國立交通大學

資訊工程學系

博士論文

大型資訊檢索系統之轉置檔案設計

Inverted File Design for Large-Scale Information Retrieval System

研究生：鄭哲聖

指導教授：單智君 教授

中華民國九十四年八月

大 型 資 訊 檢 索 系 統 之 轉 置 檔 案 設 計
Inverted File Design for Large-Scale Information Retrieval System

研 究 生：鄭哲聖　　Student：Cher-Sheng Cheng

指導教授：單智君 教授　　Advisor：Prof. Jean Jyh-Jiun Shann

國 立 交 通 大 學

資 訊 工 程 學 系

博 士 論 文

A Dissertation

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science and Information Engineering

Aug 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年八月

國立交通大學

資訊工程系博士班

論文口試委員會審定書

本校 資 訊 工 程 系 鄭哲聖 君

所提論文 大 型 資 訊 檢 索 系 統 之 轉 置 檔 案 設 計

合於博士資格水準、業經本委員會評審認可。

口試委員：蔡尚學 李素瑛

鍾葆斌 單智君

[illegible] 陳淳誾

郭大弘

指導教授：單智君

系 主 任：[illegible]

中華民國 九十四 年 八 月 三十 日

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

Date: Aug 30, 2005

We have carefully read the dissertation entitled **Inverted File Design for Large-Scale Information Retrieval System** *submitted by* **Cher-Sheng Cheng** *in partial fulfillment of the requirements of the degree of* Doctor *of* Philosophy *and recommend its acceptance.*

Thesis Advisor: _____

Chairman: _____

國 立 交 通 大 學

博碩士論文全文電子檔著作權授權書

本授權書所授權之學位論文，為本人於國立交通大學資訊工程學系所____組，94 學年度第 1 學期取得博士學位之論文。

論文題目：大型資訊檢索系統之轉置檔案設計

指導教授：單智君 教授

■ 同意 □不同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：	
本校及台灣聯合大學系統區域網路	■中華民國 95 年 9 月 9 日公開
校外網際網路	■中華民國 96 年 9 月 9 日公開

授 權 人：鄭哲聖

親筆簽名： 鄭哲聖

中華民國 94 年 09 月 09 日

國 立 交 通 大 學

博碩士紙本論文著作權授權書

本授權書所授權之學位論文，為本人於國立交通大學資訊工程學系所 ____組，94 學年度第 1 學期取得博士學位之論文。

論文題目：大型資訊檢索系統之轉置檔案設計

指導教授：單智君 教授

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：_____，請將論文延至____年____月____日再公開。

授 權 人：鄭哲聖

親筆簽名： 鄭哲聖

中華民國 94 年 09 月 09 日

國家圖書館博碩士論文電子檔案上網授權書

ID:GT008517802

本授權書所授權之學位論文，為本人於國立交通大學資訊工程學系所____組，94學年度第1學期取得博士學位之論文。

論文題目：大型資訊檢索系統之轉置檔案設計

指導教授：單智君 教授

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：鄭哲聖

親筆簽名： 鄭哲聖

民國 94 年 09 月 09 日

1. 本授權書請以黑筆撰寫，並列印二份，其中一份影印裝訂於附錄三之二(博碩士紙本論文著作權授權書)之次頁；另一份於辦理離校時繳交給系所助理，由圖書館彙總寄交國家圖書館。

誌 謝

這本論文得以順利完成，首先要感謝我的指導教授單智君老師以及實驗室的鍾崇斌老師。由於兩位老師在研究上的嚴格督促，以及在生活上的鼓勵與關懷，協助我克服許多研究上的困難並順利完成這本論文。還要感謝口試委員蔡尚榮教授、李素瑛教授、邱舉明教授、陳添福教授、和郭大維教授諸多的寶貴意見與指正，使得本這本論文的內容能更加完整。

感謝盧能彬、邱日清、江衍源、徐日明、林光彬、馬詠程、謝萬雲、喬偉豪等學長與同學，在各方面對我的啟發與協助。也感謝實驗室中的每一個成員與歷屆學弟妹在我研究生涯中所給予的幫助與鼓勵，支持我度過這漫長而充實的求學歲月。

感謝父母親與岳父母對我的諄諄教誨，因為您們的關心與包容，才讓我可以無後顧之憂地完成學位。最後感謝我的妻子葉妍伶，謝謝妳一路走來始終如一的鼓勵與支持，讓我可以全心地完成這本論文。

謹以此論文，獻給各位。

鄭哲聖 2005 年 8 月

大 型 資 訊 檢 索 系 統 之 轉 置 檔 案 設 計

研究生：鄭哲聖　　指導教授：單智君教授

國立交通大學資訊工程學系

摘要

本論文主要在探討各種改善資訊檢索效率的技術。最近幾年來，資訊檢索系統已廣泛地使用於各種應用中，例如：搜尋引擎、數位圖書館、基因序列分析等。為了在大量的資料中有效率地搜尋，資訊檢索系統採用已壓縮的轉置檔案來迅速地找到使用者所需要的資料。在轉置檔案中，每一個字彙都有一個相對應的文件編號串列(稱為轉置串列)來指示那一個文件包含這個字彙。大型資訊檢索系統的查詢處理時間大多花在讀取與解壓縮各個出現在查詢中的字彙所對應到的轉置串列。由於每新增一個文件就會使得出現在文件中的字彙所對應的轉置串列長度增加，因此轉置串列的長度與文件數目呈現正比關係。這意味著查詢處理時間與文件數目亦呈現正比關係。所以，發展有效率的演算法來降低轉置串列的處理(讀取、解壓縮、與合併)時間就成了設計大型資訊檢索系統的成功關鍵。

本論文將探討下列的研究議題：

(1) 發展一個有效率的編碼方法來縮減轉置檔案所佔用的空間

在這個議題中我們透過轉置檔案的壓縮來減少磁碟的輸出與輸入所需的時間並藉以改善查詢處理時間，但這卻會帶來額外的解壓縮時間。本議題的目標即是設計一個可節省空間並可快速解碼的方法來壓縮轉置檔案。我們以壓縮率最高的內插編碼法為基礎，採用遞迴移除與迴圈展開的技巧來加速內插編碼法的編碼與解碼速度。實驗顯示與其他已知的編碼法比較，我們所提的方法提供了快速的解碼與良好的壓縮效能。

(2) 發展雙層可跳躍式轉置檔案來除去多餘的解碼

在這個議題中我們提出一個雙層可跳躍式轉置檔案來減少查詢處理時所需的轉置串列解壓縮時間。這個議題所面臨的困難是利用目前的可跳躍式機制來實作雙層可跳躍式轉置檔案所需耗用的空間太大。為了設計一個節省空間並可有效加速查詢處理的雙層可跳躍式轉置檔案，在以區塊空間預估為基礎，我們發展了一個新的跳躍機制。這個機制可以

搭配目前已知的可跳躍式機制在相當小的空間耗用下實作雙層可跳躍式轉置檔案。實驗顯示我們所提的雙層可跳躍式轉置檔案可以同時加速連接式布林查詢與排名查詢。

(3) 利用文件編號來使得轉置檔案最佳化

在這個議題中我們提出一個文件編號演算法以加速查詢地處理速度。我們觀察到透過指派合適的編號給文件可以讓轉置串列在使用相同的編碼方法下被壓縮的更好，並提升查詢處理的速度。本議題我們提出一個新的演算法，稱為 *Partition-based document identifier assignment* (PBDIA) 演算法，來為文件產生合適的編號。這個演算法可以有效率地指派連續的編號給那些包含有經常被查詢的字彙之文件，使得經查被查詢的字彙之轉置串列可以被壓縮得更好。實驗顯示我們所提的 PBDIA 演算法可以有效縮短查詢處理時間。

(4) 發展平行資訊檢索系統上的轉置檔案切割方法

在這個議題中我們針對平行資訊檢索系統提出一個轉置檔案切割方法。叢集系統利用分散在各工作站上的轉置檔案，以平行計算的方式處理查詢。此演算法的目的是降低處理查詢地平均時間。我們首先採用 PBDIA 演算法讓包含經查被查詢的字彙之文件可以被指派連續的編號。然後，再採用交錯式切割方案來分配轉置檔案到各工作站上。實驗顯示利用此步驟切割轉置檔案，可以達到負載與儲存量的平衡，以及幾近理想的速度提升。

本論文之研究成果包括：

- 在縮減轉置檔案所佔用的空間方面，我們所提出的編碼方法除了可提供優越的壓縮效果外，在查詢處理速度上也比目前最常使用的Golomb coding還快了大約30%。
- 在除去多餘的解碼方面，我們所提出的雙層可跳躍式轉置檔案比起目前的單層可跳躍式轉置檔案在連接式布林查詢的處理速度上最高可提升16%，而在排名查詢的處理速度上最高可提升44%。
- 在轉置檔案最佳化方面，我們所提出的PBDIA演算法可以在數秒的時間內為1GB大小的文件集產生合適的文件編號並使得查詢處理速度最高可提升25%。
- 在平行資訊檢索方面，我們所提出轉置檔案切割步驟可以改善只使用交錯式切割方案的平行查詢處理速度達14%到17%，無論一個叢集有多少台工作站。

Inverted File Design for Large-Scale Information Retrieval System

Student: Cher-Sheng Cheng Advisor: Prof. Jean Jyh-Jiun Shann

Department of Computer Science and Information Engineering
National Chiao Tung University

Abstract

This dissertation investigates a variety of techniques to improve efficiency in *information retrieval* (IR). *Information retrieval systems* (IRSs) are widely used in many applications, such as search engines, digital libraries, genomic sequence analyses, etc. To efficiently search vast amount of data, a compressed inverted file is used in an IRS to locate the desired data quickly. An inverted file contains, for each distinct term in the collection, a posting list. The query processing time of a large-scale IRS is dominated by the time needed to read and decompress the posting list for each query term. Moreover, adding a document into the collection is to add one document identifier into the posting list for each term appearing in the document, hence the length of a posting list increases with the size of document collection. This implies that the time needed to process posting lists increase as the size of document collection grows. Therefore, efficient approaches to reduce the time needed to read, decompress, and merge the posting lists are the key issues in designing a large-scale IRS. Research topics to be studied in this dissertation are

(1) Efficient coding method for inverted file size reduction

The first topic is to propose a novel size reduction method for compressing inverted files. Compressing an inverted file can greatly improve query performance by reducing disk I/Os, but

this adds to the decompression time required. The objective of this topic is to develop a method that has both the advantages of compression ratio and fast decompression. Our approach is as follows. The foundation is interpolative coding, which compresses the document identifiers with a recursive process taking care of clustering property and yields superior compression. However, interpolative coding is computationally expensive due to a stack required in its implementation. The key idea of our proposed method is to facilitate coding and decoding processes for interpolative coding by using recursion elimination and loop unwinding. Experimental results show that our method provides fast decoding speed and excellent compression.

(2) Two-level skipped inverted file for redundant decoding elimination

The second topic is to propose a two-level skipped inverted file, in which a two-level skipped index is created on each compressed posting list, to reduce decompression time. A two-level skipped index can greatly reduce decompress time by skipping over unnecessary portions of the list. However, well-known skipping mechanisms are unable to efficiently implement the two-level skipped index due to their high storage overheads. The objective of this topic is to develop a space-economical two-level skipped inverted file to eliminate redundant decoding and allow fast query evaluation. For this purpose, we propose a novel skipping mechanism based on block size calculation, which can create a skipped index on each compressed posting list with very little or no storage overhead, particularly if the posting list is divided into very small blocks. Using a combination of our skipping mechanism and well-known skipping mechanisms can implement a two-level skipped index with very little storage overheads. Experimental results showed that using such a two-level skipped index can simultaneously allow extremely fast query processing of both conjunctive Boolean queries and ranked queries.

(3) Document identifier assignment algorithm design for inverted file optimization

The third topic is to propose a *document identifier assignment* (DIA) algorithm for fast query evaluation. We observe that a good DIA can make the document identifiers in the posting lists more clustered, and result in better compression as well as shorter query processing time. The objective of this topic is to develop a fast algorithm that finds an optimal DIA to minimize the average query processing time in an IRS. In a typical IRS, the distribution of query terms is skewed. Based on this fact, we propose a *partition-based DIA* (PBDIA) algorithm, which can efficiently assign consecutive document identifiers to those documents containing frequently used query terms. Therefore, the posting lists for frequently used query terms can be compressed better without increasing the complexity of decoding processes. This can result in reduced query processing time.

(4) Inverted file partitioning for parallel IR

The fourth topic is to propose an inverted file partitioning approach for parallel IR. The inverted file is generally partitioned into disjoint sub-files, each for one workstation, in an IRS that runs on a cluster of workstations. When processing a query, all workstations have to consult only their own sub-files in parallel. The objective of this topic is to develop an inverted file partitioning approach that minimizes the average query processing time of parallel query processing. Our approach is as follows. The foundation is interleaving partitioning scheme, which generates a partitioned inverted file with interleaved mapping rule and produces a near-ideal speedup. The key idea of our proposed approach is to use the PBDIA algorithm to enhance the clustering property of posting lists for frequently used query terms before performing the interleaving partitioning scheme. This can aid the interleaving partitioning scheme to produce superior query performance.

The results of this dissertation include:

- For inverted file size reduction, the proposed coding method allows query throughput rate of approximately 30% higher than well-known Golomb coding and still provides superior compression.
- For redundant decoding elimination, the proposed two-level skipped inverted file improves the query speed for conjunctive Boolean queries by up to 16%, and for ranked queries up to 44%, compared with the conventional one-level skipped inverted file.
- For inverted file optimization, the PBDIA algorithm only takes a few seconds to generate a DIA for a collection of 1GB, and improves query speed by up to 25%.
- For parallel IR, the proposed approach can further improve the parallel query speed for interleaving partitioning scheme by 14% to 17% no matter how many workstations are in the cluster.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Interest in *information retrieval* (IR) is growing rapidly, and many systems such as search engines, digital libraries, genomic sequence analyses, etc., are developed to efficiently search through terabytes of data and quickly identify the data relevant to the user query. One of the major problems faced by those systems is that the information explosion overwhelms the load of CPU and disk on an information retrieval server. For example, the size of Web has doubled in less than two years (Lawrence & Giles, 1999). This requires using parallel architectures to speed up search. Recently, cluster computing has revived the field of parallelism for IR. This dissertation proposes methodologies to improve the efficiency of an IRS that runs on a cluster of workstations. Efficiency here means that queries are processed faster without upgrading the hardware or the same throughput is achieved by a smaller machine configuration. The key idea is developing efficient algorithms to reduce space and time needed to store and operate on the most-widely-used indexing structure, called the inverted file. The objective is to increase the efficiency of an IRS without increasing the hardware cost of the cluster. To achieve the objective, this dissertation deals with inverted file size reduction, redundant decoding elimination, inverted file optimization, and parallel IR.

This chapter is outlined as follows. Section 1.1 and Section 1.2 present research background and research objectives. Section 1.3 presents an overview on all research topics in this dissertation. Section 1.4 presents the organization of this dissertation.

1.1 Background: IRS Runs on Cluster of Workstations

Parallel computing hardware has been used extensively to increase the data handling and query handling capacity of IRSs. Recently, the *Multiple Instruction Multiple Data* (MIMD) model of

parallelism, implemented as a cluster of workstations, has become the dominant parallel IR architecture. Inktomi, FAST, and Google are all understood to use it.

In this dissertation, we intend to reduce query processing time of an IRS by using a cluster as the server architecture. The cluster consists of identical workstations − each has its own CPU, memory, and disk − interconnected by a local area network (cf. Figure 1.1). Such an IRS works as follows. Each query is broadcast to all workstations in the cluster and each of them processes the query over the index for the piece of the collection for which they are responsible. The workstation may need to communicate with each other to exchange global statistical information. They definitely need to communication with each other to form merged results.

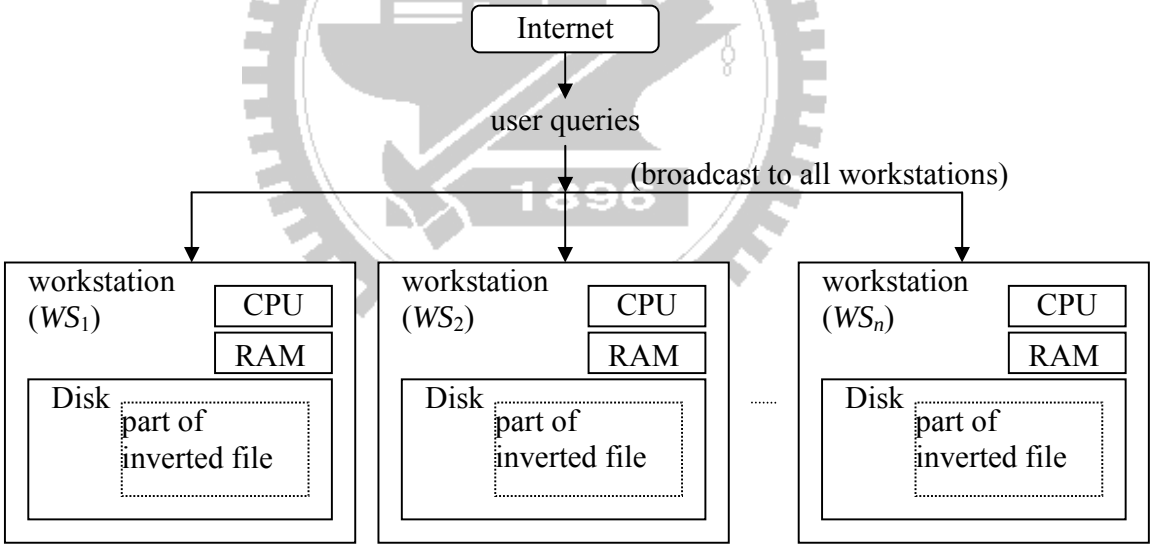


Figure 1.1 The concerned clustered architecture.

A specific data structure, called “inverted index”, is consulted to find answers for a query (cf. Figure 1.2). An inverted index consists of an index file and an inverted file. An index file is a set of records, each containing a keyword term *t* and a pointer to the posting list for term *t*. An inverted file contains, for each distinct term *t* in the collection, a posting list of the form

$$PL_t = <id_1, id_2, \ldots, id_{f_t}>,$$

where id_i is the identifier of the document that contains t, and frequency f_t is the number of documents in which t appears. The document identifiers are within the range 1...N, where N is the number of documents in the indexed collection. For ranking evaluation, each id_i may be stored with a within-document frequency fq_i to indicate term t appears in the document id_i a total of fq_i times. In a large document collection, posting lists are usually compressed, and decompression of posting lists is hence required during query processing.

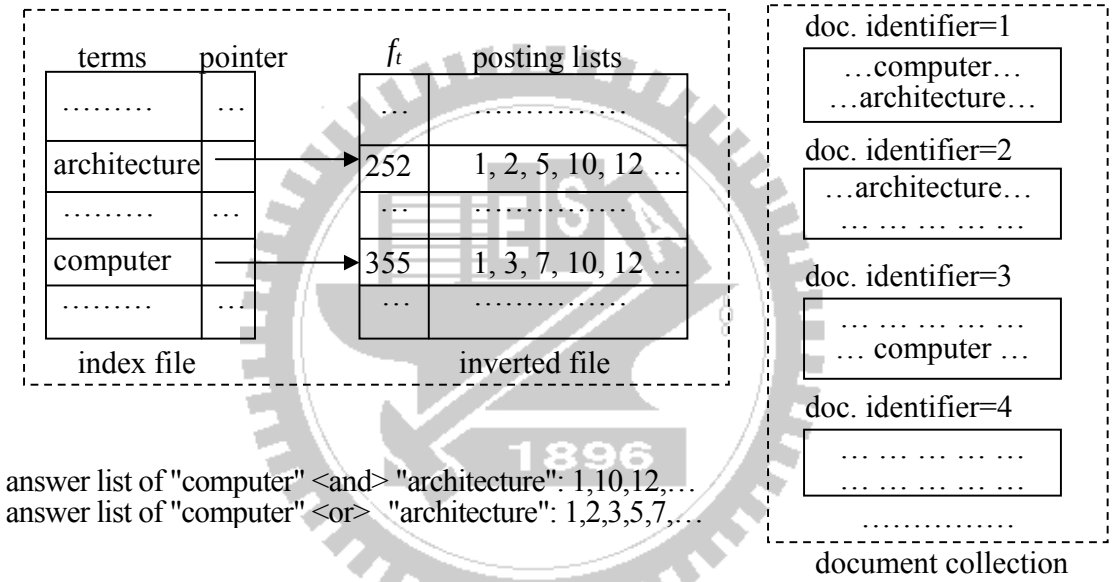


Figure 1.2 Inverted index and document collection.

In a typical IRS, a few frequently used query terms constitute a large portion of all term occurrences in queries (Jansen et al., 1997). This suggests that it is advisable to store the index records for frequently used query terms in RAM to greatly reduce index search time. Hence, the significant portion of query processing time is to read and decompress the compressed posting list for each query term. This paper restricts attention to inverted file side only and investigates the efficient approaches to reduce space and time needed to store and operate on the inverted file and improve the overall *IR* performance.

The major challenges imposed by very large scale IR (particularly on World Wide Web) are:

1. For a large-scale IRS, the access time and storage space of an inverted file become considerably large (Rillof & Hollaar, 1996; Baeza-Yates & Ribeiro-Neto, 1999). The challenge is how to improve IR performance while reducing storage requirements for a large inverted file.
2. As a document collection grows, the number of occurrences of common terms is likely to increase in proportion. This means that posting lists for common terms will be longer, increasing processing time during query processing. The challenge is how to speed up query processing by skipping over unnecessary portions of the lists without degrading retrieval effectiveness.
3. For an IRS running on a cluster of workstations, an inverted file should be partitioned and distributed onto disks of multiple workstations. The challenge is how to partition the inverted file such that, during query processing, all workstations have to consult only their own local portion of the partitioned inverted file in parallel and obtain high parallel efficiency.

1.2 Objective: Inverted File Design for Large-Scale Information Retrieval System

The objective of this dissertation is to increase the efficiency of an IRS without increasing the hardware cost of the cluster by developing efficient algorithms that reduce the time needed to read, decompress, and merge posting lists for query terms. To achieve our research objective, we investigate several issues as follows:

● Inverted file size reduction

Since large inverted files demand greater I/O to read them, the size directly affects the processing time. To solve problems such as the slow response time and the large disk space required in large

scale IRSs, a coding method with fast decoding and good compression should be developed. We notice that in an inverted file the document identifiers for a given word are usually clustered. If a coding can take advantage of clustering property, excellent compression can be achieved. However, the mechanisms of decoding for all well-known coding methods that can exploit clustering property well are more complex, which reduce the ability of searching performance at some degree. Therefore, the key to this issue is to develop a new coding method that can exploit clustering property well and allow extremely fast decompression.

- Redundant decoding elimination

The query performance on a compressed inverted file can be improved by using skipping mechanisms (Moffat et al., 1995; Moffat & Zobel, 1996; Anh & Moffat, 1998). Although compression can greatly reduce disk access time, the compressed posting list for each query term must be completely decompressed in order to be randomly accessed to any posting in it. When processing queries, it is usually that only a subset of the postings in a posting list needs to be examined. To remove redundant decoding, skipping mechanisms (Moffat et al., 1995; Moffat & Zobel, 1996; Anh & Moffat, 1998) that allow queries to be processed with only partial decoding of the list have been proposed. A common technique of skipping mechanisms is to divide the posting list into blocks and add auxiliary information into each block, so that the postings within a block can be quickly skipped without decoding them if they are useless in set operations during query processing. There are two important types of queries: conjunctive Boolean queries and ranked queries. For conjunctive Boolean queries large blocks provide faster searching for candidates, whereas for ranked queries small blocks do (Moffat & Zobel, 1996; Anh & Moffat, 1998). Although all well-known skipping mechanisms can work well for large blocks, we observe that they can incur high storage overheads if the posting lists are divided into small

blocks. The increase in disk I/O time outweighs the reduction in decompression time. Therefore, the key to this issue is developing a novel skipping mechanism that can support small blocks with very little storage overhead should be developed.

- Inverted file optimization

  The query processing time in a large-scale *IRS* is dominated by the time needed to read and decompress the posting lists for the terms involved in the query (Moffat & Zobel 1996), and we observe that the query processing time grows with the total encoded size of the corresponding posting lists. This is because the disk transfer rate is near constant, and the decoding processes of most encoding methods used for compressing inverted files are on a bit-by-bit basis. If we can reduce the total encoded size of the corresponding posting lists without increasing decompression times, a shorter query processing time can be obtained. A *document identifier assignment* (DIA) can make the document identifiers in the posting lists evenly distributed, or clustered. Clustered document identifiers generally can improve the compression efficiency without increasing the complexity of decoding process, hence reduce the query processing time. The key to this issue is developing a fast algorithm to finding a near-optimal DIA that reduces the average query processing time in an IRS when the probability distribution of query terms is given.

- Parallel IR

  To process the ever-increasing volume of data while still providing acceptable response times, parallel processing algorithms specifically for IR were developed. The key to this issue is to partition the inverted file into sub-files each for one workstation such that, during query processing, all workstations have to consult their own sub-files in parallel and query processing time can be reduced. To achieve high parallel efficiency, a partitioned inverted file to be distributed on the set of workstations should: (1) eliminate the communication overhead of

transferring postings between workstations during query processing, (2) balance amount of postings to be processed during parallel query processing, and (3) keep compression efficiency in the partitioned compressed inverted file.

1.3 Research Topics

This dissertation proceeds by dealing with the following research topics:

(1) Efficient coding method for inverted file size reduction,

(2) Two-level skipped inverted file for redundant decoding elimination,

(3) Document identifier assignment algorithm design for inverted file optimization, and

(4) Inverted file partitioning for parallel IR.

The first topic is to propose a novel size reduction method for compressing inverted files. Compressing an inverted file can greatly improve query performance by reducing disk I/Os, but this adds to the decompression time required. The objective of this topic is to develop a method that has both the advantages of compression ratio and fast decompression. Our approach is as follows. The foundation is interpolative coding, which compresses the document identifiers with a recursive process taking care of clustering property and yields superior compression. However, interpolative coding is computationally expensive due to a stack required in its implementation. The key idea of our proposed method is to facilitate coding and decoding processes for interpolative coding by using recursion elimination and loop unwinding. Experimental results show that our method provides fast decoding speed and excellent compression.

The second topic is to propose a two-level skipped inverted file, in which a two-level skipped index is created on each compressed posting list, to reduce decompression time. A two-level skipped index can greatly reduce decompress time by skipping over unnecessary portions of the list.

However, well-known skipping mechanisms are unable to efficiently implement the two-level skipped index due to their high storage overheads. The objective of this topic is to develop a space-economical two-level skipped inverted file to eliminate redundant decoding and allow fast query evaluation. For this purpose, we propose a novel skipping mechanism based on block size calculation, which can create a skipped index on each compressed posting list with very little or no storage overhead, particularly if the posting list is divided into very small blocks. Using a combination of our skipping mechanism and well-known skipping mechanisms can implement a two-level skipped index with very little storage overheads. Experimental results showed that using such a two-level skipped index can simultaneously allow extremely fast query processing of both conjunctive Boolean queries and ranked queries.

The third topic is to propose a *document identifier assignment* (DIA) algorithm for fast query evaluation. We observe that a good DIA can make the document identifiers in the posting lists more clustered, and result in better compression as well as shorter query processing time. The objective of this topic is to develop a fast algorithm that finds an optimal DIA to minimize the average query processing time in an IRS. In a typical IRS, the distribution of query terms is skewed. Based on this fact, we propose a *partition-based DIA* (PBDIA) algorithm, which can efficiently assign consecutive document identifiers to those documents containing frequently used query terms. Therefore, the posting lists for frequently used query terms can be compressed better without increasing the complexity of decoding processes. This can result in reduced query processing time.

The fourth topic is to propose an inverted file partitioning approach for parallel IR. The inverted file is generally partitioned into disjoint sub-files, each for one workstation, in an IRS that runs on a cluster of workstations. When processing a query, all workstations have to consult only their own sub-files in parallel. The objective of this topic is to develop an inverted file partitioning

approach that minimizes the average query processing time of parallel query processing. Our approach is as follows. The foundation is interleaving partitioning scheme, which generates a partitioned inverted file with interleaved mapping rule and produces a near-ideal speedup. The key idea of our proposed approach is to use the PBDIA algorithm to enhance the clustering property of posting lists for frequently used query terms before performing the interleaving partitioning scheme. This can aid the interleaving partitioning scheme to produce superior query performance.

We show the overview of the research topics in Table 1.1 and the recommended inverted file design flowchart in Figure 1.3.

Table 1.1 The overview of the research topics.

Posting list processing	Topic 1: Inverted file size reduction	Topic 2: Redundant decoding elimination	Topic 3: DIA-based inverted file optimization	Topic 4: Parallel IR
load time	+ + +	–	+	+ + + (parallelized)
decompression time	–	+ + +	+	+ + + (parallelized)
merge time	no change	+ + +	no change	+ + + (parallelized)

Notation: "+": advantage, and "–": disadvantage.

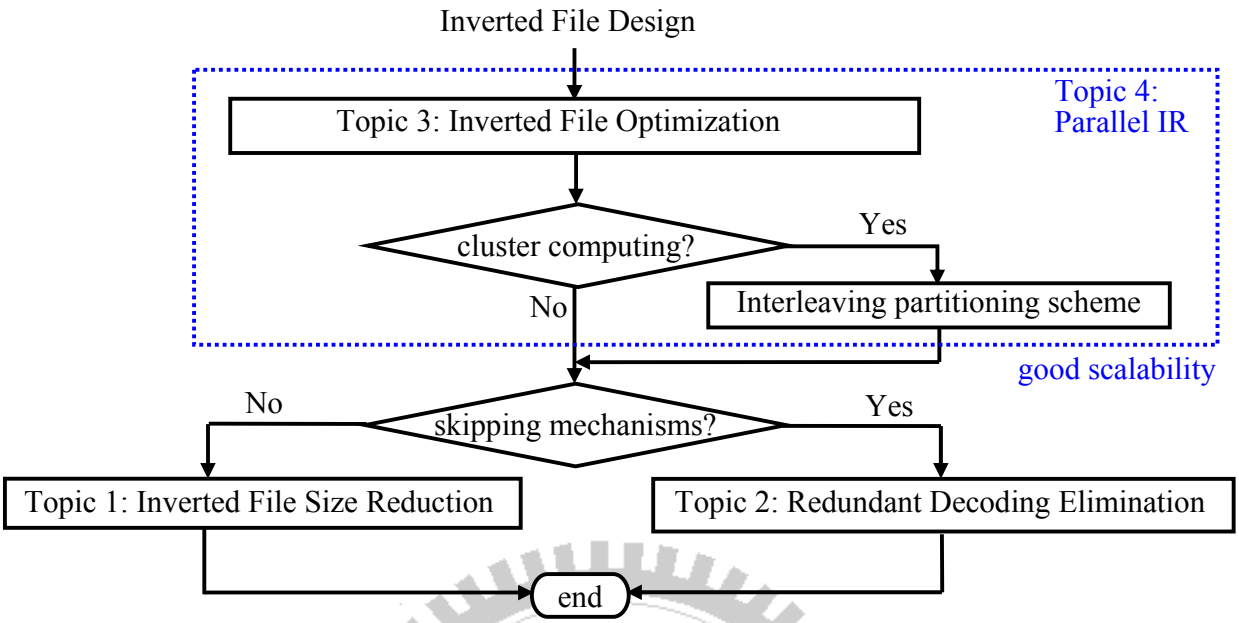


Figure 1.3 Recommended inverted file design flowchart.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a novel size reduction method, which has both the advantages of compression ratio and fast decompression, for compressing inverted files. Chapter 3 presents the proposed two-level skipped inverted file, in which a two-level skipped index is created on each compressed posting list, to reduce decompression time. Chapter 4 presents the proposed DIA algorithm for fast query evaluation. Chapter 5 presents a novel inverted file partitioning approach for parallel IR. Chapter 6 presents the conclusion.

Chapter 2 Inverted File Size Reduction

An inverted file contains, for each distinct term t in the collection, a posting list of the form

$$PL_t = < id_1, id_2, \ldots, id_{ft} >,$$

where id_i is the identifier of the document that contains t, and f_t is the total number of documents in which t appears. To process a query, the IRS retrieves the posting lists for the terms appearing in the query, and then performs some set operations, such as intersection and union, on the posting lists to obtain the answer list (Frankes & Baeza-Yates, 1992; Witten et al., 1999).

Compression of inverted files has significant advantages for large-scale IRSs. This is because the total time of transferring a compressed posting list and subsequently decompressing it is potentially much less than that of transferring an uncompressed posting list. A popular compression technique (Witten et al., 1999) is to sort the document identifiers of each posting list in increasing order, and then replace each document identifier (except the first one) with the distance between itself and its predecessor to form a list of *d*-gaps. For example, the posting list <13, 18, 22, 35, 42> can be transformed into the *d*-gap list as <13, 5, 4, 13, 7>. Although every document identifier is distinct, their *d*-gaps show some probability distributions. Many coding methods, such as unary coding (Elias, 1975), γ coding (Elias, 1975), Golomb coding (Golomb, 1966; Witten et al., 1999), skewed Golomb coding (Teuhola, 1978), batched LLRUN coding (Fraenkel & Klein, 1985; Moffat & Zobel, 1992), variable byte coding (Scholer et al., 2002), and word-aligned "Carryover-12" mechanism (Anh & Moffat, 2005), have been proposed for compressing posting lists by estimates for these *d*-gaps probability distributions. The more accurately the estimate, the greater the compression can be achieved.

The document identifiers for any given word are not uniformly distributed, since the documents in the collection are inserted in chronological order and the word's popularity changes over time (Moffat and Stuiver, 2000). These document identifiers tend to be clustered, and inverted file compression may benefit if this clustering can be taken into account. Based on the *d*-gap technique, some coding methods, such as skewed Golomb coding and batched LLRUN coding, can capture clustering of documents via accurate estimates to achieve satisfactory compression performance. However, the estimates in these methods are relatively sophisticated, which require more decompression time, so they are not yet applied in real IRSs.

Recently, Moffat and Stuiver (2000) have proposed interpolative coding. It is independent of the estimates for the *d*-gaps probability distributions. By using clustering with a recursive process of calculating ranges and codes in an interpolative order, superior compression performance can be achieved. However, interpolative coding is computationally expensive due to a stack required in its implementation, which prohibits it from being widely used in real-world IRSs.

In terms of query throughput rates, Trotman (2003) shows that for small posting lists Golomb coding is recommended, whereas for large posting lists variable byte coding is recommended. Furthermore, Anh and Moffat (2005) show that word-aligned "Carryover-12" mechanism allows a query throughput rate that is higher than Golomb coding and variable byte coding, regardless of the lengths of the posting lists. Although these compression methods provide high query throughput rates, their compression efficiencies need to be improved.

In this chapter, we develop a new coding method based on interpolative coding combined with a *d*-gap compression scheme. We call it the unique-order interpolative coding. The results of this research showed that the unique-order interpolative coding can take advantage of document identifier clustering in posting lists to achieve good compression performance. Nevertheless, the

decoding speed of this new method is even faster than that of Golomb coding and word-aligned "Carryover-12" mechanism.

This chapter is organized as follows. In Section 2.1, we present the interpolative coding that is the most space efficient method known to compress inverted files. In Section 2.2, we present the unique-order interpolative coding. Then we show the quantitative analysis and the performance evaluation in Section 2.3 and Section 2.4. In Section 2.5, we present the possible application of the unique-order interpolative coding. Finally, Section 2.6 presents our summary.

2.1 Well-known Interpolative Coding

2.1.1 Algorithm description

Moffat and Stuiver (2000) have proposed a compression technique called interpolative coding. It makes full use of the clustering in a recursive process of calculating ranges and codes, and demonstrates superior compression performance. In this method, the storing order as well as lower bound *lo* and upper bound *hi* of every document identifier *x* are calculated, and then function *Binary_Code*(*x*, *lo*, *hi*) is called to encode *x* in some appropriate manner. The simplest mechanism uses only binary code to encode *x* in $\lceil \log_2(hi - lo + 1) \rceil$ bits. The algorithm is described in Figure 2.1.

This interpolative coding is best illustrated with an example. Consider the posting list <1, 2, 5, 6, 8, 10, 13> of f_t=7 document identifiers in a collection of N=20 documents. According to the algorithm in Figure 2.1, the middle item in the list, the identifier 6, is encoded. This identifier must take on a value ranged from 1 to 20. Additionally, since there are three other identifiers on each side of this middle item, its possible value range is further limited to from 4 to 17. We represent this fact with (*x*, *lo*, *hi*) = (6, 4, 17), indicating that the document identifier *x* is within the range *lo*…*hi*. Once

the coding of document identifier 6 is accomplished, the three document identifiers on the left hand side may take on values 1 to 5 and those three on the right hand side 7 to 20. According to the same rule, the three document identifiers on the left can be processed first, followed by those three on the right. Therefore, the complete sequence of (*x*, *lo*, *hi*) triples generated by algorithm *Interpolative_Code* are (6, 4, 17), (2, 2, 4), (1, 1, 1), (5, 3, 5), (10, 8, 19), (8, 7, 9), and (13, 11, 20). Using the simplest implementation of *Binary_Code*, the corresponding codewords are 4, 2, 0, 2, 4, 2, and 4 bits long.

Using a centered minimal binary code, the compression efficiency of interpolative coding can be further improved (Moffat and Stuiver, 2000). The centered minimal binary code works in the following way. Support that a number in the range 1…*r* is to be coded. A simple binary code assigns codewords $\lceil \log_2 r \rceil$ bits long to all values 1 through *r*, and wastes $2^{\lceil \log_2 r \rceil} - r$ codewords. That is, $2^{\lceil \log_2 r \rceil} - r$ of the codewords can be shortened by one bit without loss of unique decodability. These minimal codewords are then centered on the encoding range. Numbers at the extremes of the range requires one bit more for storage than those in the center.

Algorithm *Interpolative_Code*(*PL*, *f*, *lo*, *hi*);
Input: *PL* ($PL[1...f]$ is a sorted list of *f* document identifiers, all in the range *lo*...*hi*)
Output: bitstring to represent $PL[1...f]$
begin
    if *f* = 0 then return;
    if *f* = 1 then output bitstring by invoking *Binary_Code*(*PL*[1], *lo*, *hi*) and then return;
    h:=(*f* +1) div 2;
    f_1:=*h*-1;
    f_2:=*f*-*h*;
    $L_1 := PL[1...(h-1)]$;
    $L_2 := PL[(h+1)...f]$;
    Output bitstring by invoking *Binary_Code*(*PL*[*h*], *lo*+f_1, *hi*-f_2);
    Call *Interpolative_Code*(L_1, f_1, *lo*, *PL*[*h*]-1);
    Call *Interpolative_Code*(L_2, f_2, *PL*[*h*]+1, *hi*);
end

Figure 2.1 Interpolative coding.

2.1.2 Observation and improvement

The major overhead of interpolative coding is that a recursive process is used to calculate the order and range of every document identifier. Although a recursive process can be converted to a non-recursive one (Tenenbaum et al., 1990), the converted code requires a stack, which makes the coding and decoding very slow. This is why interpolative coding is not widely used in IRSs.

We observed that the calculation of the order and range for every document identifier can be accelerated by storing partial results in memory. Consider a general posting list $PL_t = <id_1, id_2, \ldots, id_{f_t}>$, where f_t is the number of documents containing term t, $id_k<id_{k+1}$, and all document identifiers are within the range 1…N. Using the interpolative coding method in Figure 2.1, for every f_t, we can obtain the full sequence of triples for the list. Some examples are shown in Table 2.1. These triple sequences are useful for interpolative coding to calculate the order and range for each document identifier. For example, consider the posting list $PL_t = <id_1=1, id_2=2, id_3=5, id_4=7, id_5=8>$ of $f_t=5$ document identifiers in a collection of N=10 documents. The values of this list can be calculated using $f_t = 5$ triples in Table 2.1. The full sequence of triples are (id_3, 3, N-2) = (5, 3, 8), (id_1, 1, id_3-2) = (1, 1, 3), (id_2, id_1+1, id_3-1) = (2, 2, 4), (id_4, id_3+1, N-1) = (7, 6, 9), and (id_5, id_4+1, N) = (8, 8, 10). Storing such a table containing a full set of triple sequences in memory is helpful for the coding and decoding processes of interpolative coding. Compared with the method in Figure 2.1, this improved method eliminates need for a stack in the document identifier order and range calculation, saving a large amount of time.

Table 2.1 Some examples of the full sequence of triples for the general posting list.

f_t	*The full sequence of triples for the general posting list*
1	(id_1, 1, N)
2	(id_1, 1, N-1), (id_2, id_1+1, N)
3	(id_2, 2, N-1), (id_1, 1, id_2-1), (id_3 , id_2+1, N)
4	(id_2, 2, N-2), (id_1, 1, id_2-1), (id_3, id_2+1, N-1), (id_4, id_3+1, N)
5	(id_3, 3, N-2), (id_1, 1, id_3-2), (id_2, id_1+1, id_3-1), (id_4, id_3+1, N-1), (id_5, id_4+1, N)

The triples for each f_t can easily be represented as a two-dimensional array *I_Triple* consisting of f_t rows and 5 columns. This representation for f_t=5 is shown in Figure 2.2. The first row of the array represents the first triple, and the second row represents the second triple, and so forth. The first column is used to denote the index of the document identifier in the posting list for the first element of the triple. For example, *I_Triple*[3][1] is 2, meaning the first value of No. 3 triple is id_2. The second and third columns denote the index of the document identifier in the posting list and the offset for the second element of the triple. For example, *I_Triple*[3][2] and *I_Triple*[3][3] are two 1s, meaning the second value of No. 3 triple is id_1+1. Finally, the fourth and fifth columns denote the index of the document identifier in the posting list and the offset for the third element of the triple. For example, *I_Triple*[3][4] and *I_Triple*[3][5] are 3 and –1, meaning the third value of No. 3 triple is id_3-1. To make this representation more practical and convenient, two extra values are used for each posting list: id_{f_t+1}=0 and id_{f_t+2}=*N*. Therefore, the first triple (id_3, 3, *N*-2) in Figure 2.2 can be represented as 3, 6, 3, 7, and -2.

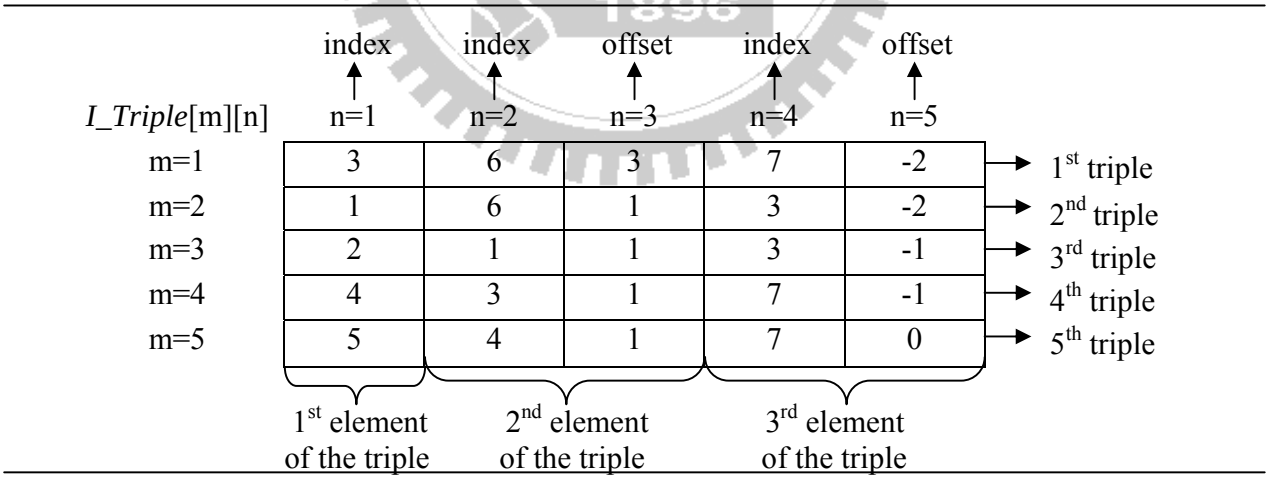


Figure 2.2 Given a general posting list PL_t: <id_1, id_2, id_3, id_4, id_5> of f_t=5 document identifiers, and set id_{f_t+1}= id_6=0 and id_{f_t+2}= id_7=*N*. The corresponding triples: (id_3, 3, *N*-2), (id_1, 1, id_3-2), (id_2, id_1+1, id_3-1), (id_4, id_3+1, *N*-1), (id_5, id_4+1, *N*) can be represented with the *I_Triple*[f_t][5].

The algorithm in Figure 2.3 can be used to generate the corresponding triples for each f_t and store them in *I_Triple*[f_t][5]. For a sub-posting list *PL*[*index* …(*index*+*k*-1)] among id_{lo_index}+*lo* and

id_{hi_index}+*hi*, *Compute_I_Triple*(*index*, *k*, *lo_index*, *lo*, *hi_index*, *hi*) can be called to generate the corresponding triples and store them in a two-dimensional array *I_Triple*.

Algorithm *Generate_I_Triple*(*PL*, *f*, *N*);
Input: *PL* ($PL[1...f]$ is a sorted list of *f* document identifiers, all in the range 1...*N*, and to simplify the algorithm we set $PL[(f+1)]$ to 0, and $PL[(f+2)]$ to *N*)
Output: *I_Triple*[*f*][5] to represent the triples
begin
  n:=1; /* *n* is a global variable*/
  Compute_I_Triple(1, *f*, *f*+1, 1, *f*+2, 0); /* generate *I_Triple*[*f*][5] */
  return *I_Triple*;
end

procedure *Compute_I_Triple*(*index*, *k*, *lo_index*, *lo*, *hi_index*, *hi*)
begin
  if *k*=0 then return;
  if *k*=1 then
    I_Triple[*n*][1]:=*index*;
    I_Triple[*n*][2]:=*lo_index*;
    I_Triple[*n*][3]:=*lo*;
    I_Triple[*n*][4]:=*hi_index*;
    I_Triple[*n*][5]:=*hi*;
    n++;
    return;
  h:=(*k*-1)/2;
  f_1:=*h*;
  f_2:=*k*-*h*-1;
  I_Triple[*n*][1]:=*h*+*index*;
  I_Triple[*n*][2]:=*lo_index*;
  I_Triple[*n*][3]:=*lo*+f_1;
  I_Triple[*n*][4]:=*hi_index*;
  I_Triple[*n*][5]:=*hi*-f_2;
  n++;
  Compute_I_Triple (*index*, f_1, *lo_index*, *lo*, *index*+*h*, -1);
  Compute_I_Triple (*index*+*h*+1, f_2, *index*+*h*, 1, *hi_index*, *hi*);
end

Figure 2.3 The algorithm for generating *I_Triple*.

2.1.3 Remarks

Although the procedure *Compute_I_Triple* in Figure 2.3 also uses recursive process, it can be processed off-line and one can store the *I_Triple*s of different f_ts in memory. This can reduce the on-line decoding time dramatically. With the *I_Triple*, one can easily find minimal binary code in encoding a posting list, as shown in the following:

```
for m:=1 to f_t do
   output bitstring by invoking Binary_Code(PL[I_Triple[m][1]],
                                            PL[I_Triple[m][2]]+I_Triple[m][3],
                                            PL[I_Triple[m][4]]+I_Triple[m][5] );
```

However, this improved method still requires large memory space. For example, each triple contains five integers. If an integer takes 4-byte storage space, the memory requirement for a triple is 20 bytes. Therefore, in a posting list with f_t document identifiers, $20 \times f_t$ bytes are required. The maximum f_t in present IRSs can reach up to thousands or millions, which means the memory space required for *I_Triple* storage is ten thousands or even ten millions of bytes. This makes it impossible using memory to accelerate coding and decoding with interpolative code. Furthermore, using *I_Triple* to encode and decode requires extra memory access time, which makes the decoding speed slow.

2.2 Proposed Method: Unique-Order Interpolative Coding

The recursive process makes the decoding of interpolative coding slow. Although using memory to store partial results of the recursive process can accelerate the coding and decoding of interpolative coding, a large amount of memory is required to store the *I_Triple* for each f_t. We develop a new method called unique-order interpolative coding in which only one *I_Triple* is required for the entire coding and decoding process of all posting lists no matter how many different values of f_t are present. Then we introduce loop unwinding to replace *I_Triple* with

constant values. The number of memory accesses to *I_Triple* can therefore be reduced, which accelerates the whole process.

2.2.1 The coding method

This subsection presents the details of our proposed coding method. Two key decisions are to be made in the coding method.

A. Decomposition of a posting list into blocks to take advantage of interpolative coding

In a posting list PL_t=<id_1, id_2, …, id_{ft} > of f_t document identifiers, where id_k<id_{k+1} and all document identifiers are within the range 1…N. A group size g is first determined. Then PL_t is divided into $m = \left\lceil \frac{f_t}{g} \right\rceil$ blocks, each having g document identifiers except possibly the last block. We define the first document identifier in each block to be a boundary pointer, the document identifiers between boundary pointers to be inner pointers, and those in the last block except the boundary pointer to be residual pointers. The PL_t can then be compressed as follows. The boundary pointers and their subsequent residual pointers together can be regarded as a sub-posting list, and a suitable *d*-gap compression scheme with high decoding speed can be used for compression. The inner pointers in each block are compressed via interpolative coding. With this new method (see Figure 2.4), each inner block contains a constant number (g-1) of inner pointers, enabling the use of only one *I_Triple* in coding and decoding. Compared with interpolative coding, this new method allows document identifiers to be stored in a fixed order, hence the name unique-order interpolative coding. When $f_t \le g$ or m=1 or g=1, no inner pointers are present, and we apply only a *d*-gap compression scheme.

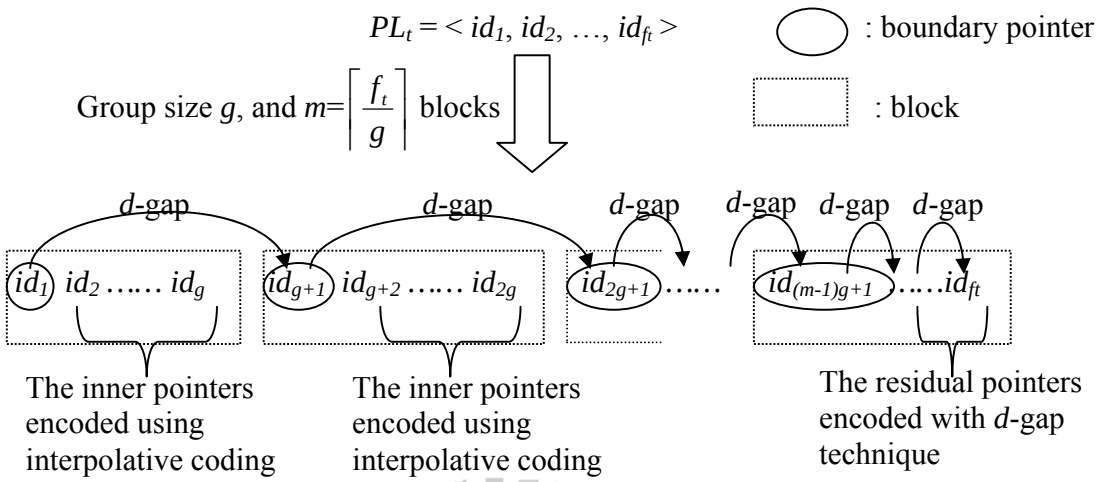


Figure 2.4 The illustration of unique-order interpolative coding.

B. Choice of a suitable coding method for boundary and residual pointers

Compared with the *d*-gaps of a traditional *d*-gap compression scheme, the *d*-gaps of unique-order interpolative coding extracted from every group of document identifiers are potentially much larger and may cause a decrease in compression efficiency. Therefore, a suitable coding method is required to encode the boundary pointers to improve compression efficiency. To simplify implementation, the boundary and residual pointers are encoded with the same method.

In this chapter, we recommend Golomb coding and *r* coding for encoding the *d*-gaps of unique-order interpolative coding. Golomb coding is very suitable for encoding the *d*-gaps of unique-order interpolative coding, since the *d*-gaps extracted from every group of document identifiers are roughly of the same length. Using γ coding is also a relatively economical choice when the document identifiers in a posting list are also close together, and the *d*-gaps are small. Other coding methods are not disregarded. We are still looking for a faster and more compact coding method to encode the *d*-gaps of unique-order interpolative coding.

To improve the compression efficiency of the *d*-gaps of unique-order interpolative coding, the value *g* is subtracted from the *d*-gap of all boundary pointers (except the first one) without loss of unique decodability. This approach works the best when the original *d*-gaps are small.

2.2.2 Illustration

This unique-order interpolative coding is best illustrated with an example. Given a posting list <5, 8, 12, 13, 15, 18, 23, 28, 29, 32, 33> of 11 document identifiers, let the group size *g* be 4, the document identifiers 5, 15, and 29 are therefore the boundary pointers, the document identifiers 32 and 33 are the residual pointers, and the others are the inner pointers. Let [id_i, id_{i+1}, …, id_j] represent id_i, id_{i+1}, …, id_j encoded in interpolative code. Since the two successive boundary pointers must be known before interpolative coding of the inner pointers, the boundary pointer of each block is stored before coding of the inner pointers. Therefore, the posting list is to be stored as

<5, 15, [8, 12, 13], 29, [18, 23, 28], 32, 33>,

where [8, 12, 13]and [18, 23, 28] are in interpolative codes, and 5, 15, 29, 32, 33 in *d*-gaps. The resulted list is

<5, 10(=15-5), [8, 12, 13], 14(=29-15), [18, 23, 28], 3(=32-29), 1(=33-32)>.

Next, since there are three document numbers between each pair of boundary pointers, the list can be simplified as

<5, 7(=10-3), [8, 12, 13], 11(=14-3), [18, 23, 28], 3, 1>.

In decoding, the first two *d*-gaps, 5 and 7, are retrieved to obtain the first two boundary pointers, which are 5 and 15(=5+7+3). Interpolative coding is then used to obtain [8, 12, 13]. Then, the *d*-gap, 11, is retrieved to obtain the next boundary point, 29(=15+11+3), and interpolative coding is used to obtain [18, 23, 28]. Finally, the residual pointers 32(=3+29) and 33(=1+32) are obtained by the remaining *d*-gaps.

Now, consider a general posting list $PL_t = <id_1, id_2, \ldots, id_{ft}>$ encoded using unique-order interpolative coding with group size $g=4$, the PL_t can be represented as

$$<id_1, \quad id_5, \quad [id_2, \quad id_3, \quad id_4],$$
$$id_9, \quad [id_6, \quad id_7, \quad id_8],$$
$$id_{13}, \quad [id_{10}, \quad id_{11}, \quad id_{12}], \ldots >,$$

where id_1, id_5, id_9, id_{13} are encoded using a *d*-gap coding method and [id_2, id_3, id_4], [id_6, id_7, id_8], [id_{10}, id_{11}, id_{12}] are encoded using interpolative coding. The example list can be further represented (using triple representation in Section 2) as

$$<id_1, \quad id_5 - id_1 - 3, \ (id_3, id_1+2, id_5-2), (id_2, id_1+1, id_3-1), (id_4, id_3+1, id_5-1),$$
$$id_9 - id_5 - 3, \ (id_7, id_5+2, id_9-2), (id_6, id_5+1, id_7-1), (id_8, id_7+1, id_9-1),$$
$$id_{13} - id_9 - 3, \ (id_{11}, id_9+2, id_{13}-2), (id_{10}, id_9+1, id_{11}-1), (id_{12}, id_{11}+1, id_{13}-1), \ldots >.$$

We observed that the *I_Triple* for [id_i, id_{i+1}, id_{i+2}] can be converted to the *I_Triple* for [id_{i+4}, id_{i+5}, id_{i+6}] by adding 4 (which is the value of *g*) to the indices of document identifiers in the *I_Triple* for [id_i, id_{i+1}, id_{i+2}]. Therefore, only one *I_Triple* is required in coding and decoding, which accelerates the whole process. If we use Golomb coding to encode boundary pointers and residual pointers, this new coding method can be shown as the following program in Figure 2.5.

```
Algorithm Unique_Order_Interpolative_Code(PL, f, N, g);

Input: PL (PL[1...f] is a sorted list of document numbers, all in the range 1...N), and
       group size g(an integer);
Output: Bitstring (the compressed posting list PL)
begin
 if f ≤ g then // compressed by Golomb coding
        b := ⌈0.69 × N / f ⌉;
        prev_document_identifier:=0;
        for i:=1 to f
              append Golomb_Code(PL[i]-prev_document_identifier, b) to Bitstring;
              prev_document_identifier:= PL[i];
 else // compressed by unique-order interpolative coding
        m = ⌈ f / g ⌉;
        b := ⌈0.69 × N /( f − (m − 1) × (g − 1)) ⌉;

        // encode the first boundary pointer
        append Golomb_Code(PL[1], b) to Bitstring;

        // generate I_Triple
        n:=0;
        I_Triple:=Compute_I_Triple(2, g-1, 1, 1, g+1, -1);

        for i:=0 to (m-2) do
              index:=i×g;

              // encode boundary pointer
              append Golomb_Code(PL[index+g+1]-PL[index+1]-g+1, b) to Bitstring;

              // encode inner pointers
              for j:=1 to g-1 do
                append Binary_Code(PL[index+I_Triple[j][1]],
                                   PL[index+I_Triple[j][2]]+I_Triple[j][3],
                                   PL[index+I_Triple[j][4]]+I_Triple[j][5]) to Bitstring;

        // encode residual pointers
        for i:=(m-1)×g+2 to f
           append Golomb_Code(PL[i]-PL[i-1], b) to Bitstring;

 return BitString;
end
```

Figure 2.5 Unique-order interpolative coding (using Golomb coding to encode boundary and residual pointers).

2.2.3 Implementation optimization

This subsection presents how to use loop unwinding to accelerate the encoding and decoding of unique-order interpolative coding. Note that once the group size g is determined, the program in Figure 2.5 can be further accelerated. For example, for g=4, the following program segment in Figure 2.5

```
for i:=0 to (m-1) do
        index:=i×g;

        // encode boundary pointer
        append Golomb_Code(PL[index+g+1]-PL[index+1]-g+1, b) to Bitstring;

        // encode inner pointers, 8 memory accesses are required for encoding each inner
        // pointer: 5 for I_Triple accesses and 3 for PL accesses
        for j:=1 to g-1 do
                append Binary_Code(PL[index+I_Triple[j][1]],
                                   PL[index+I_Triple[j][2]]+I_Triple[j][3],
                                   PL[index+I_Triple[j][4]]+I_Triple[j][5]) to Bitstring;
```

can be converted to

```
for i:=0 to (m-1) do
        index:=i×4;

        // encode boundary pointer
        append Golomb_Code(PL[index+5]-PL[index+1]-3, b) to Bitstring;

        // loop unwinding, only 3 memory accesses of PL are required for encoding each
        // inner pointer
        append Binary_Code(PL[index+3], PL[index+1]+2, PL[index+5]-2) to Bitstring;
        append Binary_Code(PL[index+2], PL[index+1]+1, PL[index+3]-1) to Bitstring;
        append Binary_Code(PL[index+4], PL[index+3]+1, PL[index+5]-1) to Bitstring;
```

In other words, once the group size g has been determined, the *I_Triple* accesses in loop can be eliminated in unique-order interpolative coding. So the 8-3=5 times memory accesses for each document identifier can be avoided, which in turn accelerates the encoding process. By using the same approach, the decoding of unique-order interpolative coding can also be accelerated.

2.3 Quantitative Analysis

Give a posting list $PL = < d_1, id_2, ..., id_f >$ of f document identifiers, where $id_k < id_{k+1}$, and all document identifiers are within the range 1...*N*. As stated in Section 2.2, the first step in unique-order interpolative coding is to determine the group size *g*. Once *g* is determined, the *PL* will be divided into $m = \left\lceil \frac{f}{g} \right\rceil$ blocks, with the first (*m*-1) blocks containing *g* document identifiers and the last block containing $f - (m-1)g$ document identifiers. The boundary pointers and the residual pointers will be coded by efficient prefix-free coding methods such as Golomb coding and γ coding, in *d*-gap manner, and the inner document identifiers will be coded by the interpolative coding.

Let the function $F(N, f)$ represent bits needed for compressing the f document identifiers ranging from 1 to *N*. Theoretically, the following approximate formulas can then be achieved (Golomb, 1966; Gallager & Van Voorhis, 1975; Mcllroy, 1982; Elias, 1975; Moffat & Stuiver, 2000).

$$\text{Golomb coding: } G(N, f) \le f \times (2 + \log_2 \frac{N}{f}) \tag{2.1}$$

$$\gamma \text{ coding: } \gamma(N, f) \le f \times (1 + 2 \times \log_2 \frac{N}{f}) \tag{2.2}$$

$$\text{Interpolative coding: } I(N, f) \le f \times (2.5783 + \log_2 \frac{N}{f}) \tag{2.3}$$

If Golomb coding is used to encode the boundary pointers and residual pointers, then the maximum number of bits required to store these *f*-(*m*-1)(*g*-1) boundary and residual pointers is

$$(f - (m-1)(g-1)) \times (2 + \log_2 \frac{N}{f - (m-1)(g-1)}) \tag{2.4}$$

If we use γ coding to encode these pointers, then the maximum number of bits required is

$$(f-(m-1)(g-1))\times(1+2\times\log_2\frac{N}{f-(m-1)(g-1)}) \tag{2.5}$$

Based on Eq.(2.3), the number of bits required to code the inner pointers ((m-1) groups, (g-1) document identifiers in each group) is

$$\sum_{i=1}^{m-1}\left[(g-1)\times(2.5783+\log_2\frac{N_i}{g-1})\right], \text{ where } N_i = id_{g\times i+1} - id_{g\times(i-1)+1} - 1 \tag{2.6}$$

Since

$$\sum_{i=1}^{m-1} N_i \le N \tag{2.7}$$

and the sum of the logarithms of the (m-1) individual ranges is maximized when all $\frac{N_i}{g-1}$ are equal, one obtains

$$\sum_{i=1}^{m-1}\left[(g-1)\times(2.5783+\log_2\frac{N_i}{g-1})\right] \le (m-1)(g-1)\times(2.5783+\log_2\frac{N}{(m-1)(g-1)}) \tag{2.8}$$

Therefore, if Golomb coding is used to encode the boundary and residual pointers, then the maximum number of bits required by the unique-order interpolative coding is at most

$$(f-(m-1)(g-1))\times(2+\log_2\frac{N}{f-(m-1)(g-1)})+(m-1)(g-1)\times(2.5783+\log_2\frac{N}{(m-1)(g-1)}) \tag{2.9}$$

Or if we use γ coding, it is

$$(f-(m-1)(g-1))\times(1+2\times\log_2\frac{N}{f-(m-1)(g-1)})+(m-1)(g-1)\times(2.5783+\log_2\frac{N}{(m-1)(g-1)}) \tag{2.10}$$

Eqs. (2.9) and (2.10) can be simplified under the condition that no residual pointers exist. For example, when f=(m-1)g+1, Eq. (2.9) can be rewritten as:

$$f\times[\frac{2+\log_2 g+2.5783\times(g-1)+(g-1)\times\log_2(\frac{g}{g-1})}{g}+\log_2\frac{N}{f}] \tag{2.11}$$

and some examples of the maximum number of bits required for unique-order interpolative coding are derived in Table 2.2.

Table 2.2 Some examples of the maximum number of bits required for unique-order interpolative coding if Golomb coding is used to encode boundary pointers under the condition that no residual pointers exist.

g	maximum number of bits required
2	$f \times [3.29 + \log_2 \frac{N}{f}]$
4	$f \times [3.25 + \log_2 \frac{N}{f}]$
8	$f \times [3.05 + \log_2 \frac{N}{f}]$
16	$f \times [2.88 + \log_2 \frac{N}{f}]$
32	$f \times [2.76 + \log_2 \frac{N}{f}]$

The results in Table 2.2 showed that when Golomb coding is used to encode boundary pointers, the maximum number of bits required in unique-order interpolative coding has inverse relationship with group size g: the maximum number of bits decreases with increase in group size g and increases with decrease in g. On the other hand, if the number of document identifiers is less than (g+1), unique-order interpolative coding cannot be used. We design an experiment in Section 2.4 to find a suitable group size g.

The results in Eqs. (2.9) and (2.10), and Table 2.2 can be improved if Eq.(2.3) can be improved. For example, the maximum number of bits required for interpolative coding to encode a posting list with 3 document identifiers ranging from 1 to N is

$$\lceil \log_2 (N-2) \rceil + \lceil \log_2 a \rceil + \lceil \log_2 b \rceil \tag{2.12}$$

since the middle item requires $\lceil \log_2 (N-2) \rceil$ bits, and the left and right items require $\lceil \log_2 a \rceil + \lceil \log_2 b \rceil$ bits where a, b are two positive integers and a+b=(N-1). Since

$$\lceil \log_2(N-2) \rceil < 1 + \log_2 N \tag{2.13}$$

and

$$\lceil \log_2 a \rceil + \lceil \log_2 b \rceil < (1 + \log_2 a) + (1 + \log_2 b) < 2 + \log_2 \frac{N}{2} + \log_2 \frac{N}{2} \tag{2.14}$$

hence

$$\lceil \log_2(N-2) \rceil + \lceil \log_2 a \rceil + \lceil \log_2 b \rceil < 3 \times (1.92 + \log_2 \frac{N}{3}) \tag{2.15}$$

We replace Eq.(2.3) with Eq.(2.15) when group size g=4, and the maximum number of bits required for the unique-order interpolative coding under the condition that no residual pointers exist is therefore

$$f \times [2.76 + \log_2 \frac{N}{f}] \tag{2.16}$$

Compared with the figure in Table 2.2, a much tighter upper bound is obtained.

To further understand the characteristics of unique-order interpolative coding, we conducted following experiments. We used encoding methods such as Golomb coding, skewed Golomb coding, batched LLRUN coding, interpolative coding, variable byte coding, Carryover-12 mechanism, unique-order interpolative coding 1 (group size g=4; boundary pointers and residual pointers by Golomb coding), unique-order interpolative coding 2 (group size g=4; boundary pointers and residual pointers by γ coding) in compression. In the first experiment (Table 2.3(a)), f = 1,000,000 gaps were drawn from a geometric distribution and compressed using the eight methods. The Golomb coding performs the best, since it is a minimum-redundancy code for geometric gap distribution (Gallager and Van Voorhis 1975). Compared with other methods, unique-order interpolative coding is not suitable for a geometric distribution when $2 < \frac{N}{f} < 256$.

But when $\frac{N}{f}$ increases, the performance of unique-order interpolative coding 1 improves proportionally. When $\frac{N}{f} \le 2$, the results of unique-order interpolative coding 2 are satisfying. For most cases in the first experiment, both variable byte coding and Carryover-12 mechanism are inefficient in compression.

Table 2.3 Compression results for geometric and skew geometric distributions of f = 1,000,000 gaps: average bits per gap

Coding Methods	Average gap (N/f), Geometric Distribution											
	1	2	4	8	16	32	64	128	256	512	1024	2048
Golomb coding	1.00	2.33	3.30	4.39	5.43	6.45	7.46	8.47	9.47	10.47	11.47	12.47
Skewed Golomb coding	1.00	2.53	3.51	4.60	5.64	6.66	7.67	8.68	9.68	10.68	11.68	12.68
Batched LLRUN coding	1.00	2.27	3.46	4.50	5.53	6.52	7.52	8.52	9.52	10.52	11.52	12.53
Interpolative coding	0.00	2.15	3.45	4.59	5.66	6.69	7.70	8.71	9.71	10.71	11.71	12.72
Variable byte coding	8.00	8.00	8.00	8.00	8.00	8.14	9.08	10.93	12.87	14.24	15.07	15.52
Carryover-12 mechanism	1.07	2.88	4.11	5.17	6.18	7.38	8.75	9.90	10.58	12.30	14.41	15.56
Unique-order interpolative coding 1	3.00	4.19	5.13	5.97	6.76	7.53	8.29	9.06	9.89	10.77	11.68	12.77
Unique-order interpolative coding 2	0.25	2.33	3.91	5.31	6.64	7.92	9.19	10.45	11.70	12.96	14.21	15.46
Self-entropy	0.00	2.00	3.24	4.35	5.40	6.42	7.43	8.44	9.44	10.44	11.43	12.43

(a) Geometric distribution

Coding Methods	Average gap (N/f), Skewed Distribution											
	1	2	4	8	16	32	64	128	256	512	1024	2048
Golomb coding	1.40	2.60	3.30	4.29	5.33	6.37	7.39	8.40	9.40	10.40	11.40	12.41
Skewed Golomb coding	1.80	2.31	2.92	3.76	4.80	5.79	6.80	7.82	8.82	9.83	10.83	11.83
Batched LLRUN coding	1.40	2.31	2.86	3.60	4.61	5.66	6.70	7.71	8.71	9.71	10.70	11.71
Interpolative coding	0.84	1.53	2.07	2.90	3.97	5.07	6.15	7.19	8.21	9.23	10.23	11.24
Variable byte coding	8.00	8.00	8.00	8.00	8.10	8.58	9.38	10.11	10.63	11.28	12.43	13.80
Carryover-12 mechanism	1.07	2.36	2.90	3.72	4.84	6.02	6.98	7.9	9.35	10.90	12.08	12.57
Unique-order interpolative coding 1	3.60	3.96	4.30	4.80	5.51	6.30	7.11	7.94	8.76	9.60	10.51	11.62
Unique-order interpolative coding 2	1.25	1.90	2.47	3.33	4.53	5.88	7.21	8.53	9.81	11.07	12.33	13.60
Self-entropy	0.97	1.77	2.30	3.05	4.06	5.10	6.15	7.18	8.19	9.19	10.19	11.20

(b) Skewed geometric distribution

In the second experiment, for each value of $\frac{N}{f}$ the sequence of f = 1,000,000 geometrically distributed gaps was broken into chunks of 200 contiguous values. The chunks were then placed in groups of five. In the first three chunks of each group, all gaps were multiplied by a factor of 0.1; whereas in the other two chunks all gaps were multiplied by a factor of 2.35. This process created

artificial clusters of gaps much similar than the average, and about 60% of the values were coded into these clusters, while the overall average gap remained the same. This better resembles the distribution of real document collections. The results are shown in Table 2.3(b). Compared with skewed Golomb coding, batched LLRUN coding, and interpolative coding, the compression efficiency of Golomb coding is not as good as others, meaning it is unable to exploit clustering well. The compression results of unique-order interpolative coding for a skewed geometric distribution are better than that for a geometric distribution. This means that unique-order interpolative coding does take a good advantage of the clustering property. For $\frac{N}{f} \leq 32$, we prefer to use the unique-order interpolative coding 2; while for $\frac{N}{f}$>32, we suggest unique-order interpolative coding 1. Similar to that for a geometric distribution, the unique-order interpolative coding 1 performs better as $\frac{N}{f}$ becomes larger. Again, both variable byte coding and Carryover-12 mechanism are inefficient in compression for most cases in the second experiment. From Table 2.3(b), interpolative coding can even outperform self-entropy. This is due to the fact that interpolative coding does not use the gap value in encoding directly, but instead uses a minimal binary code to encode every gap after it is converted to a triple.

2.4 Performance Evaluation

An experimental information retrieval system was implemented to evaluate the various coding methods. Experiments were conducted on some real-life document collections, and query processing time and storage requirements for each coding method were measured.

2.4.1 Document collections and queries

Five document collections were used in the experiments. Their statistics are listed in Table 2.4. In this table, N denotes the number of documents; n is the number of distinct terms; F is the total number of terms in the collection; and f indicates the number of document identifiers that appear in an inverted file.

Table 2.4 Statistics of document collections

		Collection				
		Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Documents	N	31,101	32,472	130,471	131,896	262,367
# of terms	F	884,746	2,320,610	72,922,893	72,087,460	145,010,353
Distinct terms	n	8,965	58,536	214,310	168,251	317,393
# of document identifier count	f	701,304	1,694,491	28,628,698	32,483,656	61,112,354
Average gap size	$N \times n / f$	398	1122	977	683	1363
Total size (Mbytes)		4.69	21.30	470	475	945

Collection *Bible* is the King James version of the Bible, in which each verse is considered as a document. The second collection, *DBbib*, is a set of citations to chapters appearing in the database literature. The third and forth collections, *FBIS* (Foreign Broadcast Information Service) and *LAT* (LA Times), are disk 5 of the TREC-6 collection that is used internationally as a test bed for research in information retrieval techniques (Voorhees and Harman, 1997). The final collection *TREC* includes the *FBIS* and *LAT* collections.

Since effectiveness of coding methods relies heavily on clustering of documents, inverted files for these collections were built with a Greedy-NN algorithm (Shieh et al., 2003). These inverted files were then used to test the advantages and shortcomings of various coding methods.

We followed the method (Moffat and Zobel, 1996) to evaluate performance with random queries. For each document collection, 1000 documents were randomly selected to generate a query set. A query was generated by selecting words from a word list of a specific document, combined by some randomly generated Boolean operators ANDs and ORs. To form the document word list,

words in the document were case folded, and stop words such as “the” and “this” were eliminated. For example, a query word list may be “inverted file document collection built”, a query may be “(inverted <AND> file <AND> document <AND> collection) <OR> built”. For each query, there existed at least one document in the document collection that satisfied the query. The generated queries followed a Zipf-like distribution $P \sim 1/\rho^{0.55}$, where P is the probability of accessing each query, and ρ is the popularity rank for the test query stream. This is widely believed to closely resemble the distribution of real queries (Breslau et al., 1999).

2.4.2 Performance results

In this subsection, we first present the compression performance of unique-order interpolative coding versus different group size g. We then present the compression performance of different coding methods. Finally, we present the search performance of different coding methods.

Compression performance of unique-order interpolative coding

In this subsection, Golomb coding was used to code both boundary pointers and residual pointers. This is due to the fact that the average gap sizes in Table 2.4 are relatively big, Golomb coding was recommended according to Table 2.3(b). The compression result is shown in Table 2.5, and the metric used is the average number of *bits per document identifier BPI*, defined as follows:

$$BPI = \frac{\text{The size of the compressed inverted file}}{\text{number of document identfiers } f}.$$

For each term t, the cost of using r coding to encode the frequency f_t is calculated and included in the presented results.

Note that for group size g=4 and g=8, unique-order interpolative coding achieved good compression. For a simple implementation, we suggest using g=4. In the following experiments, Golomb coding was used to code both boundary pointers and residual pointers for unique-order interpolative coding, and group size g was set to 4 unless otherwise stated.

Table 2.5 Compression performance of unique-order interpolative coding versus different group size *g*

Group Size	Collection				
g	*Bible*	*DBbib*	*FBIS*	*LAT*	*TREC*
1	6.11	6.20	5.27	5.31	5.49
2	5.64	5.47	4.84	4.91	4.99
3	5.61	5.31	4.80	4.89	4.94
4	5.46	5.11	4.66	4.74	4.78
5	5.52	5.13	4.71	4.80	4.82
6	5.52	5.10	4.71	4.79	4.81
7	5.47	5.04	4.65	4.74	4.75
8	5.42	4.98	4.59	4.68	4.69
9	5.47	5.01	4.64	4.72	4.73
10	5.51	5.03	4.67	4.75	4.76

Compression performance of different coding methods

We now compare the effectiveness of the eight coding methods: γ coding, Golomb coding, batched LLRUN coding, skewed Golomb coding, interpolative coding, variable byte coding, Carryover-12 mechanism, and unique-order interpolative coding. For each term *t*, the cost of using *r* coding to encode the frequency f_t is calculated and included in the presented results. Moreover, any necessary overheads, such as the complete set of models and model selectors for the batched LLRUN coding, are also calculated and included. However, the cost of storing the parameter *b* for each posting list in Golomb coding (Witten et al., 1999) is not calculated nor included. This is because the parameter *b* for each posting list in Golomb coding can be calculated via stored frequency f_t using Witten's approximation. The results are shown in Table 2.6. Notice that:

1. Both variable byte coding and Carryover-12 mechanism are inefficient in compression of inverted files.
2. For the other coding methods, the compression efficiencies of both γ coding and Golomb coding are relatively low because of the simple models they use.

3. The compression efficiencies of batched LLRUN, skewed Golomb, interpolative, and unique order interpolative coding methods are relatively good. This shows that clustering is a good compression aid.
4. The compression efficiency of unique-order interpolative coding is only inferior to that of interpolative coding, meaning that it does take a good advantage of the clustering property.

Table 2.6 Compression Performance of different coding methods.

Coding Methods	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
γ coding	6.58	5.96	5.38	5.63	5.63
Golomb coding	6.11	6.20	5.27	5.31	5.49
Batched LLRUN coding	5.52	4.88	4.63	4.78	4.84
Skewed Golomb coding	5.92	5.75	5.04	5.07	5.10
Interpolative coding	5.37	4.89	4.58	4.65	4.62
Variable byte coding	9.10	9.54	8.88	8.89	8.84
Carryover-12 mechanism	7.14	7.99	6.23	6.13	5.95
Unique-order interpolative coding	5.46	5.11	4.66	4.74	4.78

Search performance of different coding methods

The query processing time includes (1) disk access time, (2) decompression time, and (3) document identifiers comparison time. Experiments showed that disk access time and decompression time occupy more than 90% of query processing time. And document identifier comparison time is not a function of the coding method used. Therefore the search performance metric is defined as

Search Time (ST) = Disk Access Time (AT) + Decompression Time (DT).

And the speedups of all coding methods relative to Golomb coding, for all test collections, were calculated.

All experiments described in this subsection were run on an Intel P4 2.4GHz PC with 256MB DDR memory running Linux operating system 2.4.12. The hard disk was 40GB, and the data transfer rate was 25MB/sec. Intervening processes and disk activities were minimized during

experimentation. All decoding mechanisms were written in *C*, compiled with *gcc*, and optimized as follows:

1. Replaced subroutines with macros.
2. Careful choice for compiler optimization flags.
3. Implementation used 32-bit integers, as that is the internal register size of the Intel P4 CPU.
4. Implemented the integer logarithm function $\lceil \log_2(i) \rceil$ with a lookup table.

   Let z be a 256-entry array, and z[*k*] be $\lceil \log_2(k+1) \rceil$ where $0 \le k \le 255$. The function $\lceil \log_2(i) \rceil$ can be implemented in *C* as follows (v is the returned value of $\lceil \log_2(i) \rceil$):

```
do {
     register int __i  = (i) - 1;
      (v) = _B_i>>16 ?    (_B_i>>24 ? 24 + z[_B_i>>24] : 16 + z[_B_i>>16]) :
                          (_B_i>>  8 ?  8 + z[_B_i>>8]  : z[_B_i]) ;
} while (0);
```

5. Implemented the integer logarithm function $\lfloor \log_2(i) \rfloor$ also with a lookup table.

   The array z is the same as that used in the function $\lceil \log_2(i) \rceil$. The function $\lfloor \log_2(i) \rfloor$ can be implemented in *C* as follows (v is the returned value of $\lfloor \log_2(i) \rfloor$):

```
do {
     register int __i  = (i) ;
     (v) = _B_i>>16 ?    (_B_i>>24 ? 23 + z[_B_i>>24] : 15 + z[_B_i>>16]) :
                         (_B_i>>  8 ?  7 + z[_B_i>>8]  : z[_B_i] - 1) ;
} while (0);
```

6. A 256-entry lookup table is used to locate the exact bit location of the first "1" bit in a byte.

   For example, in the byte 00101000 the first "1" bit is in location 3. This can accelerate the decoding process of unary codes because no bit-by-bit decoding is required.
7. Access to binary codes with masking and shifting operations, and no bit-by-bit decoding is required.

With these optimizations, decoding of a document identifier only required tens of ns, and no bit-by-bit decoding is required.

Other optimizations included: The Huffman code of batched LLRUN coding was implemented with canonical prefix codes (Turpin, 1998). The canonical prefix codes can be decoded via fast table look-up. And for the interpolative coding method, recursive process was transformed to non-recursive process, at the cost of an explicit stack (Tenenbaum et al., 1990).

The search performance measurements are shown in Table 2.7. Key findings are:

1. Although variable byte coding and Carryover-12 mechanism gave fast decoding, *r* coding and unique-order interpolative coding achieved higher query throughput rates. This is because the disk access time (AT) of variable byte coding and Carryover-12 mechanism is much higher than that of *r* coding and unique-order interpolative coding.
2. For collection *DBbib*, the decoding times (DT) of *r* coding and unique-order interpolative coding are less than that of Carryover-12. This is because a large portion of the *d*-gaps of frequently used query terms for *DBbib* is of value 1. Both *r* coding and unique-order interpolative coding can encode these *d*-gaps very economically. This also makes the decoding times of *r* coding and unique-order interpolative coding for these *d*-gaps very low.
3. Batched LLRUN coding, skewed Golomb coding, and interpolative coding gave better compression rates than Golomb coding. However, their complex decoding mechanisms prohibited them from being used in real-world IRSs.
4. Experimental results showed that *r* coding, Carryover-12 mechanism, and unique-order interpolative coding were recommended for real-world IRSs. Their query throughput rates were all much higher than that of Golomb coding.

Table 2.7 Search performance of different coding methods (AT is the disk access time, DT is the decoding time, ST=AT+DT is the search time, and SP is the performance relative to the Golomb coding)

Coding Method		Collection				
		Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
γ coding	AT(us)	125	202	1125	1168	2149
	DT(us)	70	188	952	980	1696
	ST(us)	195	390	2077	2148	3845
	SP	1.14	1.50	1.20	1.23	1.20
Golomb coding	AT(us)	131	306	1282	1321	2422
	DT(us)	92	280	1200	1314	2179
	ST(us)	223	586	2482	2635	4601
	SP	1.00	1.00	1.00	1.00	1.00
Batched LLRUN coding	AT(us)	116	381	1101	1134	2086
	DT(us)	130	192	1688	1771	3013
	ST(us)	246	573	2789	2905	5099
	SP	0.91	1.02	0.89	0.91	0.90
Skewed Golomb coding	AT(us)	117	331	1120	1150	2097
	DT(us)	122	201	1492	1577	2696
	ST(us)	239	532	2612	2727	4793
	SP	0.93	1.10	0.95	0.97	0.96
Interpolative coding	AT(us)	111	137	1024	995	1916
	DT(us)	243	688	3094	3266	5598
	ST(us)	354	825	4118	4261	7514
	SP	0.63	0.71	0.60	0.62	0.61
Variable byte coding	AT(us)	214	918	3134	3489	5506
	DT(us)	22	90	336	388	633
	ST(us)	236	1008	3470	3877	6139
	SP	0.95	0.58	0.72	0.68	0.75
Carryover-12 mechanism	AT(us)	145	311	1498	1491	2566
	DT(us)	52	190	765	825	1368
	ST(us)	197	501	2263	2316	3934
	SP	1.13	1.17	1.10	1.14	1.17
Unique-order interpolative coding	AT(us)	113	182	1066	1076	2011
	DT(us)	82	169	1041	1041	1909
	ST(us)	195	351	2107	2117	3920
	SP	1.14	1.67	1.18	1.24	1.17

5. To obtain better compression rates, Golomb coding and unique-order interpolative coding use a minimal binary code in their codewords. To decode a minimal binary code, “toggle point” calculations are required and slow down query evaluation. Rice coding is a variant of Golomb

coding where the value *b* is restricted to be a power of 2. The advantage of this restriction is that there is no "toggle point" calculation required. The disadvantage of this restriction is the slightly worse compression than that of Golomb coding. If we use Rice coding to encode the boundary and residual pointers in unique-order interpolative coding and use a simple binary code to encode the (*x*, *lo*, *hi*) triples for the inner pointers, there is no "toggle point" calculation required for unique-order interpolative coding. Table 2.8 showed that Rice coding allowed query throughput rates of approximately 8% higher than Golomb coding, and unique-order interpolative coding without "toggle point" calculation allowed query throughput rates of approximately 30% higher than Golomb coding. Experimental results further showed that the decoding time of unique-order interpolative coding without "toggle point" calculation is even less than that of Carryover-12 mechanism.

6. Experimental results showed that a good coding method must be characterized by both high compression ratio and high decompression rate. The unique-order interpolative coding is such a good method.

Table 2.8 Search performance of Rice coding and unique-order interpolative coding (AT is the disk access time, DT is the decoding time, ST=AT+DT is the search time, and SP is the performance relative to the Golomb coding).

Coding Method		Collection				
		Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Rice coding	AT(us)	133	286	1305	1345	2462
	DT(us)	74	267	1004	1069	1808
	ST(us)	207	553	2309	2414	4270
	SP	1.08	1.06	1.07	1.09	1.08
Unique-order interpolative coding[a]	AT(us)	119	193	1128	1137	2127
	DT(us)	55	141	747	772	1363
	ST(us)	174	334	1875	1909	3490
	SP	1.28	1.75	1.32	1.38	1.32

[a] The boundary and residual pointers are encoded in Rice codes, the (*x*, *lo*, *hi*) triples for the inner pointers are encoded in simple binary codes, and group size *g* is 4.

2.5 Other Application

Unique-order interpolative coding, like interpolative coding, can be directly applied to encode strictly ascending integer sequences. One such example is encoding of within-document frequencies of posting lists. If ranked queries are to be supported, it is also necessary to store with each document identifier the frequency of the term appearing within that document, giving the posting list the form:

$$<(id_1, f_{t,1}), (id_2, f_{t,2}), \ldots, (id_{ft}, f_{t,ft}) >,$$

where f_t is the number of documents containing term t, $id_k<id_{k+1}$, and $f_{t,i}$ is the frequency of term t in document i, $1 \le i \le f_t$. The within-document frequencies can be compressed in exactly the same manner of compressing document pointers: if there are f_t entries in a posting list and a total of F_t occurrences of that term in the collection, the sequence of cumulative sums of the $f_{t,i}$ values also forms a strictly increasing integer sequence, and all of the existing compression methods are applicable. Because the within-document frequencies are typically small, according to Table 2.3(b), unique-order interpolative coding should use γ coding to encode within-document frequencies. Table 2.9 shows the cost, in bits per pointer, of storing the within-document frequencies for the five test collections. Test results showed that unique-order interpolative coding achieved very good compression, second to only the interpolative coding. Considering also the performance results in Tables 2.7, we conclude that the unique-order interpolative coding is very suitable for encoding within-document frequencies of posting lists.

Table 2.9 Within-document frequency index compression of all posting lists, in average bits per pointer.

Coding Methods	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Unary coding	1.26	1.37	2.55	2.22	2.37
γ coding	1.38	1.44	2.14	2.00	2.07
Golomb coding	1.30	1.50	2.29	2.09	2.20
Batched LLRUN coding	1.38	1.44	2.14	2.00	2.05
Skewed Golomb coding	1.45	1.60	2.39	2.26	2.35
Interpolative coding	0.86	0.92	1.78	1.77	1.75
Variable byte coding	8.11	8.19	8.04	8.02	8.03
Carryover-12 mechanism	2.04	2.75	3.22	2.99	3.07
Unique-order interpolative coding[a]	0.96	1.02	1.92	1.76	1.84

[a] The boundary and residual pointers are encoded in γ codes and group size g is 4.

2.6 Summary

This chapter proposes a novel coding method, the unique-order interpolative coding, to compress inverted files in IRSs. This method is much easier to implement than interpolative coding. Furthermore, it is custom designed to suit the clustering property of document identifiers, a property that has been observed in real-world document collections. Experiments with the inverted files of five test databases show that this method yields superior performance in both fast querying and space-efficient indexing. This work shows a feasible way in building a responsive and storage-economical IRS.

Chapter 3 Redundant Decoding Elimination

To provide fast query processing, inverted indexes are widely used in *information retrieval systems* (IRSs) (Witten et al., 1999; Zobel et al., 1998). An inverted index consists of an index file and an inverted file. An index file is a set of records, each containing a keyword term *t* and a pointer to the posting list for term *t*. An inverted file contains, for each distinct term *t* in the collection, a posting list of the form

$$PL_t = < (id_1, fq_1), (id_2, fq_2), ..., (id_{f_t}, fq_{f_t}) >,$$

where a posting (*id*,*fq*) indicates that term *t* appears in the document whose identifier is *id* a total of *fq* times (*fq* is referred to as the within-document frequency), and f_t (referred to as the document frequency of term *t*) is the number of documents in which term *t* appears. In a large-scale IRS, posting lists are usually compressed, and decompression of posting lists is hence required during query processing (Zobel & Moffat, 1995; Witten et al., 1999). A query consists of keyword terms. To process a query, the query evaluation engine searches the index file for the query terms to retrieve and decompress the corresponding posting lists. Set operations, such as intersection, union, and difference, are then performed on the posting lists to obtain the query output. The results in the query output are possibly ranked by calculating and examining the score of each document, where the score is usually a function of the within-document frequency and the document frequency of term *t* (Salton, 1989; Salton & McGill, 1983).

In a typical IRS, a few frequently used query terms constitute a large portion of all term occurrences in queries (Jansen et al., 1997). This suggests that it is advisable to store the index records for frequently used query terms in RAM to greatly reduce index search time. Hence, the query processing time of a large-scale IRS is dominated by the time needed to read and decompress

the compressed posting list for each query term (Moffat & Zobel, 1996). Moreover, adding a document into the collection is to add one document identifier into the posting list for each term appearing in the document, hence the length of a posting list increases with the size of document collection. This implies that the time to process posting lists increase as the size of document collection grows. Therefore, further improvement in retrieving and decompressing posting lists becomes necessary.

Compression of an inverted file is the most popular technique used to increase query throughput (Zobel & Moffat, 1995; Williams & Zobel, 1999; Trotman, 2003). This is because the total time of transferring a compressed posting list and subsequently decompressing it is potentially much less than that of transferring an uncompressed posting list. To achieve good compression, the postings in a posting list should be sorted in order of increasing document identifier. Two popular approaches for compressing the document identifiers in the identifier-ordered postings are *d*-gap compression approach (Moffat & Zobel, 1992; Witten et al., 1999) and interpolative coding approach (Moffat & Stuiver, 2000). The *d*-gap compression approach consists of two steps. It first replaces each document identifier (except the first one) with the distance between itself and its predecessor. For example, the document identifiers in the identifier-ordered postings <13, 18, 22, 35, 42> can be transformed into the *d*-gaps as <13, 5, 4, 13, 7>. And the second step is to encode these *d*-gaps using an appropriate coding method, such as unary coding (Elias, 1975), γ coding (Elias, 1975), or Golomb coding (Golomb, 1966; Witten et al., 1999). The common nature of these coding methods is their variable-length representations in which small *d*-gaps can be coded more economically than large ones. Interpolative coding approach, on the other hand, directly compresses the original document identifiers with a recursive process calculating the lower and upper bounds of every document identifier. Then every document identifier is encoded in a binary code. Moffat & Stuiver

(2000) showed that the compression result of interpolative coding is better than that of *d*-gap compression approach. The drawback of interpolative coding is its slow decompression due to a stack required in its decoding loops. The within-document frequencies in the identifier-ordered postings can also be encoded efficiently by using γ coding or interpolative coding (Bell et al., 1993; Moffat & Zobel, 1992; Moffat & Stuiver, 2000).

The query performance on a compressed inverted file can be further improved by using skipping mechanisms (Moffat et al., 1995; Moffat & Zobel, 1996; Anh & Moffat, 1998). Although compression can greatly reduce disk access time, the compressed posting list for each query term must be completely decompressed in order to be randomly accessed to any posting in it. Where as in processing queries, usually only a subset of the postings in a posting list needs to be examined. To save redundant decoding, skipping mechanisms (Moffat et al., 1995; Moffat & Zobel, 1996; Anh & Moffat, 1998) that allow queries to be processed with only partial decoding of the list have been proposed. A common skipping mechanism is to divide the posting list into blocks and add auxiliary information into each block, so that the postings within a block can be quickly skipped without decoding them if they are useless in set operations during query processing. There are two important types of queries: conjunctive Boolean queries and ranked queries. For conjunctive Boolean queries large blocks provide faster searching for candidates, whereas for ranked queries small blocks are favored (Moffat & Zobel, 1996; Anh & Moffat, 1998). We observed that all well-known skipping mechanisms can incur high storage overheads if the posting lists are divided into small blocks. The increase in disk I/O time outweighs the reduction in decompression time. Therefore, a novel skipping mechanism that can support small blocks with very little storage overhead should be developed.

In this chapter, we deal with posting list skipping problem for both the conjunctive Boolean queries and ranked queries in one design. We propose a two-level skipped inverted file, in which a two-level skipped index is created on each compressed posting list, to remove redundant decoding and allow fast query evaluation. We first employ well-known skipping mechanisms to create the first-level index on each posting list by dividing the list into blocks. The first-level index is constructed with large blocks and designed for optimizing the query performance of conjunctive Boolean queries. A novel skipping mechanism is then proposed to create the second-level index on each block for optimizing the query performance of ranked queries. It first divides each block into sub-blocks, each containing a fixed number of postings. Then it employs functions to accurately calculate the maximum required bits that will be allocated and reserved to store the postings within a sub-block, and that can be easily skipped. The novel skipping mechanism works the best for small sub-blocks and has significant advantages for ranked queries. Experimental results show that the proposed two-level skipped inverted file provides excellent query speed on both conjunctive Boolean queries and ranked queries with very little or no storage overhead.

The remainder of this chapter is organized as follows. Section 3.1 describes two well-known skipping mechanisms and their posting list structures for inverted files. Our test document collection is described in Section 3.2. In Section 3.3, we present the proposed two-level skipped inverted file. The performance evaluation is presented in Section 3.4. Finally, Section 3.5 presents our summary.

3.1 Two Well-known Skipping Mechanisms and Their Posting List Structures

Moffat & Zobel (1996) and Moffat et al. (1995) proposed two well-known skipping mechanisms to eliminate redundant decoding and allow fast candidate searching. Two posting list

structures are employed in their proposed skipping mechanisms. This section presents these two posting list structures, and comments on them.

3.1.1 Skipped inverted file

Moffat & Zobel (1996) proposed the skipped inverted file to avoid redundant decoding and allow fast processing of conjunctive search queries. The idea is to divide the compressed posting list into blocks each containing a fixed number, k, of postings. The first document identifier of each block is referred to as the critical document identifier, and it is associated with some extra bits that specify the location of the next critical document identifier. For example, consider the set of (id,fq) postings in a given posting list

(4,2), (6,1), (11,1), (13,2), (14,1), (19,2), (24,1), (27,2), (30,2), (42,1)…

For the number of postings per block k=3, the posting list can be represented as

((4,a_1),2), (6,1), (11,1), ((13,a_2),2), (14,1), (19,2), ((24,a_3),1), (27,2), (30,2), ((42,a_4),1)…,

where a_i is the address of the first bit of the $(i+1)^{th}$ critical document identifier. The document identifiers (except the critical document identifier) within a block can be stored as d-gaps:

((4,a_1),2), (2,1), (5,1), ((13,a_2),2), (1,1), (5,2), ((24,a_3),1), (3,2), (3,2), ((42,a_4),1)…,

Finally, the critical document identifiers and the addresses can also be stored as d-gaps:

((4,a_1),2), (2,1), (5,1), ((9, a_2-a_1),2), (1,1), (5,2), ((11,a_3-a_2),1), (3,2), (3,2), ((18,a_4-a_3),1)…,

To search the compressed posting list for a document identifier id, the first step is searching in the critical document identifier list and the second step is searching in one targeted block. Note that within each block each (id,fq) posting is still code-dependent upon its predecessor. If the candidate answers do not exist in that block, the postings (except the critical document identifier) within a block can be quickly skipped without decoding, resulting in reduced decompression time. When implementing a skipped inverted file, Golomb coding is used to code the d-gaps of document

identifiers and the addresses, whereas γ coding is used to code the within-document frequencies (Moffat & Zobel, 1996).

3.1.2 Blocked inverted file

The skipped inverted file uses *k*-posting blocks, so the blocks themselves are of differing length. On alternative, called blocked inverted file in which the posting list can be modified to provide faster checking of individual candidates, is to break the posting lists into blocks of the same size in bits (Moffat et al., 1995). The first document identifier of each block is also called critical document identifier. Let b be the number of bits for each block, then the i^{th} block starts at bit location $1+(i-1)\times b$. Therefore, the address that specify the location of next critical document identifier can be omitted in a blocked posting list. Counterbalancing this gain, on average half a (*id*,*fq*) posting per block will be lost. Each compressed (*id*,*fq*) posting in a posting list occupies about 8 bits (Witten et al., 1999), so 4 bits per block will be unused.

In a blocked posting list, the critical document identifier can be stored completely uncompressed, and a binary search for critical document identifier can be carried out. This clearly offers much faster accesses to candidates than the skipped inverted file since there is no decoding of the critical document identifier required, and only logarithmically many of them have to be examined (Moffat et al., 1995). However, this leads to additional space wastage. For a collection of N=1,000,000 documents to be indexed and the number of bits per block b=128, the use of an uncompressed critical document identifier adds about 10% to the size of the compressed inverted file (Moffat et al., 1995). The space overhead ratio will increase if the size of document collection N grows or the number of bits per block b decreases. In implementing a blocked inverted file, Golomb coding is used to code the *d*-gaps of document identifiers (except critical document identifiers), whereas γ coding is used to code the within-document frequencies (Moffat et al., 1995).

Moffat et al. (1995) indicated that a binary search for the compressed critical document identifiers can be carried out if the blocks are stored in interpolative manner, and reduced space overheads can be achieved. This, however, leads to slow decompression for critical document identifiers due to a stack required in its decoding loops. Therefore, the blocked inverted file to be implemented in this chapter uses uncompressed critical document identifiers.

3.1.3 Remarks

For both skipped inverted files and blocked inverted files, we cannot find a fixed value of k or b to simultaneously optimize the query evaluation of conjunctive Boolean queries and ranked queries. This is because conjunctive Boolean queries favor large blocks, whereas for ranked queries favor small blocks. Two different indexes might be constructed if speed on both types of query is at a premium (Moffat & Zobel, 1996). A trivial solution to this problem is to employ a two-level skipping mechanism, where the first level of skipping divides the compressed posting list into large blocks for optimizing the query performance of conjunctive Boolean queries, and the second level divides each large block into small sub-blocks for optimizing the query performance of ranked queries. However, both skipped inverted file and blocked inverted file are inappropriately used for smaller sub-blocks due to their high storage overheads. To create a space-efficient two-level skipped index for providing excellent speed on both types of query, we propose a novel skipping mechanism to support smaller sub-blocks with very little or no storage overhead in Section 3.3.

3.2 Test Data

The document collection used for the experiments in this research is the disk 5 of the TREC-6 collection (Voorhees & Harman, 1997). We have broken the longer documents into pages of around 1000 bytes to ensure that retrieved text is always of a size that can be digested by the user (Zobel et al., 1995). In the paged form of the test document collection, there are 1,025,469 pages totaling 945MB, an average of 141.4 terms per page, and 317,393 distinct terms, after folding all letters to lowercase and removing variant endings using Lovin's stemming algorithm (Lovins, 1968). Each page is mapped to a unique document identifier. The inverted file comprises 93,226,576 stored (*id*,*fq*) postings.

3.2.1 Conjunctive Boolean queries

We followed the method (Moffat & Zobel, 1996) to generate random conjunctive Boolean queries. For the test document collection, 300 pages were randomly selected to generate a query set. A query was generated by selecting words from the word list of a specific page. The number of terms per query ranged from 1 to 8. For example, a query containing 5 terms may be "inverted file document collection built". For each query, there existed at least one page that was relevant to the query. We also made the generated query set have the following characteristics: (1) Query repetition frequencies followed a Zipf distribution (Xie & O'Hallaron, 2002); (2) The terms per query distribution followed a shifted negative binomial distribution (Wolfram, 1992). This made the distribution of generated queries closely resemble the distribution of real queries. Table 3.1 shows the average number of candidate pages and the average number of (*id*,*fq*) postings considered when processing the generated queries, for each query size.

Table 3.1 Processing of generated conjunctive Boolean queries.

Number of terms	Average number of candidate pages	Average number of (*id*,*fq*) postings
1	42,763	42,763
2	4,814	85,223
3	1,096	127,491
4	459	169,343
5	211	210,665
6	113	251,728
7	43	292,476
8	22	333,082

3.2.2 Ranked queries

50 pages were randomly selected to generate the test ranked query set. For each of the selected pages, we eliminated stopwords and removed all nonalphabetic characters, and case-folded and stemmed the resulting words. This gave a set of 50 queries containing, on average 50.2 distinct terms, and on average 2,050,000 of the (*id*,*fq*) postings processed per query, and 41,000 postings per term per query. We allowed multiple appearances of terms to influence the weighting given to that term. When using the continue algorithm (Moffat & Zobel, 1996) to evaluate ranked queries, the average number of (*id*,*fq*) postings needed to be checked against the posting list for each query term may range from 0.2 to 2.0 percent of *N*, where *N* is the number of pages in the collection.

3.3 Proposed Two-level Skipped Inverted Files

In this section, we first describe the framework of the proposed two-level skipped inverted file. Then we propose a novel skipping mechanism to optimize the query performance of ranked queries with very little or no storage overhead.

3.3.1 Framework of proposed approach

For skipped inverted files, Moffat & Zobel (1996) showed that the total decoding time required to search a posting list containing *p* postings for *c* candidates can be minimized if the posting list is

divided into blocks each containing $2\sqrt{cp}/c$ postings. According to Table 3.1, this indicates that the number of postings per block, *k*, should be set at a value ranged from 6 to 88 for optimizing the query performance of conjunctive Boolean queries; while according to Section 3.2.2, this indicates that the number of postings per block, *k*, should be set at a value ranged from 3 to 9 for optimizing the query performance of ranked queries. However, we observed that a skipped inverted file is inappropriately used for ranked queries. When $k \leq 8$ the size of the skipped inverted file is much larger than that of an un-skipped compressed inverted file, this incurs more read time and dramatically absorbs the CPU gains. A novel skipping mechanism that can support smaller blocks with little space overhead should be developed. We also observed that blocked inverted files are faced with the same problem.

In this chapter, we propose a two-level inverted file, in which a two-level index is created on each compressed posting list, to simultaneously optimize the query performance of conjunctive Boolean queries and ranked queries. The idea is that the first-level index is designed for optimizing the query performance of conjunctive Boolean queries, whereas the second-level index is designed for ranked queries. We observed that well-known skipping mechanisms can work well for the first-level indexing; hence the key to the proposed two-level skipped inverted file is to develop a novel skipping mechanism that can efficiently support the second-level indexing. The framework of the proposed two-level skipped index on each compressed posting list is as follows:

The first-level index: One of the skipping mechanisms proposed by Moffat et al. (1995) and Moffat & Zobel (1996) is first used to create the first-level index on each compressed posting list by dividing the posting list into large blocks and adding auxiliary information into each block to skip over unnecessary portions of the list.

The second-level index: A novel skipping mechanism is then proposed to create the second-level index on each large block by dividing the block into sub-blocks and adding auxiliary information into each sub-block to skip over unnecessary portions of the block.

To ensure that skipped inverted files do not become too large, we require that every block contains at least 17 postings. This adds about 10% to the size of the un-skipped compressed inverted file and can reduce considerable decompression time with acceptable space overhead. For blocked inverted files, we also require that every block contains at least 128 bits. This is because that each compressed (*id*,*fq*) posting occupies about 8 bits (Witten et al., 1999); a skipped inverted file with k=17 corresponds to a blocked inverted file with b of about 128.

The next section describes the proposed skipping mechanism for the second-level index.

3.3.2 Proposed skipping mechanism

In this section, we first describe the proposed skipping mechanism based on *maximum required bits* (*MRB*) calculation. Then we present the recommended coding method and its *MRB* function for the document identifiers and the within-document frequencies within a sub-block. Finally, we present the implementation optimization technique.

The design

In this sub-section, we propose a novel skipping mechanism based on *maximum required bits* (*MRB*) calculation (cf. Fig. 3.1) to efficiently create a second-level index on each block for the first level of skipping. Consider a given block containing n postings

$$(id_1,fq_1), (id_2,fq_2), (id_3,fq_3), \ldots, (id_n,fq_n)$$

where $id_i<id_{i+1}$. We first replace the within-document frequency fq_i with the F_i, where $F_i = \sum_{j=1}^{i} fq_j$ is referred to as the cumulative within-document frequency. Next a sub-block size g is determined. The block is then divided into $m = \lceil n/g \rceil$ sub-blocks, each having g postings except possibly the

last block. We define the first posting in each sub-block to be a critical pair consisting of a document identifier and a cumulative within-document frequency, the postings between critical pairs to be inner postings, and those in the last sub-block except the critical pair to be the residual postings. The critical pairs and their subsequent residual postings together can be regarded as a sub-posting list, on which the document identifiers can be encoded in Golomb coding with the *d*-gap technique and the cumulative within-document frequencies can be encoded in γ coding also with the *d*-gap technique. For the inner postings within a sub-block, the document identifiers and the cumulative within-document frequencies are stored separately (cf. Fig. 3.1). Assume that the document identifiers in the inner postings are to be compressed with compression method *C*1,

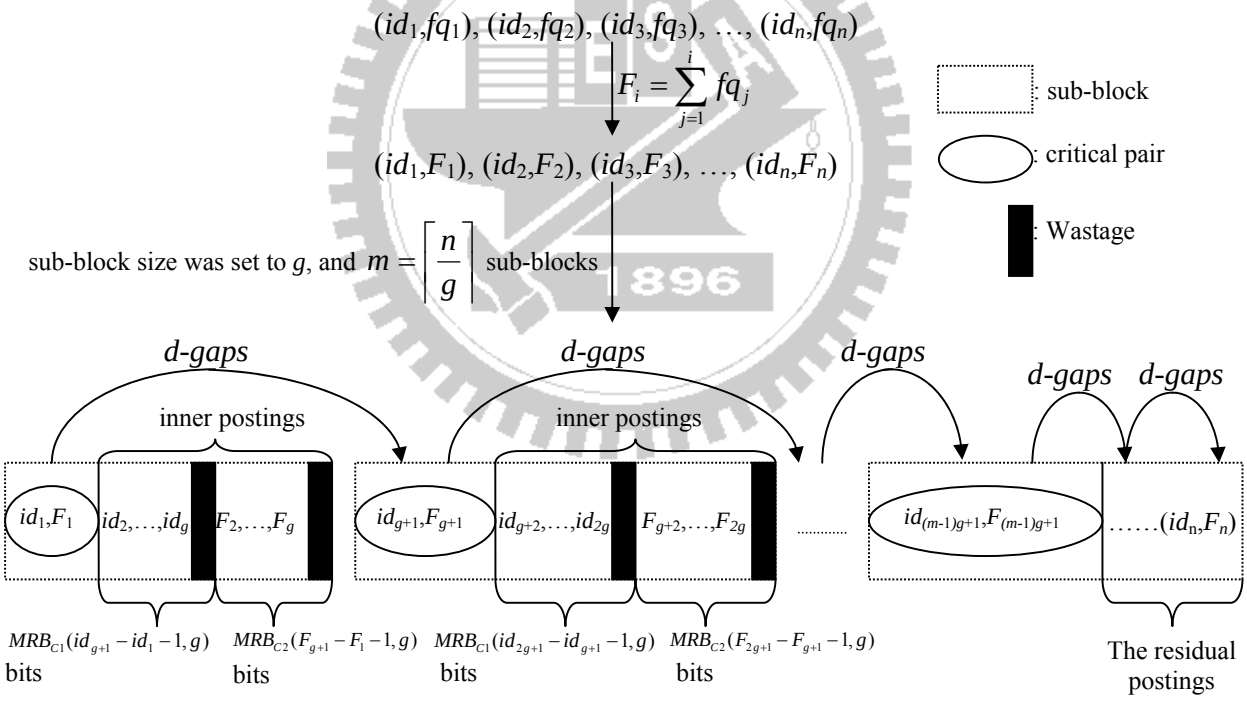


Figure 3.1 Illustration of the proposed skipping mechanism. Assume that the document identifiers in the inner postings are to be compressed with compression method *C*1, and the cumulative within-document frequencies are with compression method *C*2. The function $MRB_C(x_{j+g}-x_j-1,g)$ can calculate the maximum required bits that need to be allocated to store the strictly ascending integer sequences $x_{j+1},x_{j+2},\ldots x_{j+g-1}$ compressed with method *C*, where *x* can be either *id* or *F* and *C* can be either *C*1 or *C*2.

and the cumulative within-document frequencies are with compression method $C2$. We want to find two functions $MRB_{C1}(DI_i,g)$ and $MRB_{C2}(DF_i,g)$ to precisely calculate the maximum required bits that need to be allocated to store the document identifiers compressed with method $C1$ and the cumulative within-document frequencies compressed with method $C2$, respectively, in the inner postings within the i^{th} sub-block, where $DI_i=IC_i-IC_{i+1}-1$ and IC_i is the document identifier for the i^{th} critical pair, and $DF_i=FC_i-FC_{i+1}-1$ and FC_i is the cumulative within-document frequency for the i^{th} critical pair. Since the maximum number of bits for the document identifiers and the cumulative within-document frequencies in the inner postings within a sub-block is known, those identifiers and frequencies that are useless in set operations during query processing can be skipped easily. In this mechanism, the critical pair for the $(i+1)^{th}$ sub-block should be stored before the inner postings for the i^{th} sub-block. Compared with the skipping mechanism proposed by Moffat & Zobel (1996), this mechanism does not require extra bits to specify the location of critical document identifiers. However, the space overhead of this mechanism is still possibly high if the estimation function is not accurate. The key to the success of this skipping mechanism is to find efficient coding methods with accurate functions for compressing the document identifiers and the cumulative within-document frequencies in the inner postings within a sub-block.

Recommended coding method and its *MRB* function for inner postings

For the proposed skipping mechanism, interpolative coding is recommended for compressing both the document identifiers and the cumulative within-document frequencies. The reasons are:

(1) Interpolative coding can yield superior compression performance for both document identifiers and cumulative within-document frequencies (Moffat & Stuiver, 2000).

(2) When the group size g is known, Chapter 2 showed that the decoding process for interpolative coding can be greatly facilitated using recursion elimination and loop unwinding, this provides high query throughput rate.

(3) Consider a sequence of $(g-1)$ numbers x_{j+1} to x_{j+g-1} constrained by $x_j<x_{j+1}<x_{j+2}<\ldots<x_{j+g-1}< x_{j+g}$. When the group size g=4, we can show that the maximum required bits for the interpolative coding can be derived as

$$MRB_{interp}(D, g=4) = \begin{cases} 0 & if\ D=3 \\ 2 & if\ D=4 \\ 3(h+1)+1 & if\ 4<D<3\times 2^h+3 \\ 3(h+1)+2 & if\ 3\times 2^h+3\le D \end{cases} \quad (3.1)$$

where $D=(x_{j+g}-x_j-1)$ and $h=\lceil \log_2(D-2)\rceil-2$. This function is the closed form of Eq.(2.12) and can calculate the maximum required bits for the document identifiers and the cumulative within-document frequencies in the inner postings within a sub-block with very little space overhead.

With interpolative coding, to allow different values of g, one can easily show that

$$MRB_{interp}(D, g=8) = \lceil \log_2(D-6)\rceil + MRB_{interp}(p,4) + MRB_{interp}(q,4)$$

and this can be converted to

$$MRB_{interp}(D, g=8) = \begin{cases} 0 & if\ D=7 \\ 3 & if\ D=8 \\ 6 & if\ D=9 \\ 8 & if\ D=10 \\ 7(h+1)+4 & if\ 10<D\le 5\times 2^h+7 \\ 7(h+1)+5 & if\ 5\times 2^h+7<D\le 6\times 2^h+7 \\ 7(h+1)+7 & if\ 6\times 2^h+7<D\le 7\times 2^h+7 \\ 7(h+1)+8 & if\ 7\times 2^h+7<D \end{cases} \quad (3.2)$$

where $D=(x_{j+g}-x_j-1)$, $h=\left\lceil \log_2\frac{(D-6)}{2}\right\rceil-2$, and p, q are two positive integers and p+q=D-1.

Applying the same approach, we have

$$MRB_{interp}(D, g=16) = \lceil \log_2(D-14) \rceil + MRB_{interp}(p,8) + MRB_{interp}(q,8)$$

and this can be converted to

$$MRB_{interp}(D, g=16) = \begin{cases} 0 & if \quad D=15 \\ 4 & if \quad D=16 \\ 8 & if \quad D=17 \\ 11 & if \quad D=18 \\ 15 & if \quad D=19 \\ 17 & if \quad D=20 \\ 20 & if \quad D=21 \\ 22 & if \quad D=22 \\ 15(h+1)+11 & if \quad 22 < D \le 9\times 2^h + 15 \\ 15(h+1)+12 & if \quad 9\times 2^h + 15 < D \le 10\times 2^h + 15 \\ 15(h+1)+14 & if \quad 10\times 2^h + 15 < D \le 11\times 2^h + 15 \\ 15(h+1)+15 & if \quad 11\times 2^h + 15 < D \le 12\times 2^h + 15 \\ 15(h+1)+18 & if \quad 12\times 2^h + 15 < D \le 13\times 2^h + 15 \\ 15(h+1)+19 & if \quad 13\times 2^h + 15 < D \le 14\times 2^h + 15 \\ 15(h+1)+21 & if \quad 14\times 2^h + 15 < D \le 15\times 2^h + 15 \\ 15(h+1)+22 & if \quad 15\times 2^h + 15 < D \end{cases} \tag{3.3}$$

where $D = (x_{j+g} - x_j - 1)$, $h = \left\lfloor \log_2 \frac{(D-14)}{4} \right\rfloor - 2$, and p, q are two positive integers and p+q=D-1.

The proposed skipping mechanism can be directly employed to create the first-level index by dividing the compressed posting list into blocks each containing g postings. Table 2 shows the size of the inverted files constructed using the proposed skipping mechanism with different g values. The results show that this skipping mechanism can efficiently support smaller sub-blocks. The size of inverted files constructed using this mechanism can be even smaller than that of a compressed inverted file in which the document identifiers are compressed in Golomb codes with the d-gap technique and the within-document frequencies are in γ codes. Note that the file size increases as the value of g increases, so this skipping mechanism works the best for smaller blocks.

When this skipping mechanism is employed to create the second-level index, to optimize the query performance of ranked queries requires that the sub-block size be set at smaller values of g. For a simple implementation and which requires space efficiency, we suggest g=4. Note that when applying this skipping mechanism to a blocked inverted file to create the second-level index on each block, a unary code should be added in each block to indicate the number of sub-blocks in the block. Other coding methods are not disregarded. We are still looking for a faster and more effective coding method to encode the document identifiers or the cumulative within-document frequencies.

Table 3.2 Sizes of inverted files constructed using the proposed skipping mechanism with different g values.

Inverted file organization	Size	
	MB	%
compressed inverted file	93.28	100.0
the inverted file by the proposed skipping mechanism		
g=4	89.33	95.8
g=8	93.06	99.8
g=16	96.21	103.1

Implementation optimization

To skip over unnecessary inner postings, this skipping mechanism requires calculating the maximum required bits for both document identifiers and cumulative within-document frequencies. We observed that in most cases the gap value D in Eq. (3.1) is less than 256. Therefore, a 256-entry array z is used to facilitate the calculation of the maximum required bits, and $z[i]=MRB_{interp}(i, g=4)$, $i= x_{j+g}- x_j-1$, for $3 \le i \le 255$. Whenever the gap value in Eq. (3.1) is less than 256, we can obtain the corresponding maximum required bits with only one array access. This greatly reduces the CPU time and improves query performance.

3.4 Performance Evaluation

This section presents our experiments to evaluate the efficiency of various inverted file organizations. We used the standard (un-skipped) compressed inverted file as the baseline, in which *d*-gaps are encoded in Golomb codes with the parameter *b* chosen appropriately for each posting list (Witten et al., 1999), and within-document frequencies are encoded in γ codes (Bell et al., 1993; Moffat & Zobel, 1992). This baseline is then used to evaluate other fine-tuned skipped inverted file organizations.

Four skipped inverted file organizations are evaluated in our experiments: the skipped inverted file (described in Section 3.1.1), the blocked inverted file (described in Section 3.1.2), the skipped inverted file with the 2^{nd}-level index, and the blocked inverted file with the 2^{nd}-level index. The 2^{nd}-level index is created using the skipping mechanism (g=4) described in Section 3.3.2.

All experiments were run on an Intel P4 2.4GHz PC with 512MB DDR memory running Linux operating system 2.4.12. The hard disk was 40GB, and the data transfer rate was 25MB/sec. Intervening processes and disk activities were minimized with best effort during experimentation.

In Section 3.4.1, we present the sizes for various inverted file organizations. In Section 3.4.2, we present the time taken to process the generated queries described in Section 3.2 to measure the query performance of various inverted file organizations.

3.4.1 Sizes for various inverted file organizations

The actual size for each inverted file organization is shown in Table 3.3. As expected, the sizes of the skipped inverted files and the blocked inverted files are larger than that of standard compressed inverted file. The space overheads associated with both the skipped inverted files and the blocked inverted files increase as the block length decreases. This confirms that smaller blocks are inappropriate for both the skipped inverted file and the blocked inverted file. The skipping

mechanism proposed in Section 3.3.2 is used to create the 2nd-level index on each block for both the skipped inverted files and the blocked inverted files. Experimental results show that the skipping mechanism can incur no space overhead in creating the 2nd-level index. Furthermore, we observed that the size of the skipped inverted file with the 2nd-level index can even be less than that of standard compressed inverted file for larger *k* values. This provides a space-economical way to implementing a two-level skipped inverted file.

Table 3.3 Sizes of various inverted file organizations. The sizes are presented in both megabytes and ratio to the standard compressed inverted file size. For skipped inverted files, *k* is the number of postings per block. For blocked inverted files, *b* is the length of each block in bits.

Inverted file organization	Size	
	MB	%
compressed inverted file	93.28	100.0
skipped inverted file		
k=17	102.74	110.1
k=33	98.63	105.7
k=65	96.24	103.2
blocked inverted file		
b=128	107.89	115.7
b=256	101.11	108.4
b=512	98.10	105.2
skipped inverted file with 2nd-level index[a]		
k=17	98.34	105.4
k=33	94.50	101.3
k=65	92.27	98.9
blocked inverted file with 2nd-level index[a]		
b=128	107.61	115.4
b=256	100.40	107.6
b=512	97.20	104.2

[a] the 2nd-level index is created by using the novel skipping mechanism (*g*=4) described in Section 3.3.2

3.4.2 Elapsed time required to process queries

In this subsection, we present the time taken to process the conjunctive Boolean queries and the ranked queries, with various inverted file organizations. The *query processing time* (QPT) presented includes: (1) the disk read time of compressed posting list for each query term, and (2) the CPU time measured from the query being issued until the list of answer document identifiers being finalized. The QPT does not include the time taken to retrieve and display answers. All programs were optimized as follows:

1. Replaced subroutines with macros.
2. Careful choice for compiler optimization flags.
3. Implementation used 32-bit integers, as that is the internal register size of the Intel P4 CPU.
4. Implemented the integer logarithm functions $\lceil \log_2(i) \rceil$ and $\lfloor \log_2(i) \rfloor$ with a 256-entry lookup table.
5. Another 256-entry lookup table was used to locate the exact bit location of the first "1" bit in a byte.
6. Accessed to binary codes with masking and shifting operations, and no bit-by-bit decoding were required.

With these optimizations, decoding of a document identifier only required tens of ns, and no bit-by-bit decoding is required.

Conjunctive Boolean queries

When processing a conjunctive Boolean query, the posting lists for the query terms are processed in order of increasing document frequency f_t. The time taken to process the conjunctive Boolean queries with various inverted file organizations is shown in Table 3.4. Except for single-term query, both the skipped inverted files and the blocked inverted files can improve query

performance by skipping over unnecessary portions of the compressed posting lists. Experimental results show that the skipped inverted file can achieve an average speedup of 2.80 to 3.04, and the blocked inverted file 3.08 to 3.33, compared with the standard compressed inverted file. When the number of terms ≥ 4, the blocked inverted files far outperform the skipped inverted files. This is because the number of candidate answers is much less when the number of terms ≥ 4 (cf. Table 3.1) and the binary search supported by blocked inverted file works well.

Experimental results also show that the 2^{nd}-level index created by the skipping mechanism has substantial and consistent potential to improve the query performance. For the skipped inverted file, the 2^{nd}-level index can improve the average query speed by up to 11%; while for the blocked inverted file, it can be up to 16%.

Ranked queries

Ranked queries are disjunctive rather than conjunctive, in that any document containing any of the queried terms is considered a candidate. Skipping mechanisms do not necessarily yield significant benefits in the evaluation of ranked queries. To improve ranked query evaluation with skipping mechanisms, Moffat & Zobel (1996) proposed a pruning algorithm, called *continue* algorithm, to reduce the number of candidates during the evaluation of ranked queries. They showed that the continue algorithm for ranked queries can exploit fast search made possible by skipping mechanisms, and results in improved ranked query evaluation without any substantial degradation in retrieval effectiveness. We adopt the continue algorithm to evaluate ranked queries in this experiment. The similarity of a query and a document was calculated by the cosine measure (Salton, 1989; Salton & McGill, 1983). The maximum number of accumulators was set at 0.2, 0.5, 1.0 and 2.0 percent of N (the number of pages in the test collection).

Table 3.4 Conjunctive Boolean query performance of various inverted file organizations (QPT is the average query processing time of conjunctive Boolean queries, in ms; SP is the speedup relative to the standard compressed inverted file). For skipped inverted files, *k* is the number of postings per block. For blocked inverted files, *b* is the length of each block in bits.

Number of terms		Inverted file organization												
		compressed inverted file	skipped inverted file			blocked inverted file			skipped inverted file with 2nd-level index[a]			blocked inverted file with 2nd-level index[a]		
			k=17	*k*=33	*k*=65	*b*=128	*b*=256	*b*=512	*k*=17	*k*=33	*k*=65	*b*=128	*b*=256	*b*=512
1	QPT	3.60	3.68	3.65	3.63	3.84	3.82	3.81	3.92	3.83	3.75	3.92	3.94	3.90
	SP	1.00	0.98	0.99	0.99	0.94	0.94	0.94	0.92	0.94	0.96	0.92	0.91	0.92
2	QPT	6.82	5.16	5.43	5.79	6.10	6.16	6.37	5.10	5.05	5.04	5.83	5.50	5.31
	SP	1.00	1.32	1.26	1.18	1.12	1.11	1.07	1.34	1.35	1.35	1.17	1.24	1.28
3	QPT	9.42	4.25	4.27	4.61	4.30	4.53	4.98	4.22	3.88	3.83	4.09	3.93	3.92
	SP	1.00	2.22	2.21	2.04	2.19	2.08	1.89	2.23	2.43	2.46	2.30	2.40	2.40
4	QPT	11.23	3.72	3.45	3.53	3.08	3.23	3.51	3.69	3.20	3.01	2.97	2.87	2.86
	SP	1.00	3.02	3.26	3.18	3.65	3.48	3.20	3.04	3.51	3.73	3.78	3.91	3.93
5	QPT	11.98	3.37	2.95	2.89	2.39	2.46	2.66	3.35	2.80	2.53	2.35	2.26	2.25
	SP	1.00	3.55	4.06	4.15	5.01	4.87	4.50	3.58	4.28	4.74	5.10	5.30	5.32
6	QPT	12.40	3.15	2.63	2.48	1.96	2.01	2.15	3.10	2.45	2.21	1.95	1.87	1.87
	SP	1.00	3.94	4.71	5.00	6.33	6.17	5.77	4.00	5.06	5.61	6.36	6.63	6.63
7	QPT	13.04	3.05	2.45	2.24	1.67	1.68	1.80	3.01	2.35	2.03	1.70	1.64	1.62
	SP	1.00	4.28	5.32	5.82	7.81	7.76	7.24	4.33	5.55	6.42	7.67	7.95	8.05
8	QPT	13.99	3.04	2.33	2.01	1.43	1.43	1.50	3.02	2.15	1.89	1.48	1.42	1.42
	SP	1.00	4.60	6.00	6.96	9.78	9.78	9.33	4.63	6.51	7.40	9.45	9.85	9.85
Avg	QPT	10.31	3.68	3.40	3.39	3.10	3.17	3.35	3.68	3.21	3.04	3.04	2.93	2.89
	SP	1.00	2.80	3.03	3.04	3.33	3.25	3.08	2.80	3.21	3.39	3.39	3.52	3.57

[a] the 2nd-level index is created by the skipping mechanism (*g*=4) described in Section 3.3.2

The time taken to process the ranked queries with various inverted file organizations is shown in Table 3.5. Experimental results show that the skipped inverted file can achieve an average speedup of 1.23 to 1.59, and the blocked inverted file can achieve an average speedup of 1.09 to 1.36, compared with the standard compressed inverted file. In most cases, the skipped inverted files outperform the blocked inverted files. This is because that the number of candidate answers is larger and the binary search supported by blocked inverted files cannot be used to produce good performance. For both the skipped inverted files and the blocked inverted files, smaller blocks provide better query performance. This confirms our assessment that small blocks have significant

advantages for ranked queries. As the maximum number of accumulators increases, the query speedup of both the skipped inverted files and the blocked inverted files decreases. When the maximum number of accumulators was set at 2.0 percent of N, the query performance of the skipped inverted file (k=65) and all the blocked inverted files were even worse than that of the standard compressed inverted file. To improve the ranked query performance, the 2nd-level index created by the skipping mechanism is applied to the skipped inverted files and the blocked inverted files. For the skipped inverted file, the 2nd-level index can improve the average query speed by up to 38%; while for the blocked inverted file, it can improve the average query speed by up to 44%. This fact shows that a space-efficient 2nd-level index can provide fast candidate search for ranked queries.

Table 3.5 Ranked query performance of various inverted file organizations (QPT is the average query processing time of ranked queries, in mini-seconds; SP is the speedup relative to the standard compressed inverted file). For skipped inverted files, k is the number of postings per block. For blocked inverted files, b is the length in bits of each block.

% of N		Inverted file organization												
		compressed inverted file	skipped inverted file			blocked inverted file			skipped inverted file with 2nd-level index[a]			blocked inverted file with 2nd-level index[a]		
			k=17	k=33	k=65	b=128	b=256	b=512	k=17	k=33	k=65	b=128	b=256	b=512
0.2	QPT	100.4	38.2	41.2	50.3	36.0	43.7	55.8	36.1	33.0	33.9	33.1	34.2	36.6
	SP	1.00	2.63	2.44	2.00	2.79	2.30	1.80	2.78	3.04	2.96	3.03	2.94	2.74
0.5	QPT	109.0	57.6	65.2	78.1	64.5	74.3	99.4	53.7	51.9	53.7	59.1	58.2	59.2
	SP	1.00	1.89	1.67	1.40	1.69	1.47	1.10	2.03	2.10	2.03	1.84	1.87	1.84
1.0	QPT	116.1	79.8	90.7	105.2	96.5	105.0	116.8	74.5	73.4	75.4	88.7	84.3	82.6
	SP	1.00	1.45	1.28	1.10	1.20	1.11	0.99	1.56	1.58	1.54	1.31	1.38	1.41
2.0	QPT	124.1	107.7	119.7	132.7	133.4	136.2	142.1	102.0	100.9	101.6	124.3	114.3	108.8
	SP	1.00	1.15	1.03	0.94	0.93	0.91	0.87	1.22	1.23	1.22	1.00	1.09	1.14
Avg	QPT	112.4	70.8	79.2	91.6	82.6	89.8	103.5	66.6	64.8	66.2	76.3	72.8	71.8
	SP	1.00	1.59	1.42	1.23	1.36	1.25	1.09	1.69	1.73	1.70	1.47	1.54	1.57

[a] the 2nd-level index is created by using the novel skipping mechanism (g=4) described in Section 3.3.2

3.5 Summary

This chapter proposes a two-level skipped inverted file to facilitate fast conjunctive Boolean queries and ranked queries. For this purpose, well-known skipping mechanisms are first used to create the first-level index on each compressed posting list by dividing the posting list into large blocks for optimizing conjunctive Boolean queries. Then a skipping mechanism is proposed to create the second-level index on each block by dividing the large block into small sub-blocks for optimizing ranked queries. Compared with well-known skipping mechanisms, this novel skipping mechanism can support second level of skipping with no storage overhead. Experiments clearly indicate that the proposed two-level skipped inverted file can improve the query speed for conjunctive Boolean queries by up to 16%, and for ranked queries by up to 44%, compared with the conventional one-level skipped index. This provides a very simple and attractive way to building a fast and space-economical IRS.

Chapter 4 Inverted File Optimization

Inverted files are widely used in modern large-scale IRSs for fast query evaluation. Compressing an inverted file can greatly increase query throughput (Zobel & Moffat, 1995; Williams & Zobel, 1999). This is because the total time of transferring a compressed posting list and subsequently decompressing it is potentially much less than that of transferring an uncompressed posting list. The query processing time in a large-scale IRS is dominated by the time needed to read and decompress the posting lists for the terms involved in the query (Moffat & Zobel 1996), and we observe that the query processing time grows with the total encoded size of the corresponding posting lists. This is because the disk transfer rate is near constant, and the decoding processes of most encoding methods used in the *d*-gap compression approach are on a bit-by-bit basis. If we can reduce the total encoded size of the corresponding posting lists without increasing decompression times, a shorter query processing time can be obtained.

A *document identifier assignment* (DIA) can make the document identifiers in the posting lists evenly distributed, or clustered. Clustered document identifiers generally result in better compression efficiency of the coding methods used for compressing inverted files without increasing the complexity of decoding process, hence reduce the query processing time. In this chapter, we consider the problem of finding an optimal DIA for the inverted file to minimize the average query processing time when the probability distribution of query terms is given. The DIA problem, that is known to be NP-complete via a reduction to the rectilinear *traveling salesman problem* (TSP), is a generalization of the problems solved by Olken & Rotem (1986), Shieh et al. (2003), and Gelbukh et al. (2003). Their research results showed that this kind of optimization problem can be effectively solved by the well-known TSP heuristic algorithms. The *greedy nearest*

neighbor (Greedy-NN) algorithm performs the best on average, but its high complexity discourages its use in modern large-scale IRSs.

In this chapter, we propose a fast heuristic, called *partition-based document identifier assignment* (PBDIA) algorithm, to find a good DIA that can make the document identifiers in the posting lists for frequently used query terms more clustered. This can greatly improve the compression efficiency of the posting lists for frequently used query terms. Where the probability distribution of query terms is skewed, as is the typical case in a real-world IRS, the experimental results show that the PBDIA algorithm can yield a competitive performance versus the Greedy-NN for the DIA problem. The experimental results also show that the DIA problem has significant advantages for both long queries.

The remainder of this chapter is organized as follows. Section 4.1 describes the inverted index and explains why a DIA can affect the storage space required and change query performance. Section 4.2 derives a cost model for the DIA problem, and presents how to use the well-known TSP heuristic algorithms to solve this optimization problem. In Section 4.3, we propose a fast PBDIA algorithm. We show the performance evaluation in Section 4.4. Finally, Section 4.5 presents our summary.

4.1 General Framework

The data structures of an inverted index are depicted in Figure 4.1. An inverted index consists of an index file and an inverted file. An index file is a set of records, each containing a keyword term t and a pointer to the posting list for term t. An inverted file contains, for each distinct term t in the collection, a posting list of the form

$$PL_t = <id_1, id_2, \ldots, id_{f_t}>,$$

where id_i is the identifier of the document that contains t, and frequency f_t is the number of documents in which t appears. The document identifiers are within the range 1...N, where N is the number of documents in the indexed collection. In a large document collection, posting lists are usually compressed, and decompression of posting lists is hence required during query processing.

Zipf (1949) observed that the set of frequently used terms is small. According to Zipf's law, 95% of words in all documents fall in a vocabulary with no more than 8000 distinct terms. This suggests that it is advisable to store the index records of frequently used terms in RAM to greatly reduce index search time. Hence, the significant portion of query processing time is to read and decompress the compressed posting list for each query term. This chapter restricts attention to inverted file side only and investigates the DIA problem to improve the efficiency of an inverted file and the overall *information retrieval* (IR) performance.

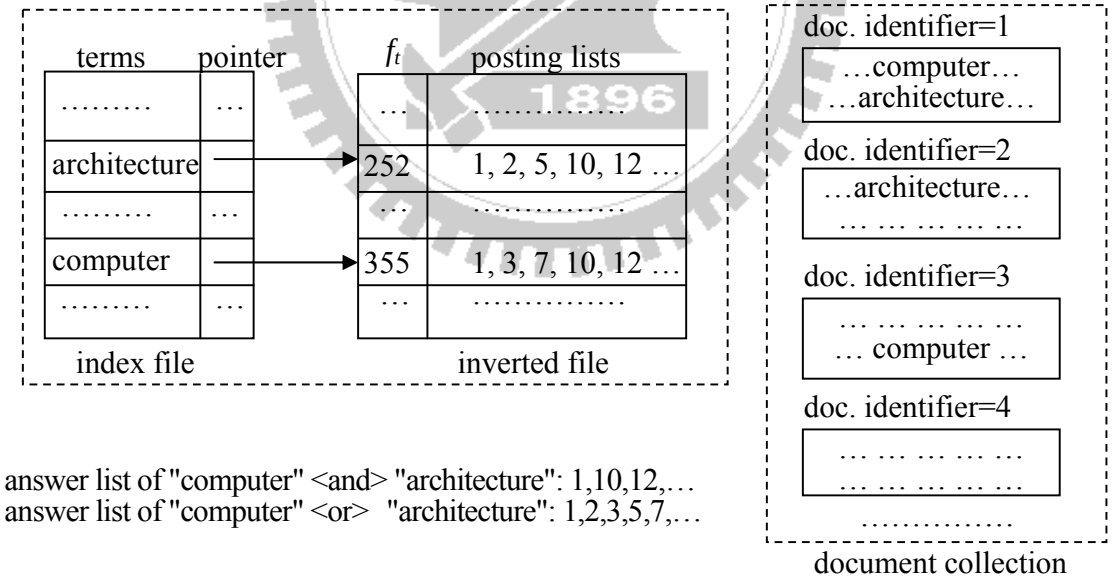
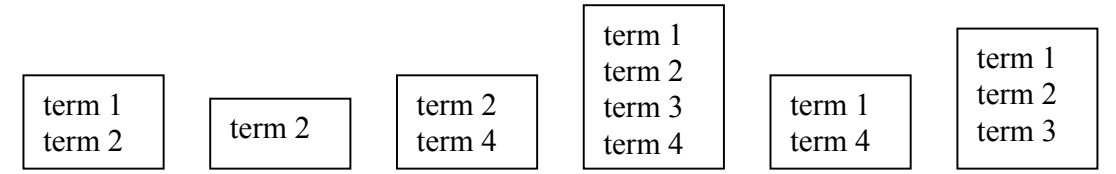


Figure 4.1 Inverted index and document collection

Compression of an inverted file is the most popular technique used to increase query throughput (Zobel & Moffat, 1995; Williams & Zobel, 1999; Trotman, 2003). This is because the

total time of transferring a compressed posting list and subsequently decompressing it is potentially much less than that of transferring an uncompressed posting list. To achieve good compression, the document identifiers in a posting list should be sorted in increasing order and compressed using the *d*-gap compression approach (Moffat & Zobel, 1992; Witten et al., 1999) or the interpolative coding approach (Moffat & Stuiver, 2000). Both approaches can yield superior compression if the document identifiers in the posting lists are clustered.

Consider a document collection of 6 documents shown in Figure 4.2(a). Each document contains one or more terms. The document d_1 contains term 1 and term 2, document d_2 contains term 2, etc. In Figures 2.2(b) and 2.2(c), the notation $d_i \rightarrow j$ in DIAs I and II denotes that the document identifier j is assigned to the document d_i. According to the documents in Figure 4.2(a) and the DIAs I and II, the obtained posting lists and *d*-gap lists are shown in Figures 4.2(b) and 4.2(c). For DIA I, the *d*-gap values have nine 1s, two 2s, two 3s and one 4; whereas for DIA II, the *d*-gap values have eleven 1s, one 2 and two 3s. With γ coding in Table 4.1, the compressed inverted file requires 26 bits for DIA I, whereas it requires 20 bits for DIA II. If every term is queried with equal probability, the query processing costs for DIA II will be much lower than that of DIA I. This is because DIA II can result in better compression for the given coding method without increasing the complexity of decoding process, hence improve query throughput by reducing both the retrieval and decompression times of posting lists. This example shows that different DIAs can result in different compression results and different query throughputs for a given coding method. In next section, we will introduce a query cost function for the DIA problem, and then derive a method to find a good DIA to shorten average query processing time when the probability distribution of query terms is given.



(a) Example documents

DIA I: { d_1→1, d_2→2, d_3→3, d_4→4, d_5→5, d_6→6}

Posting list of term 1: <1, 4, 5, 6>
Posting list of term 2: <1, 2, 3, 4, 6>
Posting list of term 3: <4, 6>
Posting list of term 4: <3, 4, 5>

d-gap list of term 1: <1, 3, 1, 1>
d-gap list of term 2: <1, 1, 1, 1, 2>
d-gap list of term 3: <4, 2>
d-gap list of term 4: <3, 1, 1>

Total bits required to encode *d*-gaps with γ code = 26 bits

(b) DIA I result

DIA II: { d_1→3, d_2→5, d_3→4, d_4→1, d_5→6, d_6→2}

Posting list of term 1: <1, 2, 3, 6>
Posting list of term 2: <1, 2, 3, 4, 5>
Posting list of term 3: <1, 2>
Posting list of term 4: <1, 4, 6>

d-gap list of term 1: <1, 1, 1, 3>
d-gap list of term 2: <1, 1, 1, 1, 1>
d-gap list of term 3: <1, 1>
d-gap list of term 4: <1, 3, 2>

Total bits required to encode *d*-gaps with γ code = 20 bits

(c) DIA II result

Figure 4.2 An example to show different DIAs result in different compression results

d-gap value *x*	γ code
1	0
2	10 0
3	10 1
4	110 00

Table 4.1 Some example codes for γ coding

4.2 Document Identifier Assignment Problem and Its Algorithm

The DIA problem is the problem of assigning document identifiers to a set of documents in an inverted file-based IRS in order to minimize the average query processing time when the probability distribution of query terms is given. In this section, we first formalize the problem, and then show how to use the well-known *greedy nearest neighbor* (Greedy-NN) algorithm to solve this problem.

4.2.1 Problem mathematical formulation

Let $D=\{d_1, d_2, \ldots, d_N\}$ be a collection of N documents to be indexed, and $\pi :\{ d_1, d_2, \ldots, d_N \}\rightarrow\{1, 2, \ldots, N\}$ be a DIA that assigns a unique identifier within the range $1\ldots N$ to each document in D. Let f_t be the total number of documents in which term t appears and $d_{t(1)}, d_{t(2)}, \ldots, d_{t(f_t)}$ be documents containing term t, then the posting list of the term t can be represented as $PL_t=<\pi(d_{t(1)}), \pi(d_{t(2)}),\ldots, \pi(d_{t(f_t)})>$. Without loss of generality, we assume that $\pi(d_{t(1)})<\pi(d_{t(2)})<\ldots<\pi(d_{t(f_t)})$. Assume a coding method C which requires $C(x)$ bits to encode a d-gap x. The size of a posting list PL_t for term t can then be expressed as

$$\sum_{i=1}^{ft} C(\pi(d_{t(i)}) - \pi(d_{t(i-1)})) \tag{4.1}$$

where we let $d_{t(0)}=0$ and $\pi(d_{t(0)})=0$ to simplify the expression of Eq.(4.1). Assume that the probability of a term t appearing in a query is p_t. Let X_t be a random Boolean variable representing whether term t appears in a query: $X_t=1$ if term t appears in a query and $X_t=0$ otherwise. The query processing time $Time_{QP}$ of posting list processing includes (1) retrieval time $Time_R$ of posting list PL_t for each query term t, (2) decompression time $Time_D$ of posting list PL_t for each query term t, and (3) document identifier comparison time $Time_{Comp}$. Since the document identifier comparison

time is relatively small (about 10% of query processing time) and does not change with different DIAs, the query processing time in this chapter is defined only as

$$Time_{QP} = \sum_t X_t \times (Time_R(PL_t) + Time_D(PL_t)) \tag{4.2}$$

The average query processing time $AvgTime_{QP}$ is the expected value of $Time_{QP}$. That is,

$$AvgTime_{QP} = \sum_t p_t \times (Time_R(PL_t) + Time_D(PL_t)) \tag{4.3}$$

Since the disk transfer rate is near constant and the decoding processes of most coding methods used in *d*-gap compression approach are on a bit-by-bit basis, the retrieval and decompression times of a posting list PL_t for the term t appearing in a query grows with the size of the posting list PL_t. So

$$Time_R(PL_t) + Time_D(PL_t) = \text{constant} \times \sum_{i=1}^{f_t} C(\pi(d_{t(i)}) - \pi(d_{t(i-1)})) \tag{4.4}$$

Substituting Eq.(4.4) into Eq.(4.3), we obtain

$$AvgTime_{QP} = \text{constant} \times \sum_t p_t \times \sum_{i=1}^{f_t} C(\pi(d_{t(i)}) - \pi(d_{t(i-1)})) \tag{4.5}$$

We thus define the objective function $Cost(\pi)$ to reflect the average query processing time $AvgTime_{QP}$:

$$Cost(\pi) = \sum_t p_t \times \sum_{i=1}^{f_t} C(\pi(d_{t(i)}) - \pi(d_{t(i-1)})) \tag{4.6}$$

The objective of this research is to find a *DIA* $\pi : D \rightarrow \{1,2,3\ldots,N\}$ such that $Cost(\pi)$ is minimal. This optimization problem is called the DIA problem, and it is reduced to the *simple DIA* (SDIA) problem if the value of p_t for each term t is set to 1. The SDIA problem is the problem of finding a DIA to minimize the size of inverted file, and it is known to be NP-complete via a reduction to the rectilinear traveling salesman problem (Olken & Rotem 1986). Since the DIA problem is a generalization of the SDIA problem, the DIA problem is also a NP-complete problem.

4.2.2 Solving DIA problem via the well-known Greedy-NN algorithm

The research works of Shieh et al. (2003) and Gelbukh et al. (2003) indicated that finding the near-optimal solution for the SDIA problem can be recast as the *traveling salesman problem* (TSP), and also showed that heuristic algorithms for the TSP can be applied to the SDIA problem to find a near-optimal DIA. Compared with those well-known TSP heuristic algorithms, such as insertion heuristic algorithm and spanning tree based algorithm, Shieh et al. (2003) showed that the Greedy-NN algorithm performs better for the SDIA problem on average. In this section, we first show how to solve the SDIA problem using the Greedy-NN algorithm. Then, we show how to transform the DIA problem into the SDIA problem, and explain why the Greedy-NN algorithm can provide better performance than the other TSP heuristic algorithms for the DIA problem.

Solving SDIA problem via Greedy-NN algorithm

Shieh et al. (2003) showed that the SDIA problem can be solved by using TSP heuristic algorithms. Given a collection of N documents, a *document similarity graph* (DSG) can be constructed. In a DSG, each vertex represents a document, and the weight on an edge between two vertices represents the similarity of these two corresponding documents. The similarity $Sim(d_i, d_j)$ between two documents d_i and d_j is defined as:

$$Sim(d_i, d_j) = \sum_{t \in (T(d_i) \cap T(d_j))} 1 \tag{4.7}$$

where $T(d_i)$ and $T(d_j)$ denote the set of terms appearing in d_i and d_j, respectively, and $\cap$ denotes the intersection operator. Hence, the similarity between two documents is the number of common terms appearing in both documents. The DSG for the example documents in Figure 4.2(a) is shown in Figure 4.3. A TSP heuristic algorithm can then be used to find a path of the DSG visiting each vertex exactly once with maximal sum of similarities. If we follow the visiting order of vertices on the path to assign document identifiers, the sum of d-gap values for an inverted file can be

decreased, and the size of inverted file compressed via the *d*-gap compression approach can be reduced. Shieh et al. (2003) showed that the Greedy-NN algorithm (Figure 4.4) can provide excellent performance for the SDIA problem.

We now show how to obtain a DIA for the example documents in Figure 4.2. In Step 1, we construct the DSG (Figure 4.3) for the given documents, where $V=\{d_1, d_2, d_3, d_4, d_5, d_6\}$. In Step 2, we pick d_4 as v_1 since the sum of similarity values associated with its adjacent edges is maximal (=10). In Step 3, we have $V'=\{d_1, d_2, d_3, d_5, d_6\}$. In Step 4, we pick d_6 as v_2 since d_6 is the vertex v in V' such that the edge (v,v_1) has the maximal similarity value. In Step 5, we have $V'=\{d_1, d_2, d_3, d_5\}$. Repeat Steps 4 and 5 as needed, we can then sequentially pick d_1 as v_3, d_3 as v_4, d_2 as v_5, and d_5 as v_6. Hence, we have a TSP path: $\{d_4, d_6, d_1, d_3, d_2, d_5\}$, and a DIA π = $\{d_1 \rightarrow 3, d_2 \rightarrow 5, d_3 \rightarrow 4, d_4 \rightarrow 1, d_5 \rightarrow 6, d_6 \rightarrow 2\}$.

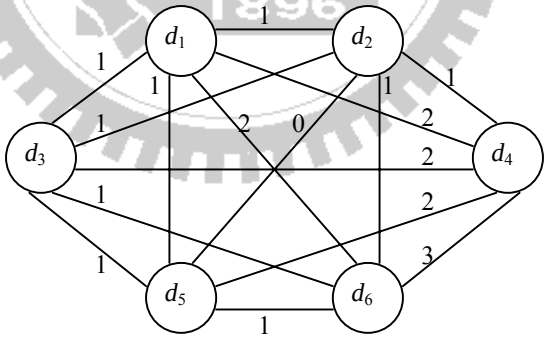


Figure 4.3 The DSG for the example documents in Figure 4.2(a).

Algorithm Greedy_nearest_neighbor

Input:

$D=\{d_1, d_2, \ldots, d_N\}$: a collection of N documents to be indexed.

Output:

A TSP path: the visiting order of vertices is $\{v_1, v_2, ..., v_N\}$

Method:

1. Construct the DSG(V, E), where V is a set of vertices (in which each vertex represents a document) and E is a set of edges (in which each edge has a similarity value associated with it);
2. Pick a vertex $v \in V$ as v_1 such that the sum of similarity values associated with the adjacent edges of v is maximal;
3. $V' := V - \{v_1\}$; $i := 1$;
4. Find v in V' such that the similarity value of the edge (v,v_i) is maximal: if more than one such vertex exist, select one randomly;
5. $i := i + 1$; $v_i := v$; $V' := V' - \{v_i\}$;
6. If $i<N$ then goto 3;
7. Output a TSP path with its visiting order of vertices being $\{v_1, v_2, ..., v_N\}$

Figure 4.4 The Greedy-NN algorithm for the SDIA problem.

Transforming DIA problem into SDIA problem

We use a matrix A to represent the input document collection, in which a row corresponds to a term and a column corresponds to a document. The entry $A_{i,j}$ is a 1 if term i appears in document d_j, and 0 otherwise. The SDIA problem is to determine whether there exists a permutation of the columns of A that results in a matrix B such that

$$\sum_{i=1}^{n}\left(\sum_{j=2}^{f_i} C\big(z(i, j) - z(i, j-1)\big) + C\big(z(i,1)\big)\right) \le k \qquad (4.8)$$

where C is a coding method which requires $C(x)$ bits to encode a d-gap x, n is the number of terms, f_i is the total number of documents in which term i appears, $z(i,j)$ is a function that returns the column index of the j^{th} nonzero entry at row i, and k is a given integer used to determine whether there exists a permutation of columns of A such that the total encoded size of an inverted file is less than k. The DIA problem is to determine whether there exists a permutation of the columns of A that results in a matrix B such that

$$\sum_{i=1}^{n} p_i \times \left(\sum_{j=2}^{f_i} C\big(z(i,j) - z(i,j-1)\big) + C\big(z(i,1)\big) \right) \leq k' \tag{4.9}$$

where p_i is the probability of a term i appearing in a query and k' is a given integer used to determine whether there exists a permutation of columns of A such that the mean encoded size needed to read and decompress a posting list during query processing is less than k'.

To show how to transform the DIA problem into the SDIA problem, we use the document collection in Figure 4.2(a) as an example instance of the DIA problem, and assume that the probabilities of terms being queried are p_1=0.2, p_2=0.3, p_3=0.1, and p_4=0.4. Figure 4.5(a) shows the matrix A of Figure 4.2(a). Then we construct a new matrix A' for the SDIA problem by duplicating each row of matrix A in a certain number of times based on the given probabilities of terms appearing in a query, as shown in Figure 4.5(b). In matrix A', the row of matrix A corresponding to term i is duplicated m_i times, where m_i=*rows*(A')×p_i and *rows*(A') denotes the number of rows of matrix A'. The *rows*(A') can be any positive integer such that m_i=*rows*(A') ×p_i is an integer for every i. In this example, we let *rows*(A') be 10. One can easily show that the optimal solution of matrix A' for the SDIA problem is also the optimal solution of matrix A for the DIA problem when the probabilities p_1=0.2, p_2=0.3, p_3=0.1, and p_4=0.4 are given.

Using the same approach, it is obvious that one can transform any instance A of the DIA problem into an instance A' of the SDIA problem such that the optimal solution of matrix A' for the SDIA problem is also the optimal solution of matrix A for the DIA problem when the probabilities p_i for $1 \leq i \leq n$ are given, where n denotes the number of distinct terms. Since the research work of Shieh et al. (2003) showed that the Greedy-NN algorithm performs the best for the SDIA problem on average, one can show that the Greedy-NN algorithm can provide better performance than the other TSP heuristic algorithms for the DIA problem. Therefore, the DIA problem can be solved

using the Greedy-NN algorithm described in Figure 4.4, if the similarity $Sim(d_i, d_j)$ between two documents d_i and d_j in a DSG is redefined as:

$$Sim(d_i, d_j) = \sum_{t \in (T(d_i) \cap T(d_j))} p_t \tag{4.10}$$

where the probability of a term t appearing in a query is known to be p_t.

Matrix A:

probability		d_1	d_2	d_3	d_4	d_5	d_6
p_1=0.2	term 1	1	0	0	1	1	1
p_2=0.3	term 2	1	1	1	1	0	1
p_3=0.1	term 3	0	0	0	1	0	1
p_4=0.4	term 4	0	0	1	1	1	0

(a) An example instance for the DIA problem: Matrix A corresponds to the document collection in Figure 4.2(a), and the probabilities of terms appearing in a query are p_1=0.2, p_2=0.3, p_3=0.1, and p_4=0.4.

Matrix A':

	d_1	d_2	d_3	d_4	d_5	d_6
Row_{term1} of matrix A is duplicated $m_1=rows(A') \times p_1$=2 times	1	0	0	1	1	1
	1	0	0	1	1	1
Row_{term2} of matrix A is duplicated $m_2=rows(A') \times p_2$=3 times	1	1	1	1	0	1
	1	1	1	1	0	1
	1	1	1	1	0	1
Row_{term3} the matrix A is duplicated $m_3=rows(A') \times p_3$=1 time	0	0	0	1	0	1
Row_{term4} of matrix A is duplicated $m_4=rows(A') \times p_4$=4 times	0	0	1	1	1	0
	0	0	1	1	1	0
	0	0	1	1	1	0
	0	0	1	1	1	0

(b) Matrix A' is the corresponding instance of Figure 4.5(a) for the SDIA problem. In matrix A', Row_{termi} of matrix A is duplicated m_i times, where $m_i=rows(A') \times p_i$ and $rows(A')$ denotes the number of rows of matrix A'.

Figure 4.5 An example to illustrate how to transform an instance of the DIA problem into an instance of the SDIA problem

Although the Greedy-NN algorithm is very simple to implement, it is not very applicable to large-scale IRSs due to its high complexity. Given a collection of N documents and n distinct terms, the number of comparisons for calculating $Sim(d_i,d_j)$ given fixed i and j is $O(n)$, hence the total

number of comparisons to construct a DSG for the Greedy-NN algorithm is $O(N^2 \times n)$. An algorithm with lower complexity yet still generates satisfactory results should be developed.

4.3 Partition-based Document Identifier Assignment Algorithm

Since the DIA problem is an NP-complete problem, the effort in search for an effective low-complexity method is needed. Although the Greedy-NN algorithm can be used to solve the DIA problem, its complexity is too high. In this section, we first present an optimal DIA algorithm for a single query term, and then propose an efficient *partition-based document identifier assignment* (PBDIA) algorithm for the DIA problem.

4.3.1 Generating an optimal DIA for a single query term

Consider a posting list PL_t for term t with f_t document identifiers in a collection of N documents. Using the d-gap technique, we can obtain f_t d-gap values: $d\text{-}gap_1$, $d\text{-}gap_2$,…, $d\text{-}gap_{ft}$. Assume a coding method C which requires $C(x)$ bits to encode a d-gap x. We want to know which d-gap probability distribution can minimize the size of posting list PL_t after compression using method C. That is, we want to know which d-gap probability distribution can minimize

$$\sum_{i=1}^{f_t} C(d\text{-}gap_i) \tag{4.11}$$

subject to

$$f_t \le \sum_{i=1}^{f_t} d\text{-}gap_i \le k \text{ and} \tag{4.12}$$

$$1 \le d\text{-}gap_i \le k \text{ for all } i,\ 1 \le i \le k \tag{4.13}$$

where k is the largest document identifier in the posting list PL_t. It is known that $C(x)$ is approximately proportional to $\log_2(x)$ for many popular coding methods, such as γ coding, skewed

Golomb coding, and batched LLRUN coding. For these coding methods, we can use dynamic programming technique (Bellman and Dreyfus, 1962) and find that minimizing Eq.(4.11) should meet two requirements: (1) maximize the number of *d*-gap values of 1; and (2) minimize the largest document identifier, i.e., *k*, in the posting list PL_t. If a DIA for term *t* can satisfy the above two requirements, the best compression and the fastest query speed for the posting list PL_t can be achieved.

According to the above observation, we propose the *simple partition-based document identifier assignment* (SPBDIA) algorithm to generate optimal DIAs for a given query term *t*. The SPBDIA algorithm consists of a partitioning procedure, an ordering procedure, and a document identifier assignment procedure. The partitioning procedure divides the given documents into two partitions in terms of query term *t*: one partition P(*t*) consists of documents containing query term *t* ; the other partition P(*t*') is made up of the documents without *t*. Then, the ordering procedure sets the order of partitions as P(*t*) followed by P(*t*'). Finally, the document identifier assignment procedure generates an appropriate DIA for the ordered partitions according to query term *t*: the documents in partition P(*t*) are assigned smaller consecutive document identifiers, while the documents in partition P(*t*') assigned larger consecutive document identifiers. The SPBDIA algorithm is illustrated in the following Example.

Example. There is a collection of 500 documents, among which 300 documents contain query term *t*. After partitioning, P(*t*) has 300 documents and P(*t*') has 200 documents. Then, the ordering procedure sets the order of partitions P(*t*) followed by P(*t*'). Finally, the document identifier assignment procedure assigns the document identifiers 1~300 to the 300 documents in partition P(*t*) and assigns the document identifiers 301~500 to the 200 documents in partition P(*t*'). ■

Documents in a partition can be arbitrarily assigned identifiers within the given range, hence the

number of possible DIAs for the above Example is 300!×200!. Each of the 300!×200! DIAs satisfies the two requirements for minimizing Eq.(4.11), and hence gives both the best posting list compression and fastest query speed for query term *t*. The SPBDIA algorithm is simple, and its complexity is $O(N)$.

4.3.2 Efficient PBDIA algorithm for DIA problem

In a real-world IRS, a few frequently used query terms constitute a large portion of all term occurrences in queries (Jansen et al. 1998). This fact indicates that a DIA algorithm that allows those frequently used query terms to have better posting list compression can result in reduced average query processing time. Based on the SPBDIA algorithm, an efficient *partition-based document identifier assignment* (PBDIA) algorithm for the DIA problem can be developed.

Like the SPBDIA algorithm, the PBDIA algorithm also partitions the document set, orders these partitions, and then assigns document identifiers. The flowchart of the PBDIA algorithm is shown in Figure 4.6. The partitioning and ordering procedures of the PBDIA algorithm iterate *n* times given that there are *n* query terms. Then, the document identifier assignment procedure is performed as the last step of the PBDIA algorithm. Terms that are queried more frequently should take higher priority in document partitioning and partition ordering. Let the most frequently queried term be assigned rank 1, the second most frequently queried term rank 2, and so on. We use $t_{\text{rank } i}$ to represent the i^{th} ranked query term. The partitioning and ordering procedures of the PBDIA algorithm should proceed by considering $t_{\text{rank } 1}$ first, then $t_{\text{rank } 2}$, and so on.

Both the PBDIA partitioning and ordering procedures are invoked once per iteration. The PBDIA partitioning procedure first divides each partition generated in the previous iteration into two partitions using the SPBDIA partitioning procedure. The PBDIA ordering procedure then assigns each newly generated partition a partition order. Each partition *P* in the PBDIA

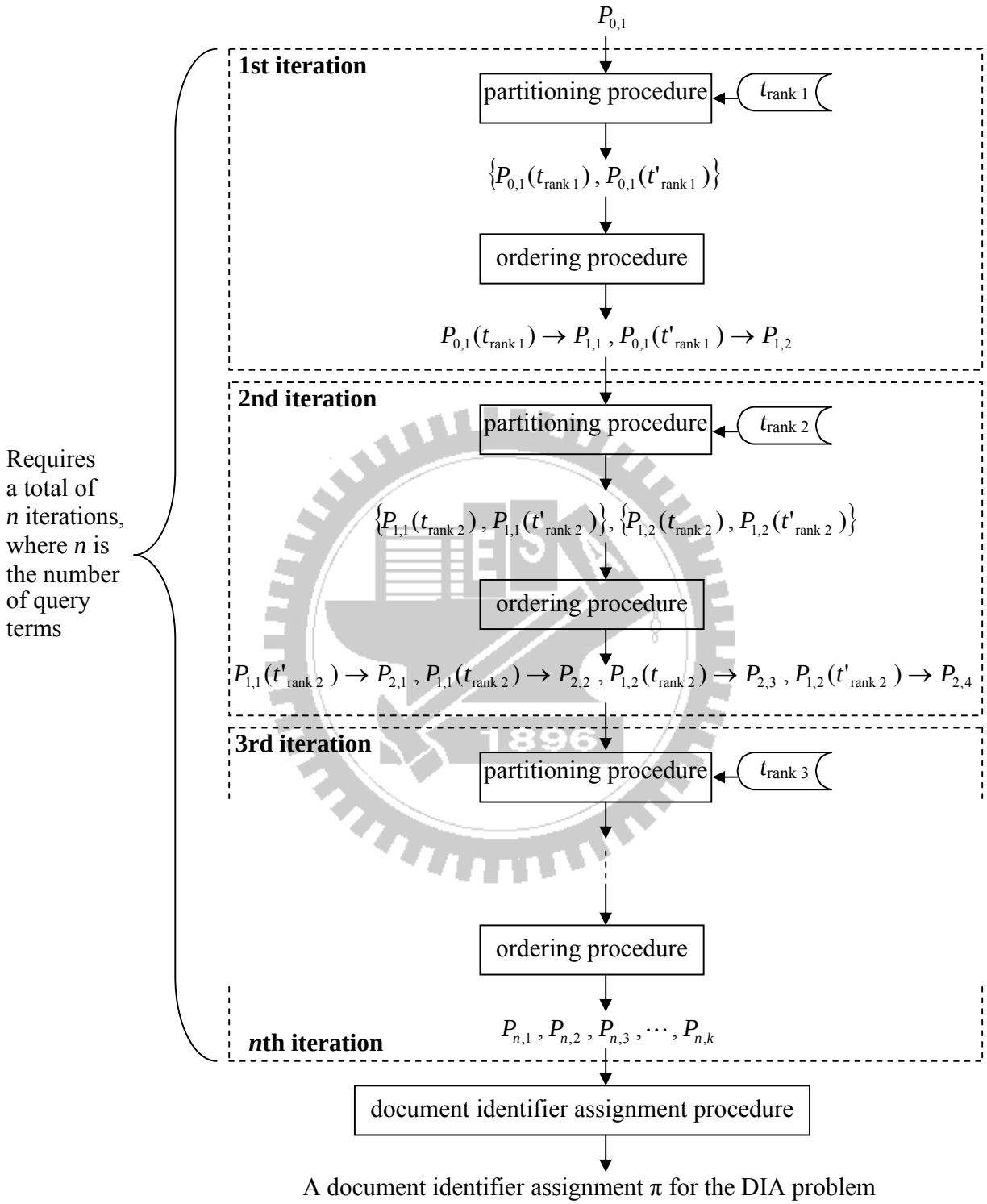


Figure 4.6 The flowchart for the PBDIA algorithm

algorithm hence can be uniquely identified by an iteration number i and a partition order j, and we use the notation $P_{i,j}$ to represent the j^{th} ordered partition of the i^{th} iteration. For example, the notation $P_{2,3}$ represents the 3rd ordered partition of the 2nd iteration. Initially, we use the notation $P_{0,1}$ to represent the partition that contains all documents in an input document collection. In the following, we describe in detail the partitioning, ordering, and document identifier assignment procedures of the PBDIA algorithm.

PBDIA partition procedure

Let $P_{i\text{-}1,1}$, $P_{i\text{-}1,2}$, …, and $P_{i\text{-}1,k}$ be nonempty partitions generated in iteration i-1. The PBDIA partitioning procedure invoked in the i^{th} iteration divides each partition $P_{i\text{-}1,j}$ into a partition pair $\{P_{i\text{-}1,j}(t_{\text{rank } i}), P_{i\text{-}1,j}(t'_{\text{rank } i})\}$ for j=1,2,…,k, where the partition $P_{i\text{-}1,j}(t_{\text{rank } i})$ consists of the documents in $P_{i\text{-}1,j}$ containing the query term $t_{\text{rank } i}$, and $P_{i\text{-}1,j}(t'_{\text{rank } i})$ consists of the documents in $P_{i\text{-}1,j}$ without the query term $t_{\text{rank } i}$. Since $P_{i\text{-}1,j}$ is nonempty, at least one of the two partitions $P_{i\text{-}1,j}(t_{\text{rank } i})$ and $P_{i\text{-}1,j}(t'_{\text{rank } i})$ is nonempty for j=1,2,…,k.

PBDIA ordering procedure

Let $\{P_{i\text{-}1,1}(t_{\text{rank } i}), P_{i\text{-}1,1}(t'_{\text{rank } i})\}$, $\{P_{i\text{-}1,2}(t_{\text{rank } i}), P_{i\text{-}1,2}(t'_{\text{rank } i})\}$, …, and $\{P_{i\text{-}1,k}(t_{\text{rank } i}), P_{i\text{-}1,k}(t'_{\text{rank } i})\}$ be the partition pairs generated by PBDIA partitioning procedure in iteration i. Let $|P_i|$ denote the number of nonempty partitions of the above partitions. The PBDIA ordering procedure invoked in the i^{th} iteration assigns a unique partition order, from $|P_i|$ to 1 and in descending order, to each nonempty partition, starting from $\{P_{i\text{-}1,k}(t_{\text{rank } i}), P_{i\text{-}1,k}(t'_{\text{rank } i})\}$, then $\{P_{i\text{-}1,k\text{-}1}(t_{\text{rank } i}), P_{i\text{-}1,k\text{-}1}(t'_{\text{rank } i})\}$, and so on.

Now let us consider the ordering of partition pair $\{P_{i\text{-}1,k}(t_{\text{rank } i}), P_{i\text{-}1,k}(t'_{\text{rank } i})\}$. Three cases exist.

Case 1: Both $P_{i\text{-}1,k}(t_{\text{rank } i})$ and $P_{i\text{-}1,k}(t'_{\text{rank } i})$ are nonempty

The ordering procedure assigns $|P_i|$ to $P_{i\text{-}1,k}(t'_{\text{rank } i})$, and $|P_i|$-1 to $P_{i\text{-}1,k}(t_{\text{rank } i})$. $P_{i\text{-}1,k}(t'_{\text{rank } i})$ is

hereafter denoted as $P_{i,|P_i|}$, and $P_{i\text{-}1,k}(t_{\text{rank }i})$ as $P_{i,|P_i|-1}$.

<u>*Case 2*</u>: $P_{i\text{-}1,k}(t_{\text{rank }i})$ is empty, and $P_{i\text{-}1,k}(t'_{\text{rank }i})$ is nonempty

The ordering procedure assigns $|P_i|$ to $P_{i\text{-}1,k}(t'_{\text{rank }i})$, and ignores $P_{i\text{-}1,k}(t_{\text{rank }i})$. $P_{i\text{-}1,k}(t'_{\text{rank }i})$ is hereafter denoted as $P_{i,|P_i|}$.

<u>*Case 3*</u>: $P_{i\text{-}1,k}(t_{\text{rank }i})$ is nonempty, and $P_{i\text{-}1,k}(t'_{\text{rank }i})$ is empty

The ordering procedure assigns $|P_i|$ to $P_{i\text{-}1,k}(t_{\text{rank }i})$, and ignores $P_{i\text{-}1,k}(t'_{\text{rank }i})$. $P_{i\text{-}1,k}(t_{\text{rank }i})$ is hereafter denoted as $P_{i,|P_i|}$.

Next we consider the ordering of partition pairs $\{P_{i\text{-}1,j}(t_{\text{rank }i}), P_{i\text{-}1,j}(t'_{\text{rank }i})\}$, where j=1,2,…,k-1. Let the next largest partition order to be assigned be s. Since PBDIA ordering procedure orders $\{P_{i\text{-}1,j+1}(t_{\text{rank }i}), P_{i\text{-}1,j+1}(t'_{\text{rank }i})\}$ before $\{P_{i\text{-}1,j}(t_{\text{rank }i}), P_{i\text{-}1,j}(t'_{\text{rank }i})\}$, $P_{i,s+1}$ is hence used to denote either $P_{i\text{-}1,j+1}(t_{\text{rank }i})$ or $P_{i\text{-}1,j+1}(t'_{\text{rank }i})$. Again, three cases exist for $\{P_{i\text{-}1,j}(t_{\text{rank }i}), P_{i\text{-}1,j}(t'_{\text{rank }i})\}$:

<u>*Case 1*</u>: Both $P_{i\text{-}1,j}(t_{\text{rank }i})$ and $P_{i\text{-}1,j}(t'_{\text{rank }i})$ are nonempty

There exist two subcases.

<u>*SubCase1.a*</u>: $P_{i,s+1}$ is used to denote $P_{i\text{-}1,j+1}(t_{\text{rank }i})$

The ordering procedure assigns s to $P_{i\text{-}1,j}(t_{\text{rank }i})$, and s-1 to $P_{i\text{-}1,j}(t'_{\text{rank }i})$. $P_{i\text{-}1,j}(t_{\text{rank }i})$ is hereafter denoted as $P_{i,s}$, and $P_{i\text{-}1,j}(t'_{\text{rank }i})$ as $P_{i,s\text{-}1}$.

<u>*SubCase1.b*</u>: $P_{i,s+1}$ is used to denote $P_{i\text{-}1,j+1}(t'_{\text{rank }i})$

The ordering procedure assigns s to $P_{i\text{-}1,j}(t'_{\text{rank }i})$, and s-1 to $P_{i\text{-}1,j}(t_{\text{rank }i})$. $P_{i\text{-}1,j}(t'_{\text{rank }i})$ is hereafter denoted as $P_{i,s}$, and $P_{i\text{-}1,j}(t_{\text{rank }i})$ as $P_{i,s\text{-}1}$.

<u>*Case 2*</u>: $P_{i\text{-}1,j}(t_{\text{rank }i})$ is empty, and $P_{i\text{-}1,j}(t'_{\text{rank }i})$ is nonempty

The ordering procedure assigns s to $P_{i\text{-}1,j}(t'_{\text{rank }i})$, and ignores $P_{i\text{-}1,j}(t_{\text{rank }i})$. $P_{i\text{-}1,j}(t'_{\text{rank }i})$ is hereafter denoted as $P_{i,s}$.

<u>*Case 3*</u>: $P_{i\text{-}1,j}(t_{\text{rank }i})$ is nonempty, and $P_{i\text{-}1,j}(t'_{\text{rank }i})$ is empty

The ordering procedure assigns s to $P_{i\text{-}1,j}(t_{\text{rank } i})$, and ignores $P_{i\text{-}1,j}(t'_{\text{rank } i})$. $P_{i\text{-}1,j}(t_{\text{rank } i})$ is hereafter denoted as $P_{i,s}$.

PBDIA document identifier assignment procedure

The document identifier assignment procedure, the last step of PBDIA algorithm, is straightforward. Let $P_{n,1}$, $P_{n,2}$, …, and $P_{n,k}$ be the generated ordered partitions of the iteration n. This procedure assigns consecutive document identifiers to documents in the same partition, and consecutive identifier groups to consecutive ordered partitions. The first (smallest) document identifier is assigned to a document in the first ordered partition ($P_{n,1}$). And the ordering of documents in a partition is irrelevant and can be arbitrary.

To obtain a good DIA, the partitions must be properly ordered. We explain why the PBDIA ordering procedure is proper: Note that the PBDIA ordering procedure always assigns consecutive partition orders to two nonempty partitions of a partition pair. This makes documents in the same partition in iteration i remain in the same or neighboring partitions in iteration i+1. According to the PBDIA document identifier assignment procedure, documents in the same partition in iteration i will eventually be assigned consecutive or at least adjacent document identifiers. That is, once the order of partitions is generated at the end of iteration i, the compression performance for the posting list of $t_{\text{rank } i}$ is determined. Hence, the posting list of $t_{\text{rank } 1}$ has the best compression, then that of t_{rank2}, and so on. This is because the PBDIA algorithm considers the $t_{\text{rank } 1}$ first, then $t_{\text{rank } 2}$, and so on, in its iterations.

Algorithm Partition_based_document_identifier_assignment

Input:

$D=\{d_1, d_2, \ldots, d_N\}$: a collection of N documents to be indexed.

$T=\{t_1, t_2, \ldots, t_n\}$: a set of n distinct terms appearing in D.

$Prob=\{p_1, p_2, \ldots, p_n\}$: p_i denotes the probability of the term $t_i \in T$ appearing in a query.

Output:

A document identifier assignment $\pi : \{ d_1, d_2, \ldots, d_N \} \rightarrow \{1, 2, \ldots, N\}$ for the DIA.

Method:

1. Create an empty doubly linked list *PartList*; // to store partition
2. Create an empty doubly linked list *TempList*; //to store partition pairs
3. Assign all documents in D to a new partition P, and add P to the *PartList*;
4. Sort the terms in T in descending order according to their probabilities. Let $t_{\text{rank } 1}$, $t_{\text{rank } 2}$, …, $t_{\text{rank } n}$ represent the sorted list.
5. **for** i:=1 to n **do**
   - 5.1 **while** *PartList* is not empty **do** **/*partitioning procedure*/**
     - 5.1.1 Get a partition P from the head of *PartList*, and then remove P from *PartList*;
     - 5.1.2 // At least one of the partitions $P(t_{\text{rank } i})$ and $P(t'_{\text{rank } i})$ should be nonempty
       Let $P(t_{\text{rank } i})$ be the partition containing the documents that are included in P and do contain the term $t_{\text{rank } i}$; let $P(t'_{\text{rank } i})$ be the partition containing the documents that are included in P and do not contain the term $t_{\text{rank } i}$;
     - 5.1.3 Add the partition pair $\{P(t_{\text{rank } i}),P(t'_{\text{rank } i})\}$ to the tail of *TempList*;
   - 5.2 **while** *TempList* is not empty **do** **/*ordering procedure*/**
     - 5.2.1 Get a partition pair $\{P(t_{\text{rank } i}),P(t'_{\text{rank } i})\}$ from the tail of *TempList*, and then remove $\{P(t_{\text{rank } i}),P(t'_{\text{rank } i})\}$ from *TempList*;
     - 5.2.2 **if** $P(t_{\text{rank } i})$ is empty **then** add $P(t'_{\text{rank } i})$ to the front of *PartList* and go to step 5.2;
     - 5.2.3 **if** $P(t'_{\text{rank } i})$ is empty **then** add $P(t_{\text{rank } i})$ to the front of *PartList* and go to step 5.2;
     - 5.2.4 **if** *PartList* is empty **then**
       Add $P(t'_{\text{rank } i})$ to the *PartList*; add $P(t_{\text{rank } i})$ to the front of *PartList*;
       else //*PartList* is not empty
       Get a partition P from the head of *PartList*, and get a document $d \in P$;
       if the document d contain the term $t_{\text{rank } i}$ **then**
       Add $P(t_{\text{rank } i})$ to the front of *PartList*; add $P(t'_{\text{rank } i})$ to the front of *PartList*;
       else // the document d does not contain the term $t_{\text{rank } i}$
       Add $P(t'_{\text{rank } i})$ to the front of *PartList*; add $P(t_{\text{rank } i})$ to the front of *PartList*;
6. i:=1;
7. **while** *PartList* is not empty **do** **/*document identifier assignment procedure*/**
   - 7.1 Get a partition P from the head of *PartList*, and then remove P from *PartList*;
   - 7.2 **while** P is not empty **do**
     - **7**.2.1 Get a document $d \in P$, and remove d from P;
     - 7.2.2 Assign document identifier i to the document d, and then i:=i+1;

Figure 4.7 The PBDIA algorithm for the DIA problem

The PBDIA algorithm is given in Figure 4.7. A doubly linked list is used to store the partitions, and the two links of a partition maintain the ordering among these partitions. Given a collection of N documents and n distinct query terms, the number of comparisons for assigning documents to partitions in each iteration is $O(N)$. Since the PBDIA algorithm iterates for n times, the total number of comparisons for the PBDIA algorithm is $O(N \times n)$. Compared with the Greedy-NN algorithm, this complexity of PBDIA algorithm is distinctively low. This advantage brings the PBDIA algorithm a dark side, of course. Although the PBDIA algorithm targets on improving the compression efficiency for the frequently used query terms, it unavoidably decreases that for the other query terms. In reality, it is often the case that the popularities of the assorted query terms are very unbalanced. And this imbalance nature makes the PBDIA algorithm achieve very good query performance. In Section 4.4, we compare the search performance of the Greedy-NN and PBDIA algorithms for real-life document collections.

4.4 Performance Evaluation

This section describes our experiments for evaluating the different DIA algorithms. Experiments were conducted on real-life document collections, and the average query processing time and the storage requirement for each DIA algorithm were measured.

4.4.1 Document collections and queries

Three document collections were used in the experiments. Their statistics are listed in Table 4.2. In this table, N denotes the number of documents; n is the number of distinct terms; F is the total number of terms in the collection; and f indicates the number of document identifiers that appear in an inverted file. The collections *FBIS* (Foreign Broadcast Information Service) and *LAT* (LA Times) are disk 5 of the TREC-6 collection that is used internationally as a test bed for research

in IR techniques (Voorhees and Harman 1997). The collection *TREC* includes the *FBIS* and *LAT*.

Table 4.2 Statistics of document collections

		Collection		
		FBIS	*LAT*	*TREC*
# of documents	N	130,471	131,896	262,367
# of terms	F	72,922,893	72,087,460	145,010,353
# of distinct terms	n	214,310	168,251	317,393
# of document identifier count	f	28,628,698	32,483,656	61,112,354
Total size (Mbytes)		470	475	945

We followed the method (Moffat & Zobel, 1996) to evaluate performance with random queries. For each document collection, 300 documents were randomly selected to generate a query set. A query was generated by selecting words from the word list of a specific document. To form the word list of a document, words in the document were folded to lower case, and stop words such as "the" and "this" were eliminated. The number of terms per query ranged from 1 to 65. For example, a query containing 5 terms may be "inverted file document collection built". For each query, there existed at least one document in the document collection that is relevant to the query. We also made the generated query set for each document collection have the following characteristics: (1) Query repetition frequencies followed a Zipf distribution $Pr(q) \sim \frac{1}{\rho^{0.6}}$, where $Pr(q)$ is the probability of query q appearing in generated query set, and ρ is the popularity rank of query q; (2) The terms per query distribution followed the shifted negative binomial distribution $f(x) = \binom{x-0.8}{x-2}(0.85)^{1.2}(0.15)^{x-1}$, where $f(x)$ is the probability of a query containing x words. This made the distribution of generated queries closely resemble the distribution of real queries (Xie & O'Hallaron, 2002; Wolfram, 1992).

4.4.2 Performance results

In this sub-section, we first present the actual times taken by the Greedy-NN and the PBDIA algorithms. Then we present the query performance of different DIA algorithms. Finally, we present the compression performance of different DIA algorithms.

The inverted files of the three test collections were constructed according to the DIAs generated by different DIA algorithms. We tested four different DIA algorithms: "Random", "Default", "Greedy-NN", and "PBDIA". The Random algorithm means that the document in a collection is randomly assigned document identifier. The Default algorithm means that the document in a collection is assigned document identifier in chronological order. The Greedy-NN and PBDIA algorithms were described in Section 3.2 and Section 4.2, respectively. For each DIA algorithm, we also tested five coding methods: γ coding (Elias 1975), Golomb coding (Golomb 1966; Witten et al. 1999), skewed Golomb coding (Teuhola 1978), batched LLRUN coding (Fraenkel & Klein 1985), and unique-order interpolative coding method (Cheng et al. 2004). For the following experiments, the parameter *b* for each posting list in Golomb coding was calculated using Witten's approximation (Witten et al. 1999), and the parameter *g* for unique-order interpolative coding was set to 4 (Cheng et al. 2004).

All experiments were run on an Intel P4 2.4GHz PC with 512MB DDR memory running Linux operating system 2.4.12. The hard disk was 40GB, and the data transfer rate was 25MB/sec. Intervening processes and disk activities were minimized during experimentation.

Time taken by Greedy-NN and PBDIA algorithms

In Table 4.3, the performance in terms of completion time is shown. The times reported are the actual times taken by the algorithms to generate a DIA for the given document collection that has been inverted. Please note that the times presented in Table 4.3 consider neither the time spent

in preliminary inversion of the document collection, nor the time needed to rebuild an inverted file with a new DIA.

Table 4.3 shows that the PBDIA algorithm is much faster than the Greedy-NN algorithm. This fact makes the PBDIA algorithm viable for use in large-scale IRSs. Such a fast DIA algorithm can be very useful for situations such as:

1. Dynamically changing probability distribution of query terms, and
2. Dynamically changing document collection.

Table 4.3 Time consumed by the Greedy-NN and the PBDIA algorithms

DIA algorithm	Collection		
	FBIS	*LAT*	*TREC*
Greedy-NN	23 hrs 59 mins	24 hrs 37 mins	198 hrs 2 mins
PBDIA	9 secs	10 secs	18 secs

Query performance of different *DIA* algorithms

In Table 4.4, the average query processing time ($AvgTime_{QP}$) and the speedup relative to the Default algorithm (*SP*) were measured according to Eq.(4.3). In Table 4.5, the average number of bits required to retrieve and decode an identifier during query processing ($AvgBPI_{QP}$) and the improvement over the Default algorithm (*Imp*) were measured according to Eq.(4.6). For each document collection, the generated query set was divided into three subsets: the short query set, the medium-length query set, and the long query set. The number of terms per query for the short, medium-length, and long query sets range from 1 to 8, 9 to 20, and 21 to 65, respectively.

All decoding mechanisms were optimized, including:

1. Replaced subroutines with macros.
2. Replaced calls to the log function with fast bit shifts.
3. Careful choice for compiler optimization flags.
4. Implementation used 32-bit integers, as that is the internal register size of the Intel P4 CPU.

Furthermore, the Huffman code of batched LLRUN coding was implemented with canonical prefix codes that can be decoded via a fast table look-up (Turpin 1998). With these optimizations, decoding of a document identifier only required tens of ns.

The experimental results are shown in Tables 4.4 and 4.5. Key findings are:

1. Table 4.4 shows that the query performance of the Default algorithm can be 10% faster than the Random algorithm. This indicates that the Default algorithm already captures some clustering nature, thus can serve as a rigid baseline in comparison with other fine-tuned algorithms.
2. Comparing Tables 4.4 and 4.5, one should observe that $AvgTime_{QP}$ is proportional to $AvgBPI_{QP}$. This verifies Eq. (4.4) in Section 3.1, and explains why a good DIA can result in better compression and reduced query processing time.
3. From Table 4.5, one should observe that both the Greedy-NN and PBDIA algorithms can result in better compression of posting lists for all tested coding methods except Golomb coding. This indicates that the Greedy-NN and PBDIA algorithms can improve the cache efficiency if a posting list cache is implemented.
4. Table 4.4 shows that both the Greedy-NN and PBDIA algorithms can reduce average query processing time for all tested coding methods except Golomb coding. And the query speedup differences between the Greedy-NN and PBDIA algorithms were only 3% on average. Considering the algorithm complexity, the PBDIA algorithm is a good choice for the DIA problem.
5. From Table 4.4, one should observe that Golomb coding cannot benefit much from the Greedy-NN and PBDIA algorithms in terms of query performance. This is because Golomb coding assumes that the *d*-gap values in a posting list following a Bernoulli model (Witten et al. 1999),

hence both the compression result and the query processing time of Golomb coding are independent of *d*-gap distribution.

6. From Table 4.4, one should observe that the query speedup obtained by the PBDIA algorithm becomes higher as the query length increases. This is because that, as the number of query terms increases, more frequently used query terms are likely to be included, resulting in more advantage due to the PBDIA algorithm.
7. Table 4.4 shows that both γ coding and unique-order interpolative coding are recommended for real-world IRSs due to their fast query throughputs. In addition, compared with the other tested coding methods, these two coding methods benefit more from the PBDIA algorithm. We conclude that the PBDIA algorithm is viable for use in real-world IRSs.
8. Table 4.4 shows that the PBDIA algorithm can reduce average query processing time by up to 20% for an inverted file in which the document identifiers in a posting list are sorted in ascending order. To allow extremely fast processing of conjunctive queries and ranked queries using the same index, most IRSs in use today adopt the skipped inverted files (Moffat & Zobel, 1996) or the blocked inverted files (Moffat et al., 1995). Both the skipped and blocked inverted files are identifier-ordered arrangement. Therefore, the PBDIA algorithm can also be applied to those inverted files, and reduce the time needed to process a query against those inverted files. Since skipped inverted files and blocked inverted files are widely used in modern large-scale IRSs, we believe that the PBDIA algorithm can contribute in real-world IRSs.

Table 4.4 Query performance of different DIA algorithms ($AvgTime_{QP}$ is the average query processing time, and *SP* is the speedup relative to the Default algorithm)

(a) short queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*
FBIS	Random	2989	0.93	2858	0.98	3894	0.96	3748	0.97	2746	0.95
	Default	2789	1.00	2802	1.00	3754	1.00	3636	1.00	2614	1.00
	Greedy-NN	2431	1.15	2790	1.00	3348	1.12	3275	1.11	2315	1.13
	PBDIA	2529	1.10	2808	1.00	3427	1.10	3320	1.10	2333	1.12
LAT	Random	2829	0.96	2704	0.99	3737	0.98	3654	0.97	2564	0.97
	Default	2724	1.00	2688	1.00	3645	1.00	3542	1.00	2476	1.00
	Greedy-NN	2268	1.20	2653	1.01	3137	1.16	3143	1.13	2085	1.19
	PBDIA	2379	1.15	2644	1.02	3234	1.13	3231	1.10	2150	1.15
TREC	Random	5822	0.90	5573	0.97	7556	0.93	7217	0.94	5448	0.91
	Default	5244	1.00	5380	1.00	7026	1.00	6781	1.00	4942	1.00
	Greedy-NN	4431	1.18	5353	1.01	6139	1.14	6032	1.12	4256	1.16
	PBDIA	4606	1.14	5292	1.02	6254	1.12	6171	1.10	4313	1.15

(b) medium-length queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*
FBIS	Random	9388	0.93	8972	0.98	12222	0.97	11749	0.97	8613	0.95
	Default	8758	1.00	8795	1.00	11795	1.00	11402	1.00	8201	1.00
	Greedy-NN	7563	1.16	8746	1.01	10426	1.13	10225	1.12	7205	1.14
	PBDIA	7838	1.12	8798	1.00	10650	1.11	10387	1.10	7223	1.14
LAT	Random	8997	0.97	8605	1.00	11842	0.98	11562	0.97	8192	0.97
	Default	8684	1.00	8564	1.00	11580	1.00	11229	1.00	7932	1.00
	Greedy-NN	7126	1.22	8407	1.02	9851	1.18	9852	1.14	6607	1.20
	PBDIA	7434	1.17	8359	1.02	10098	1.15	9982	1.12	6755	1.17
TREC	Random	18475	0.92	17689	0.97	23936	0.94	22724	0.95	17273	0.93
	Default	16935	1.00	17153	1.00	22594	1.00	21666	1.00	16004	1.00
	Greedy-NN	14069	1.20	16942	1.01	19493	1.16	19058	1.14	13598	1.18
	PBDIA	14611	1.16	16713	1.03	19809	1.14	19280	1.12	13722	1.17

(c) long queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*	$AvgTime_{QP}$ (us)	*SP*
FBIS	Random	20210	0.92	19399	0.98	26526	0.95	26049	0.96	18423	0.94
	Default	18594	1.00	18939	1.00	25316	1.00	24984	1.00	17269	1.00
	Greedy-NN	15882	1.17	18971	1.00	22131	1.14	21957	1.14	14979	1.15
	PBDIA	15871	1.17	18953	1.00	21972	1.15	22143	1.13	14377	1.20
LAT	Random	18029	0.96	17116	1.00	23591	0.98	22646	0.97	16477	0.97
	Default	17392	1.00	17035	1.00	23011	1.00	22033	1.00	15964	1.00
	Greedy-NN	13875	1.25	16624	1.02	19173	1.20	18984	1.16	13046	1.22
	PBDIA	13996	1.24	16298	1.05	19023	1.21	19212	1.15	12817	1.25
TREC	Random	37881	0.93	36023	0.98	49012	0.95	46584	0.96	35266	0.94
	Default	35096	1.00	35231	1.00	46547	1.00	44588	1.00	33008	1.00
	Greedy-NN	28372	1.24	34469	1.02	39489	1.18	38592	1.16	27523	1.20
	PBDIA	29152	1.20	33809	1.04	39766	1.17	39089	1.14	27401	1.20

Table 4.5 $AvgBPI_{QP}$ of different DIA algorithms ($AvgBPI_{QP}$ is the average number of bits required to retrieve and decode an identifier during query processing, and *Imp* is the improvement over the Default algorithm)

(a) short queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)
FBIS	Random	3.56	-10.6	3.21	0.3	3.31	-7.1	3.25	-5.5	3.15	-7.9
	Default	3.22	---	3.22	---	3.09	---	3.08	---	2.92	---
	Greedy-NN	2.78	13.7	3.24	-0.6	2.73	11.7	2.69	12.7	2.63	9.9
	PBDIA	2.95	8.4	3.23	-0.3	2.84	8.1	2.76	10.4	2.69	7.9
LAT	Random	3.32	-6.8	2.98	0.0	3.05	-4.8	3.00	-3.8	2.87	-4.7
	Default	3.11	---	2.98	---	2.91	---	2.89	---	2.74	---
	Greedy-NN	2.56	17.7	3.00	-0.7	2.48	14.8	2.47	14.5	2.35	14.2
	PBDIA	2.73	12.2	2.97	0.3	2.59	11.0	2.59	10.4	2.42	11.7
TREC	Random	3.75	-13.3	3.38	0.3	3.46	-9.5	3.40	-8.2	3.34	-10.6
	Default	3.31	---	3.39	---	3.16	---	3.14	---	3.02	---
	Greedy-NN	2.78	16.0	3.41	-0.6	2.72	13.9	2.69	14.3	2.65	12.3
	PBDIA	2.94	11.2	3.37	0.6	2.81	11.1	2.81	10.5	2.70	10.6

(b) medium-length queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)
FBIS	Random	3.57	-10.9	3.21	0.3	3.31	-6.8	3.25	-5.5	3.15	-7.9
	Default	3.22	---	3.22	---	3.10	---	3.08	---	2.92	---
	Greedy-NN	2.75	14.6	3.24	-0.6	2.70	12.9	2.66	13.6	2.61	10.6
	PBDIA	2.92	9.3	3.24	-0.6	2.81	9.4	2.75	10.7	2.66	8.9
LAT	Random	3.37	-6.3	3.03	0.3	3.11	-4.4	3.06	-3.7	2.94	-4.6
	Default	3.17	---	3.04	---	2.98	---	2.95	---	2.81	---
	Greedy-NN	2.58	18.6	3.06	-0.7	2.50	16.1	2.48	15.9	2.39	14.9
	PBDIA	2.73	13.9	3.02	0.7	2.59	13.1	2.60	11.9	2.44	13.1
TREC	Random	3.83	-12.0	3.42	0.3	3.53	-8.3	3.47	-7.1	3.40	-9.0
	Default	3.42	---	3.43	---	3.26	---	3.24	---	3.12	---
	Greedy-NN	2.82	17.5	3.45	-0.6	2.76	15.3	2.74	15.4	2.71	13.1
	PBDIA	2.99	12.6	3.41	0.6	2.85	12.6	2.86	11.7	2.75	11.9

(c) long queries

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)	$AvgBPI_{QP}$	*Imp* (%)
FBIS	Random	3.31	-12.2	3.02	0.3	3.09	-8.4	3.03	-6.7	2.90	-9.0
	Default	2.95	---	3.03	---	2.85	---	2.84	---	2.66	---
	Greedy-NN	2.50	15.3	3.06	-1.0	2.47	13.3	2.43	14.4	2.37	10.9
	PBDIA	2.57	12.9	3.05	-0.7	2.47	13.3	2.48	12.7	2.34	12.0
LAT	Random	3.58	-6.2	3.21	0.3	3.28	-4.1	3.23	-3.5	3.13	-4.3
	Default	3.37	---	3.22	---	3.15	---	3.12	---	3.00	---
	Greedy-NN	2.66	21.1	3.24	-0.6	2.58	18.1	2.55	18.2	2.50	16.7
	PBDIA	2.73	19.0	3.19	0.9	2.58	18.1	2.63	15.7	2.48	17.3
TREC	Random	3.85	-10.6	3.43	0.3	3.54	-7.3	3.47	-6.1	3.41	-7.9
	Default	3.48	---	3.44	---	3.30	---	3.27	---	3.16	---
	Greedy-NN	2.78	20.1	3.46	-0.6	2.73	17.3	2.70	17.4	2.69	14.9
	PBDIA	2.92	16.1	3.41	0.9	2.79	15.5	2.81	14.1	2.71	14.2

Compression performance of different DIA algorithms

The compression results are shown in Table 4.6, and the metric used is the average number of *bits per identifier BPI*, defined as follows:

$$BPI = \frac{\text{The size of the compressed inverted file}}{\text{number of document identfiers } f}.$$

To reduce average query processing time, both the Greedy-NN and PBDIA algorithms target on improving the compression efficiency for the frequently used query terms. However, this is at the cost of sacrificing the compression efficiency for the less frequently used query terms. We need to know how much space overhead is needed to trade for this speed advantage. Results in Table 4.6 show that the Greedy-NN and PBDIA algorithms can speed up query processing with very little or no storage overhead.

Table 4.6 Compression performance of different DIA algorithms (*BPI* is the average bits per identifier of the inverted file for the test collection, and *Imp* is the improvement over the Default algorithm)

Collection	DIA algorithm	Coding Methods									
		γ coding		Golomb coding		Skewed Golomb coding		Batched LLRUN coding		Unique-order Interpolative coding	
		BPI	*Imp* (%)	*BPI*	*Imp* (%)	*BPI*	*Imp* (%)	*BPI*	*Imp* (%)	*BPI*	*Imp* (%)
FBIS	Random	7.06	-19.7	5.28	0.0	5.75	-10.6	5.38	-8.5	5.36	-10.3
	Default	5.90	---	5.28	---	5.20	---	4.96	---	4.86	---
	Greedy-NN	5.86	0.7	5.28	0.0	5.33	-2.5	4.88	1.6	4.85	0.2
	PBDIA	6.17	-4.6	5.28	0.0	5.42	-4.2	5.06	-2.0	4.95	-1.9
LAT	Random	7.12	-6.6	5.33	0.0	5.73	-3.2	5.43	-2.8	5.42	-3.8
	Default	6.68	---	5.33	---	5.55	---	5.28	---	5.22	---
	Greedy-NN	6.06	9.3	5.32	0.2	5.26	5.2	5.00	5.3	4.91	5.9
	PBDIA	6.35	4.9	5.32	0.2	5.33	4.0	5.12	3.0	5.01	4.0
TREC	Random	7.39	-16.7	5.50	-0.4	5.92	-9.2	5.59	-7.5	5.59	-9.6
	Default	6.33	---	5.48	---	5.42	---	5.20	---	5.10	---
	Greedy-NN	6.08	3.95	5.49	-0.2	5.39	0.6	5.03	3.3	4.99	2.2
	PBDIA	6.36	-0.5	5.49	-0.2	5.45	-0.6	5.18	0.4	5.08	0.4

4.5 Summary

In this chapter, we study the DIA-based inverted file optimization techniques for an IRS. With an inverted file, we first define a cost model for query evaluation. Based on this model, we propose an efficient heuristic, called *partition-based document identifier assignment* (PBDIA) algorithm, to generate a good DIA for the inverted file to reduce average query processing time. The PBDIA algorithm can efficiently assign consecutive document identifiers to the documents containing frequently used query terms. This makes the *d*-gaps of posting lists for frequently used query terms very small, and results in better compression for popular coding methods without increasing the complexity of decoding processes. This can result in reduced query processing time. For the fastest unique-order interpolative coding, experimental results show that the PBDIA algorithm can reduce the average query processing time by up to 20%. We also point out that the DIA problem has vital effects on the performance of long queries. Compared with the well-known Greedy-NN algorithm, the PBDIA algorithm is much faster and yields very competitive performance for the DIA problem. This fact should make the PBDIA algorithm viable for use in modern large-scale inverted file-based IRSs.

Chapter 5 Parallel IR

The rapid growth in Internet usage brings wide variety of applications as well as new system design challenges on *information retrieval systems* (IRSs). The problem of information explosion overwhelms the load of CPU and disk on an *information retrieval* (IR) server. In this chapter, we intend to reduce query processing time of an IRS by using a cluster as the server architecture. Queries are processed on a cluster of workstations − each has its own CPU, memory, and disk − interconnected by a local area network. For example, Google search engine is a cluster of more than 6000 PCs and each PC contains Gigabytes of random access memory. The key research issue here is to partition the inverted file into sub-files each for one workstation such that, during query processing, all workstations have to consult their own sub-files in parallel and query processing time can be reduced.

Two main approaches for partitioning inverted files are in general use: the *TermID* partitioning approach (Reddaway, 1991; Stanfill et al., 1989; Ribeiro-Neto et al., 1999) and the *DocID* partitioning approach (Hawking, 1996; Aalbersberg & Sijstermans, 1990; Stanfill & Thau, 1991; Hollaar, 1991). The *TermID* partitioning approach takes a posting list as an object to be allocated, whereas the *DocID* partitioning approach takes the set of all postings referring to a document identifier as an object. MacFarlane (2000) and Ma et al. (2002) showed that the *DocID* partitioning approach is a better inverted file distribution method. This is because that the *DocID* partitioning approach can parallelize both CPU computation and disk accesses without inducing communication overhead of transferring posting lists between workstations.

With the *DocID* partitioning approach, Ma et al. (2002) proposed some partitioning algorithms to partition and distribute the inverted file onto disks of workstations such that the average query

processing time of parallel query processing can be minimized. They have shown that the interleaving partitioning scheme can partition an inverted file with good load balance and produce a near-ideal speedup. We observe that the document identifier clustering plays an important role for this interleaving partitioning scheme in load balance and query speedup. Hence, we propose using the PBDIA algorithm (described in Section 4.3) to enhance the clustering property of document identifiers in posting lists by assigning consecutive identifiers to those documents containing frequently used query terms. Experimental results show that the PBDIA algorithm can aid the interleaving partitioning scheme to achieve a better load balance and improve the parallel query performance by a factor of 1.13 to 1.18 no matter how many workstations are in the cluster. The PBDIA algorithm has substantial and consistent potential to improve the performance of an IRS run on a cluster of workstations. This shows that the clustering property should deserve much attention in parallel IR.

The remainder of this chapter is organized as follows. Section 5.1 describes the concerned inverted file partitioning problem. The interleaving partitioning scheme is described in Section 5.2. In Section 5.3, we present the framework of the proposed approach to partition an inverted file. Performance evaluation is presented in Section 5.4. Finally, Section 5.5 presents our summary.

5.1 Inverted File Partitioning Problem

The inverted file partitioning problem considered in this chapter is as follows. The inputs to an inverted file partitioning algorithm are

- a compressed inverted file for sequential processing,
- popularities of terms appearing in queries, and
- number of workstations.

The output is a partitioned compressed inverted file to be distributed on the set of workstations. The objective is to minimize the average query processing time of parallel query processing. Issues to the objective are

- eliminating the communication overhead of transferring postings between workstations during query processing,
- balancing amount of postings to be processed during parallel query processing, and
- keeping compression efficiency in the partitioned compressed inverted file.

Ma et al. (2002) have proven that this problem is known to be NP-complete since it is identical to the multiprocessor scheduling problem defined in Garey & Johnson (1979). Hence, a heuristic algorithm for this optimization problem should be developed.

5.2 Fundamental: Interleaving Partitioning Scheme

In Section 5.2.1, we describe the well-known interleaving partitioning scheme that apply interleaved mapping rule to generate a partitioned inverted file and produce a near-ideal speedup. In Section 5.2.2, we describe how to improve the average processing time through document identifier assignment on the partitioned inverted file generated with the interleaving partitioning scheme.

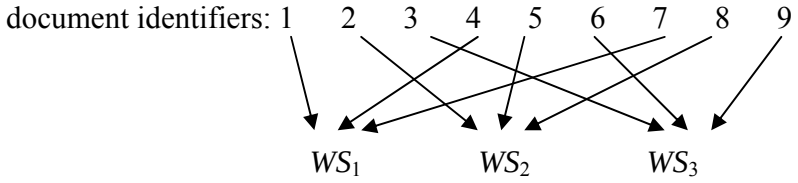
5.2.1 Algorithm description

Figure 5.1 shows the idea of the interleaving mapping rule. Each workstation is mapped with a set of interleaved document identifiers. Let M be the number of workstations and N be the number of documents. The rule for mapping document identifiers to workstations is as follows.

Rule 1 *The interleaved mapping rule maps a document identifier i to a workstation* WS_k *with a function* A_{intlv}:

$$k = A_{intlv}(i) = i - \left\lfloor \frac{(i-1)}{M} \right\rfloor \times M \qquad (5.1)$$

With the interleaved mapped rule, postings in a posting list are supposed to be evenly distributed regardless of the document identifier clustering.



(a) Mapping document identifiers to workstation IDs

posting list: 2, 3, 5, 7, 8, 11, 12, 13, 15, 16

	WS_1	WS_2	WS_3
represented using original document identifier:	7, 13, 16	2, 5, 8, 11	3, 12, 15
represented using local document identifier:	3, 5, 6	1, 2, 3, 4	1, 4, 5

(b) Partitioning a posting list

Figure 5.1 Partitioning with interleaved mapping rule

To keep compression efficiency, each workstation represents documents using local document identifiers. The mapping rule A_{intlv} increases the gap between document identifiers after partitioning. The gap between document identifiers in a local posting list is at least M. And compression methods can not work well on the local inverted file if documents are presented with the original document identifiers. We notice that, for a workstation WS_k, the local document identifier for a document identifier i mapped to WS_k can be obtained as following rule.

Rule 2 *In the partitioned inverted file generated by interleaved mapping rule, a document i is represented as local document identifier* $LID_{intlv}(i)$*:*

$$LID_{intlv}(i) = \lfloor (i-1)/M \rfloor + 1 \qquad (5.2)$$

Note that the original document identifier i mapped to WS_k then can be obtained using the following equation

$$i = M \times (LID_{intlv}(i) - 1) + k \tag{5.3}$$

Figure 5.2 presents the algorithm to generate a partitioned inverted file with interleaved mapping rule. The time complexity is $O(f)$ where f is the number of postings in the input inverted file.

Algorithm Interleaving_partitioning_scheme

Input:

IF: the inverted file for sequential query processing. *IF* consists of a set of posting lists PL_t for each term t.

Output:

$LIF=\{LIF_1, LIF_2, \ldots, LIF_M\}$: the set of local inverted files LIF_k for each workstation WS_k. Each LIF_k consists of a set of local posting lists $PL_t(WS_k)$ for each term t.

Method:

1. **for** each term t **do**
   1.1 **for** each document identifier $i \in PL_t$ **do**
      1.1.1 $k \leftarrow i - \lfloor (i-1)/M \rfloor \times M$
      1.1.2 $i' \leftarrow \lfloor (i-1)/M \rfloor + 1$
      1.1.3 append i' to $PL_t(WS_k)$

Figure 5.2 Interleaving partitioning scheme

5.2.2 How to improve parallel query processing through document identifier assignment

In this subsection, we use an example to show how to improve parallel query processing through document identifier assignment. Consider term t appears in documents d_1, d_3, d_4, d_6, d_8, d_{10}, d_{18}, d_{22}, d_{23}, d_{26}, d_{34}, d_{35}, d_{45}, d_{46}, d_{47}. There are two workstations in the cluster. We have two document identifier assignments DIA I and DIA II (cf. Figure 5.3). The notation $d_i \rightarrow j$ in DIAs I and II denotes that the document identifier j is assigned to the document d_i. For each DIA, we can obtain

Term t appears in documents d_1, d_3, d_4, d_6, d_8, d_{10}, d_{18}, d_{22}, d_{23}, d_{26}, d_{34}, d_{35}, d_{45}, d_{46}, d_{47}.

(a) DIA I: { $d_1 \rightarrow 1$, $d_3 \rightarrow 3$, $d_4 \rightarrow 4$, $d_6 \rightarrow 6$, $d_8 \rightarrow 8$, $d_{10} \rightarrow 10$, $d_{18} \rightarrow 18$, $d_{22} \rightarrow 22$, $d_{23} \rightarrow 23$, $d_{26} \rightarrow 26$, $d_{34} \rightarrow 34$, $d_{35} \rightarrow 35$, $d_{45} \rightarrow 45$, $d_{46} \rightarrow 46$, $d_{47} \rightarrow 47$}.

(1) The posting list PL_t for DIA I

PL_t: <1, 3, 4, 6, 8, 10, 18, 22, 23, 26, 34, 35, 45, 46, 47>

(2) The posting list PL_t for DIA I is partitioned into two sub-posting lists $PL_t(WS_1)$ and $PL_t(WS_2)$ using the interleaving partitioning scheme (α is a constant)

(i) original document identifier representation

sub-posting lists	bits after compression	QPT	PQPT
$PL_t(WS_1)$: <1, 3, 23, 35, 45, 47>	30 bits	30α	45α
$PL_t(WS_2)$: <4, 6, 8, 10, 18, 22, 26, 34, 46>	45 bits	45α	

(ii) local document identifier representation

sub-posting lists	bits after compression	QPT	PQPT
$PL_t(WS_1)$: <1, 2, 12, 18, 23, 24>	20 bits	20α	27α
$PL_t(WS_2)$: <2, 3, 4, 5, 9, 11, 13, 17, 23>	27 bits	27α	

(b) DIA II: { $d_1 \rightarrow 1$, $d_3 \rightarrow 2$, $d_4 \rightarrow 3$, $d_6 \rightarrow 4$, $d_8 \rightarrow 5$, $d_{10} \rightarrow 6$, $d_{18} \rightarrow 7$, $d_{22} \rightarrow 8$, $d_{23} \rightarrow 9$, $d_{26} \rightarrow 10$, $d_{34} \rightarrow 11$, $d_{35} \rightarrow 12$, $d_{45} \rightarrow 13$, $d_{46} \rightarrow 14$, $d_{47} \rightarrow 15$}

(1) The posting list PL_t for DIA II

PL_t: <1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15>

(2) The posting list PL_t for DIA II is partitioned into two sub-posting lists $PL_t(WS_1)$ and $PL_t(WS_2)$ using the interleaving partitioning scheme (α is a constant)

(i) original document identifier representation

sub-posting lists	bits after compression	QPT	PQPT
$PL_t(WS_1)$: <1, 3, 5, 7, 9, 11, 13, 15>	22 bits	22α	22α
$PL_t(WS_2)$: <2, 4, 6, 8, 10, 12, 14>	21 bits	21α	

(ii) local document identifier representation

sub-posting lists	bits after compression	QPT	PQPT
$PL_t(WS_1)$: <1, 2, 3, 4, 5, 6, 7, 8>	8 bits	8α	8α
$PL_t(WS_2)$: <1, 2, 3, 4, 5, 6, 7>	7 bits	7α	

Figure 5.3 An example to show how to improve parallel query processing through document identifier assignment. There are two workstations in the cluster. The interleaving partitioning scheme is employed to partition the posting list PL_t. All sub-posting lists are encoded in γ codes with the *d*-gap technique. QPT is the query processing time and PQPT is the parallel query processing time.

a posting list PL_t for term t and the PL_t can be partitioned into two sub-posting lists $PL_t(WS_1)$ and $PL_t(WS_2)$ using the interleaving partitioning scheme. Assume that all sub-posting lists are encoded

in γ codes with the *d*-gap technique, where the γ code represents an integer x in $1+2\lfloor \log_2 x \rfloor$ bits. Based on Eq.(4.4), we can derive the *query processing time* (QPT) of WS_1 for term t and that of WS_2 for term t. Then the parallel query processing time can be calculated using the time the last workstation finishes its job. This example confirms that local document identifier representation can improve the compression efficiency. We then observe that the compression efficiency of DIA II is better than that of DIA I. This implies that the query processing time of DIA II is shorter than that of DIA I since the query processing time is proportional to the total size of encoded posting list. The parallel query processing time of DIA II is also shorter than that of DIA I. Hence, this example shows that the clustering property in the posting list plays an important role in interleaving partition scheme.

5.3 Framework of Proposed Approach

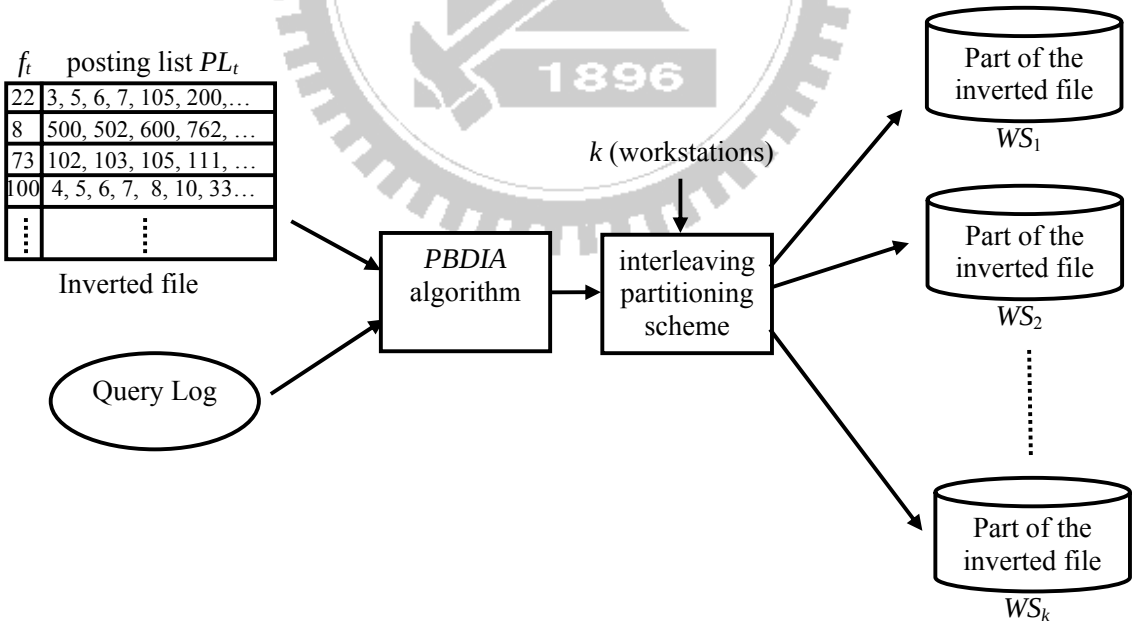


Figure 5.4 The proposed approach to partition an inverted file for an IRS that runs on a cluster of workstations.

In Section 5.2.2, we have observed that the clustering property plays an important role for interleaving partitioning scheme in both the load balance and the compression efficiency. Since the distribution of query terms is skewed, we recognize that the PBDIA can be employed to aid interleaving partitioning scheme to produce superior performance. The Figure 5.4 shows our proposed approach to partition an inverted file for an IRS that runs on a cluster of workstations. The performance evaluation is shown in next section.

5.4 Performance Evaluation

This section investigates the *document identifier assignment* (DIA) problem in an IRS that runs on a cluster of workstations. Experiments were conducted on real-life document collections. We measured the sequential query processing time for each workstation and calculated the parallel query processing time. The storage requirement of the partitioned inverted files was also presented.

5.4.1 Test collection and query set

Three document collections were used in the experiments. Their statistics are listed in Table 2. In this table, *N* denotes the number of documents; *n* is the number of distinct terms; *F* is the total number of terms in the collection; and *f* indicates the number of document identifiers that appear in an inverted file. The collections *FBIS* (Foreign Broadcast Information Service) and *LAT* (LA Times) are disk 5 of the TREC-6 collection that is used internationally as a test bed for research in *IR* techniques (Voorhees and Harman 1997). The collection *TREC* includes the *FBIS* and *LAT*.

Table 5.1 Statistics of document collections

		Collection		
		FBIS	*LAT*	*TREC*
# of documents	*N*	130,471	131,896	262,367
# of terms	*F*	72,922,893	72,087,460	145,010,353
# of distinct terms	*n*	214,310	168,251	317,393
# of document identifier count	*f*	28,628,698	32,483,656	61,112,354
Total size (Mbytes)		470	475	945

We followed the method (Moffat & Zobel, 1996) to evaluate performance with random queries. For each document collection, 300 documents were randomly selected to generate a query set. A query was generated by selecting words from the word list of a specific document. To form the word list of a document, words in the document were folded to lower case, and stop words such as "the" and "this" were eliminated. The number of terms per query ranged from 1 to 65. For example, a query containing 5 terms may be "inverted file document collection built". For each query, there existed at least one document in the document collection that is relevant to the query. We also made the generated query set for each document collection have the following characteristics: (1) Query repetition frequencies followed a Zipf distribution $Pr(q) \sim \frac{1}{\rho^{0.6}}$, where $Pr(q)$ is the probability of query q appearing in generated query set, and ρ is the popularity rank of query q; (2) The terms per query distribution followed the shifted negative binomial distribution $f(x) = \binom{x-0.8}{x-2}(0.85)^{1.2}(0.15)^{x-1}$, where $f(x)$ is the probability of a query containing x words. This made the distribution of generated queries closely resemble the distribution of real queries (Xie & O'Hallaron, 2002; Wolfram, 1992).

5.4.2 Performance results

This subsection shows the experimental results. These results include: (1) speedup of parallel query processing, and (2) compression efficiency.

Speedup of parallel query processing

This subsection investigates the DIA problem in an IRS that runs on a cluster of workstations. Assuming k workstations, the inverted file is generally partitioned into k disjoint sub-files, each for one workstation. Table 5.2 shows the performance of parallel query processing using interleaving partitioning scheme with either the Default algorithm or the PBDIA algorithm, where the Default

algorithm means that the documents in a collection are assigned document identifiers in chronological order. The Default algorithm is widely used in modern IRSs, and it already captures some clustering nature. Hence, the Default algorithm can serve a rigid baseline in comparison with the PBDIA algorithm. The metric is the speedup relative to sequential query processing with Default algorithm. Experiments were conducted on the *TREC* collection. The sub-file on each workstation was compressed using the unique-order interpolative coding method (g=4). The parallel query processing time was defined as $\max[T_1,T_2,\ldots,T_k]$, where T_i ($1 \le i \le k$) was the time needed to retrieve and decompress the (partial) posting lists for the query terms on the i^{th} workstation. The experimental results show that the interleaving partitioning scheme can yield near-ideal speedups, as reported in Ma et al. (2002). In addition, using the PBDIA algorithm to enhance the clustering property of posting lists for frequently used query terms, the interleaving partitioning scheme yields super-linear speedups. Hence the DIA problem should deserve much attention in parallel IR.

Table 5.2 Speedup of parallel query processing

Collection	Approach	The number of workstations					
		1[a]	2	4	6	8	10
FBIS	*Default* + Interleaving partitioning	1.00	1.89	3.73	5.58	7.41	9.30
	PBDIA + Interleaving partitioning	1.14	2.16	4.26	6.37	8.45	10.60
LAT	*Default* + Interleaving partitioning	1.00	1.90	3.76	5.63	7.46	9.37
	PBDIA + Interleaving partitioning	1.18	2.25	4.44	6.65	8.80	11.04
TREC	*Default* + Interleaving partitioning	1.00	1.90	3.75	5.61	7.44	9.35
	PBDIA + Interleaving partitioning	1.17	2.23	4.41	6.57	8.70	10.93

[a] Without interleaving partitioning

Compression Efficiency

To reduce average query processing time of parallel query processing, the PBDIA algorithm improves the compression efficiency for the frequently used query terms. However, this is at the

cost of sacrificing the compression efficiency for the less frequently used query terms. We need to know how much space overhead is needed to trade for this speed advantage. Average bits per document identifier of the different partitioning approaches are shown in Table 5.3. The sub-file on each workstation was compressed using the unique-order interpolative coding method (g=4). Results in Table 5.3 show that the PBDIA algorithms can speed up query processing with very little or no storage overhead.

Table 5.3 Compression performance of different partitioning approaches

Collection	Approach	The number of workstations					
		1[a]	2	4	6	8	10
FBIS	*Default* + Interleaving partitioning	4.86	4.88	4.86	4.85	4.83	4.82
	PBDIA + Interleaving partitioning	4.95	4.98	4.96	4.95	4.95	4.94
LAT	*Default* + Interleaving partitioning	5.22	5.23	5.23	5.21	5.19	5.17
	PBDIA + Interleaving partitioning	5.01	5.02	5.01	5.01	4.99	4.97
TREC	*Default* + Interleaving partitioning	5.10	5.13	5.12	5.10	5.07	5.05
	PBDIA + Interleaving partitioning	5.08	5.11	5.08	5.07	5.05	5.04

[a] Without interleaving partitioning

5.5 Summary

This chapter is to propose an inverted file partitioning algorithm for parallel information retrieval. The inverted file is generally partitioned into disjoint sub-files, each for one workstation, in an IRS that runs on a cluster of workstations. When processing a query, all workstations have to consult only their own sub-files in parallel. The objective of this chapter is to develop an inverted file partitioning algorithm that minimizes the average query processing time of parallel query processing. Our approach is as follows. The foundation is interleaving partitioning scheme, which generates a partitioned inverted file with interleaved mapping rule and produces near-ideal speedup.

The key idea of our proposed algorithm is to use the document identifier assignment algorithm to enhance the clustering property of posting lists for frequently used query terms. This can aid the interleaving partitioning scheme to produce superior query performance.

Chapter 6 Conclusions

This dissertation studies methodologies to improve the efficiency of an IRS that runs on a cluster of workstations. The key idea is developing efficient algorithms to reduce space and time needed to store and operate on the most-widely-used indexing structure, called the inverted file. The objective is to increase the efficiency of an IRS without increasing the hardware cost of the cluster. Research issues are

(1) Inverted file size reduction,

(2) Redundant decoding elimination,

(3) Inverted file optimization, and

(4) Parallel IR

The contributions of this dissertation are involved in the two important research directions:

(1) Efficient indexing and fast searching for large scale IRSs, and

(2) Parallel IR.

Based on the results of this dissertation, various new research topics in these two directions can be studied.

6.1 Dissertation Summary

The research topics in the dissertation are

(1) Efficient coding method for inverted file size reduction,

(2) Two-level skipped inverted file for redundant decoding elimination,

(3) Document identifier assignment algorithm design for inverted file optimization, and

(4) Inverted file partitioning for parallel IR.

The primary results of these research topics are:

(1) For inverted file size reduction, we propose a novel coding method, called unique-order interpolative coding, to compress inverted files. This method facilitates the decoding process for interpolative coding using recursive elimination and loop unwinding. This method has both the advantages of compression ratio and fast decompression. Experimental results show that this method allows query throughput rate of approximately 30% higher than well-known Golomb coding and still provides superior compression.

(2) For redundant decoding elimination, we propose a two-level skipped inverted file to simultaneously provide excellent query speed on both conjunctive Boolean queries and ranked queries with very little or no space overhead. We first employ well-known skipping mechanisms to create the first-level index on each compressed posting list by dividing the list into large blocks. Then we propose a novel skipping mechanism to create the second-level index on each large block by dividing the block into small sub-blocks. The first-level index is constructed to optimize the query performance of conjunctive Boolean quires, whereas the second-level index is to optimize the query performance of ranked queries. Experimental results show that the proposed two-level skipped inverted file improves the query speed for conjunctive Boolean queries by up to 16%, and for ranked queries up to 44%, compared with the conventional one-level skipped inverted file.

(3) For inverted file optimization, we propose a fast *document identifier assignment* (DIA) algorithm, called *partition-based DIA* (PBDIA) algorithm, to generate a good DIA for the inverted file to optimize average query processing time when the distribution of query terms is known. In a typical IRS, a few frequently used query terms constitute a large portion of all term occurrences in queries. Based on this fact, the PBDIA algorithm assigns consecutive document

identifiers to those documents containing frequently used query terms. Experimental results show that the PBDIA algorithm only takes a few seconds to generate a DIA for a collection of 1GB, and improves query speed by up to 25%.

(4) For parallel IR, we propose a novel approach that partitions an inverted file to minimize parallel query processing time. The interleaving partitioning scheme has been proven that it can partition an inverted file with good load balance and produce near-ideal speedup. We observe that the cluster property plays an important role for interleaving partitioning scheme in the load balance and the query speed. We propose using the PBDIA algorithm to enhance the cluster property of document identifiers in posting lists. Experimental results show that the PBDIA algorithm can further improve the parallel query speed for interleaving partitioning scheme by 14% to 17% no matter how many workstations are in the cluster.

To verify scalability of our research works, we concatenated the *FBIS* and *LAT* to form a bigger collection *TREC*. Except for the topic 2 (2-level skipped inverted file), *FBIS*, *LAT*, and *TREC* were used to evaluate our proposed methods in the other three research topics. In these topics, the performance of our proposed methods for *TREC* is not worse than that for *FBIS* and *LAT*. This indicates that our proposed methods provide good scalability. We believe that this is also true for topic 2 since the topic 2 adopts the same idea of topic 1 to accelerate the decoding process of interpolative coding.

There are several issues that need to be discussed:

(1) Inverted file updating

Although our research works focus only on static document collections, they can still work well for dynamically changing collections with very few modifications.

For dealing with changes due to inserted documents, sparing free space for each posting list can be allocated to allow future expansion (Brown et al., 1994), and the postings in the posting lists should be stored in descending order by document identifier since it is typically more efficient to insert at the head of the list than in any other location. This does not affect the performance of our research works.

For dealing with changes due to deleted documents, a searchable update log can be used to store the postings of deleted documents between periodic rebuilds. When (partially) rebuilding inverted file, query processing is used to search both the inverted file and the update log, and merge the results of both. This can be accelerated by using our research works.

(2) Disk design considerations

We use an IDE hard disk per workstation in our experiments. However, low disk throughput is one of the main impediments to improving the performance of our research works (see Table 2.8). How to increase disk throughput with different disk organizations/architectures is a very interesting research topic. For example, SCSI disk drives and disk arrays can be employed to improve disk throughput. For SCSI disk drives, to amortize the cost of a disk access, the controller read a fixed number of consecutive blocks ahead and stores them in its cache. How to adjust the block size and the number of read-ahead blocks is an important issue. For disk arrays, the simplest and best-known technique for balancing load is striping. Striping groups several sequential disk blocks in units of fixed size and lays those units out across the physical disks in round-robin fashion. How to adjust the size of striping unit is an important issue.

(3) Fast document retrieval

There are two techniques used to evaluate queries in modern large-scale IRSs: eager (or term-at-a-time) query evaluation and lazy (or document-at- a-time) query evaluation (Turtle & Flood, 1995). In the first case, the posting list of one of query terms is computed first (usually, choosing the rarest term), and then, it is merged or filtered with the other lists. When evaluating is lazy, instead, posting lists are scanned in parallel, retrieving in sequence each document satisfying the query. The latter approach is essential in very large document collections, where the actual number of documents that could be retrieved is guessed, and the scan for documents satisfying the query is stopped as soon as enough documents have been retrieved. In this dissertation, we use eager query evaluation to verify our research works. However, we believe our research can still work well for lazy query evaluation. This is because lazy evaluation requires keeping constantly in sync several posting lists. To perform this operation efficiently, it is essential that a skip method is available that allows the caller to quickly reach the first document identifier larger than or equal to a given one. Our proposed two-level skipped inverted file can work well for this problem.

6.2 Contribution and Suggested Work

The contributions of this dissertation are involved in the two important research directions:

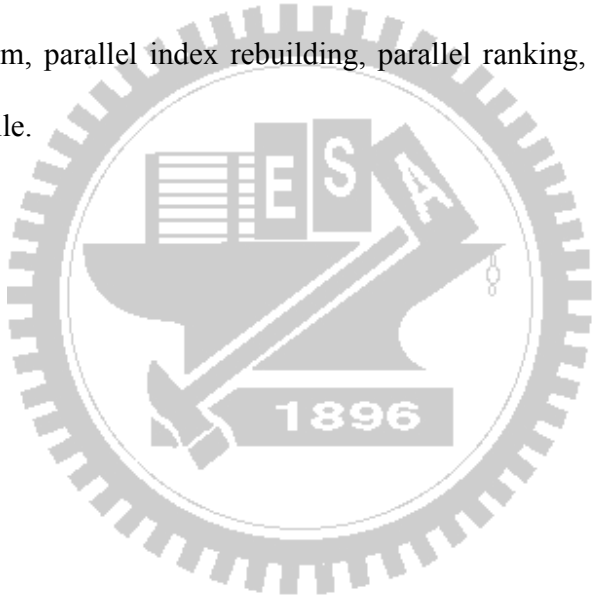
(1) Efficient indexing and fast searching for large scale IRSs

With the Internet explosive growth, the index structuring for large scale IRSs become more and more important. The barriers to make the efficient index structuring feasible were the changes of IRS scale and user query behavior. This dissertation presents the keys to eliminate the barriers: inverted file compression, skipped inverted file, and inverted file optimization. Based

on the results in this dissertation, various new topics can be investigated, including multimedia IR indexing, various search techniques, and inverted file caching.

(2) Parallel IR

The importance of parallel IR comes from the high performance requirement brought by Internet growth. The barrier to make parallel IR feasible was the lack of inverted file partitioning method to achieve ideal speedup. This dissertation presents the key to eliminate the barrier: the interleaving partitioning scheme with the PBDIA algorithm. Based on results in this dissertation, various research topics on parallel IR can be studies. These research topics include parallel DIA algorithm, parallel index rebuilding, parallel ranking, and incremental update of partitioned inverted file.



References

Aalbersberg, I.J. & Sijstermans, F. (1990). InfoGuide: A full-text document retrieval system. In A.M. Tjoa & R. Wagner (Eds.), *Proceedings of the international conference of database and expert systems applications* (*DEXA'90*), (pp.12-21). Berlin: Springer-Verlag.

Anh, V.N. & Moffat, A. (1998). Compressed inverted files with reduced decoding overheads. In R. Wilkinson, B. Croft, and C.V. Rijsbergen (Eds.), *Proceedings of the 21st annual international ACM SIGIR conference on Research and Development in Information Retrieval*, (pp. 290-297), Melbourne. New York: ACM Press.

Anh, V.N. & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1), 151-166.

Bell, T.C., Moffat, A., Nevill-Manning, C.G., Witten, I.H., and Zobel, J. (1993). Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9), 508-531.

Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. (1999). Web caching and zipf-like distributions: evidence and implications. In *Proceedings of Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies* (*IEEE INFOCOM '99*), (pp. 126-134), New York, Mar. Los Alamitos, CA: IEEE Computer Society Press.

Brown, E.W., Callan, J.P., and Croft, W.B. (1994). Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th Very Large Data Base Conference* (*VLDB'94*) , (pp. 192-202).

Cheng, C.S., Shann, J.J.J., and Chung, C.P. (2005). Unique-order interpolative coding for fast querying and space-efficient indexing in information retrieval systems. To appear in *Information Processing and Management*.

Cheng, C.S., Shann, J.J.J., and Chung, C.P. (2004). A Unique-Order Interpolative Code for Fast Querying and Space-Efficient Indexing in Information Retrieval Systems. In P.K. Srimani et al. (Eds.), *Proceedings of ITCC 2004 International Conference on Information Technology: Coding and Communications Volume 2*, (pp. 229-235), Las Vegas, Nevada, Apr. Los Alamitos, CA: IEEE Computer Society Press.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2), 194-203.

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, 17(1), 49-74.

Fraenkel, A.S. & Klein, S.T. (1985). Novel Compression of sparse bit-string—Preliminary report. In A. Apostolico & Z. Galil (Eds.) *Combinatorial Algorithms on Words*: *Vol.* 12, NATO ASI Serials F. (pp. 169-183). Berlin: Springer-Verlag.

Frakes, W.B. & Baeza-Yates, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Upper Saddle River, NJ: Prentice Hall.

Gallager, R.G. & Van Voorhis, D.C. (1975). Optimal source codes for geometrically distributed alphabets. *IEEE Transactions on Information Theory*, IT-21(2), 228-230.

Gelbukh, A., Han, S.Y., and Sidorov, G. (2003). Compression of boolean inverted files by document ordering. In *Proceedings of 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering* (*IEEE NLPKE-2003*), (pp. 244-249), Beijing, China, Oct. Los Alamitos, CA: IEEE Computer Society Press.

Golomb, S.W. (1966). Run Length Encoding. *IEEE Transactions on Information Theory*, IT-12(3), 399-401.

Hawking, D. (1996). Document retrieval performance on parallel systems. In H.R. Arabnial, ed, *Proceedings of the 1996 International Conference on Parallel and Distributed Processing*

Techniques and Applications, Sunnyvale, (pp. 1354-1365), California, August. Athens: CSREA Press.

Hollaar, L.A. (1991). Special-purpose hardware for text searching: past experience, future potential. *Information Processing & Management*, 27 (4): 371-378.

Janson, B.J., Spink, A., Bateman, J., and Saracevic, T. (1998). Real life information retrieval: a study of user queries on the Web. *SIGIR Forum*, 32(1), 5-17.

Kobayashi, M. & Takeda, K. (2000). Information retrieval on the web. *ACM Computing Surveys*, 32(2), 144-173.

Lawrence, S. & Giles, C. (1999). Accessibility of information on the web. *Nature*, 400, 107-109.

Lovins, J.B. (1968). Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11, 22-31.

Ma, Y.C., Chen, T.F., and Chung, C.P. (2002). Posting file partitioning and parallel information retrieval. *Journal of Systems and Software*, 63(2), 113-127.

MacFarlane, A. (2000). *Distributed inverted files and performance: a study of parallelism and data distribution methods in IR* (Ph.D. thesis). London: City University.

Mcllroy, M.D. (1982). Development of a spelling list. *IEEE Transactions on Communications*, COM-30(1), 91-99.

Moffat, A. & Stuiver, L. (2000). Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1), 25-47.

Moffat, A. & Zobel, J. (1992). Parameterised compression for sparse bitmaps. In N. Belkin, P. Ingwersen, and A.M. Pejtersen (Eds.), *Proceedings of 15th annual international ACM-SIGIR Conference on Research and Development in Information Retrieval*, (pp. 274-285), Copenhagen, Jun. New York: ACM Press.

Moffat, A. & Zobel J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4), 349-379.

Moffat, A., Zobel, J., and Klein, S.T. (1995). Improved inverted file processing for large text databases. In R. Sacks-Davis & J. Zobel (Eds.), *Proceedings of 6th Australasian Database Conference*, (pp. 162-171), Adelaide, Australia, Jan.

Olken, F. & Rotem, D. (1986). Rearranging data to maximize the efficiency of compression. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, (pp. 78-90), Cambridge, Massachusetts, United States, Mar. New York: ACM Press.

Reddaway, S.F. (1991). High speed text retrieval from large databases on a massively parallel processor. *Information Processing & Management*, 27 (4): 311-316.

Ribeiro-Neto, B., Moura, E.S., Neubert, M.S., and Ziviani, N. (1999). Efficient distributed algorithms to build inverted files. In M. Hearst, F. Gey, and R. Tong (Eds.), *Proceedings for the 22nd International Conference on the Research and Development in Information Retrieval* (*SIGIR'99*), (pp. 105-112). New York: ACM Press.

Salton, G. (1989). *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Reading, Mass: Addison-Wesley.

Salton, G. & McGill, M.J. (1983). *Introduction to Modern Information Retrieval*. New York: McGraw-Hill.

Scholer, F., Williams, H.E., Yiannis, J., and Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S.H. Myaeng, and K. Järvelin (Eds.), *Proceedings of the 25th annual international ACM SIGIR conference on Research*

and Development in Information Retrieval, (pp. 222-229), Tampere, Finland. New York: ACM Press.

Shieh, W.Y., Chen, T.F., Shann, J.J., and Chung, C.P. (2003). Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1), 117-131.

Stanfill, C., Thau, R., and Waltz, D. (1989). A parallel Indexed algorithm for Information Retrieval. In N.J. Belkin & C.J. Van Rijsbergen (Eds.), *Proceedings of the 12th annual conference on research and development in Information Retrieval* (*SIGIR'89*), (pp. 88-97). New York:ACM Press.

Stanfill, C. & Thau, R. (1991). Information retrieval on the connection machine: 1 to 8192 Gigabytes. *Information Processing & Management*, 27 (4): 285-310.

Tenenbaum, A.M., Langsam, Y., and Augenstein, M.J. (1990). *Data structures using C*. Englewood CLiffs, N.J. 07632: Prentice-Hall.

Teuhola, J. (1978). A Compression method for clustered bit-vectors. *Information Processing Letters*, 7(6), 308-311.

Trotman, A. (2003). Compressing inverted files. *Information Retrieval*, 6(1), 5-19.

Turpin, A. (1998). *Efficient prefix coding* (Ph.D. thesis). Melbourne: University of Melbourne.

Turtle, H. & Flood, J. (1995). Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(6): 831-850.

Voorhees, E. & Harman, D. (1997). Overview of the sixth text retrieval conference (TREC-6). In E.M. Voorhees & D.K. Harman (Eds.), *Proceedings of the Sixth Text REtrieval Conference* (*TREC-6*), (pp. 1-24). Gaithersburg, MD: NIST.

Williams, H.E. & Zobel, J. (2002). Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1), 63-78.

Williams, H.E. & Zobel, J. (1999). Compressing integers for fast file access. The Computer Journal, 42(3), 193-201.

Witten, I.H., Moffat, A., and Bell, T.C. (1999). *Managing Gigabytes: Compressing and Indexing on Documents and Images, Second Edition*. San Francisco, CA: Morgan Kaufmann Publishers.

Wolfram, D. (1992). Applying informetric characteristics of databases to ir system file design, part i: informetric models. *Information Processing and Management*, 28(1), 121-133.

Xie, Y. & O'Hallaron, D. (2002). Locality in search engine queries and its implications for caching. In P. Kermani, F. Bauer, and P. Morreale (Eds.), *Proceedings of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies* (*INFOCOM'02*), (pp. 1238-1247), New York, Jun.

Zipf G. (1949). *Human Behavior and the Principle of Least Effort*. New York: Addison-Wesley.

Zobel, J. & Moffat, A. (1995). Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8), 891-903.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4), 453-490.