Efficient Phrase Querying with Flat Position Index

Dongdong Shan, Wayne Xin Zhao, Jing He, Rui Yan, Hongfei Yan and Xiaoming Li
School of Electronics Engineering and Computer Science, Peking University, China
{shandd2008, batmanfly, peaceful.he, yhf1029}@gmail.com,{ r.yan, lxm}@pku.edu.cn

ABSTRACT

A large proportion of search engine queries contain phrases, namely a sequence of adjacent words. In this paper, we propose to use flat position index (a.k.a schema-independent index) for phrase query evaluation. In the flat position index, the entire document collection is viewed as a huge sequence of tokens. Each token is represented by one flat position, which is a unique position offset from the beginning of the collection. Each indexed term is associated with a list of the flat positions about that term in the sequence. To recover *DocID* from flat positions efficiently, we propose a novel cache sensitive look-up table (CSLT), which is much faster than existing search algorithms. Experiments on TREC GOV2 data collection show that flat position index can reduce the index size and speed up phrase querying substantially, compared with traditional word-level index.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: [Search process, efficiency]

General Terms

Algorithms, Performance, Experimentation

Keywords

Flat position index, Phrase query evaluation

1. INTRODUCTION

With the explosive growth of Web data, how to seek information efficiently and effectively has been a very important problem to both research community and industry. Search engines have become the most commonly used tools for Web information retrieval, and it is necessary to process thousands of queries per second. A significant fraction of the submitted queries contain phrases, namely a sequence of adjacent words. Users can submit explicit phrase queries to search engines typically by enclosing them in quotation marks. [3] reported that there were 8.3% of explicit phrase queries in excite log during 1997-1999. In addition, users can also submit implicit phrase queries. [7] analyzed a large amount of AltaVista's query logs, finding that many queries without explicit phrase operator were actually implicit phrase searches. Recently, [9]analyzed Twitter search log and found that about 15.22% percent of the queries are celebrity names, which are possibe phrase queries. It indicates that phrase querying is very important for social network websites, too.

Usually, search engine employs a two-step approach for evaluating phrase queries based on the traditional word-level index [2]. In the traditional word-level index, each indexed term is associated with a posting list, and each posting is a triplet, namely a document identifier (*DocID*), an in-document term frequency (*TF*), and a list of term offsets in that document. With the traditional word-level index, search engines first intersect *DocID* sets to get a list of candidate documents that possibly contain the phrase, and then checks whether the query terms are adjacent or not in the candidate documents. However, as suggested in [3], the traditional word-level index is not efficient since the cost for processing common term's posting list is very high. To solve this problem, one crude method is to remove stop words in queries, which may result in incorrect query evaluation. Other methods [3, 10] add auxiliary structures (e.g. N-gram, partial next word index, phrase index, etc) to speed up phrase query evaluation. One shortcoming of these methods is that not all the candidate documents retrieved in the first step contain the target phrase, which adds un-necessary overloads and slows down the overall processing.

To seek one more efficient way for phrase search, in this paper, we propose to use the flat position index for phrase query evaluation. Flat position index (a.k.a *schema-independent index*) was first proposed by [5] to process queries on structured text, and [6] presented an overall description of its structure. In the flat position index, it views the whole document collection as a single sequence of tokens. Thus, each token can be represented by a unique position offset from the beginning of the collection. For each indexed term, the posting list is composed by a list of position offsets. Flat position index has many potential advantages over the traditional word-level index due to the simple structure. First, *DocID* and *TF* information need not be stored, which results in smaller index. Second, the flat position structure is able to support the queries with position constraints very flexibly, e.g. proximity search and phrase search.

Though the structure of flat position index is very simple and elegant, few studies have presented the implementation details of one real retrieval system with it, and none reported the real performance of it in public literature. It is challenging to deploy flat position index into real systems at least in three aspects:

- How to effectively represent all the flat positions when the document collection is extremely large, i.e. the number of

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

tokens exceeds the maximum range a 32-bit number can represent.

- Since *DocID* is not contained in flat posting lists, how to efficiently recover the original *DocID* from the flat position.
- How to apply existing state-of-art index techniques in flat position index, e.g. compression algorithms.

In this paper, we build a real retrieval system with the flat position index, and we empirically examine the performance of phrase query evaluation based on that. To the best of our knowledge, it's the first study to explore the flat position index for evaluating phrase queries. To solve the first challenge, we divide the whole collection space into several subspaces. Each subspace is a subset of the whole document collection, and each token in that can be represented as a 32-bit integer identifier. To recover *DocID* efficiently, we propose and implement a novel cache sensitive look-up table(CSLT) to map the flat position into corresponding DocID efficiently. The algorithmic complexity for each mapping operation is $O(1)$. The experiments show that it is three times faster than CSS (cache sensitive search) tree [4].

We construct extensive experiments based on *TREC GOV2* collection and compare our results with the traditional word-level index for phrase evaluation. Without nextword structures, flat position index can reduce the index size by 6.3% and speed up phrase evaluation by 31.3%, compared with traditional word-level index. With nextword structures, flat position index can reduce the index size by 7.3% and speed up phrase evaluation by 33.5%, compared with traditional word-level index. We also examine how auxiliary structures affect the performance of phrase querying for flat position index.

2. FLAT POSITION INDEX

In the traditional word-level index [2], each posting of a term t is represented by a triplet, formally we have

$$(d, f_{d,t}, [o_{d,1}, ..., o_{d,f_{d,t}}]), \quad (1)$$

where d is the document identifier, $f_{d,t}$ is the number of occurrences of t in the document (*TF*) and the $o_{d,(\cdot)}$ are the increasingly ordered offsets of t occuring in the document d. With this structure, traditional phrase querying is a two-step approach: perform *DocID* set intersection to get a list of candidate documents, and then check whether candidate documents contain the target phrase query. However, not all the documents retrieved in the first step are the final results. To address this problem, in this section, we introduce flat position index and discuss how to use it for phrase querying.

2.1 Structure of Flat Position Index

Flat position index [5, 6] views the whole document collection as a single sequence of tokens, and each token in the collection has one global and unique identifier (i.e. one flat position). Flat position index is composed by three main components: dictionary, inverted index file and a document boundary array. Similar to that in traditional word-level index, the dictionary contains various statistics information such as document frequency (*DF*) and collection frequency (*CF*) for the terms in the collection. Posting lists of indexed terms are stored in the inverted index file. The posting list is composed by a list of flat positions, i.e. global offset values. The document boundary array stores all boundaries of documents in the collection, which is used to map a flat position into corresponding *DocID*. We present one example in Figure 1 to illustrate the structure of the flat position index. In this example, the whole document collection consists of five documents. The first occurrence of term

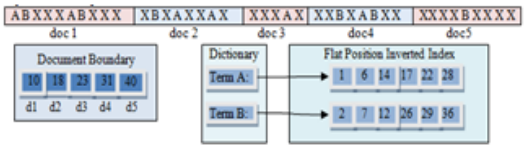

Figure 1: An example for the structure of flat position index.

A is the first token in this collection, so its flat position is 1. Similarly, the flat positions for the first and the third occurrences of term B are respectively 2 and 12. Interestingly, we can see that flat position index doesn't store *DocID* and *TF* information.

When the size of collection is very large, the maximum flat position in the collection may not be represented by a 32-bit integer, e.g., GOV2 collection has 23 billion tokens. One naïve method is to use a 64-bit integer to represent a flat position. However, it has two limitations: 1) more space is needed to store a position in memory; 2) some state-of-the-art compression algorithms can't work for 64-bit integers [1]. To address those problems, we propose to use the *subspace dividing* method, which divides the whole collection into several subspaces. In each subspace, flat positions can be represented by a 32-bit integer, and each subspace is indexed by an subspace identifier. After subspace dividing, each flat position is represented by two 32-bit integers: the subspace identifier and the offset in that subspace. In practice, flat positions in one block usually have the same subspace identifier, so we can just store one subspace identifier for one block to reduce the space.

2.2 Flat Position Mapping

Since there's no explicit *DocID* information in the flat position index, we have to recover the *DocID* from flat positions in query evaluation. The problem of recovering *DocID* can be formulated in a more general problem: given an integer m and an ordered array $\{T_i\}_{i=0}^{n-1}$, find out an index number k ($0 \leq k \leq n-1$) so that $T_k < m$ and $T_{k+1} \geq m$.

A couple of methods can be used to solve this problem: binary search, m-array search trees, CSS-tree (cache sensitive search tree) and CPSS-tree (cache/page sensitive search tree) [4] . Though CSS-tree and CPSS-tree are cache conscious data structures, the time complexity is still $O(\log_m(n))$, which is not efficient enough for an extremely large arrays.

To speed up the position mapping, we propose a novel cache sensitive look-up table (CSLT). The idea is to reduce the number of cache misses by making most of searching operations in a cache line [4].

Let the size of cache line be 2^a bytes , the average document length be l and each element in document boundary array cost 2^b bytes. We assume that the lengths of documents in the collection do not vary too much. The number of documents in a cache line is nearly 2^{a-b}, and a cache line can cover $l * 2^{a-b}$ positions on average. The position space that a cache line cover is called as a cache line position space (CLPS), which is a sub-range of flat positions in the whole collection. The whole position space can be divided into several CLPSs. If the biggest position value in the collection is L, then the number of CLPS is $\lfloor \frac{L}{l*2^{a-b}} \rfloor + 1$. We use a two-layer structure to implement CSLT. The first layer is a *index array* to record the offsets where each CLPS starts; the second layer is the document boundary array. The index array can be built very efficiently as follows: linear scan the document boundary array, assign each boundary value in it to the corresponding CLPS and set the i-th entry in the index array with the offset of the smallest document boundary value in the i-th CLPS.

The time complexity of building the index array is $O(n)$, n is the number of documents in collection. For recovering *DocID* from

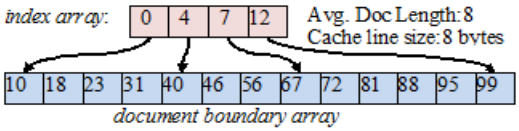

Figure 2: An example for the structure of CSLT.

one flat position, it includes two steps: 1) Compute the offset in the index array by dividing the position by the size of CLPS. Read the value S in corresponding entry of the *index array*. 2) Linear scan the document boundary array from S, and stop when the value of current entry is bigger than the searching position. Return the boundary offset in document boundary array.

Figure 2 illustrates the structure of CSLT. In this example, we have $a = 3$, $b = 1$ and $l = 8$, so a cache line can cover 32 positions and there are totally four CLPS. Then we show how to set the entries of the index array. For example, the second CLPS ranges from 32 to 63, and the fifth entry of the document boundary array is the first boundary value belonging to this CLPS. So we set second entry of the index array to the corresponding offset, i.e. 4. For recovering *DocID* from one flat position 45, we first derive that this position belongs to the second CLPS (45 divided by 32), then we load the second entry of index array to get 4. Second we begin to scan the document boundary array from the fifth element. We skip the first entry 40 since it's smaller than 45. We stop at the second entry 46 (bigger than the searched number 45). The offset of this entry in the document boundary array is 5. So we know that the flat position 45 is mapped to *DocID* 5.

The time complexity of CSLT is O(1). The expected number of cache miss for mapping a position to *DocID* is 2 on average: one in the index array and the other in the document boundary array.

2.3 Auxiliary Structures

For flat position index, we consider two kinds of auxiliary structures common used in traditional word-level index, namely skip lists and nextword structure [10]. Skip lists can reduce the amount of data for decoding while nextword can speed up phrase querying. Both of them can be easily adaptive to flat position index since flat position index is also one variation of word-level inverted index. See Section 3.1 for details.

3. EXPERIMENTS AND RESULTS

3.1 Experiment Setup

Dataset: We use the TREC GOV2 collection, which consists of 25.2 million web pages crawled from the .gov Internet domain. We use two different sets of queries in our experiments. One query set was used in TREC06 efficiency task of Terabyte track. The other query set is sampled from the MSN Search Spring 2006 Query logs. The detailed statistics are shown in Table 1. We run all the experiments on the server with two Quad-Core Intel Xeon 5310(1.66GHz) processors (we only use one core for our experiments) and 8GB of RAM. For each query evaluation, we first load all the necessary posting lists into the main memory.

Query processing: There are two main approaches to process queries [2], either using *DAAT(document-at-a-time)* or using *TAAT(term-at-a-time)* for the standard word-level inverted index. These two approaches can be employed in flat position index, too. In our experiments, we empirically find that *DAAT* is more efficient for phrase querying for both flat position index and the traditional word-level inverted index, so in the following sections, all our experiments are based on *DAAT*.

Compression algorithms: We test four state-of-art compressions

Table 1: Statistics of two query sets.

	TREC query set	MSN query set
#queries	100K	61K
average query length	3.1	2.5

Table 2: Skip distance for different posting list lengths.

Posting list length	Flat position index	Word-level index
$x > 10^7$	384	128
$10^5 < x < 10^7$	64	32
$10^3 < x < 10^5$	32	32
Otherwise	0	0

algorithms in both the standard word-level inverted index and flat position index, including PForDelta, Group Varint [6], Rice and VarByte. In our experiments, we find that PForDelta and Rice results in smaller index size than the others, and PForDelta is much faster than Rice (about 6 times) in decoding data. To make a trade-off between decoding speed and index size, we select PForDelta as the major compression algorithm for both the standard word-level inverted index and flat position index in the following experiments. When the length of one posting list is less than 128, we use VarByte algorithm.

Skip lists: Skip list has been commonly used to speed up processing queries. When adding skip list to an index, it's very important to set a reasonable skip distance. Here we follow [8]'s approach to set list-length-dependent skip lengths. We show our setting of skip lengths for different posting lists in Table 2. After adding skip list, the index size increased by about 1%. Skip lists affect the index size very little but speed up the query evaluation substantially. In the following experiments, all indexes are added with skip lists by default if not particularly mentioned.

Nextword index: Nextword index [10] is proposed to speed up phrase querying. To see how it affects the performance of phrase query evaluation, we add top-5 partial nextword index to both the traditional word-level index and the flat position index. For comparisons, we also keep the original indexes without nextword index.

3.2 Results

Performance of *DocID* mapping: We first evaluate the performance of CSLT for *DocID* mapping. We compare four search algorithms, namely linear search, binary search, CSS tree, and our proposed CSLT. We collect the posting lists of $5K$ query terms as our test data. We use a $25M$ document boundary array, which is sampled from GOV2 test collection. In Figure 3 , we can see that 1) our CSLT performs much better than all the others when the length of posting list is smaller than 5×10^6, the major reason is that the number of expected cache misses for CSLT is smaller than the others; 2) while for very long posting lists (e.g. posting lists of stopwords), linear search performs best since we need to perform *DocID* mapping for most of documents in the collection. Based on the analysis above, we use the linear search algorithm when the length of boundary array is bigger than $\frac{N}{4}$, where N is the document number in collection; otherwise, we use the proposed CSLT.

Index size: We examine the index sizes with/without top 5 partial nextword in both traditional word-level index and flat position index. As shown in Table 3, we find that flat position index reduces the index size substantially. The major reason is that it doesn't store *DocID* and *TF*.

Performance of phrase querying: Generally, the overall phrase query evaluation into four steps: 1) loading data; 2) decoding data;

Table 3: Index size with/without nextword structure.

	Flat	Trad.	Flat+NW	Trad.+NW
Index size(GB)	29.6(-6.3%)	31.6	34.1(-7.3%)	36.6

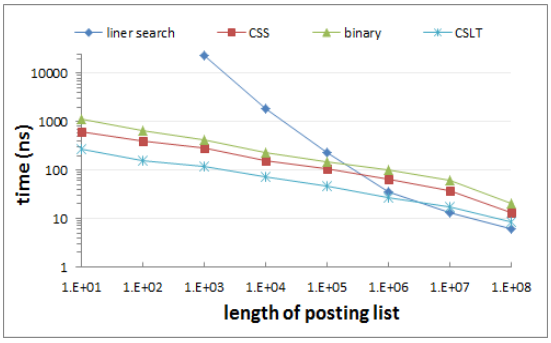

Figure 3: Results of recovering *DocID* for different algorithms.

Table 4: Performance of phrase querying in TREC query set.

	Inverted		Inverted+NW	
	Flat	Trad.	Flat	Trad.
Decoding	0.0556	0.0998	0.0418	0.0758
Finding phrase	0.0711	0.0848	0.0479	0.0590
Total time	0.1267	0.1846	0.0897	0.1348

3) finding phrase; 4) scoring documents. To see why flat position index is more effective in phrase querying, we further examine time costs in each individual step. The time for loading data is not considered since we've loaded all the necessary data into main memory before each query evaluation. We also ignore the fourth step since it is the same for traditional word-level inverted index and flat position index. We further consider two kinds of index structures: a) inverted index without nextword (denoted as *Inverted*); b) inverted index with top 5 nextword structure (denoted as *Inverted+NW*). The results on TREC and MSN query sets are shown respectively in Table 4 and 5. We can see that 1) flat position index is more efficient for query evaluation with/without nextword; 2) flat position index reduces decoding time significantly; 3) nextword can improve query evaluation for both traditional word-level index and flat position index.

Examine the effect of different auxiliary structures: In our experiments, we consider two kinds of auxiliary structures for flat position index, skip list and nextword index. We examine how these auxiliary structures affect the performance of phrase querying for flat position index. We consider four kinds of index as comparisons: raw index, raw index with only skip list, raw index with only top 5 nextwords index, raw index with both skip list and top 5 nextwords index. Fig 4 presents the average processing time of different query lengths using flat position index with different auxiliary structures. The major findings are 1) both skip list and nextword index improve the performance of phrase querying for flat position index, and skip list is more helpful than nextword; 2) skip list together with nextword gives the best performance; 3) when the length of phrase query is long, auxiliary structures is very necessary to improve the performance of phrase querying. We also performed these comparison experiment in traditional word-level index, and we got the similar findings.

4. DISCUSSION AND CONCLUSIONS

In this paper, we propose to use flat position index for efficient phrase querying. We construct extensive experiments based on TREC GOV2 collection. We find that flat position index is very efficient for phrase evaluation. In the future, we plan to do more exploration on how to apply flat position index to general query evaluation. One possible way is to explicitly store *DocID* and *TF* information in flat position index; since flat position index is very efficient to deal with proximity information, another promising way is to transform non-phrase queries into equivalent or approximate queries with proximity constraints. Flat position index can be also used as an auxiliary structure to support efficient proximity related queries.

Table 5: Performance of phrase querying in MSN query set.

	Inverted		Inverted+NW	
	Flat	Trad.	Flat	Trad.
Decoding	0.0602	0.1066	0.0487	0.0860
Finding phrase	0.0685	0.0756	0.0558	0.0618
Total time	0.1287	0.1822	0.1045	0.1478

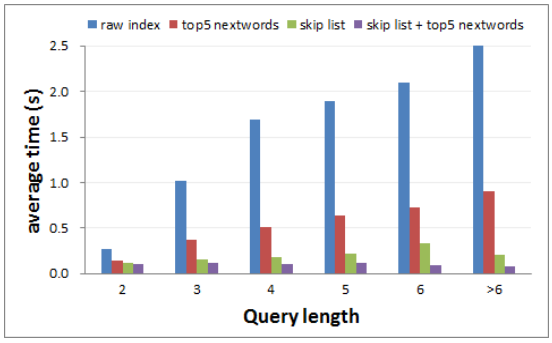

Figure 4: Average processing time of different query lengths using flat position index with different auxiliary structures.

Acknowledgments

This work has been partially supported by HGJ 2010 Grant 2011ZX01042-001-001 and NSFC with Grant No.61073082, 60933004.

5. REFERENCES

[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 2005.

[2] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *SPIRE*, 2006.

[3] D. Bahle, H. E. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR*, 2002.

[4] S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In *CIKM*, pages 761–770, New York, NY, USA, 2007. ACM.

[5] C. L. Clarke, G. V. Cormack, and F. J. Burkowski. An Algebra for Structured Text Search and A Framework for its Implementation. *The Computer Journal*, 1995.

[6] J. Dean. Invited talk: Challenges in building large-scale information retrieval systems. In *WSDM*, 2009.

[7] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 1999.

[8] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *SIGIR*, 2007.

[9] J. Teevan, D. Ramage, and M. R. Morris. #twittersearch: a comparison of microblog search and web search. In *WSDM*, 2011.

[10] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 2004.