The Sixth National Computer Science and Engineering Conference, October 2002, Pattaya, Thailand.

Building Inverted Files Through Efficient Dynamic Hashing

Chuleerat Jaruskulchai and Canasai Kruengkrai
Intelligent Information Retrieval and Database Laboratory
Department of Computer Science, Faculty of Science
Kasetsart University, Bangkok, Thailand
Email: fscichj@ku.ac.th, canasai@hotmail.com

Abstract

We consider the problem of using an inversion algorithm to build inverted files from large document collections. In particular, we focus on applying a hash table to represent the dynamic dictionary data structure. The dictionary is used to store all distinct terms in the document collection during indexing process. It is the challenge to efficiently construct the hash table, since we do not know in advance the table size. In this paper, we introduce an efficient algorithm for building an inverted file based on the combination of the sort-based inversion algorithm and the cuckoo hashing. Our approach is to exploit the cuckoo hashing scheme that guarantees linear space used and worst case constant look up time. The experimental results show that our algorithm performs well on two large datasets.

Keywords: Information Retrieval, Inverted Files, Cuckoo Hashing, Dynamic Hashing.

1. Introduction

In Information Retrieval (IR), the first step in developing retrieval systems for text document collections is to consider the access methods. *Inverted files* or *inverted indexes* [1] have shown their advantages in comparison with other access methods. It is commonly used in commercial search engines. Building an inverted file can be done by using simple algorithms. However, this process is costly when applies to large document collections. The space and time optimization issues remain important. Therefore it is necessary to realize the efficient algorithms for building inverted files from such collections.

Many algorithms have been proposed to build an inverted file. With the exponential growth of text data on the Internet, a simple algorithm is now inadequate in that it requires too much memory or time. The *sort-based inversion algorithm* is one of efficient inversion algorithms, which requires reasonable time, memory, and disk space [10]. During indexing process, the algorithm needs to store a dictionary and some buffer in main memory. The dictionary stores a set of all distinct terms in the collection of documents and their occurrences, which almost dominates the main memory consumption. Thus the dictionary data structure is one of the major keys of this algorithm. We have to consider two important parameters of a dictionary including the amount of space occupied by the dictionary and the search time. Hashing technique is an interesting choice, since its data structure can ideally fit into main memory and provide efficient search operation.

Unfortunately, basic hashing schemes such as separate changing or liner probing do not always guarantee the parameters as mentioned earlier. Recently, researchers have focused on algorithms for obtaining $O(1)$ worst case lookup time with $O(n)$ worst case space known as *perfect* (collision-free) hash functions. Fredman et al. [5] and Fox et al. [3] introduced randomized hashing algorithms that can reach the requirements. However, the algorithms only performed on a static dataset, which did not support insert and deletion. Dietzfelbinger et al. [2] presented a modified algorithm of [5] with amortized expected constant time insert and deletion. Here we focus on an efficient hashing scheme called *cuckoo hashing*, which is a new hashing technique with very interesting worst case properties proposed by Pagh and Rodler [7]. The analysis shows that cuckoo hashing scheme achieves the same theoretical properties as the dynamic perfect hashing in [2].

In this paper, we propose to improve the performance of an inversion algorithm for building inverted files. Specifically, we present the inversion algorithm that combines the sort-based inversion algorithm and cuckoo hashing. To the best of our knowledge, the combination of sort-based inversion algorithm with cuckoo hashing has not been reported in the literature. The empirical results show that our algorithm performs well on two large document collections.

The paper is organized as follows. In Section 2, we briefly review the cuckoo hashing that is used for storing our dynamic dictionary data structure. Section 3 describes the sort-based inversion algorithm that combines the cuckoo hashing. In Section 4, we show ex-

periments on two document collections, in particular we examined the results on a subset of TREC-6 collection called Ziff-Davis and the OHSUMED corpus. In section 5, we discuss the implication of the results. Finally, Section 6 concludes with a short summary and also gives the directions of future work.

2. Cuckoo Hashing Scheme

In this section, we review the cuckoo hashing, which is used for representing the dictionary structure S. The main idea of cuckoo hashing is to use two random hashing functions h_1, h_2, and two tables t_1, t_2 of size $m \geq (1+\varepsilon)n$ for some constant $\varepsilon > 0$, where n is the number of all elements. Cuckoo hashing guarantees $O(n)$ space and does not need perfect hash functions that is very complicated if the set of elements stored changes dynamically under the insertion and deletion.

Let us describe two important operations of the cuckoo hashing used intensively in our task. The first operation is the search or lookup. This operation is very simple. The hash table only checks x whether it is stored in $t_1[h_1(x)]$ or $t_2[h_2(x)]$. Figure 1 shows the lookup function. The second operation is the insertion. Given an element x, it is placed in the first table $t_1[h_1(x)]$. If this cell is empty, the insertion is complete. If the cell was previously occupied by an element x', then the element x insists to go into the first table and x' is kicked out. Instead, The element x' now goes into the second table $t_2[h_2(x')]$, which possibly kicks out an element x'' that must go back to $t_1[h_1(x'')]$. This process continues until an unoccupied cell is found. The name "cuckoo" hashing comes from this process that mimics a cuckoo that puts its eggs in other bird's nests.

However, It can be seen that the cuckoo process may not terminate. As a result, the number of iterations is bounded by a specified value called *MaxLoop*. In [7], it proves the case that the insertion procedure loops without limit with probability $O(1/n)$. If MaxLoop value is reached, everything is rehashed by reorganizing the hash table with new hash functions, and we try to find the empty cell again. Figure 2 gives the outline of insertion procedure. The lookup is first called in the procedure to ensure the robustness if the element is already in the two hash tables. In our implementation, we add an attribute to each cell of the hash table to store term occurrence.

```
Function lookup(x)
    return t1[h1(x)] ∨ t2[h2(x)]
End
```

Figure 1. Lookup function

```
Procedure CuckooInsert(x)
  x' ← ∅
  if lookup(x) then return
  for 1 ≤ i ≤ MaxLoop
      x' ← t1[h1(x)]
      t1[h1(x)] ← x
      if x' = ∅ then return
      x ← t2[h2(x')]
      t2[h2(x')] ← x'
      if x = ∅ then return
  rehash()
  CuckooInsert(x)
End
```

Figure 2. Insertion procedure

3. Combining Sort-based Inversion Algorithm with Cuckoo Hashing

In this section, we present the sort-based inversion algorithm that combines the cuckoo hashing scheme for storing the dynamic dictionary data structure. For simplicity, we describe our combination algorithm by giving an example. Suppose that we have a set of four documents $D = \{d_1, d_2, d_3, d_4\}$ as shown in Figure 3. We need to create an inverted file from the original set of D. Figure 2 illustrates the inverted file structure of D.

The sort-based inversion algorithm starts by creating an empty dictionary structure S. We use the hash table to represent S with the cuckoo hashing scheme. Then, the algorithm allocates memory buffer M called a block to store k recodes of $\langle i, j, f_{ij} \rangle$. Each record is composed of a term number i that is assigned in order of first occurrence, a document number j, and a frequency of a term i in a document j. These records are all the same length consisting of 4 bytes for term number, 4 bytes for the document number, and 4 bytes for term frequency. The size in bytes can determine by using the operator sizeof(int) in C programming.

Next, the algorithm passes through the entire set of documents to extract all records. However, we cannot keep all of them during processing due to the limit of size of main memory. As an example in Figure 5, if 84 bytes of main memory are available and we have 20 records (each with 12 bytes long), then we can keep only $k = 7$ records. When the buffer M is full, we do internal sorting by using Quicksort, and write the sorted block to a temporary file. We refer to each block of sorted records as a *run*. This step iterates until all documents in the collection are read. From the example, we obtain three temporary files.

Document	Text
d_1	One love one blood
d_2	One life you have got to do what you should
d_3	One life with each other
d_4	Sisters, brothers

Figure 3. Sample text documents

Number	Term	Document			
		d_1	d_2	d_3	d_4
1	one	2	1	1	-
2	love	1	-	-	-
3	blood	1	-	-	-
4	life	-	1	1	-
5	you	-	2	-	-
6	have	-	1	-	-
7	got	-	1	-	-
8	to	-	1	-	-
9	do	-	1	-	-
10	what	-	1	-	-
11	should	-	1	-	-
12	with	-	-	1	-
13	each	-	-	1	-
14	other	-	-	1	-
15	sisters	-	-	-	1
16	brothers	-	-	-	1

Figure 4. An inverted file corresponding to Figure 3

Term	Term Number
to	8
should	11
one	1
each	13
have	6
other	14
love	2
what	10
got	7
brothers	16
blood	3
sisters	15
you	5
do	9
life	4
with	12

Dictionary

Sorted runs
⟨1,1,2⟩
⟨2,1,1⟩
⟨3,1,1⟩
⟨1,2,1⟩
⟨4,2,1⟩
⟨5,2,2⟩
⟨6,2,1⟩
⟨7,2,1⟩
⟨8,2,1⟩
⟨9,2,1⟩
⟨10,2,1⟩
⟨11,2,1⟩
⟨1,3,1⟩
⟨4,3,1⟩
⟨12,3,1⟩
⟨13,3,1⟩
⟨14,3,1⟩
⟨15,4,1⟩
⟨16,4,1⟩

Sorted runs

Merged runs
⟨1,1,2⟩
⟨1,2,1⟩
⟨1,3,1⟩
⟨2,1,1⟩
⟨3,1,1⟩
⟨4,2,1⟩
⟨4,3,1⟩
⟨5,2,2⟩
⟨6,2,1⟩
⟨7,2,1⟩
⟨8,2,1⟩
⟨9,2,1⟩
⟨10,2,1⟩
⟨11,2,1⟩
⟨12,3,1⟩
⟨13,3,1⟩
⟨14,3,1⟩
⟨15,4,1⟩
⟨16,4,1⟩

Merged runs

Figure 5. Sort-based inversion

1. **Initialization**
   create an empty dictionary S
   allocate memory buffer M
   set $R \leftarrow 0$
2. **Process text to build runs**
   for each document d_j in the collection, $1 \leq j \leq N$
   read d_j, parsing it into index terms
   for each index term $w_i \in d_j$
   let f_{ij} be the frequency in d_j of term w_i
   call CuckooInsert(w_i)
   $M \leftarrow M + \langle i, j, f_{ij} \rangle$, where i is represented by its term number in S
   if M is full
   set run $\leftarrow$ Quicksort(M)
   write run to a temporary file
   set $M \leftarrow \varnothing$, $R \leftarrow R + 1$
3. **Merge runs**
   perform R-way merge runs

Figure 6. The combination of the sorted-based inversion algorithm and the cuckoo hashing

Finally, we perform external sorting to merge all the runs until there remains one run. The final run is equivalent to the inverted file. We can sequentially read the merged run in Figure 5 to get the inverted file in Figure 4. The simple external sorting algorithm merges the runs together by making several passes. For example, we have 100 runs and use a 2-way merge sort, so the sorting requires $\lceil \log_2 100 \rceil = 7$ passes. In each pass, we must read records of all runs from hard disk, which can increase the I/O costs. Therefore we require more sophisticated sorting algorithm to reduce the number of passes through the files. We apply multiway merging algorithm using priority queues [10]. Our inversion algorithm is outlined in Figure 6.

4. Experiments

In this section, we give experimental results for the entire indexing process including time, memory, and disk consumed by the algorithm. In order to examine the algorithm performance, we applied it to the problem of indexing full text documents. We conducted our experiments on a Linux workstation with a 1 GHz single CPU and 512 MBytes of main memory.

4.1 Datasets and Protocol

We used two datasets in our experiments. The Ziff-Davis corpus, taken from the Computer Selects (articles) of the TREC-6 collection [9], consists of 74,815 documents. The OHSUMED corpus, constructed by Hersh et al. [6], contains bibliographic entries and abstracts for 348,566 MEDLINE medical articles.

In our implementation, we did not need the preprocessing phase of preparing the input data for the algorithm. We could build an inverted file directly from the original corpus. By using Porter's suffix-stripping algorithm [4], all words in the corpus were stemmed during the process of reading and parsing each document into index term.

To measure the difficulty of finding an empty cell by hashing, we employed the load factor computed as the ratio of the number of elements in the hash table to the table size. The load factor should be kept as low as possible to maintain good performance. The hash table is designed to keep the load factor between 0.2 and 0.5 taken from empirical experiments in [7]. If it reaches this value, the table size is either halved or doubled, and a rehash occurs.

4.2 Results

Table 1 shows the summary of performance of our algorithm design. It can be observed that building and writing runs (step 2) from the entire document collection are so fast. The computational time of the algorithm is mostly dominated by merging runs (step 3), since we did the external sorting. For the Ziff-Davis corpus, it takes less than 4 minutes to build all runs and less than 5 minutes to merge them. For the OHSUMED corpus, it takes less than 7 minutes and 17 minutes to build and merge runs, respectively.

Table 2 gives the summary of the space used in our algorithm. We fixed the size of block buffer at 1 million records, which consumed 12 MBytes of main memory. When the buffer was full, we did the internal Quicksort and wrote a run to disk. We used 21.2 MBytes for storing the entire distinct terms with their term numbers, so we consumed less than 34 MBytes of main memory to process the Ziff-Davis corpus.

Table 1. Sort-based inversion with cuckoo hashing performance

Corpus	Number of Documents	Size (MBytes)	Build Sorted Runs Time	Merge Runs Time	Overall Time
Ziff-Davis	74,815	250	3m 31s	4m 36s	8m 7s
OHSUMED	348,566	400	6m 14s	16m 4s	22m 18s

Table 2. Space used for building inverted files

Corpus	Number of Distinct terms	Number of Cells in Hash Table (x2)	Size of Hash Table (MBytes)	Number of Records	Size of Final Merged Run (MBytes)
Ziff-Davis	732,132	1,048,576	21.2	17,500,270	216
OHSUMED	1,908,189	2,097,152	45.10	34,080,121	410

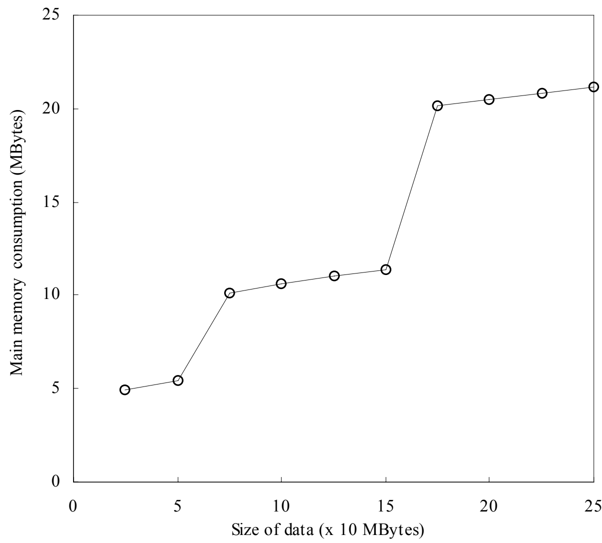

Figure 7. Main memory consumption vs. size of data on Ziff-Davis

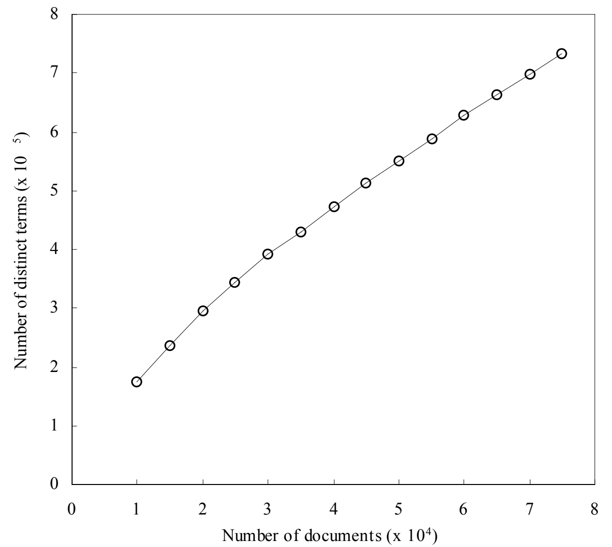

Figure 8. Vocabulary size vs. number of documents on Ziff-Davis

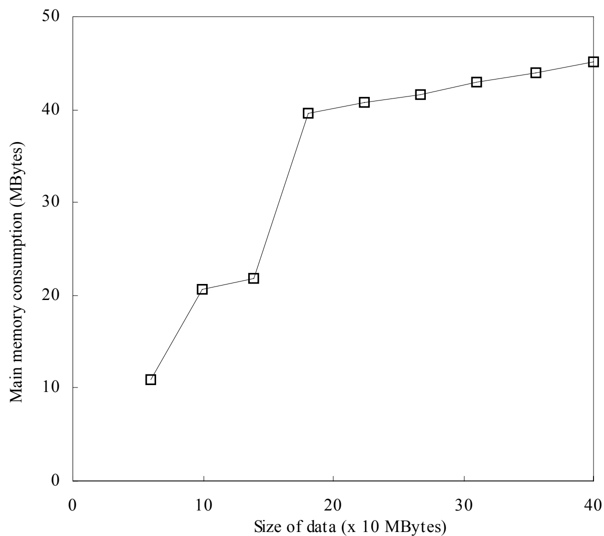

Figure 9. Main memory consumption vs. size of data on OHSUMED

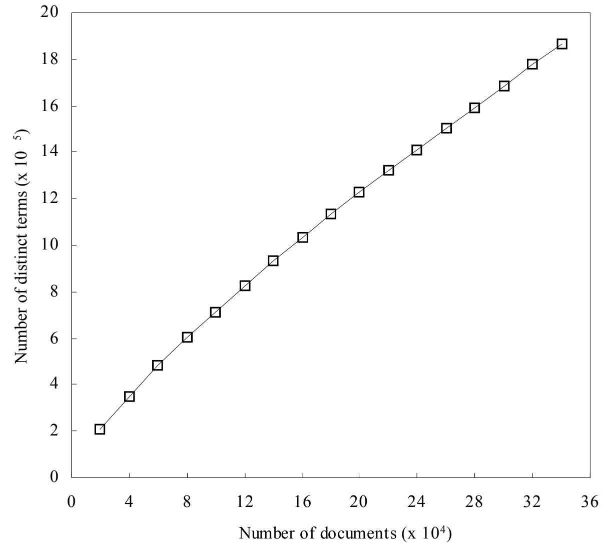

Figure 10. Vocabulary size vs. number of documents on OHSUMED

For the OHSUMED corpus, we used 45.1 MBytes for storing the dictionary and consumed less than 58 MBytes for the entire task. This indicates that our algorithm can carry out from start to end the indexing process on a single workstation with the modest size of main memory.

5. Discussions

The experimental results show that the sort-based inversion algorithm combined with the cuckoo hashing performs well on two datasets. Overall times used for building inverted files take less than 9 minutes and 23 minutes, respectively. We observe that the external sorting mostly dominates the total time. The step 3 of our algorithm takes significantly longer execution times than the step 2, when the data size increases, e.g. on OHSUMED corpus. More efficient merge techniques are required, such as double buffering.

We can see that the algorithm consumes modest main memory during processing. Furthermore, we can roughly predict the main memory consumption of our algorithm in relation to the data size. Figure 7 shows the curve of the main memory consumption versus the data size on Ziff-Davis corpus. The hash table doubles itself two times. Figure 9 shows the same trend on OHSUMED corpus. Consequently, we roughly calculate that our algorithm needs main memory at least 0.1 of data size. For the growth of the size of distinct terms, we see in our experiments that it follows *Heap's Law*, which states that the relation between the vocabulary V and text size n is $V = O(n^{\beta})$, where β is a positive values less than one depending on the particular text [1]. This relation can be seen from Figure 8 and 10.

Further experiments with larger datasets are needed to determine the scalability of our algorithm design. However, our preliminary results indicate that applying cuckoo hashing can provide a useful way of building inverted files.

6. Conclusions and Future Work

In this paper, we have presented a time and memory efficient scheme for building inverted files. We propose the combination of the sort-based inversion algorithm with cuckoo hashing for storing the dynamic dictionary structure. The experimental results show that we are able to build inverted files from the Ziff-Davis and the OHSUMED corpus in reasonable time with modest memory consumption.

In future work, we will continue our focus on improving the efficiency and scalability of our scheme. The merge runs process is the computational bottleneck in sort-based inversion algorithm, since it performs the external merge sort. The efficient techniques for sorting and compressing the runs are required. Distributing our algorithm [8] will also be studied in the future work to examine the scalability issue.

7. Acknowledgement

This research was supported by the grant of the National Research Council of Thailand under Topic Identification and Text Summarization for Thai Text project, 2002.

8. References

[1] R. Baeza-Yates and B. Rebeiro-Neto. Modern Information Retrieval. ACM Press, Addison Wesley, New York, 1999.

[2] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. SIAM J. Computing, 23(4):738-761, 1994.

[3] E.A. Fox, L.S. Heath, Q.F. Chen, and A.M. Daoud. Practical Minimal Perfect Hash Functions for Large Databases. CACM 35(1), pp. 105-121, 1992.

[4] W.B. Frakes and R. Baeza-Yates. Information Retrieval Data Structures & Algorithms. Prentice Hall, Englewood Cliffs, N.J., 1992.

[5] M.L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. Journal of the ACM, 31(3):538-544, 1984.

[6] W. Hersh, C. Buckley, T.J. Leone, and D. Hickam. OHSUMED: An Interactive Retrieval Evaluation and New Large Test Collection for Research. In Proc. ACM SIGIR '94, pp. 192-201, 1994.

[7] R. Pagh and F.F. Rodler. Cuckoo hashing. In ESA: Annual European Symposium on Algorithms, volume 2161 of LNCS. Springer, 2001.

[8] B. Ribeiro-Neto, N. Ziviani, E. Moura, and M. Neuber. Efficient Distributed Algorithms to Build Inverted Files. Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR Forum, pp. 105-112, August 1999.

[9] TREC. Text Retrieval conference. http://trec.nist.gov.

[10] I.H. Witten, A. Moffat, and T.C. Bell. Managing Gigabytes. Van Nostrand Reinhold, New York, 1994.