find the document $d$ appears in more inverted lists, its $W(d)$ value becomes larger since the latest document-term relevant score should be counted in.

**Upper Bound:** Given a document $d$ and subset $S(d) = \{i_1, i_2, \cdots, i_t\} \subseteq \{1, \cdots, m\}$ of known inverted lists, with known scores $x_{i_1}, x_{i_2}, \cdots, x_{i_t}$, $B_S(d)$ is defined as the maximum (or best) score the ranking function $t$ can attain for the document $d$. The best ranking score a document can get depends on other information we have. Here we only use the bottom values in each inverted list, defined as in TA: $x_i$ is the last (smallest) value obtained via sorted access in the inverted list $L_i$. When $t$ is monotone, this maximum value is obtained by assigning $x_i$ to each missing inverted list $i \in \{1, \cdots, m\} \setminus S$, and applying $t$ to the result. Thus, the maximum ranking score can be represented as $B_S(d) = t(x_{i_1}, x_{i_2}, \cdots, x_{i_t}, x_{i_t+1}, \cdots, x_m)$.

From the above definition, we can immediately derive a property: if $S$ is the set of known inverted lists where the document $d$ has appeared, then $t(d) \leq B_S(d)$.

That is to say, $B(d)$ represents an upper bound on the ranking score $t(d)$, given the information we have so far. In general, as an algorithm progresses and we find the
document $d$ in more inverted lists, the bottom values $x_d$ decrease, and thus $B(d)$ will also decrease.

With the definitions of lower bound and upper bound, let $W_k$ be the set of the $K$ documents with the largest lower bounds. If the smallest lower bound in $W_k$ is at no smaller than the largest upper bound of any document $x$ not in $W_k$, then $W_k$ is returned as the top-$K$ result and the algorithm terminates. Then NRA can be described by the following pseudo code:

1. Do sorted access in parallel to each of the $m$ sorted lists $L_i$. At each depth $d$ (when $d$ objects have been accessed under sorted access in each list) maintain the following:
   - The bottom values $x_1(d), x_2(d), \ldots, x_m(d)$ encountered in the lists.
   - For every object $R$ with discovered fields $S = S^{(d)}(R) \subseteq \{1, \ldots, m\}$ the values $W^{(d)}(R) = W_S(R)$ and $B^{(d)}(R) = B_S(R)$.
   - The $k$ objects with the largest $W^{(d)}$ values seen so far (and their grades); if two objects have the same $W^{(d)}$ value, then ties are broken using the $B^{(d)}$ values, such that the object with the highest $B^{(d)}$ wins (and arbitrarily if there is a tie for the highest $B^{(d)}$ value). Denote this top $k$ list by $T^{(d)}_k$.

Let $M^{(d)}_k$ be the $k$th largest $W^{(d)}$ value in $T^{(d)}_k$.

2. Call an object $R$ viable if $B^{(d)}(R) > M^{(d)}_k$. Halt when (a) at least $k$ distinct objects have been seen (so that in particular $T^{(d)}_k$ contains $k$ objects) and (b)
there are no viable objects left outside \( T_k^{(d)} \), that is, when \( B^{(d)}(R) \leq M_k^{(d)} \) for all \( R \not\in T_k^{(d)} \). Return the objects in \( T_k^{(d)} \).

Let's see how NRA processes the top-1 query for the ranked inputs of Figure 4 and the ranking function is \( S_{d,q} = \sum_{c \subseteq q} S_{d,c} \) assuming that atomic scores in each inverted list range from 0 to 1 and denoting lower bound of document \( x \) as \( lb_x \) and upper bound as \( ub_x \). In the first iteration, NRA accesses \( h, c \) and \( a \). The object with the highest upper bound is \( a \) or \( c \), with \( ub_a = ub_c = 2.5 \). Since \( lb_a < lb_c \), NRA loops to access a new iteration of documents. After the fifth iteration, \( W_1 = \{b\} \), where \( lb_b = 1.9 \) and the highest upper bound is \( ub_e = 1.6 \).

Since \( lb_b > ub_e \), the algorithm terminates. Although it seems that NRA goes to the end of the input inverted list, it'll stop here if the lists are longer with other information.

Although NRA solves the problem of random access forbidden, its execution may require a lot of book keeping at each step. The reason is that each time NRA does a sorted access, the bottom value is updated with a smaller score, and thus the upper bound scores must be updated for each document \( d \) that has been seen so far. Given the depth of the inverted index is \( l \), this may be up to \( lm \) updates for each depth, which yields a total of \( \Omega(l^2) \) updates by depth \( l \). Furthermore, unlike TA, it no longer suffices to have bounded buffers. This is an issue for further investigation.

### 3.2.3 Lattice-Based Rank Aggregation

As mentioned in the previous section, in practice NRA algorithms can have significant performance difficulties in terms of accesses, computational cost and memory requirements. The number of accesses is a serious cost factor, and the computational cost is critical for real-time applications, whereas memory is an issue
for NRA algorithms, which, as opposed to random-access based methods, have large buffer requirements.

To overcome these problems, some key observations have been overlooked by past research and applied on the whole family of “no random accesses” (NRA) algorithms that perform top-$K$ search with monotone ranking functions. These observations point out two phases that any NRA algorithm should go through: a growing phase, during which the set of top-$K$ candidates grows and no pruning can be performed and a shrinking phase, during which the set of candidates shrinks until the top-$K$ result is finalized.

Based on these observations, the Lattice-based Rank Aggregation (LARA) algorithms proposed to optimize “no random accesses” method. Identifying the operations required in each (growing and shrinking) phase, LARA chooses appropriate data structures in orders to support them efficiently. LARA takes its name from the lattice it uses to reduce the computational cost and the number of sorted accesses in the shrinking phase.

A set of lemmas impose some useful rules that indicate the observations of the two phases. We now have a look at those lemmas. Let $t$ be the $k$-th highest score in $W_k$ and $T$ be the sum of the bottom scores of all the inverted lists. The following lemma can be proved:

**LEMMA 1.** If $t < T$, every object which has not been seen so far at any input can end up in the top-$K$ result.

**LEMMA 2.** If $t < T$, any of the documents seen so far can end up in the top-$K$ result.

From lemmas 1 and 2 we can derive that the set of candidate documents can only grow and there is nothing that we can do with it. Thus, while $t < T$, we should only update $W_k$ and $T$ when accessing documents from the inverted lists and need not apply expensive updates and comparisons on the upper bounds.
At the moment when \( t \geq T \) is true, NRA should start maintain upper bounds and compare the highest upper bound with \( t \), in order to verify the termination condition as said above. It is important to notice that if \( t \geq T \), all documents that have not been seen so far have no chance to be within the top-\( K \) result set.

**LEMMA 3.** If \( t \geq T \), no document that has not been seen in any inverted list can end up in the top-\( K \) result.

Lemma 3 implies that once \( t \geq T \) holds, the newly met documents should just be ignored and the memory required by the algorithm can only shrink.

Lemma 1 to 3 imply the two phases: a growing phase during which \( t < T \) and the set of top-\( K \) candidates can only grow and a shrinking phase during which \( t \geq T \), and the set of candidates can only shrink until the algorithm terminates at the termination condition being satisfied.

Now we study how LARA does in each phase.

**The growing phase**

In this phase, due to the reason that the set of candidates can only grow and it is meaningless to do any pruning, LARA only maintains (i) the set of documents with their partial ranking scores and the record showing in which inverted lists the documents have been accessed, (ii) \( \mathcal{W}_k \), the set of documents with the \( K \) highest lower bound scores (and \( t \) is the minimum value in \( \mathcal{W}_k \)), and (iii) an array \( L \) recording the bottom score of each inverted list, and \( T \) is the sum of each element of this array.

We use two hash tables \( H \) and \( \mathcal{W}_k \) (with docID as search key) to store each seen document, with its ID, a bitmap indicating form which inverted lists it has been
accessed, and its lower bound. $W_k$ is the set of the documents with the $K$ highest lower bounds. H is the set of the documents that have been seen but is not in $W_k$.

Whenever a document $x$ is accessed from in inverted list $IL_i$, we check whether $x$ is already in $W_k$. If so, we update its lower bound. Otherwise, we update its lower bound in $H$ and then switch it with the $k$-th document in $W_k$ if $lb_x \geq t$. Finally, $L$ and $T$ are updated. $T = T^{prev} - t^i + l_i$, where $T^{prev}$ is the previous value of $T$ and $l_i$ denotes the bottom value in the inverted list $IL_i$.

After each sorted access, the data structures are updated and the condition $t \geq T$ is checked. Once this condition is satisfied, the algorithm goes into the shrinking phase, which will be discussed in the next paragraph.

The shrinking phase

The shrinking phase starts when $t \geq T$ is true. In this phase, upper bounds are maintained and compared to $t$, until the largest upper bound is no greater than $t$, when the top-$K$ result is finalized.

As Lemma 3 indicates, no new documents will be included in the top-$K$ result set in the shrinking phase. Thus, if a newly accessed document is not found in $H$ or $W_k$, it is just ignored and the algorithm proceeds to the next access. This avoids many unnecessary computations, and also saves the memory space required to store the information of the accessed documents. No more memory is required during the shrinking phase until the termination of the algorithm.

Let $x$ be the document in $H$ with the greatest upper bound score $ub_x$, then the algorithm terminates when the condition $ub_x \leq t$ becomes true. The key point is how to efficiently maintain $ub_x$. The tedious technique used by NRA is to explicitly
update the upper bound score for all documents in $H$ and recomputed the document with the largest upper bound score after each access. A large computation cost is involved in this technique, due to that all documents must be accessed and updated.

To avoid explicitly maintaining the upper bound scores for each document that belongs to $H$, LARA proposes a solution to reduce the computation cost based on the following idea. For each combination $v$ of the $m$ inverted lists, we keep a set $S^v$ of documents $x^v$ such that (i) $x$ has been accessed exactly in $v$ inverted lists, (ii) $x \in H$, and (iii) the first document $x_0^v$ in $S^v$ has the highest partial ranking score in that set. This idea indicates that if $ UB_{x^v} \leq t$, it can be immediately concluded that no document in the set $S^v$ may end up in the top-$K$ result. Thus, by maintaining the set $S^v$ of documents $x^v$ for each combination $v$, we can verify the termination condition by only comparing a small number ($2^m$) of upper bound scores to the $k$-th lower bound score in $W_k$.

To summarize, when LARA enters the shrinking phase, it constructs a lattice $\mathbb{Q}$, implemented as a vector of size $2^m$. For each combination $v$ (node in lattice $\mathbb{Q}$) of inputs, it maintains the set of document $x^v$ ($<docID, partial score>$) that are accessed only in $v$, but not in $W_k$, with the highest partial score the first element in the set. As soon as $t$ is not smaller than any upper bound score for each first document in each $v$, LARA terminates reporting $W_k$ as the top-$K$ result set.

As presented so far, LARA can be described by the pseudo code of Figure 5.
3.2.4 Impact-based Pruning Method

As mentioned in the introduction, there are three kinds of similarity computation techniques, term-at-a-time (TAAT), document-at-a-time (DAAT) and score-at-a-time (SAAT). The first two have been discussed in the above sections, and we’ll describe one kind of SAAT methodology in this part.

The impact-based pruning method [4] in essential is similar to LARA, but it makes use of impact-sorted indexes and the exact ranking score of a document can be calculated in this algorithm. To understand the whole algorithm, we first introduce the impact score computations adopted by this algorithm and then the evaluation methodology.

Algorithm LARA (ranked inputs $S_1, S_2, \ldots, S_m$)

1. $\text{growing} := \text{true}; /* \text{initially in growing phase} */$
2. access next object $x$ from next input $S_i$;
3. if $\text{growing}$ then
4. update $\gamma_{lb}^x$; /* partial aggregate score */
5. if $\gamma_{lb}^x > t$ then
6. update $W_k$ to include $x$ in the correct position;
7. update $T$;
8. if $t \geq T$ then
9. $\text{growing} := \text{false};$ construct lattice;
10. goto Line 2;
11. else /* shrinking phase */
12. if $x$ in $H$ then
13. update $\gamma_{lb}^x$; /* partial aggregate score */
14. if $x \in W_k$ then /* already in $W_k$ */
15. update $W_k$ to include $x$ in the correct position;
16. else /* $x$ was not in $W_k$ */
17. $v_{prew}^x :=$ lattice node where $x$ belonged;
18. if $x$ was leader in $v_{prew}^x$ then
19. update leader for $v_{prew}^x$;
20. if $\gamma_{lb}^x > t$ then
21. update $W_k$ to include $x$ in the correct position;
22. check if $y$ (evicted from $W_k$) is leader of $\sigma_y$;
23. else check if $x$ is leader of node $v_x := v_{pre}^x \cup S_i$;
24. $u := \max\{\gamma_{lb}^x : x \in G\}$; /* use lattice leaders */
25. if $t < u$ then goto Line 2;
26. report $W_k$ as the top-$k$ result;

Figure 5: The LARA algorithm
Impact Score Computations

In each document, each term that appears in this document can be sorted by their in-document frequency. Based on that ordering, each term is assigned an impact varying from $K$ down to 1, with exponentially growing numbers of terms assigned to the lower-valued buckets. The upper limit $K$ is decided at index construction time, and is typically assigned to a value such as $K = 8$.

Here’s an example to illustrate the concept of impact. Consider a document which contains $n_d = 55$ distinct terms, of which $n_s = 10$ are stop words. The base $B$ of the exponentially growing set of bucket sizes is calculated via the function $B = (n_d - n_s + 1)^{1/k}$, and the impact buckets are allocated to contain $(B - 1)B^i$ terms, where $i \in 0, 1, \ldots, 7$. After rounded off to integers, these values correspond to 1, 1, 1, 3, 4, 7, 11, 17 terms. That is, the most frequent non-stop term in the document is assigned an impact of 8 (in this document); the next most frequent non-stop term an impact of 7; and so on; until the 17 least frequent non-stop terms are all assigned an impact of 1.

For the terms with equal term frequency (in the same document), they are all assigned the impact that would accrue to the middle term in the cluster.

With regard to query-term weights, the algorithm uses the following formula:

$$\left(1 + \log_2 f_{q,t}\right) \times \left(\log_2 \left(1 + \frac{f^m}{f_t}\right)\right)$$  \hspace{1cm} (3.1),

Where $f^m$ is the maximum value of $f_t$ over the collection, and $f_{q,t}$ is the number of times term $t$ appears in the query. The set of query-term weights are then transformed to a set of integer query-term impacts by linear scaling and integer quantization so that the query-term impact is range from $K$ down to 1 (typically $K = 8$).
Finally, the relevancy of a document $d$ to a query $q$ is calculated as the inner-product of the two integer impact vectors:

$$ S_{d,q} = \sum_{t \in q} \omega_{d,t} \times \omega_{q,t} \tag{3.2}, $$

Where $\omega_{d,t}$ and $\omega_{q,t}$ denote the document-term impact and the query-term impact.

**Impact-based Pruning**

The inverted index is created with pointers applied in an order that is neither strictly term-based nor strictly document-based, but in another way in which indexes can be categorized by the type of information contained in each pointer. Here each pointer contains a single document ID ($docID$) and a document-term impact score $\omega_{d,t}$. The pointers are categorized and sorted by impact score. The documents with $\omega_{d,t} = 8$ are accessed first, then those with $\omega_{d,t} = 8$, and so on, until the termination condition is satisfied.

The processing strategy evaluates documents in four different modes: *Or*-mode, *And*-mode, *Refine*-mode, and *Ignore* mode. If a posting is accessed in *Or*-mode, the document it points to can be nominated as a potential answer and can be considered by subsequent processing steps as well, even if no other query terms appear in it. On the other hand, postings accessed in *And*-mode are allowed to boost the scores of previously seen documents, but are not permitted to be treated as candidates. The *Refine*-mode, is optional. It is useful only when the exact ordering of the top-$K$ results is required. Processing goes into this stage when the set of top-$K$ results have been determined and before their final ordering is fixed. After the *Refine*-mode, processing strategy goes into *Ignore*-mode, in which postings are simply not considered. Figure 6 shows the relationship between the four modes and the size of the accumulator set $A$ of candidate documents.
We note that in LARA, the evaluation process stops as soon as the termination condition comes true. The top-$K$ documents in the returned result set don’t get their precise scores and thus they might not be in their exact order. Instead, the third transition point in the impact-based pruning algorithm guarantees that the ordering of the top-$K$ documents is final, even if not their precise scores, and the processing terminates with the remainder of the inverted index being ignored.

The pseudo code of the impact-based pruning algorithm is displayed in Figure 7. To facilitate computation of the three transition points, each accumulator $A_d$ stores the current score of the document $d$, as well as the quantity $T_d$ that tracks in which inverted lists the document $d$ has been accessed. Besides, for each term $t$, the quantity $next_t$ indicates the $I$ (from $k^2$ down to 1) value at which term $t$ will supply its next block of postings. When the whole inverted list $IL_t$ has been traversed, the value of $next_t$ is set to zero. The quantity $M_d$ is the maximum score that document $d$ can get based on the evidence encountered so far.

Figure 6: The relationship between modes conversion and the size of accumulator set A.
Figure 7: Impact-based Pruning Algorithm

The three transition condition can be expressed as follows:

**Condition 1:** \( R_{\text{min}} \geq \sum \{ \text{next}_t \mid t \in q \} \), where \( q \) is the set of query terms.

**Condition 2:** \( R_{\text{min}} \geq \max \{ M_d \mid d \in A, d \notin R \} \), where

\[
M_d = A_d + \sum \{ \text{next}_t \mid t \in q, t \notin T_d \}.
\]
Condition 3: \( T_d = q, \forall d \in R \), but for situations in which not all of the top-\( K \) documents contain all of the query terms can also happen. In this situation, suppose the result set \( R \) is ordered by decreasing accumulator score, so that \( A_{R_1} \geq A_{R_2} \), and so on; then the processing can be stopped when \( A_{R_i} \geq M_{R_i}, \forall R_i \in R \).

3.3 Revisiting Globally Sorted Index

As mentioned at the very beginning of the thesis, to evaluate the importance of a document, there’re two kinds of information: term-dependent information, such as document-term relevant score or the impact\([4]\), and term-independent information e.g., static rank. We call the term-independent information global scores \((GS)\) since such scores can be determined globally without considering the term information for each document, while we call term-dependent information local scores\((LS)\).

All the above methodologies and many other approaches sort their inverted index by term-dependent information, here by document-term relevant score specifically. However, in reality, some term-independent information should also be considered to evaluate the importance of the document with respect to a query. The overall score of a document \( d \) to a query \( q \) is commonly a weighted sum of both kinds of scores as follows:

\[
S_{d,q} = \alpha \cdot GS(d) + \beta \cdot LS(d,q), \quad (\alpha + \beta = 1, \alpha \geq 0 \text{ and } \beta \geq 0) \quad (3.3),
\]

where \( GS(d) \) is the global score of the document \( d \), and \( LS(d,q) \) is the score of the document \( d \) with regard to the query \( q \). One popular way to calculate the \( LS \) score is the BM25 formula used in \([11]\), which has been widely used in search engine areas and thus adopted in the later sections of this paper. The function is shown in function 2.2:

\[
LS(d,q) = \sum_{t \in q} S_{d,t} = \sum_{t \in q} \omega_t \cdot \frac{(1+k_1)tf}{k_1((1-b) + b \cdot \frac{df}{avdl}) + tf}, \quad \text{and} \quad \omega_t \text{ is calculated by the function 2.3.}
\]
With the ranking function 3.3 and the original organization of the inverted index, which is sorted by document-term relevant score $S_{d,t}$, we should choose the largest global score $GS_{\text{max}}$ in the inverted list and the threshold score $T = \alpha \cdot GS_{\text{max}} + \beta \cdot \sum_{i=1}^{m} x_i^t$, where $x_i^t$ is the bottom value in the inverted list $IL_t$. Then the termination condition should be $t \leq T$, where $t$ is the $k$-th candidate score in the candidate set. As we know, $t$ increases in the process of the algorithm while $T$ decreases, so that the terminate condition will be satisfied at some moment. However, with the maximal global score $GS_{\text{max}}$ involved, $T$ will decrease very slowly and the document retrieval performance might be poor.

Thus, a lot of research work proposed to reorganize the inverted index by global scores, such as solely by static rank. This improved the performance, but as shown in [6, 12, 13], the pure GS methods based only on static rank cannot lead to the effective early termination. It has been verified (both theoretically and experimentally) that early termination can always be achieved on the artificial indexes sorted by static ranks, as long as the local scores LS are uniformly distributed; while on the other hand, early termination cannot be achieved on the real large scale data sets, where the LS scores are not uniformly distributed.

To achieve effective early termination, some new techniques to organize inverted index based on term-independent information and the new retrieval strategies on the resulting index are proposed in [3]. More global information other than static rank is included in the global score to sort the inverted index. We first see how the new indexes are built using three different methods, all of which sort inverted index by the global information except that they are based on different kinds of global information.

The global information used in the three different methods are $UBIR$ (upper bound document-term relevant score), $SR$ (static rank) and $UBTF$ (upper bound term frequency). $UBIR$ score is the maximal value of the document-term relevant scores for all terms contained in the document. Similarly, $UBTF$ score is the maximal value of
the term frequency scores for all terms contained in the document. Thus, \textit{UBIR}, \textit{SR} and \textit{UBTF} are all term-independent. Now let’s have a look at the three different indexing methods.

**Constructing New Indexes**

\textbf{MSI}: The definition of \textit{UBIR} indicates that the document-term relevant scores for all terms contained in the document are not greater than the \textit{UBIR} score. To build the inverted index, we first get the \textit{UBIR} score for each document in the collection and then combine it with the \textit{SR} score together as a global score (\textit{GS}). Then use this global score (\textit{GS}) to sort all inverted lists. The combination function is \[ GS = \max(SR, \lambda \times UBIR). \]

\textbf{SSI}: This method also combines \textit{SR} and \textit{UBIR} scores to calculate the \textit{GS} score. However, different from MSI, the function it uses to calculate the GS score is \[ GS = \alpha \times SR + (1 - \alpha) \times UBIR. \] It is easily to notice that the function is exactly the formula 3.1 and thus the resulting \textit{GS} score is directly the threshold score \textit{T}.

\textbf{MST}: This method combines the upper bound term frequency (\textit{UBTF}), instead of the \textit{UBIR} score, with the \textit{SR} score to evaluate the \textit{GS} score, It assigns GS value as \[ GS = \max(SR, \gamma \times UBTF), \] where \textit{\gamma} is a non-negative parameter. The advantage of this method is that it does not need to create the new index in case the parameter values of the ranking functions, e.g., \textit{k_1} and \textit{b} in formula 2.2, are changed, while the other two methods have to recreate the inverted index in this kind of situation. The disadvantage is that it is unable to reduce the threshold score \textit{T} for all unseen documents as much as the other two methods.

**Retrieval Strategies**

Based on the different inverted indexes constructed by the above three methods, some appropriate retrieval strategies should be proposed.
Notice a common property of these methods is that a upper bound of the maximal possible scores for all unseen documents is explicitly or implicitly embedded into their indexes. Therefore their retrieval strategies are almost the same, the only difference being the ways to estimate the threshold score $T$. Figure 8 shows the retrieval algorithm, in which a line with bold fonts is the way of estimating $T$, and will be discussed in detail as follows.

**Algorithm:** Document retrieval strategy for our algorithms

**Input:** Inverted lists $L_1, \ldots, L_Q$, for the query $Q$

**Output:** Top-$k$ documents

$R = \text{empty};$ // $R$: the current top-$k$ result list
$S_R = 0;$ // $S_R$: the score of the ith document in $R$

loop

$d = \text{NextDoc}();$

if ($d$ is empty) return $R;$

Compute $d$ score:

if ($|R| < k \text{ OR } d$ score $> S_R$)

$R.$insert($d$)

Update $S_R$

end-if

update the maximal possible score for all unseen docs

$S_T$:

if ($|R| \geq k \text{ AND } S_R \geq S_T$)

return $R$;

end-loop

return $R$.

**Figure 8:** The retrieval strategy

*For MSI:* Given that the document $d$ is the being evaluated during the query processing, its global score $GS_d$ is then available in the indexes and can be used to predict the maximal possible value $T$ as follows: since $GS_d = \max (SR, \lambda \times UBIR)$, the upper bound score $T$ can be described as $T \leq \alpha \times GS_d + (1 - \alpha) \times \frac{GS_d}{\lambda}$.

*For SSI:* $T$ is directly the value of $GS_d$.

*For MST:* Use $TF$ values to get the whole $UBIR$ score and then evaluate $T$ in a similar way as the above two methods.
CHAPTER 4  HYBRID METHOD

4.1 Problem analysis

We implemented LARA, Merge-sort algorithm, and also exhaustive searching method to verify the correctness of the other two methods. After verification of the correctness of LARA and Merge-sort methodologies, we use the experimental data to compare these two algorithms. We tested 55 pairs of queries, collected the execution time in getting top $K$ documents ($K = 1, 5, 10, 20, 40, 160, 320$) of each method, as well as the percentage of accessed documents for LARA.

Table 2 presents the average execution time of LARA, Merge-sort and exhaustive searching algorithms.

<table>
<thead>
<tr>
<th>$K$</th>
<th>LARA</th>
<th>Merge-sort</th>
<th>Exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.27</td>
<td>0.44</td>
<td>2.17</td>
</tr>
<tr>
<td>5</td>
<td>0.41</td>
<td>0.43</td>
<td>2.82</td>
</tr>
<tr>
<td>10</td>
<td>0.49</td>
<td>0.43</td>
<td>2.26</td>
</tr>
<tr>
<td>20</td>
<td>0.54</td>
<td>0.43</td>
<td>2.21</td>
</tr>
<tr>
<td>40</td>
<td>0.60</td>
<td>0.43</td>
<td>2.29</td>
</tr>
<tr>
<td>80</td>
<td>0.64</td>
<td>0.44</td>
<td>2.61</td>
</tr>
<tr>
<td>160</td>
<td>0.69</td>
<td>0.44</td>
<td>2.75</td>
</tr>
<tr>
<td>320</td>
<td>0.73</td>
<td>0.44</td>
<td>2.86</td>
</tr>
</tbody>
</table>

Table 2: Average execution time of the three algorithms

From table 2, we observe that both LARA and Merge-sort algorithms outperform the exhaustive algorithm. When $K$ is small, the performance of LARA is better than Merge-sort algorithm. With larger $K$, LARA’s performance becomes even worse than Merge-sort algorithm. There might be two reasons for this situation. One is that the cost of time spent on maintaining a lattice exceeds the time of the merging sort of the whole involved inverted lists. Another reason is that the estimated maximal possible score too far exceeds the $k$-th candidate document score.

Based on this observation, we seek to propose a new approach that is able to stop early in respect to response time and in the meanwhile avoid traversing the whole inverted index.
4.2 our approaches

Our solutions to achieve the above goal will be described in this section. We focus on two main aspects to propose our methodology: 1) build the new index structure by firstly sorting the inverted lists according to document-term relevant scores, and secondly breaking each inverted list into several parts, the postings in each parts being sorted by document ID; 2) use appropriate information retrieval strategies in a DAAT manner to process the resulting inverted index. The basic idea of this method is to make use of the overall score-sorted index to avoid accessing the whole index, and at the same time take the advantage of fast merge-sort approach in between parts of each list.

4.2.1 New Inverted Index Construction

The basic idea of re-constructing the inverted index is to divide each inverted list, into several parts, and then sort the postings within each part by docIDs. Thus, our first task is to design an appropriate partitioning method splitting the original score-sorted inverted index into a number of sections.

➢ Equal Partition

The simplest, or the most naïve partitioning method is to divide the inverted lists equally according to a given parameter $P$, which denotes the number of parts. Suppose an inverted list contains $M$ postings, after equal partitioning, it should be divided into $P$ sections, each section containing $M/P$ postings. The algorithm is shown below.

<table>
<thead>
<tr>
<th>Algorithm: Equal Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>input: inverted lists $IL_1, IL_2, \ldots, IL_t$ ($t$ is the number of lists ), number of parts $P$.</td>
</tr>
<tr>
<td>output: reconstructed inverted lists $IL'_1, IL'_2, \ldots, IL'_t$</td>
</tr>
<tr>
<td>for each single inverted list $IL_i$</td>
</tr>
<tr>
<td>partition $IL_i$ into $P$ sections</td>
</tr>
<tr>
<td>for each section $IL'_i$</td>
</tr>
<tr>
<td>sort $IL'_i$ by document ID of the inside postings</td>
</tr>
</tbody>
</table>
do the data compression
end-for
end for
return $IL'_0, IL'_1, ..., IL'_{n-1}$

Figure 9: Equal Partition Method

With the equal partition method, let’s look at an illustration. The left graph of Figure 10 is our original inverted index sorted by scores and we’ll split it into three partitions. The reorganized index by the equal partition method is shown in Figure 10.

<table>
<thead>
<tr>
<th>Original Index</th>
<th>Reorganized Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IL_1$</td>
<td>$IL_2$</td>
</tr>
<tr>
<td>h, 0.9</td>
<td>c, 0.6</td>
</tr>
<tr>
<td>d, 0.7</td>
<td>b, 0.6</td>
</tr>
<tr>
<td>c, 0.6</td>
<td>d, 0.5</td>
</tr>
<tr>
<td>e, 0.6</td>
<td>g, 0.2</td>
</tr>
<tr>
<td>g, 0.3</td>
<td>h, 0.1</td>
</tr>
</tbody>
</table>

Figure 10: Equal Partition Illustration ($P = 2$)

For different queries, the best partition might be different, that is, the best $P$ for each specific query $q$ composed of different query terms should vary. Thus, we should design a new partition method that will divide each inverted list according to its own characteristics in the purpose of achieving the fastest response time in average.

- Deviation-based Partition

To design such a partition method, we first study the distribution of postings in each inverted list which is sorted by score. The distribution diagrams of some randomly selected terms are shown in Figure 11.
Almost all of the score-sorted inverted lists are in the similar distribution to those shown in Figure 11. We can observe that the score changes extremely fast at the beginning and the decreasing speed becomes slower and slower. This means the sections to the beginning of the list should contain fewer postings, while the sections to the end of the list may include more postings. In this way, the postings with close scores will be grouped into the same section, which will reasonably increase the searching efficiency.

Based on this idea, we should design a dividing rule which should reflect the density of each section. We use the deviation data model to make this decision, since deviation is a measure of difference for interval and ratio variables between observed value and the mean. When we need to decide whether the current accessed postings should be grouped into the same part, we compare their deviation $\sigma_p^2$ with the deviation $\sigma_r^2$ of the rest of the postings in the list. If $\sigma_p^2 \geq \sigma_r^2$, it means the score difference is stronger than the rest of the lists and they should be split out into the same part. Before describing the details of the algorithm, we first look at some definition and formula that will be used in the partition method.

In deciding whether the current accessed postings should be put into a new part, besides their deviation, we use a window size $W$ to make decision. A window size
$WZ$ is the smallest number of postings each part should contain. The window size $WZ$ is relevant to the size of each inverted list. The bigger the list, the bigger the window size is. Give the size $SZ_i$ of an inverted list $L_i$, its window size $WZ$ can be obtained with formula 4.1, where $\alpha$ is a tuning parameter between $(0, 1)$.

$$WZ = SZ_i^\alpha, \alpha \epsilon (0,1) \quad (4.1)$$

The deviation $\sigma^2$ of a number $NP$ of postings is the average of the sum of the squares of the difference between each single score $s_i$ and the average score $avg_i$.

$$\sigma^2 = \frac{\sum_{i=1}^{NP} (s_i - avg_i)^2}{NP}, \text{where} \ avg_i = \frac{\sum_{i=1}^{NP} s_i}{NP} \quad (4.2)$$

With the above definitions, the deviation-based partition method can be described in Figure 12.

<table>
<thead>
<tr>
<th>Algorithm: Deviation-based Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input:</strong> inverted lists $L_1, L_2, \ldots, L_t$ (t is the number of lists), turning parameter $\alpha$.</td>
</tr>
<tr>
<td><strong>output:</strong> reconstructed inverted lists $L'_1, L'_2, \ldots, L'_t$</td>
</tr>
<tr>
<td>for each single inverted list $L_i$</td>
</tr>
<tr>
<td>$sz :=$ number of the postings contained in $L_i$</td>
</tr>
<tr>
<td>$wz := pow(sz, \alpha)$</td>
</tr>
<tr>
<td>$last_score := 0.0$</td>
</tr>
<tr>
<td>$new_index := 0$ // $new_index$: the beginning index of the next part</td>
</tr>
<tr>
<td>$ndoc := 0$ // $ndoc$: the number of documents of the last access</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>access the next document $d$ in $L_i$</td>
</tr>
<tr>
<td>if (($d_score \neq last_score$) and ($ndoc - new_index \geq wz$)) or $ndoc == sz-1$</td>
</tr>
<tr>
<td>update the local deviation $\sigma_p^2$</td>
</tr>
<tr>
<td>update the deviation of the rest postings $\sigma_r^2$</td>
</tr>
<tr>
<td>if $\sigma_p^2 \geq \sigma_r^2$</td>
</tr>
<tr>
<td>sort the postings from new_index to ndoc</td>
</tr>
</tbody>
</table>
4.2.2 Retrieval Strategies

After the new inverted indexes achieved by the above two methods, we propose two retrieval strategies for the resulting index structures. According to the index structure we construct, there are two understanding of “hybrid”. At first we aim to combine the advantages of the fast speed of merge-sort algorithm and the “early stop” idea of LARA. In this kind of processing strategy, merge-sort method is carried out in-between parallel parts of different inverted lists, and lattice structure is used in calculating the highest possible score $out_{best}$ of the documents that are not chosen in the result set $R$. However, now that our inverted lists are sorted by docIDs within each part, $out_{best}$ can be calculated directly in the merging process. This very simple way of hybrid method combines the two ways of index structure, that is, sorted by score and sorted by docID. We refer to the first retrieval strategy as “hybrid-I” method, and the second approach simply as “hybrid-II” method. In this section we’ll first describe these two approaches in detail, and then make a comparison of them, why we should bother to use the complex lattice rather than the simple naïve method in our retrieval.

➢ Hybrid-I Method

Since the hybrid-I method needs to use the lattice structure to reduce the computation cost, we first get a review of it and look at how we construct our lattice.

```
do the data compression
    new_index := ndoc
end-if
    last_score := d.score
end-if
    ndoc ++
end-loop
end-for
return $II_0', II_1', \ldots, II_{k-1}$
```

Figure 12: The deviation-based partition method
A lattice is a hash structure that will map a document \( d \) to a set showing in which inverted lists this document \( d \) has been accessed. Each lattice set stores all the documents that have been accessed in exactly the same combination of inverted lists, and these documents are sorted by their scores. Thus, if we use \(<\text{docID}, \text{score}>\) as the key of each lattice set, the first element of each lattice set presents the document with the highest score that have been accessed in this composition of lists.

In our implementation, we use vector \( \text{lattice\_vector} \) to map the composition of inverted lists and use set as each vector’s element. Suppose there are \( n \) inverted lists \( IL = \{IL_1, IL_2, \ldots, IL_n\} \), then a document \( d \) might be seen in any of \( 2^n \) combinations of lists, (including \( \emptyset \) in which case the document \( d \) appears in none of the lists). For each document, we use a \( \text{bitmap} \) of size \( n \) to record in which list this document has been accessed. If the document \( d \) has been accessed in the inverted list \( L_i \), then \( d.\text{bitmap}[IL_i] := 1 \), otherwise \( d.\text{bitmap}[IL_i] := 0 \).

Then \( d.\text{bitmap} \) match with a unique integer \( d.\text{lindex} \), and then \( d.\text{lindex} \) can be used as the index of the lattice vector. Each element of \( \text{lattice\_vector} \) is implemented as a set, using a pair \(<\text{docID}, \text{score}>\) as its key and sorted by score.

Let’s look at a simple illustration shown in Figure 13. Suppose the query is composed of three query terms, and thus three inverted lists \( IL_0, IL_1, \text{and} IL_2 \) are involved. The illustration is based on the reorganized index in Figure 10. The two diagrams present the content of the \( \text{lattice\_vector} \) after the first and the second part being accessed accordingly. Each ellipse in Figure 13 denotes a vector in the lattice. For example, the top ellipse denotes the vector that contains the documents which have been accessed in all three lists \( IL_0, IL_1, \text{and} IL_2 \). After the first partition has been accessed, we can tell that the documents \( h \) and \( d \) have been accessed in \( IL_1 \), \( b \) and \( c \) in \( IL_2 \) and \( a \) and \( e \) in \( IL_3 \). In each ellipse, documents are first sorted by their score and then by
their docID. The second graph in Figure 13 shows the updated lattice_vector after accessing to the second part of the index. To calculate the maximal possible score, we only need to calculate that score of the first document in each set and get the largest one.

Based on the concept of lattice_vector, we may start to describe our hybrid-I method. The pseudo code of the hybrid-I method is shown in Figure 14. Similar to LARA algorithm, the hybrid-I method also goes through two phases: a growing phase during which the newly accessed documents have chance to be returned as results and the set of candidate documents can only grow; and a shrinking phase during which the documents that are newly accessed together with the documents that have been accessed in all inverted lists but are not chosen as candidates can be safely neglected, and the candidate documents can only shrink, until the top-K result is finalized.

In the shrinking phase, we access one partition of all inverted lists in parallel once a time. For each single inverted list buffer, do merge-sort for the current accessed partition with previous ones. That is, after each access to a new partition, the inverted lists are sorted by docIDs. During the inside merging process of each list \( L_i \), we can get the lower bound score \( \text{Lb}[i] \) of that list. After that, merge-sort can be carried out in-between all involved inverted lists. We put the top-K documents into candidate set. For the rest of documents, if the document has been accessed in all lists, we can determine that this document will have no chance to be selected as a result, so it can
be simply neglected, otherwise it should be put into a container $S$, which keeps track of the information of the documents that will be used to maintain the lattice_vector later in the shrinking phase. The container $S$ stores docID, score and the bitmap of each document, using docID as a key to find a specific document. The container $S$ should be cleared in each iteration of the growing phase, since the in-between merge-sort will be carried from the beginning of the inverted index each time.

At the end of each iteration in the growing phase, a shrinking condition should be checked to determine whether the next iteration should be a shrinking phase, or still a growing phase. If the number of documents in candidate set equals $K$ and the $k$-th score $S_k$ no less than the sum of the lower bound score of other lists $Lb[0] + Lb[1] + ... + Lb[t-1]$, the algorithm can go to the shrinking phase.

At the first iteration of the shrinking phase, a lattice_vector should be constructed by the document information stored in the container $S$. In the shrinking phase, merge-sort is only carried out in-between the newly accessed partitions. During the merge-sort process, if a newly accessed document $d$ is found in neither of the candidate set or the container $S$, nothing needs to be done. If $d$ is found in the candidate set $R$, $d.score$ and $d.bitmap$ should be updated, and the $k$-th score $S_k$ also needs to be updated. If $d$ is found in container $S$, its score should be updated first. If its new score is greater than $S_k$, it will replace the $k$-th candidate. Then we should determine whether the replaced candidate $d'$ needs to be put into the container $S$ or not. If $d'$ has been seen in all lists, it will have no chance to be put into the candidate set $R$ anymore, and thus it can be just thrown away. Otherwise, put it into $S$. Then the lattice_vector Lattice should be updated. First use $d.bitmap$ to find its corresponding lattice set, and then erase $d$ using its id and previous score to identify it in that set. Then put the updated < $d.id$, $d.score$ > into the new lattice set mapped by the updated $d.bitmap$, in the case that $d$ doesn’t replace the $k$-th candidate and $d$ has not been accessed by all lists. Or put the replaced candidate $d'$ to its corresponding lattice set.
if it needs to. After the merge-sort in-between all inverted lists, use the *lattice_vector* to calculate the maximal possible score $T$ and check the termination condition. If the $k$-th score $S_k$ is no less than $T$, the result documents can be returned.

**Algorithm: Hybrid-I**

```plaintext
input: Inverted lists $IL_1, IL_2, \ldots, IL_t$ for the query $Q$ composed of $t$ terms
output: Top-$K$ documents

$R :=$ empty  // $R$: the current top-$K$ result list
$S :=$ empty  // $S$: the set containing the accessed documents but not in the result list

$lattice\_vector :=$ empty  // $lattice\_vector$: a set containing the information of which documents have been seen in which lists

$S_k := 0$  // $S_k$: the score of the $k$-th document in $R$.
$\text{Lb}[i] :=$ the lowest score of the parts that have been accessed in list $i$;
$\text{out\_best} := \text{Lb}[0] + \text{Lb}[1] + \ldots + \text{Lb}[t-1]$  
$\text{to\_shrink} :=$ true

loop
  if $\text{to\_shrink}$
    access each inverted list in parallel, do merge-sort with previous accessed documents inside each single list
    update $\text{Lb}[i]$ for each inverted list $IL_i$
    $\text{out\_best} := \text{Lb}[0] + \text{Lb}[1] + \ldots + \text{Lb}[t-1]$  
    do merge-sort in between all inverted lists, for each accessed document $d$
    if $R$.size() < $K$ or $S_k < d$.score
      $R$.insert($d$)
    update $S_k$
  end-if
  update $S$

  // check the shrinking condition
  if $S_k \geq \text{out\_best}$
    use $S$ to construct $lattice\_vector$
    $\text{to\_shrink} :=$ false

  // check the termination condition right after the lattice being constructed
  $lattice\_vector[m]$.best := the document of the best score in that node
  if $lattice\_vector[m]$.best $\leq S_k$ for all $m$
```
return R
end-if
end-if
else
clear the previous buffered inside-merged documents in all inverted lists
access each inverted list to the next part in parallel, do merge-sort in between all lists, for each accessed document d
if R.contains(d)
    update d.score
    update Sk
end-if
if S.contains(d)
    update d.score
    if d.score >= Sk
        R.insert(d)
        update Sk
        update S
    end-if
    update lattice_vector
end-if
//check the termination condition
lattice_vector[m].best := the document of the best score in node m
if lattice_vector[m].best <= Sk for all m
    return R
end-if
end-if
end-loop
return R

Figure 14: The hybrid-I method

Here we give a sample to illustrate the hybrid-I method. The inverted index used here is the reorganized index shown in Figure 10 and we set K to be 1. After the accessing to the first part of each list, we obtain $R = \{(a, 0.9)\}$, $S = \{(b, 0.6), (c, 0.7), (d, 0.7), (e, 0.9), (h, 0.9)\}$. $Sk = 0.9$ and $out_best = 0.7 + 0.6 + 0.9 = 2.2$. Now the shrinking condition $Sk >= out_best$ is false. Thus,
the second part of each list should be accessed. In the second iteration, we first sort the postings by docID inside each list. The index buffer we get after inside sorting should be like this:

<table>
<thead>
<tr>
<th>Index Buffer in Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL₁</td>
</tr>
<tr>
<td>c, 0.6</td>
</tr>
<tr>
<td>d, 0.7</td>
</tr>
<tr>
<td>e, 0.6</td>
</tr>
<tr>
<td>h, 0.9</td>
</tr>
</tbody>
</table>

After merge-sort in-between inverted lists, we get \( R = \{< d, 1.9 >\} \), \( S = \{< a, 0.9 >, < b, 0.6 >, < c, 1.2 >, < e, 1.5 >, < f, 0.9 >, < g, 0.2 >, < h, 0.9 >\} \), \( Sk = 1.9 \) and \( out_best = 0.6 + 0.2 + 0.7 = 1.5 \). Now \( Sk > out_best \) and thus we can access the shrinking phase. Since this is the first access to the shrinking phase, the lattice_vector should be constructed by \( S \). The lattice_vector structure is shown in the second graph in Figure 13 except that the posting \( < d, 1.9 > \) is not in ellipse \( IL₁IL₂IL₃ \) now since it is in the result set \( R \). Then, we calculate the best score of each lattice node. \( lattice_vector[IL₁IL₂]best = c.score + Lb[IL₃] = 1.2 + 0.7 = 1.9 \). In this way, we may calculate all the best scores for all lattice_vector nodes, and the values are \{1.7, 1.8, 1.9, 1.7\}. None of the best scores exceeds \( Sk \), so the document \( d \) can be returned as the top-1 result.

**Hybrid-II Method**

The hybrid-I method is a somewhat complicated, since our reorganized inverted index is sorted by document IDs within each part, we think of simply calculate the maximal possible score during the process of merging and we call this “hybrid-II” method. After implementing this method, we’ll use experimental results to compare it with the hybrid-I to see whether the hybrid-I method worth the bother of complication to outperform hybrid-II.
In this section, we will explain the hybrid-II method. First look at the pseudo code of the hybrid-II method in Figure 15. In this method, we don’t use container since the calculation of the maximal possible are calculated at the same time with merge-sort in-between inverted list. Each time all new parts of all inverted lists are accessed in parallel. First merge the new part with the previous parts within each list, and then simply do the merge sort in-between all involved lists. This process is similar to the Merge-sort algorithm introduced in the previous chapters, the only difference is that the calculation of the maximal possible score will demand additional cost. We initialize the maximal possible score $out_best$ by zero. During the merging process, if the document is not chosen as a candidate and its maximal possible score exceeds $out_best$, then $out_best$ is updated to the bigger one. Thus, after the merging process, we get the biggest possible outside the candidate set and can use this value to check the termination condition. If the $k$-th score in the candidate set is no less than $out_best$, the algorithm can terminate and the result can be returned.

**Algorithm: Hybrid-II**

- **input:** Inverted lists $IL_1, IL_2, \ldots, IL_t$, for the query $Q$, composed of $t$ terms
- **output:** Top-$K$ documents

\[
\begin{align*}
R := & \text{empty} \quad // R: \text{the current top-$K$ result list} \\
Sk := & 0 \quad // Sk: \text{the score of the $k$-th document in } R. \\
// & Lb[i]: \text{the lowest score of the parts that have been accessed in list } i \\
Lb[i] := & \text{the lowest score of } IL_i \text{'s first part} \\
out_best := & 0 \\
\text{loop} \\
& \text{access each list } Li \text{ in parallel, do merge-sort within each single list.} \\
& \text{update } Lb[i] \text{ for each list} \\
& \text{out_best := } Lb[0] + Lb[1] + \ldots + Lb[t-1] \\
& \text{do merge-sort in-between all the inverted lists, for each current} \\
& \text{accessed document } d \\
& \text{if } R.\text{size() } < K \\
& \quad R.\text{insert}(d) \\
& \quad \text{update } Sk \\
& \text{else if } d.\text{score } > Sk \\
& \quad \text{pop}_d := R.\text{pop()} \quad // \text{pop}_d: \text{the popped document from the result set } R \\
\end{align*}
\]
update pop_d's best score pop_best
R.insert(d)
update Sk
if out_best < pop_best
    out_best := pop_best
end-if
else
    d.maximal := d.score + \sum_{i \text{ is the list where } d \text{ has not been accessed}} \text{LB}[i]
    if out_best < d.maximal
        out_best := d.maximal
    end-if
endif

// if the out_best is worse than the k-th score in R, then R can be returned
if Sk >= out_best
    return R
end-if
end-loop
return R

Figure 15: The hybrid-II method

Now we use the same reorganized index and aim to choose top-1 result by hybrid-II method. In the first iteration, the hybrid-II method initialize out_best to be the sum of the lowest scores of all lists, thus we get out_best = 0.7 + 0.6 + 0.9 = 2.2 before the merging process. In the merging process, we calculate the maximal possible score of each document that is not in the result set R according to its bitmap. For example, at the very beginning R=\{<a,0.9>\} and out_best = 2.2. Then document b is accessed, since its score is less than 0.9, we need to calculate its maximal possible score b.maximal = b.score + \text{LB}[1] + \text{LB}[3] = 0.6 + 0.7 + 0.9 = 2.2. Because b.maximal is no bigger than out_best, out_best need not be updated. After the merging the first part of all lists, we get R=\{<a,0.9>\}, Sk = 0.9 and out_best = 2.4. Since Sk < out_best, we access the second part and do the merge sort in the similar way of the first iteration. At the end of the second iteration, we get R=\{<d,1.9>\}, Sk = 1.9 and out_best = 1.8. So document d can be returned as our top-1 result.
CHAPTER 5  EXPERIMENTAL RESULTS

5.1  Experimental Setup

✓ Datasets and query sets: For our experiments, we use the ICWSM 2009 Spinn3r Blog Datasets. ICWSM is a dataset available to researchers in the blog and social media fields. The dataset, provided by Spinn3r.com, is a set of 44 million blog posts made between August 1st and October 1st, 2008. The total size of the dataset is 142 GB uncompressed, (27 GB compressed). We randomly chose 100 pairs of queries, most of which are of high correlations and very few with low correlations and we restrict $K$ to $K \in \{1,5,10,20,40,80,160,320,640,1280,2560,5120\}$.

✓ Hardware and software environment: The experiments are carried out on a single machine with Intel® Core™2 Quad CPU Q6600 @ 2.40GHz, 3.25 GB RAM and 100GB local NTFS disk.

✓ Evaluation metrics: We use the response time to evaluate both the partition and the retrieval methods. We first compare the deviation partition method with the naïve equal partition method by looking at their response time for each separate retrieval approach (the hybrid-I method and the hybrid-II method). Then we choose the superior partition method to build the inverted index and based on that new index to analyze and compare the two retrieval approaches, by the average response time of the 100 query pairs.
5.2 Experimental Results

➢ Tuning Parameters of Partition Methods

Recall that in the deviation-based partition method, we use a tuning parameter $\alpha \in (0,1)$ to tune the window size. Thus, either to compare our deviation-based partition method with the naïve equal partition method, or to compare the two hybrid methods, we need to try different tuning parameter $\alpha$ in the deviation-based partition method and to try different partition number $P$ in the equal partition method to build the inverted index and chose the best tuning parameters $\alpha$ and $P$ for each evaluation method (hybrid-I and hybrid-II). We choose a default $K=320$ to do the tuning.

Figure 16 shows the average response time for hybrid-I and hybrid-II method each on different inverted index built by the deviation-based method with varying tuning parameter $\alpha$ and the equal partition method with different partition number $P$. It is shown in the first graph that when $\alpha = 0.8$, both the hybrid-I method and the hybrid-II method cost the least response time and thus performs best respectively. In the second graph, we find that partition 10 is the best for hybrid-I method while partition 5 is the best for hybrid-II. Thus, all the following experiments are based on the best tuned parameter $\alpha = 0.8$, $P=10$ for hybrid-I and $P=5$ for hybrid-II.
Figure 16: Turning parameter to get the best index partition of both equal and deviation-based partition for each method (hybrid-I & hybrid-II).

Partition Method Comparison

Firstly, we need to compare our deviation-based partition method with the naïve equal partition method and the tuning parameters are set by the previous experiment. Figure 17 shows the average response time of the 100 queries with both the hybrid-I retrieval and the hybrid-II retrieval method for only the equal partition method. In the horizontal axis, the integer “1, 5, ... , 5120” represents the required top-K answers while the vertical axis is the average response time in logarithmic scale with base 2.

In either retrieval approach, we find that the deviation-based partition method performs better than the equal partition method not only for our default $K (=320)$, but for most $K$. That means no matter in which retrieval methodology, our deviation-based partition method outperforms the naïve equal partition method in general. And in the following experiment, we’ll adopt the reorganized inverted index constructed by the best tuned deviation method.
Figure 17: Partition Method Comparison in hybrid-I and hybrid-II evaluation method

**Retrieval Approaches Comparison**

Now that we have demonstrated that the deviation-partition is better than the naïve equal partition, our later experiments of the retrieval approaches are carried out on the inverted index built by the deviation-based partition method.

The first experiment in this section aims to compare our proposed approaches, both the hybrid-I and the hybrid-II method, with the old ones. After that, we would like to compare the hybrid-I method and the hybrid-II method to explain which one is better.

Figure 18 shows the average query response time of the LARA method, Merge-sort method, Exhaust method, Hybrid-I method and the hybrid-II method for...
The first graph compares the whole query set, the second graph compares the query set of high correlated queries and the third graph compares the queries of low correlations. It is obvious that in either of the three query sets, all other four approaches are superior than the exhaustive approach, which simply traverse the whole inverted index and its performance get extremely worse when the index size becomes large. On the other hand, although LARA can “stop early”, which means it should avoid accessing the whole index, it only outperforms the merge-sort algorithm when $K$ is small. This is because updating the lattice structure at each access is incredibly expensive. It turns out that both our hybrid-I and hybrid-II method on the rebuilt index are far more efficient than all the studied retrieval methodologies performed on the classic docID sorted or score sorted index. At this moment, we haven’t seen big difference between the hybrid-I method and the hybrid-II method. Thus, we need to further compare these two methods.
In the next experiment, we execute the two retrieval methodologies on their own best partitioned index, compare their performance and make a decision which method is preferred.

Figure 18: Overall Comparison of our IR approaches with the reviewed ones
Figure 19: Overall comparison of the hybrid-I method and the hybrid-II method

Figure 19 shows the response time of the two algorithms for the set of $K$ specified in the experimental section. The graph indicates the performance trend of the two methods with the growth of $K$. When $K$ is less than 160, the efficiency of the two algorithms interchanges with each other, and there’s no big difference between them. However, when $K$ exceeds 160 and becomes larger, the hybrid-II method’s response time is in excess of that of the hybrid-I method, and the efficiency of the hybrid-II method tends to become increasingly worse.

Finally, we look at the performance of hybrid-I and hybrid-II for different correlated query sets. Here we use two query sets, each containing 20 queries. The average correlation of one set is 0.04, while the other set’s average correlation is 0.004.
Figure 20: Comparison of the hybrid-I and the hybrid-II method for different correlated query sets

Figure 20 presents the performance of hybrid-I and hybrid-II for the query sets with high correlation and low correlation. From this figure, we observe that in either set, hybrid-I performs better than hybrid-II in general, especially when $K$ becomes large, which means hybrid-I is better than hybrid-II for queries of either high or low correlations.

The reason for this result is that each time the hybrid-II method gets a new partition of the inverted index, it has to merge the new part with the sorted previous parts, and what’s more, the maximal possible score outside the candidate set has to be recalculated from the beginning of the index. No matter how fast merge-sort is, the cost of re-merging the index, re-choosing the top-$K$ candidates, and re-calculating the maximal possible score over and over is inevitably high.

Thus, it is worthwhile to implement and adopt the hybrid-I method in spite of its complication.
In this work, we have proposed a new method for information retrieval using an overall score-sorted, partial docID-sorted index and two appropriate retrieval strategies on the new index. The compelling experimental evidence of the great improvement in query response time strongly demonstrates the superior of our new method over the state-of-the-art methodologies.

There’re still many things can be improved. On the one hand, we will further study the characteristics of specific inverted lists and try to propose more accurate partition method.

On another hand, as discussed in literature review, the document-query relevance should be decided by both document global score and document-term local score. Our methodology only takes the local score into account at this moment. In future, we’ll further study how to improve the query efficiency when document global scores are considered in evaluating the document-query similarity.
BIBLIOGRAPHY


Academic Qualifications

- B.Sc. in Software Development and Application, Macau University of Science and Technology, Macau (2008).