ELSEVIER

The Journal of Systems and Software 63 (2002) 113–127

The Journal of Systems and Software

www.elsevier.com/locate/jss

Posting file partitioning and parallel information retrieval

Yung-Cheng Ma [a,*], Tien-Fu Chen [b], Chung-Ping Chung [a]

[a] *Department of Computer Science and Information Engineering, National Chiao-Tung Univeristy, 1001 Ta Hsueh Road, Hsinchu 30050, Taiwan*

[b] *Department of Computer Science and Information Engineering, National Chung-Cheng University, Chayi 600, Taiwan*

Received 30 December 2000; received in revised form 5 May 2001; accepted 10 June 2001

Abstract

The rapid growth in Internet usages brings new challenges on designing a scalable information retrieval system. To reduce the response time of a query to a large database, we parallelize both CPU computation and disk access of Boolean query processing on a cluster of workstations. The key issue is to partition the inverted file such that, during parallel query processing, each workstation consults only its own locally resident data to complete its task. To achieve this goal, we treat the set of all postings referring to a document ID as an object to be allocated in the develop data placement problem. Following the partitioning by document ID principle, we develop posting file partitioning algorithms to transform a sequential information retrieval system to a parallel information retrieval system. The advantage is that a better speed-up can be achieved by deriving from the fast sequential approach – the compressed posting file. The partitioning schemes are designed to balance work-load of workstations in parallel query processing without increasing the average disk access time per posting. The experiment shows that almost linear speed-up can be achieved and the performance bottleneck in previous work, which parallelize only disk access, can be removed. This work shows that, by using parallel processing technique, it is feasible to build a scalable information retrieval system.

1. Introduction

The rapid growth in Internet usage brings wide variety of applications as well as new system design challenges on information retrieval systems. Information retrieval systems can be found in various Web applications for searching homepages, research papers, news articles, and product information. However, the problem of information explosion overwhelms the load of CPU and disk on an information retrieval server. The response time to serve a user query scales as the size of the document collection grows. Reducing the query response time is a key issue in designing a scalable information retrieval system.

We intend to reduce the query response time by means of parallel Boolean query processing on a network of workstations. Queries are processed on a cluster of workstations – each has its own CPU, memory, and disks – interconnected by a local area network. A user query consists of keyword terms and Boolean operators. A document either matches or mismatches a Boolean query in a binary fashion. A retrieval data structure stored on disks is consulted to search for those documents matched with the given query. The key research issue here is to partition and distribute the large retrieval data structure onto disks of multiple workstations such that the query processing time can be reduced.

Two well-known retrieval data structures are the signature file (Faloutsos, 1985) and the inverted file (Frakes and Baeza-Yates, 1992; Witten et al., 1999). We use the inverted file, which consists of an index file and a posting file, as the retrieval data structure. The posting file is compressed and stored on disks. The compression reduces not only the storage space required but also the disk access time to retrieve posting lists. To the best of our knowledge, the *d*-gap scheme (Witten et al., 1999) has the best compression efficiency. The effectiveness of the *d*-gap compression relies on the clustering of documents to reduce the gaps between document IDs (Moffat and Stuiver, 1996a). We will see that document clustering also plays an important role on posting file partitioning.

Parallel query processing has been an active research area. Some researchers (Stanfill and Kahle, 1986;

[*] Corresponding author. Tel.: +886-3-5712121; fax: +886-3-5724176. *E-mail address:* ycma@csie.nctu.edu.tw (Y.-C. Ma).

Cringean et al., 1990; Lee, 1995) have proposed parallel signature-file-based query processing. However, Stone (1987) and Salton and Buckley (1988) indicates that the parallel signature-file-based method may even be slower than sequential inverted-file-based method. On the other hand, other researchers (Stanfill, 1990; Mansand and Sanfill, 1993; Reddaway, 1991) parallelize inverted-file-based query processing at the cost of synchronization overheads in SIMD machines (Stanfill, 1990; Mansand and Sanfill, 1993) or requiring additional complexity by expanding a posting list to a binary bit-map (Reddaway, 1991). Recent research efforts (Jeong and Omiecinski, 1995; Riberio-Neto et al., 1998) parallelize the disk accesses to retrieve posting lists, however the CPU computation is still performed sequentially. The shortcoming in previous work is that processing posting list is inherently sequential and the performance bottleneck still exists. The goal of this work is to attack this problem.

The key to the success of parallel query processing is to partition the posting file such that, during parallel query processing, each workstation has to consult only its own local portion of the partitioned posting file. Jeong and Omiecinski (1995) states that the posting file can be partitioned either by term or by document ID. The straightforward approach is by-term partitioning, which takes a posting list as an object to be allocated. An alternative approach, partitioning by document ID, is to take the set of all postings referring to a document ID as an object for the allocation. Jeong and Omiecinski (1995) parallelizes disk accesses with these two approaches but the CPU computation is still performed sequentially. In this paper, we show that the by-document-ID partitioning approach parallelizes both CPU computation and disk accesses without inducing communication overhead of transfering posting lists between workstations.

Following the partitioning by document ID principle, we propose algorithms to transform a sequential retrieval system to a parallel information retrieval system. The algorithm receives a posting file for sequential query processing as input and produces a partitioned posting file for parallel query processing. We assume that the input posting file is compressed using the *d*-gap compression scheme, and documents are clustered to reduce gaps between document IDs. We also apply the *d*-gap compression on each portion of the partitioned posting file to reduce the disk access time in parallel query processing. To achieve better performance, posting file should be partitioned to balance workload of each workstation in parallel query processing without reducing the effectiveness of *d*-gap compression scheme. With the document clustering assumption and representing a document as a local document ID in the corresponding workstation, we will see that load balancing can be achieved without sacrificing the *d*-gap compression effectiveness. Experiment shows that almost linear speed-up on query processing performance can be achieved.

This paper is organized as follows. Section 2 describes the inverted file and the *d*-gap compression scheme. Section 3 derives a model for parallelizing query processing. Section 4 describes two partitioning schemes that can produce the partitioned posting file after reading the original posting file once. In Section 5, we propose the third partitioning scheme which achieves a higher query processing performance at the cost of reading the input posting file twice for producing the partitioned posting file. Section 6 presents the experimental results, and a conclusion is given in Section 7.

2. Inverted file

The data structure of an inverted file (Frakes and Baeza-Yates, 1992; Witten et al., 1999) is depicted in Fig. 1. An inverted file consists of an index file and a posting file. The index file is a set of records, each containing a keyword term t and a pointer to the posting list of term t. The posting file is a set of posting lists, each being a set of document IDs to indicate the presence of a term in a document. A Boolean query consists of keyword terms and Boolean operators. To process a query, the system searches the index file for the queried terms to retrieve posting lists of corresponding terms on disks. Set operations, such as intersection ($\cap$) and union ($\cup$), are then performed on the posting lists to obtain the answer list. In a large document collection, posting lists are usually compressed on disks and CPU computation to uncompress posting lists are hence required.

Psycho-linguist Zipf (1949) observed that the set of frequently used terms is small. According to Zipf's law (Zipf, 1949; Salton, 1989), 95% of words in 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 random access memory and the average time for index searching will not scale with the size of the document collection.

We focus our attention to the posting file side instead. The time complexity of posting lists processing, including disk accesses, uncompression, and document ID comparison, is $O(f_1 + f_2 + \cdots + f_k)$, where f_i is the length of the posting list of the ith queried term (Salton, 1989). Moreover, adding a document into the collection is to add one document ID to each posting list of the terms appearing in the document, and hence the length of a posting list increases with the size of the document collection. This implies that the time to process posting lists increases as the size of document collection grows. Efficient approaches to reduce space and time to store and operate on the posting file are the key issues in the study of a scalable information retrieval system.

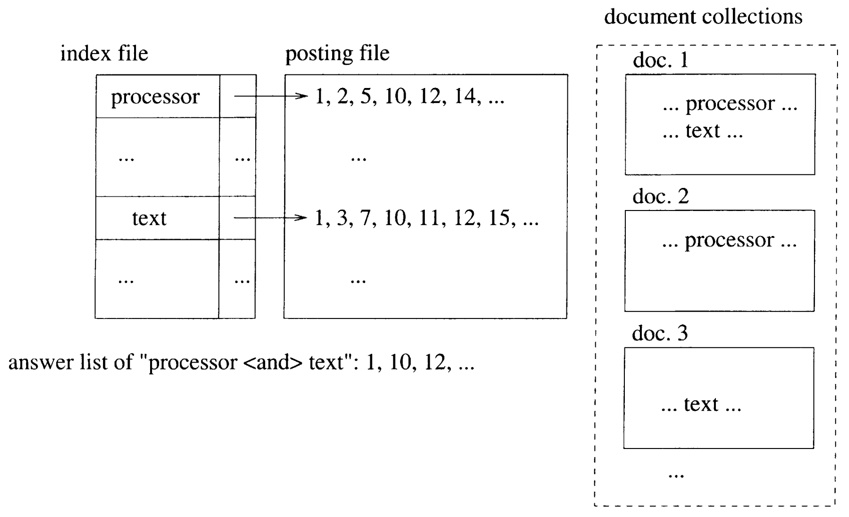


Fig. 1. Inverted file.

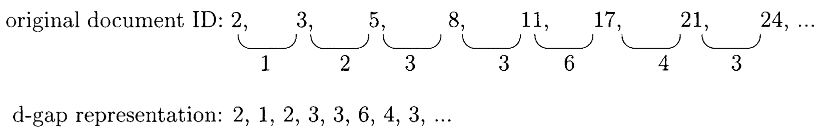

original document ID: 2, 3, 5, 8, 11, 17, 21, 24, ...
1 2 3 3 6 4 3
d-gap representation: 2, 1, 2, 3, 3, 6, 4, 3, ...

Fig. 2. *D*-gap compression scheme.

Witten et al. (1999) proposes *d*-gap compression scheme to reduce both space for storing posting lists and time for retrieving posting lists from disks. The key idea of *d*-gap compression scheme is depicted in Fig. 2. Assume that document IDs in a posting list are sorted in an increasing order. In the *d*-gap represented list, each document ID (expect for the first one) is represented as the difference of its ID to its predecessor's. The *d*-gap is then encoded using encoding schemes that can encodes a small number with fewer bits (Elias, 1975; Bently and Yao, 1976; Golumb, 1966). The effectiveness of *d*-gap compression scheme relies on the ordering of document IDs. Documents should be clustered according to the number of common terms shared, and clustered document IDs are assigned close to each other to reduce the gaps between document IDs in a posting list. Based on the *d*-gap scheme, document clustering to improve the effectiveness, is used to compress the posting file (Moffat and Stuiver, 1996a).

3. Parallelizing Boolean query processing

The key idea of parallelizing posting list processing is depicted in Fig. 3. We use the notation WS_k to denote workstation k. Each posting list is partitioned and each workstation stores a segment of the partitioned posting list. In this example, the segment of a posting list containing document IDs 0–9 is stored in WS_0. Similarly, the segment containing document IDs 11–19 is stored in WS_1, and the segment containing document ID 21–29 is stored in WS_2. During parallel query processing, each workstation performs set operations (∪, ∩, etc). on its own portion of partitioned posting lists independently. For instance, workstation WS_1 has to consult only its locally resident data to check whether document 15 matches a given query. This parallelizes the set operations without inducing communication overhead. In this section, we formalize the idea and show how a query is processed in a network of workstations.

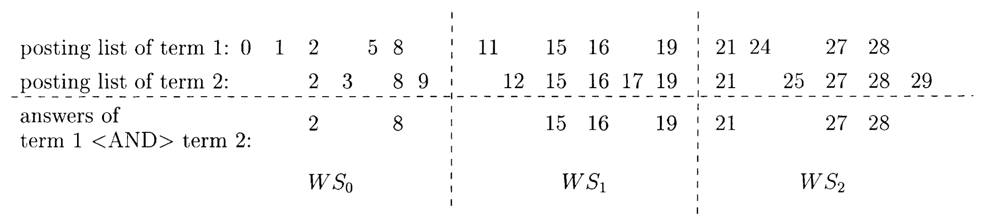

posting list of term 1: 0 1 2 5 8 | 11 15 16 19 | 21 24 27 28
posting list of term 2: 2 3 8 9 | 12 15 16 17 19 | 21 25 27 28 29
answers of term 1 <AND> term 2: 2 8 | 15 16 19 | 21 27 28
WS0
WS1
WS2

Fig. 3. Partitioning the posting file for parallel query processing.

3.1. Formalization of parallel posting list processing

The principle of parallel processing is, for any document d, all postings referring to document ID d should be placed in the same workstation. A partitioning scheme uses a mapping A to map each document ID d to a workstation WS_k. We use the notation $A(d) = k$ to denote that all postings referring to document ID d must be mapped to WS_k. Each workstation WS_k stores a *locally posting list* for each term. Let L_i be the posting list of the term i. The *local posting list* of the term i in workstation WS_k, denoted $L_i(WS_k)$, is the list of document IDs and L_i which are mapped to workstation WS_k. That is,

$$\begin{aligned} L_i(WS_k) &= \{\text{document ID } d \mid d \in L_i \text{ and } A(d) = k\} \\ &= L_i \cap D_k, \end{aligned} \tag{1}$$

where D_k is the set of document IDs covered by workstation WS_k.

$$D_k = \{\text{document ID } d \mid A(d) = k\}. \tag{2}$$

The *local posting file* of a workstation WS_k is the set of all local posting lists stored in WS_k.

To process posting lists is to perform set operations on posting lists. For a given query q, the *complete answer list*, denoted Ans_q, is the list of all document IDs matching the given query q. That is,

$$Ans_q = \{\text{document ID } d \mid \text{document } d \text{ matches query } q\}. \tag{3}$$

In parallel query processing, the task of workstation WS_k is to construct its own *partial answer list* $Ans_q(WS_k)$, which is the list of document IDs in D_k matching the given query q. That is,

$$\begin{aligned} Ans_q(WS_k) &= \{\text{document ID } d \mid A(d) \\ &\qquad = k \text{ and document } d \text{ matches } q\} \\ &= Ans_q \cap D_k. \end{aligned} \tag{4}$$

And the union of all partial answer lists of all workstations is hence the complete answer list. That is,

$$Ans_q = \bigcup_{WS_k} Ans_q(WS_k). \tag{5}$$

It is clear that each workstation WS_k can compute $Ans_q(WS_k)$ by consulting only its local posting file. A trivial method is to examine each document ID d mapped to WS_k. However, this trivial method introduces unnecessary complexity. We show that, to compute $Ans_q(WS_k)$, workstation WS_k only has to perform the ordinary sequential set operations on its local posting lists of queried terms, regardless of the irregularity of the mapping A. This means that any technique to improve sequential query processing performance (Witten et al., 1999; Moffat and Zobel, 1996b) can be used, and a good speed-up of parallel processing can be achieved by using one of the fast sequential methods.

Theorem 1 (Partial answer list construction). *The local partial answers, $Ans_q(WS_k)$, can be written as set operations on local posting lists of queried terms in WS_k.*

Proof 1. We prove this theorem by induction on the number of Boolean operators in the given query q.

The basis, that is when query q contains only one Boolean operator, is shown as follows. Query q is either "term i ⟨AND⟩ term j" or "term i ⟨OR⟩ term j". We consider the case when query q is "term i ⟨AND⟩ term j". The partial answer list to be produced by WS_k is

$$\begin{aligned} Ans_q(WS_k) &= (L_i \cap L_j) \cap D_k = (L_i \cap D_k) \cap (L_j \cap D_k) \\ &= L_i(WS_k) \cap L_j(WS_k). \end{aligned} \tag{6}$$

This rewrites $Ans_q(WS_k)$ with set operations on local posting list in WS_k. The case when q term i ⟨OR⟩ term j is similar and omitted.

The induction hypothesis is: suppose the theorem holds when the number of Boolean operators in q is less than n. We show that the theorem also holds when q contains n Boolean operators. The query q is either "(q_1) ⟨AND⟩ (q_2)" or "(q_1) ⟨OR⟩ (q_2)", where q_1 and q_2 are queries containing no more than $n-1$ Boolean operators. We first consider the case when q is (q_1) ⟨AND⟩ (q_2). The partial answer list to be produced by WS_k is

$$\begin{aligned} Ans_q(WS_k) &= (Ans_{q1} \cap Ans_{q2}) \cap D_k \\ &= (Ans_{q1} \cap D_k) \cap (Ans_{q2} \cap D_k) \\ &= Ans_{q1}(WS_k) \cap Ans_{q2}(WS_k). \end{aligned} \tag{7}$$

The induction hypothesis states that $Ans_{q1}(WS_k)$ and $Ans_{q2}(WS_k)$ can be written as set operations on local posting lists of queried terms in WS_k, and hence so is $Ans_q(WS_k)$. The case when q is (q_1) ⟨OR⟩ (q_2) is similar and omitted. This theorem is thus proved by induction. □

3.2. Guideline to load balanced partitioning

Besides communication overhead, another performance hurdle in parallel processing is load imbalancing. We analyze the time complexity of the query processing and derive the guideline for load balancing.

We formulate query processing time as follows. The time complexity of sequential query processing is $O(f_{t_1} + f_{t_2} + \cdots + f_{t_n})$, where f_{t_i} is the length of the posting list of the ith queried term t_i and n is the number of queried terms (Salton, 1989). We thus proportionate the processing time of sequential query processing to the total length of posting lists of queried terms. That is,

$$Time_{sequential} = \text{Constant} * (f_{t_1} + f_{t_2} + \cdots + f_{t_n}). \tag{8}$$

This modeling can be justified by the experiment data to be shown in Section 6. According to Theorem 1, each workstation performs ordinary sequential query processing on its local posting file independently as part of the parallel query processing. Processing time of a workstation WS_k is thus

$$Time(WS_k) = \text{Constant} * (f_{t_1}^{(k)} + f_{t_2}^{(k)} + \cdots + f_{t_n}^{(k)}), \tag{9}$$

where $f_{t_i}^{(k)}$ is the length of the local posting list of the ith queried term t_i in workstation WS_k. The processing time of parallel query processing is the time the last workstation finishes its job:

$$\begin{aligned} Time_{parallel} &= \max_{WS_k}\{Time(WS_k)\} \\ &= \text{Constant} * \max_{WS_k}\left\{\left(f_{t_1}^{(k)} + f_{t_2}^{(k)} + \cdots + f_{t_n}^{(k)}\right)\right\}. \end{aligned} \tag{10}$$

Ideally, processing time of parallel processing can be as little as the sequential processing time divided by the hardware parallelism. That is,

$$\begin{aligned} Time_{ideal} &= \frac{Time_{sequential}}{M} \\ &= \text{Constant} * \left(\frac{f_{t_1}}{M} + \frac{f_{t_2}}{M} + \cdots + \frac{f_{t_n}}{M}\right), \end{aligned} \tag{11}$$

where M is the number of workstations. The ideal case occurs when each posting list is equally split onto multiple workstations, that is, when $f_i^{(k)} = f_i/M$ for each term i and each workstation WS_k.

The worst parallel query processing time is also determined by uneven partitioning of posting file. For each term i, let α_i be the maximum skew with respect to the uniform partitioning for the posting list of term i. That is,

$$\alpha_i = \frac{\max_{WS_k}\{f_i^{(k)}\}}{(f_i/M)}. \tag{12}$$

Let α be the global maximum skew among all posting lists. That is,

$$\alpha_i = \max_{\text{term}i} \alpha_i. \tag{13}$$

The worst case parallel query processing time is no more than the ideal processing time multiplied by the skew. This is stated in the following theorem.

Theorem 2 (Worst case parallel query processing time). *For any given query, the time $Time_{parallel}$ to process the query in parallel is bounded by*

$$Time_{parallel} \leqslant \alpha * Time_{ideal}. \tag{14}$$

Proof 2. Eq. (14) is derived from estimating the bound on the processing time of all workstations. Eqs. (12) and (13) give the upper bound on the number of postings to be processed by any workstation. For any term i and any workstation WS_k, the length of the local posting list of term i in workstation WS_k is bounded by

$$f_i^{(k)} \leqslant \alpha_i * (f_i/M) \leqslant \alpha * (f_i/M). \tag{15}$$

The processing time of all workstations WS_k is thus bounded by

$$\begin{aligned} Time(WS_k) &\leqslant \text{Constant} * \alpha * \left(\frac{f_{t_1}}{M} + \frac{f_{t_2}}{M} + \cdots + \frac{f_{t_n}}{M}\right) \\ &= \alpha * Time_{ideal}. \end{aligned} \tag{16}$$

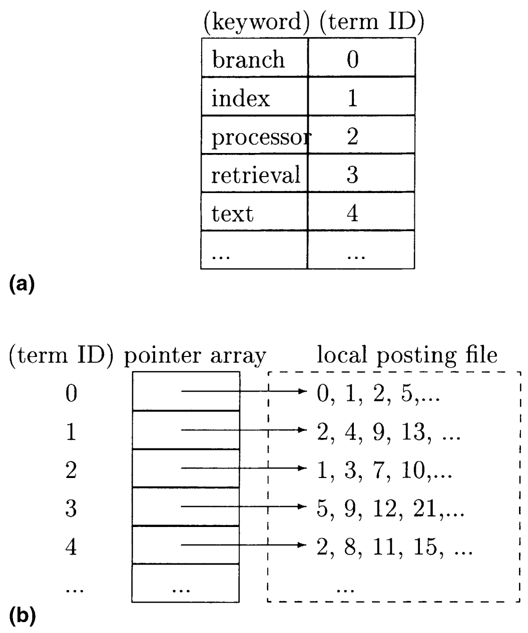
We thus obtain the upper bound on the parallel query processing time

$$Time_{parallel} = \max_{WS_k}\{Time(WS_k)\} \leqslant \alpha * Time_{ideal}. \quad \square \tag{17}$$

Above analysis indicates that, to improve parallel query processing performance, postings in a posting list should be distributed to multiple workstations as uniformly as possible. In later sections, we show how this guideline is applied to derive partitioning schemes.

3.3. Query processing on cluster of workstations

We describe the flow of processing a query on a network of workstations, starting from receiving a user query to the reply of the answer list. A specific workstation, called the gateway, is responsible for receiving user queries and performing the index file search. The gateway searches the index file for queried terms as shown in Fig. 4(a), and substitutes a term ID for each term in the query. Records of frequently used terms are



(keyword)	(term ID)
branch	0
index	1
processor	2
retrieval	3
text	4
...	...

Fig. 4. Partitioned inverted file: (a) index file in the gateway workstation; (b) pointer array and local posting file in a back-end workstation.

often stored in the random access memory such that the average index search time will not scale with the size of the document collection. The query is then broadcast to all back-end workstations for parallel posting list processing. Each workstation stores an index array of pointers to local posting lists on local disks, as shown in Fig. 4(b). While receiving a broadcast query, a workstation looks up the index array by term IDs to retrieve local posting lists and generates its own partial answer list. The partial answer list is buffered locally, and the number of matches found is sent back to the gateway.

After all the workstations are done, the remaining work is to reply answers to the user page by page. A page contains the number of matches to the query, and a few titles of matched documents that can be presented in one page. The number of matches is useful for a user to determine whether to browse further matches. When the number of matches is large, a user may decide to discard the query results and give a more specific query to reduce the number of matches. The gateway accumulates number of matches found by each back-end workstation to obtain total number of matches. The first page is then generated and delivered to the user. Parallelization of query processing reduces the response time to deliver the first page, in which the total number of matches must be contained. Remaining pages are generated and delivered upon user demands. To generate a page, the gateway polls some back-end workstation(s) to get matches just enough to fill a page. Since a user may not request all of the results to a query, the answers distributed on multiple workstations need not be collected at once.

Recent progress on parallel technology provides efficient ways to obtain ranking on matches distributed across multiple workstations. Let r be the number of matches to be presented in a page. Following the partition-by-document ID principle, each workstation can score and select the top r matches within its partial answers independently. The top r matches in the complete answer list can be obtained by parallel sort (Kumar, 1994) of all workstations' top r matches. With architectural support, Patterson's group shows that more than 1 G integers can be sorted in 2.41 s using 64 workstations (Arpaci-Dusseau et al., 1997; Arpaci-Dusseau et al., 1998). Since the number of answers r to be presented in a page is small and will not scale with the collection size, it is obvious that ranking does not create a performance bottleneck on query response time. The only challenge is to reduce the posting list processing time which we attack in this paper.

4. One-pass posting file partitioning algorithms

We now apply different partition policies to generate the partitioned posting file. The major concern in deriving partitioning schemes is load balancing in parallel query processing. Document ID assignment plays an important role on load balancing. We assume that the input posting file is d-gap compressed and documents are clustered to reduce gap between document IDs in a posting list. The straightforward scheme to be proposed first is the *consecutive scheme*. However, under the document clustering assumption, the consecutive scheme fails to follow the load balancing guideline described in Section 3.2. To overcome this drawback, we derive the second scheme – the *interleaving scheme*. These two schemes can result in the partitioned posting file by reading the input posting file only once. In the following section, we derive the third scheme – differential scheme – which achieves better query processing performance at the cost of reading the input posting file twice.

4.1. Consecutive partitioning scheme

A straightforward approach is to let each local posting list be a segment of the input posting list, as shown in Fig. 3. Each workstation contains a fixed range of consecutive documents IDs. Let D be the number of documents in the entire document collection and M be the number of workstations. The mapping A_{consec} is to map each document ID d to a workstation by

$$A_{consec}(d) = \left\lfloor \frac{d}{\lceil D/M \rceil} \right\rfloor. \tag{18}$$

In each workstation, a document can be represented by a local document ID, a d-gap compression scheme is applied on the local document IDs. We let the local document IDs in a workstation starts from zero for simplicity. The rule $LID_{consec}(d)$ to assign local document ID for each document d is

$$LID_{consec}(d) = d - A(d) * \lceil D/M \rceil. \tag{19}$$

For the example in Fig. 3, the local posting list $\{11, 15, 16, 19\}$ of term 1 in workstation WS_1 is represented as $\{1, 5, 6, 9\}$ using the local document ID representations (where $D = 30$ and $M = 3$). Use of local document IDs results in a small starting ID for each local posting list, and the size of the d-gap compressed local posting file is slightly smaller.

Fig. 5 shows the consecutive posting file partitioning scheme. The algorithm examines each posting in the input file, maps each document ID d to a workstation WS_k, and obtains a local document ID d' according to Eqs. (18) and (19) (cf. Steps (1) and (2) in Fig. 5). It then updates the local posting list in workstation by appending the local document ID (cf. Step (3) in Fig. 5). This algorithm reads the input posting file once, and the time complexity is O(f), where f is the total number of postings in the input posting file.

A serious drawback of the consecutive scheme is that it fails to take the load balancing into account (Documents are clustered to reduce gaps between document

```
Algorithm $Partition\_Consecutive(PF, LPF)$
 Input:

  • $PF$: the posting file for sequential query processing. $PF$ consists of a set of posting
    lists $L_i$ for each term $i$.

 Output:

  • $LPF = \{LPF_0, LPF_1, ..., LPF_{M-1}\}$: the set of local posting files $LPF_k$ for each
    workstation $WS_k$. Each $LPF_k$ consists of a set of local posting lists $L_i(WS_k)$ for
    each term $i$.

 Method:

   for each term $i$ do
       for each document ID $d \in L_i$ do
       1) $k \leftarrow \left\lfloor \frac{d}{\lceil D/M \rceil} \right\rfloor$
       2) $d' \leftarrow d - k * \lceil D/M \rceil$
       3) append $d'$ to $L_i(WS_k)$.
```

Fig. 5. Consecutive posting file partitioning algorithm.

IDs.) For the effectiveness of d-gap compression, documents are usually clustered such that most of the document IDs in a posting list are within a small range. When the posting list is partitioned by the consecutive scheme, most of the document IDs will be mapped to a small number of workstations, as depicted in Fig. 6, resulting in load imbalancing in parallel query processing. To overcome this drawback, we derive the second partitioning scheme, the *interleaving scheme*, to balance workload without sacrificing the effectiveness of d-gap compression.

4.2. Interleaving partitioning scheme

Fig. 7 shows how the interleaving partitioning scheme works. As shown in Fig. 7(a), each workstation is mapped with a set of interleaved document IDs. Let M be the number of workstations. The rule $A_{intlv}(d)$ is to map each document ID d to a workstation by

$$A_{intlv}(d) = d \mathrm{mod}\ M. \tag{20}$$

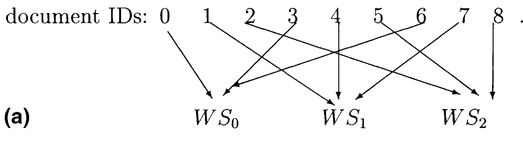
The workstation to which d is mapped to is (d mod M). With the interleaving scheme, postings in a posting list are supposed to be evenly distributed regardless of the document ID clustering.

However, the mapping rule $A_{intlv}(d)$ increases the gap between document IDs after partitioning. The gap document IDs in a local posting list is at least M. And d-gap compression does not work well on the local posting file

posting list:	2, 9,	11, 12, 15, 17, 18, 19	22, 25
	WS_0	WS_1	WS_2

Fig. 6. Imbalance due to consecutive partitioning.

(a)



posting list: 1, 2, 4, 6, 7, 10, 11, 12, 14, 15

represented using orignal document ID:	6, 12, 15	1, 4, 7, 10	2, 11, 14,
represented using local document ID:	2, 4, 5	0, 1, 2, 3	0, 3, 4
(b)	WS_0	WS_1	WS_2

Fig. 7. Interleaving partitioning scheme: (a) mapping document IDs to workstation IDs; (b) partitioning a posting list.

if documents are presented with the original document IDs. We notice that, for a workstation WS_k, a document ID d mapped to k can be written as

$$d = M * \lfloor d/M \rfloor + k. \tag{21}$$

To represent a document in a workstation, only the quotient of d/M is required. We thus use the quotient as the local document ID $LID_{intlv}(d)$ in a workstation,

$$LID_{intlv}(d) = \lfloor d/M \rfloor. \tag{22}$$

Fig. 8 shows the interleaving partitioning algorithm. The procedure reads in the whole posting file to examine each posting in the file. Each document ID in the input posting file is mapped to a workstation, and its local document ID within the workstation is calculated using Eqs. (20) and (22) (cf. Steps (1) and (2) in Fig. 8). The local document ID is then appended to the corresponding local posting list in the corresponding workstation (cf. Step (3) in Fig. 8). The time complexity is O(f), where f is the total number of postings in the input posting file.

Algorithm *Partition_Interleaving*(PF, LPF)

Input:

- PF: the posting file for sequential query processing. PF consists of a set of posting lists L_i for each term i.

Output:

- $LPF = \{LPF_0, LPF_1, ..., LPF_{M-1}\}$: the set of local posting files LPF_k for each workstation WS_k. LPF_k consists of a set of local posting lists $L_i(WS_k)$ for each term i.

Method:

```
for each term $i$ do
    for each document ID $d \in L_i$ do
    1) $k \leftarrow d \bmod M$
    2) $d' \leftarrow \lfloor d/M \rfloor$
    3) append $d'$ to $L_i(WS_k)$.
```

Fig. 8. Interleaving partitioning algorithm.

5. Differential partitioning model and heuristic

In this section, we propose the *differential partitioning scheme* to further improve the parallel query processing performance. In previous sections, we did not consider the difference in the popularity of a term to be queried. However, according to Zipf's law (Zipf, 1949; Salton, 1989), the distribution of term popularities is not uniform and thus has a significant influence on the system performance. The key idea of the differential partitioning scheme is to estimate the average query processing time according to term popularity, and derive partitioning to reduce the average query processing time. In contrast to the consecutive and interleaving scheme, the differential partitioning heuristic reads the original posting file twice to generate the partitioned posting file. Since the posting file partitioning is performed off-line, it is worthwhile to use a more complex partitioning for better query processing performance.

5.1. Optimization problem of differential partitioning

We use a probability model to formulate posting file partitioning as an optimization problem. Assuming that the probability of a term *i* appearing in a query is known to be p_i. A partitioning scheme is specified by a document ID-to-workstation mapping A. We derive an objective function $cost(A)$ to reflect the average parallel query processing time according to term popularity p_i. The optimization problem is to find a mapping A that minimizes $cost(A)$.

We first formulate the average sequential query processing time. Let X_i be a random Boolean variable representing whether term i appears in a query: $X_i = 1$ if term i appears in a query and $X_i = 0$ otherwise. The number of postings to be to be processed for a query is

$$\sum_{\text{term}i} X_i * f_i, \tag{23}$$

where f_i is the length of the posting list of term *i*. According to Eq. (8), the sequential query processing time is

$$Time_{sequential} = \text{Constant} * \sum_{\text{term}i} X_i * f_i. \tag{24}$$

The average sequential query processing time is the expected value of $Time_{sequential}$.

$$\begin{aligned} AveTime_{sequential} &= E[Time_{sequential}] \\ &= \text{Constant} * \sum_{\text{term}i} p_i * f_i. \end{aligned} \tag{25}$$

Eq. (25) is obtained by accumulating weight p_i whenever a document ID d is found in a posting list of term *i*. Each document d contributes a weight $w(d)$ to Eq. (25)

$$w(d) = \sum_{\text{term}i:d\in L_i} p_i. \tag{26}$$

Eq. (25) can also be calculated by summing the weight of each document

$$AveTime_{sequential} = \text{Constant} * \sum_{\text{document } d} w(d). \tag{27}$$

Average parallel query processing time can thus be derived. The time for workstation WS_k to compute its partial answer list is

$$Time(WS_k) = \text{Constant} * \sum_{\text{document } d:A(d)=k} w(d). \tag{28}$$

We thus define the objective function $cost(A)$ to reflect the average parallel query processing time according to Eq. (10)

$$cost(A) = \max_{WS_k} \left\{ \sum_{\text{document } d:A(d)=k} w(d) \right\}. \qquad (29)$$

We ignore the constant it will not affect the selection of mapping A.

5.2. NP-completeness of differential partitioning problem

We show that the differential partitioning problem is NP-complete by observing that it is identical to the multiprocessor scheduling problem defined (Garey, 1979). Given a set of independent tasks $T = \{t_0, t_1, \ldots, t_{N-1}\}$ and a set of processors $P = \{p_0, p_1, \ldots, p_{M-1}\}$. Each task $t_i \in T$ is associated a weight $e(t_i)$ to represent the execution time of the task. The multiprocessor scheduling problem (Garey, 1979) is to find a mapping A that maps each task $t_i \in T$ to a processor $p_k \in P$ to minimize $cost(A)$ which reflects the parallel processing time

$$cost(A) = \max_{p_k \in P} \left\{ \sum_{t_i:A(t_i)=p_k} e(t_i) \right\}. \qquad (30)$$

Taking a document as a task and the weight of a document as the execution time of the task, we find that the differential partitioning problem is theoretically the same as the multiprocessor scheduling problem. Garey (1979) has shown that the multiprocessor scheduling problem is NP-complete, and hence differential partitioning is also NP-complete. Optimal solution can be found using the branch-and-bound method with dominance relation to reduce the searching space (Ma and Chung, to appear). In this paper, we propose a heuristic for the optimization problem.

5.3. Differential partitioning heuristic

We use the bitmaps depicted in Fig. 9 to explain the proposed heuristic. The global bitmap M represents the input posting file, in which a row corresponds to a term and a column corresponds to a document. The entry at row i and column d is a 1 if term i appears in document d. A posting list is to store the positions (column numbers) of 1s in a row of the bitmap. The weight of a column is the weight of the corresponding document (Eq. (26)). To partition the posting file is to decompose the bitmap M into individual columns, and each workstation WS_k is allocated a set of columns to form its own local bitmap LM_k. We take the column number in a local bitmap LM_k as the local document ID of the corresponding document in the workstation WS_k.

Fig. 9 depicts how the heuristic partitions the bitmap. The document clustering assumption on the input posting file means that 1s on a row are distributed in a small range. To prevent postings in a posting list from being distributed only on a small number of workstations, we reorder the columns in the bitmap M to form a reordered bitmap RM. Image partitioning the bitmap using interleaving scheme and reassembling the partitions together. We formulate the rule to map each column d in M to a column $\hat{d}$ in RM

$$\hat{d} = \lceil D/M \rceil * (d \bmod M) + \lfloor d/M \rfloor. \qquad (31)$$

We then slice the reordered bitmap RM vertically such that each workstation is allocated a set of consecutive columns in RM, and sum of column weights in a local bitmap is approximately the balanced weight

$$Balanced\_Weight = \frac{Total\_Weight}{M}, \qquad (32)$$

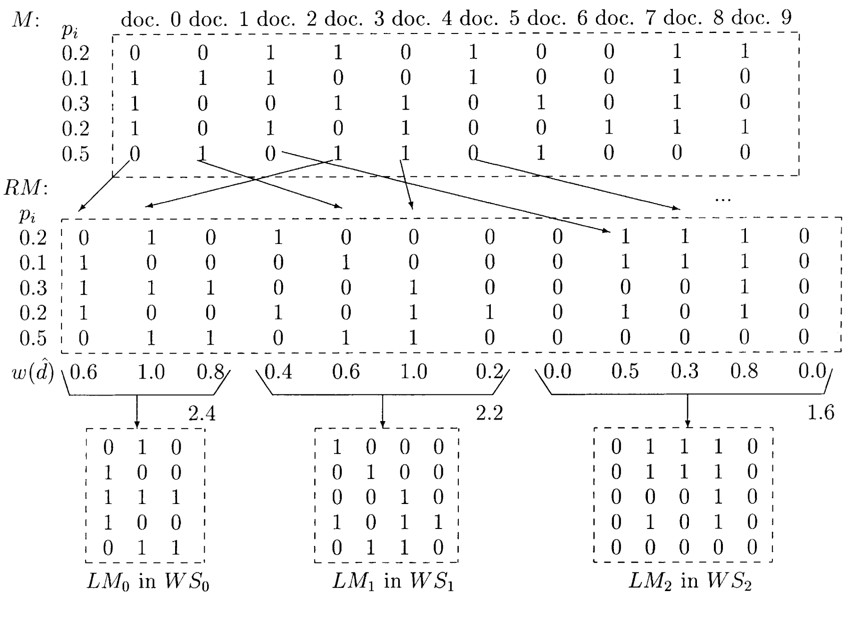


Fig. 9. Partitioning bitmap using differential partitioning scheme.

```
Algorithm $Partition\_Differential(PF, P, LPF)$
  Input:
  • $PF$: the posting file for sequential query processing. $PF$ consists of a set of posting
    lists $L_i$ for ech term $i$.
  • $P = \{p_0, p_1, ..., p_{n-1}\}$: $p_i$ is the probability that the term $i$ appears in a query for
    each term $i$.
  Output:
  • $LPF = \{LPF_0, LPF_1, ..., LPF_{M-1}\}$: the set of local posting files $LPF_k$ for each
    workstation $WS_k$. $LPF_k$ consists of a set of local posting lists $L_i(WS_k)$ for each
    term $i$.
  Method:
  1) perform $FindMapping(PF, P, Map, FirstCol)$
  2) perform $ProduceLPF(PF, Map, FirstCol, LPF)$
```

Fig. 10. Differential partitioning algorithm.

```
Algorithm $FindMapping(PF, P, Map, FirstCol)$
  Input:
  • $PF$: the posting file for sequential query processing. $PF$ consists of a set of posting
    lists $L_i$ for each term $i$.
  • $P = \{p_0, p_1, ..., p_{n-1}\}$: probability $p_i$ of term $i$ appearing in a query.
  Output:
  • $Map$: the mapping that maps each column $\hat{d}$ in $RM$ to a workstation $WS_k$.
  • $FirstCol[k]$ for each workstation $WS_k$: the first column in $RM$ that is mapped to
    $WS_k$
  Method:
  1) /* initialize */
    1.1) for $\hat{d} \leftarrow 0$ to $\lceil D/M \rceil * M - 1$ do $w(\hat{d}) \leftarrow 0$
    1.2) $Total\_Weight \leftarrow 0$
    1.3) $k \leftarrow 0$
    1.4) $FirstCol[k] \leftarrow 0$
    1.5) $Acc \leftarrow 0$
  2) for each posting list $L_i$ in $PF$ do /* calculate $w(\hat{d})$ and $Total\_Weight$ */
    2.1) for each document ID $d \in L_i$ do
      2.1.1) $\hat{d} \leftarrow \lceil D/M \rceil * (d \bmod M) + \lfloor d/M \rfloor$
      2.1.2) $w(\hat{d}) \leftarrow w(\hat{d}) + p_i$
      2.1.3) $Total\_Weight \leftarrow Total\_Weight + p_i$
  3) $Balanced\_Weight \leftarrow Total\_Weight/M$
  4) for $\hat{d} \leftarrow 0$ to $\lceil D/M \rceil * M - 1$ do
    4.1) $Map[\hat{d}] \leftarrow k$
    4.2) $Acc \leftarrow Acc + w(\hat{d})$
    4.3) if $Acc \geq Balanced\_Weight$ then
      4.3.1) $k \leftarrow k + 1$
      4.3.2) $Acc \leftarrow 0$
      4.3.3) $FirstCol[k] \leftarrow \hat{d} + 1$
```

Fig. 11. Mapping document IDs to workstations in differential partitioning scheme.

where M is the number of workstations and

$$Total\_Weight = \sum_{\text{document } d} w(d). \tag{33}$$

Fig. 10 depicts the two-pass list-based algorithm to realize the scheme. In the first pass, algorithm *FindMapping* (see Fig. 11) maps each document ID to a

```
Algorithm ProduceLPF(PF, Map, FirstCol, LPF)
  Input:
    • $PF$: posting file for sequential query processing. $PF$ consists of a set of posting
      lists $L_i$ for each term $i$.
    • $Map$: the mapping that maps each col. $\hat{d}$ in $RM$ to a workstation.
    • $FirstCol[k]$ for each workstation $WS_k$: the first column in $RM$ that is mapped to
      $WS_k$.
  Output:
    • $LPF = \{LPF_0, LPF_1, ...LPF_{M-1}\}$: a set of local posting files $LPF_k$ for each
      workstation $WS_k$. Each $LPF_k$ consists of a set of local posting lists $L_i(WS_k)$ for
      each term $i$.
  Method:
    for each posting list $L_i$ in $PF$ do
        for each document ID $d$ in $L_i$ do
        1) $\hat{d} \leftarrow \lceil D/M \rceil * (d \bmod M) + \lfloor d/M \rfloor$
        2) $k \leftarrow Map[\hat{d}]$
        3) $d' \leftarrow \hat{d} - FirstCol[k]$
        4) append local document ID $d'$ to $L_i(WS_k)$.
```

Fig. 12. Producing partitioned posting file in differential partitioning scheme.

workstation plus the local document ID within the workstation. *Map* and *FirstCol* are used for bookkeeping of the mapping. For each column $\hat{d}$ in *RM*, $Map[\hat{d}] = k$ means that column $\hat{d}$ is mapped to workstation WS_k. For each workstation WS_k, we use $FirstCol[k]$ to record the first column in *RM* that is mapped to the workstation WS_k. The algorithm *FindMapping* reads each posting in the posting file to create the two arrays. In the second pass, algorithm *ProduceLPF* reads the input posting file again and generates the partitioned posting file according to *Map* and *FirstCol* created in the first pass (see Fig. 12).

The time complexity is analyzed as follows: Let D be the number of documents in the collection and f be the total number of postings in the input posting file. The time complexity of algorithm *FindMapping*, which reads the whole input posting file and examines each column $\hat{d}$ in *RM*, is $O(f + D)$. Algorithm *ProduceLPF* reads the whole input posting file and the time complexity is $O(f)$. Hence the time complexity of algorithm *Partial_Differential* is $O(f + D)$. The partitioning algorithm reads the input posting file twice from the disk, and hence the time to generate the partitioned posting file is twice more than the time required by the one-pass algorithms.

6. Experiments

We implement an experimental information retrieval system to evaluate the proposed partitioning schemes. We measure the sequential query processing time for each workstation and the parallel query processing time is calculated according to Eq. (10).

6.1. Document collection and query generation

We evaluate the three proposed partitioning schemes using a set of 7794 documents and 486,673 randomly generated queries. The document collection is a set of ACM transaction/conference papers. We follow (Moffat and Zobel, 1996b) to generate test queries and feed them into the experimental system.

Statistics on the set of documents are depicted in Table 1. The number of documents, terms, and postings are obtained by scanning the inverted file. Average length of posting list is:

$$\text{Average length of a posting list} = \frac{\text{number of posting}}{\text{number of terms}}. \tag{34}$$

We generate an inverted file for the document collection as input to the proposed partitioning algorithm. The inverted file is compressed using the d-gap scheme and documents are clustered to reduce gaps between document IDs (Hsieh et al., submitted). Fig. 13 shows the distribution of gaps in the inverted file to demonstrate the effect of document clustering. $Pr\{x\}$ is used to denote the probability that a gap chosen from the d-gap represented posting file is x. Figs. 13(a) and (b) depicts the distribution $Pr\{x\}$ for x between 1–50 and between 5–50,

Table 1
Statistics of the test document collection

Number of documents	7794
Number of terms	13,383
Number of postings	5,877,578
Average length of a posting list	446

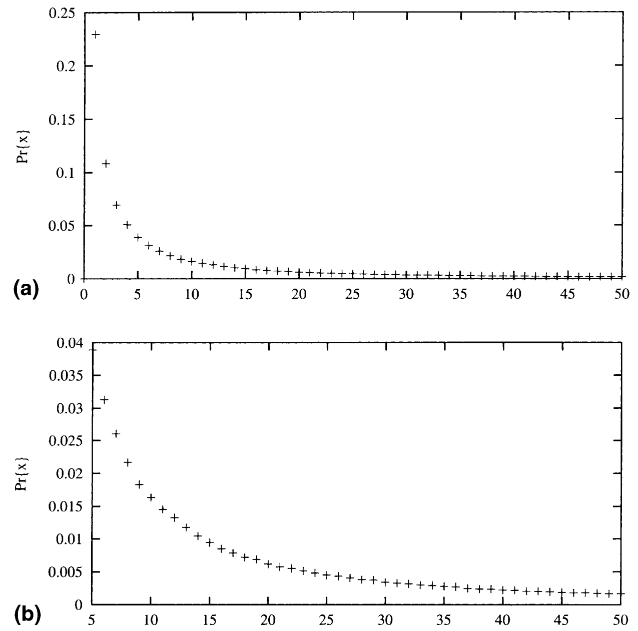


Fig. 13. Probability distribution d-gaps: (a) probability distribution for encoded symbols 1–50; (b) probability distribution for encoded symbols 5–50.

respectively. Among all postings in the d-gap represented posting file, 79.6% of the gaps fall between 1–50, and 59.36% fall between 1 and 10. $Pr\{x\}$ is a monotonically decreasing function. This shows that most of the gaps between document IDs are small and most of the document IDs in a posting list are within a small range.

We follow (Moffat and Zobel, 1996b) to randomly generate queries for performance evaluation. 100 documents are randomly selected from the document collection. A query is formed by selecting words from a word list of a specific document and inserting Boolean operators into the query word list. To form the document word list, words in the document are case folded and stop words (such as "the", "this") are eliminated. The query word list is a sub-list chosen from the document word list. We then randomly insert the Boolean OR operator into the query word list. Remaining positions between words in query word list are then inserted the Boolean AND operators and the query is in sum-of-products form. For example, let "invert file document collection built" be query word list. If an OR operator is inserted between words "collection" and "built", a query "(inverted ⟨AND⟩ file ⟨AND⟩ document ⟨AND⟩ collection) ⟨OR⟩ built" is formed. For each generated query, there exists at least one document collection matching the query.

6.2. Sequential query processing time

We justify the sequential query processing time Eq. (8). We implement the sequential query processing algorithm proposed by Witten et al. (1999), in which an OR operation is processed using list merging and an AND operation is processed by searching document IDs in a posting list using binary search. The posting file is d-gap compressed and γ-code encoded (Elias, 1975; Bently and Yao, 1976). The processing time measured includes (1) disk access time to load posting lists, (2) CPU time to uncompress posting lists, and (3) CPU time to compare document IDs in posting lists.

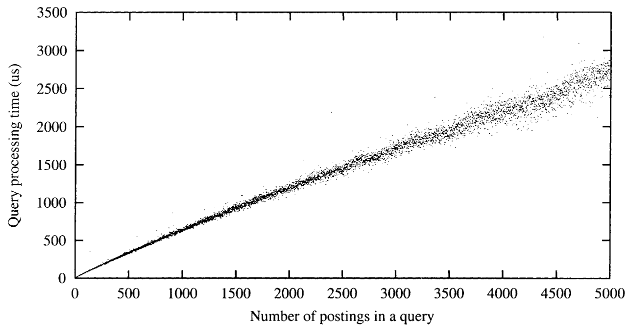


Fig. 14. Sequential query processing time as a function of number of postings.

The result is depicted in Fig. 14. The x-axis is the number of postings in a query and the y-axis is the time (in μs) to process a a query. We make an (x, y) dot if y μs is spent processing a query of x postings. In this figure, the dots spread alone a line. It indicates that the query processing time is linearly proportional to the number of postings to be processed.

6.3. Performance of parallel query processing

We now show the effectiveness of the proposed partitioning schemes. The primary difference of our schemes from the previous work on designing a scalable inverted

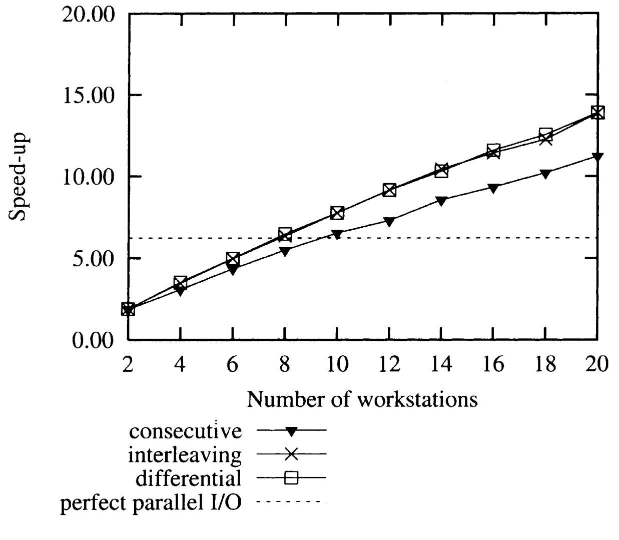


Fig. 15. Speed-up of parallel query processing.

file (Jeong and Omiecinski, 1995; Riberio-Neto et al., 1998) is that, instead of parallelizing only disk accesses, we parallelize CPU computation as well as disk accesses. Performance upper bound of parallelizing only disk accesses can be estimated by Amdahl's law (Hennesey and Patterson, 1996) as follows:

$$\text{Speed-up} \leqslant \frac{1}{1 - \text{percentage of disk access time in sequential processing time}}. \qquad (35)$$

Our experiment shows that 84.02% of the sequential query processing time is spent for disk accesses. Hence parallelizing only disk accesses cannot achieve a speed-up of more than 6.25. In the following section we show that our work can break this upper bound by parallelizing CPU and I/O systems.

Fig. 15 depicts the performance of parallel query processing using the proposed partitioning schemes. The metric is the speed-up to sequential query processing

$$\text{Speed-up} = \frac{Time_{sequential}}{Time_{parallel}}. \qquad (36)$$

We evaluate the speed-up for 2–20 workstations. This figure shows that almost linear speed-up can be achieved. As expected, both differential partitioning scheme and interleaving scheme outperform the consecutive one. Interleaving and differential partitioning schemes have

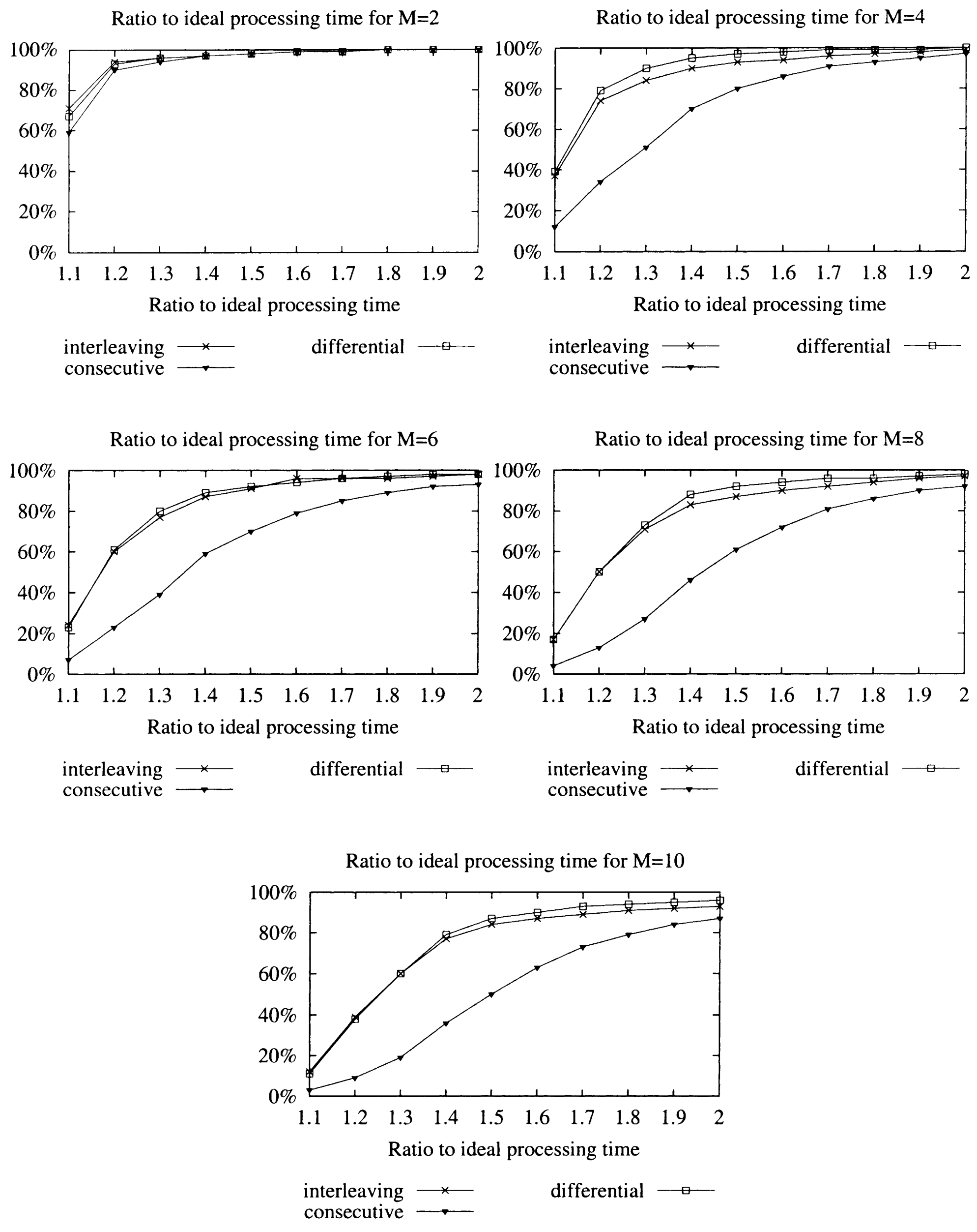


Fig. 16. Ratio to ideal speed-up within 10 workstations.

approximately the same performance. The dashed line indicates the performance upper-bound of parallelizing only disk I/O. Each of the proposed schemes distinctly breaks the performance upper bound that is determined by parallelizing only disk I/O.

We also study the variance on the performance for individual queries. Fig. 16 depicts accumulated curves on the ratio to ideal processing time for number of work stations $M = 2$ to 10. The x-axis is the ratio to ideal processing time, *RI*, defined as follows:

$$RI = \frac{Time_{parallel}}{(Time_{sequential}/M)}. \tag{37}$$

The y-axis is the percentage of all tested queries. We mark an (x, y) dot if y percent of all tested queries have a ratio to ideal processing time *RI* less than x. Where the curve reaches the top of the chart indicates the worst case performance in all tested queries. For interleaving and differential partitioning scheme, almost all queries can be processed within two times the ideal execution item. And the differential partitioning scheme has a slightly better performance than the interleaving scheme.

6.4. Compression efficiency on the partitioned posting file

We estimate the storage requirement of the partitioned posting files and compare it to the input posting file size. The metric is the average number of bits per posting (*BPP*), defined as follows:

$$BPP = \sum_{\text{symbol}\, x} Pr(x) * I(x), \tag{38}$$

where $Pr(x)$ is defined in Section 6.1 and $I(x)$ is the number of bits to encode the symbol x. The space to store the posting file is *BPP* times the number of postings. The time to retrieve a (local) posting list from the disk is also proportional to *BPP*.

Average bits per posting of the original and partitioned posting file are shown in Table 2. Various encoding schemes are used in the evaluation (Elias, 1975; Bently and Yao, 1976; Golumb, 1966). The results show that the space to store the partitioned posting file is slightly smaller than the input posting file. The reason are: (1) the gap between local document IDs are approximately equal to the gap between global document IDs, and (2) the partitioning results in a smaller starting (local) document ID in the local posting list. This shows that partitioning the posting file does not reduce the effectiveness of d-gap compression.

7. Conclusion

This paper demonstrates that information retrieval for large scale textual database systems is suitable for parallel processing by establishing the posting file partitioning model. We analyze the parallelism within set operations and point out that the posting file should be partitioned by document IDs. Posting file partitioning algorithms are proposed to transform a sequential information retrieval system, which uses a d-gap compressed inverted file, to a parallel information retrieval system. Experiments show that almost ideal speed-up on query processing can be obtained without sacrificing the effectiveness of d-gap compression scheme. The performance bottleneck in the previous work (Jeong and Omiecinski, 1995; Riberio-Neto et al., 1998) is relieved. This work shows a feasible way to build a scalable information retrieval system.

Table 2
Space requirements for the original and partitioned posting files

(a) *Average bits per posting of the posting file for sequential processing*

δ-Code encoding	7.59
γ-Code encoding	7.07
Golumb-code encoding	10.32

(b) *Average bits per posting in the partitioned posting file*

	2	4	6	8	10	12	14	16	18	20
δ-Code encoding										
Consecutive scheme	7.47	7.39	7.31	7.26	7.23	7.19	7.13	7.12	7.09	7.04
Interleaving scheme	7.61	7.60	7.58	7.56	7.53	7.49	7.47	7.45	7.43	7.40
Differential scheme	7.61	7.60	7.58	7.56	7.53	7.50	7.48	7.46	7.44	7.38
γ-Code encoding										
Consecutive scheme	6.86	6.71	6.58	6.50	6.46	6.40	6.31	6.29	6.25	6.18
Interleaving scheme	7.08	7.03	6.97	6.93	6.89	6.82	6.79	6.75	6.73	6.69
Differential scheme	7.07	7.03	6.97	6.93	6.89	6.83	6.80	6.77	6.74	6.67
Golumb-code encoding										
Consecutive scheme	9.29	9.27	8.23	8.22	8.21	8.21	7.21	7.19	7.19	7.19
Interleaving scheme	9.30	9.27	8.24	8.24	8.23	8.23	8.20	8.19	7.22	7.21
Differential scheme	9.29	9.26	8.24	8.23	8.21	8.20	8.21	8.19	7.21	7.17

References

Arpaci-Dusseau, A., Arpaci-Dusseau, R., Culler, D.E., Hellerstein, J.M., Patterson, D., 1997. High performance sorting on networks of workstations. In: Proceedings of 1997 ACM SIGMOD Conference.

Arpaci-Dusseau, A., Arpaci-Dusseau, R., Culler, D.E., Hellerstein, J.M., Patterson, D., 1998. Searching for the sorting record: experiences in tuning now-sort. In: Proceedings of 1998 Symposium on Parallel and Distributed Tools.

Bently, J., Yao, A.C.C., 1976. An almost optimal algorithm for unbounded searching. Information Processing Letters 5 (3), 82–87.

Cringean, J.K., England, R., Manson, G.A., Willett, P., 1990. Parallel text searching in serial files using a processor farm. In: Proceedings of the 13th ACM SIGIR Conference on Research and Development in Information Retriveal, pp. 429–445.

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.

Frakes, W.B., Baeza-Yates, R., 1992. Information Retrieval: Data Structures and Algorithms.

Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness.

Golumb, S.W., 1966. Run length encodings. IEEE Transaction on Information Theory IT-12 (3), 399–401.

Hennesey, J., Patterson, D.A., 1996. Computer Architecture: A Qualitative Approach, second ed.

Hsieh, W.Y., Dai, S.W., Chen, T.F., Chung, C.P., Shann, J.J., submitted. Document identification reassignment for inverted file compression. Journal of Systems and Software.

Jeong, B.S., Omiecinski, E., 1995. Inverted file partitioning schemes in multiple disk systems. IEEE Transactions on Parallel and Distributed Systems 6 (2), 142–153.

Kumar, V.P., 1994. Introduction of Parallel Computing: Design and Analysis of Algorithms.

Lee, D.K., 1995. Massive parallelism on the hybrid text-retrieval machine. Information Processing and Management 31 (6), 815–830.

Ma, Y.C., Chung, C.P., to appear. A dominance relation enhanced branch-and-bound task allocation. Journal of Systems and Software.

Mansand, B., Sanfill, C., 1993. A information retrieval test-bed on the cm-5. In: Proceedings of the Second Text Retrieval Conference, pp. 117–122.

Moffat, A., Stuiver, L., 1996a. Exploiting clustering in inverted file compression. In: Proceedings of 1996 Data Compression Conference, pp. 82–91.

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

Reddaway, S.F., 1991. High speed text retrieval from large database on a massively parallel processor. Information Processing and Management 27 (4), 311–316.

Riberio-Neto, B.A., Kitajima, J.P., Navarro, G., 1998. Parallel generation of inverted files for distributed text collections. In: Proceedings of the 18th International Conference of the Chilean Society of Computer Science, pp. 149–157.

Salton, G., Buckley, C., 1988. Parallel text search methods. Communications of the ACM 31 (2), 202–215.

Salton, G., 1989. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.

Stanfill, C., Kahle, B., 1986. Parallel free-text search on the connection machine system. Communications of the ACM 29 (12), 1229–1239.

Stanfill, C., 1990. Partitioned posting files: a parallel inverted file structure for information retrieval. In: Proceedings of the 13th International Conference on Research and Development in Information Retrieval, pp. 413–428.

Stone, H.S., 1987. Parallel querying of large database: a case study. IEEE Computer, 11–21.

Witten, H., Moffat, A., Bell, T.C., 1999. Managing Gigabytes: Compressing and Indexing on Documents and Images.

Zipf, G., 1949. Human Behavior and the Principle of Least Effort.

Yung-Cheng Ma received the B.S. degree in computer science and information engineering from the National Chiao-Tung University, Hsinchu, Taiwan, Republic of China in 1994. Currently he is pursuing the Ph.D. degree in computer science and information engineering at the National Chiao-Tung University, Hsinchu, Taiwan, Republic of China. His research interests include computer architecture, parallel and distributed systems, and information retrieval.

Tien-Fu Chen received the B.S. degree in Computer Science from National Taiwan University in 1983. After completed his millitary services, he joined Wang Computer Ltd., Taiwan as a software engineer for three years. From 1988 to 1993 he attended the University of Washington, receiving the M.S degree and Ph.D. degrees in Computer Science and Engineering in 1991 and 1993 respectively. He is currently an Associate professor in the Department of Computer Science and Information Engineering at the National Chung Cheng University, Chiayi, Taiwan. In recent years, he has published several widely-cited papers on dynamic hardware prefetching algorithms and designs. His current research interests are computer architectures, distributed operating systems, parallel and distributed systems, and performance evaluation.

Chung-Ping Chung received the B.E. degree from the National Cheng-Kung University, Hsinchu, Taiwan, Republic of China in 1976, and M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in electrical engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao-Tung University, Hsinchu, Taiwan, Republic of China, where he is a professor. From 1991 to 1992, he was a visiting associate professor of computer science at the Michigan State University. From 1998, he jointed the Computer and Communications Laboratories, Industrial Technology Research Institute, R.O.C. as the Director of the Advanced Technology Center, and then the Consultant to the General Director. He returned to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.