Managing Short Postings Lists

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand
andrew@cs.otago.ac.nz

Xiang-Fei Jia
Department of Computer Science
University of Otago
Dunedin, New Zealand
fei@cs.otago.ac.nz

Matt Crane
Department of Computer Science
University of Otago
Dunedin, New Zealand
mcrane@cs.otago.ac.nz

ABSTRACT

Previous work has examined space saving and throughput increasing techniques for long postings lists in an inverted file search engine. In this contribution we show that highly sporadic terms (terms that occur in 1 or 2 documents) are a high proportion of the unique terms in the collection and that these terms are seen in queries. The previously known space saving method of storing their short postings lists in the vocabulary is compared to storing in the postings file. We quantify the saving as about 6.5%, with no loss in precision, and suggest the adoption of this technique.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing – Indexing methods

General Terms

Experimentation, Performance

Keywords

Indexing, Storage, Efficiency, Procrastination

1. INTRODUCTION

An inverted index typically contains two parts, a vocabulary of unique terms in a document collection and a list of postings (normally a pair of ⟨document id, term frequency⟩) for each term. Postings lists can be very long and can require a substantial amount of processing time. Various techniques have been developed to manage their size including compression [10, 4] and static pruning; and various techniques have been developed to decrease processing time including impact ordering [8] and dynamic pruning [2]. However, little attention has been give to the very large number of short postings lists typically seen in an inverted index. As terms frequencies follow a power law distribution it is reasonable to expect the majority of terms to occur only once or twice and so the majority of postings lists will be short.

The management of short lists is simple if their terms are not seen in queries: they could be removed (stopped). If they are misspellings then they could be folded into other postings lists. However such terms are intentionally used or created by authors, can be important, and consequently should not be stopped. Shakespeare uses *honorificabilitudinitatibus* once; in Love's Labour's Lost. *Orc*, from Tolkien, is a derivative of *orcneas*, a term seen only once in Beowulf (to refer to the offspring of Cain).

In this contribution, we examine the management of short postings lists. By short we mean those of singletons (terms that occur in only one document, including all *hapax legomena*) and doubletons (occurring in only two) which together we refer to as *highly sporadic terms.* We first ask the question: *What is the observed frequency of highly sporadic terms relative to other terms in the collection*? We examine this by indexing 4 TREC collections and counting the number of highly sporadic terms. We then ask: *Do highly sporadic terms occur in queries*? This we do by observing the frequency of singletons and doubletons in TREC Million Query Track queries from 2007 and 2008. We find that (as expected) highly sporadic terms account for the majority of the terms in the vocabulary; and (not as expected) they do occur in queries.

We then investigate a method of reducing the space necessary to store short postings lists. The approach we take is to store the postings lists in the vocabulary rather than as separate postings lists. This is done by re-purposing integers already present in the vocabulary, a technique reminiscent of a `union` in the C programming language. This technique has been discussed in the past [11]. Our contribution is the quantification of the consequential space saving; about 6.5%. We also make the observation that storing short postings lists in the vocabulary is a form of pre-fetching. If the vocabulary is loaded into memory on search engine start-up but the postings remain on disk then a disk seek and a disk read can be saved. If the entire index is loaded into memory on start-up then the CPU cache is likely to return gains.

2. RELATED WORK

To quantify any space saving it is first necessary to construct a solid baseline, to apply the technique, and then to measure the savings on multiple collections. Considerable work has been done on fast and efficient methods to store and process an inverted file index, but much of this has concentrated on the small number of very long postings lists that take much of the index space and considerable time to process.

Postings are often described as being a tuple of ⟨document id, term frequency⟩ and a postings list as a list of tuples. Document ids form a monotonically increasing sequence but term frequency numbers do not and so different compression techniques have been developed for each. For document ids it is usual to difference (or delta) encode and then to compress the differences using a technique such as variable byte encoding, simple-9 [3], or PForDelta [14]. The term frequency numbers can either be compressed directly using variable byte encoding or a word-aligned binary code, or further processed as is seen in sigma-encoding [12]. Some document-at-a-time search engines store these two lists separately while others break the postings lists into blocks of consecutive postings and further index those - a technique known as skip lists. Efficient processing of skip lists can be done using the MaxScore [13] or WAND [5] algorithms.

In the case of a singleton, the postings list contains only one tuple. Difference encoding of the document id is ineffectual, and it is likely that the term frequency will fit in one byte (*i.e.* be less than 256). Word-aligned binary codes require the storage of a machine word (for simple-9 this is 4 bytes) and are therefore wasteful.

An alternative approach to document-at-a-time is term-at-a-time processing. In this case the ⟨document id, term frequency⟩ tuples can be re-arranged to group together all terms with the same term frequency. These groups are first sorted in decreasing term frequency order then within each group on increasing document id. The latter is done so that difference encoding can be performed on the document ids (which are then further encoded using variable byte or word aligned codes). The former is for efficiency; documents with the highest term frequency (most likely the highest term-weight) are at the head of the list and can be processed first; and those with a lower "impact" can be optionally pruned [2]. This technique is known as impact ordering.

A further throughput efficiency gain is seen in impact ordering by replacing the term frequencies with the pre-computed term-weight. At indexing time (or shortly thereafter) the term frequency and document frequency is known for each term in each document, as is the document length and the average document length. In a post process the indexer can compute the result of the ranking function for every term in every document and replace term frequency values in the postings list with these impact values [2]. Problematically the impact values are not integers. Consequently, impact values are quantized into integers [8, 2]; in practice a simple linear scaling to fit in one byte is effective.

Impact ordering is a form of compression. The worst case is that each document has a unique quantized impact value and so a tuple needs to be stored for each (no compressive gain). The best case is that all documents share the same impact value and so only one impact value is stored for the entire postings list. In this case the number of stored integers is $n + 1$ where n is the number of documents in which the term occurs and 1 is for the impact score. When term-weights are linearly scaled into one byte the maximum length of a postings list is $n + 256$ as only 256 impact scores are possible.

In the case of a singleton there is a throughput advantage to impact ordering but both the impact value and document id must be stored so there is no space advantage.

Our search engine is term-at-a-time with impact ordered postings list, and consequently our indexes are a small fraction of the collection size (for .GOV, 2.7%) without loss in precision. We use term frequencies as impact scores. These are capped at 255 and stored in one byte. This capping could affect the ranking of frequent terms in long documents, but we assume long documents are rare, that high frequency terms are noise, and observe that many ranking functions are asymptotic. Our postings lists are sorted on decreasing impact value. Within each impact, document ids are sorted on increasing value, then difference encoded, then variable byte encoded.

Efficient methods of managing the vocabulary are examined by Jia *et al.* [6]. They assume a two-level B-tree and investigate throughput and space tradeoffs. They conclude that dictionary front-encoding of the leaves is space efficient but throughput inefficient; a front-encoded dictionary must either be decompressed and searched linearly $O(m)$, or a deserialised and binary searched $O(m)$, where m is the number of terms in the leaf. They propose *embedfixed* as a good trade-off of space and efficiency. This technique is a form of front-encoding that can be binary searched without deserialisation. They first construct a dictionary as the sorted list of all unique terms in the collection; then divide this into blocks of terms that share the same common prefix of length p. The prefixes form the root of a B-tree while the terms in each block (with the common prefix removed) form the leaves. Common prefixes are stored only once (in the root) and the vocabulary can be binary searched without de-serialisation. This technique has an average search time of $O(\log(r)) + O(\log(s))$ where r is the number of prefixes in the root and s is the number of suffixes in the leaf.

Our search engine uses *embedfixed* encoding of a 2-level B-tree to store the vocabulary. A prefix length $p = 4$ is used as this has proven to be the best trade off of space verses throughput [6].

3. HIGHLY SPORADIC TERMS

In this section we illustrate the extent to which highly sporadic terms are seen in the document collection (as expected from the power law distribution) and queries.

3.1 Experiment Setup

Four document collections of different sizes were used. The small (517MB) TREC WSJ (Wall Street Journal) contains 173,252 documents. The 18.5 million document 100GB TREC WT100G is a collection of web pages sourced from the Internet Archive in 1997. The TREC WT10G collection is a 1.69 million document subset of WT100G. TREC .GOV2 is a 2004 trawl of the entire .gov domain, is 426GB and contains 25 million documents.

We used queries from the 2007 and 2008 TREC Million Query Tracks. Both contain 10,000 queries. The 2007 queries have 41,671 terms (10,783 unique) with an average query length of about 4. 2008 has 51,910 terms (10,292 unique) and an average query length of 5. These queries were sourced from "a large Internet search engine" [1].

Two investigations were carried out. The first counted the number of unique terms, singletons, and doubletons in each collection. The second counted the number of these that occur in queries from the Million Query Tracks.

3.2 Singletons and Doubletons in Collections

In this investigation we counted the number of unique terms, singletons, and doubletons in each collection.

Collection	Unique Terms	Singletons	Doubletons
WSJ	229,493	95,455 (42%)	26,984 (12%)
WT10G	5,440,378	3,006,142 (55%)	910,107 (17)%
WT100G	24,815,587	11,372,766 (46%)	3,683,966(15%)
.GOV2	40,565,854	21,080,843 (52%)	7,516,421 (19%)

Table 1: The numbers of unique terms, singletons and doubletons seen in each of the 4 collections

	2007		2008	
Collection	Singletons	Doubletons	Singletons	Doubletons
WSJ	346 (332)	224 (221)	319 (306)	176 (173)
WT10G	77 (73)	49 (49)	54 (53)	36 (36)
WT100G	26 (26)	25 (25)	24 (24)	22 (22)
.GOV2	26 (26)	13 (13)	15 (15)	16 (16)

Table 2: The number of terms in Million Query Track queries occuring in only one document (singletons) or two documents (doubletons) in each collection; parentheses indicate the number of queries

The results are shown in Table 1. The first column lists the name of the collection, the second gives the number of unique terms in the collection, the third lists the number of singletons, and the fourth shows the number of doubletons. For example, in WT100G there are 24,815,587 unique terms of which 11,372,766 (46%) occur once and 3,683,966 (15%) occur twice.

We observe that the number of unique terms, singletons, and doubletons increases as the collection gets larger. We also observe that the proportion of terms that are highly sporadic remains approximately constant, ranging from 41% to 55% for singletons, and from 11% to 18% for doubletons. Overall, between 53.5% and 72.0% of the terms are highly sporadic. By Heaps's law we expect to be unable to close the vocabulary and therefore to see an increasing vocabulary size as the collection gets larger. Due to the power law distribution we expect most terms to be highly sporadic.

3.3 Singletons and Doubletons in Queries

In this investigation we counted the number of singletons and doubletons that occur in each query set.

The results are shown in Table 2. The first column gives the name of the collection, then for each of 2007 and 2008 the number of singleton and doubleton terms seen in the queries is presented along with the number of queries containing singletons and doubletons (in parentheses). For example, in the 2007 Million Query Track queries against WT10G there were 77 singleton terms seen in 73 queries and 49 doubleton terms seen in 49 queries.

We observe that in both sets of queries the number of highly sporadic terms and the number of queries containing highly sporadic terms decreased as the collection size increased. However, highly sporadic terms were seen in both query sets against all collections.

4. SPORADIC TERM MANAGEMENT

The investigations show that highly sporadic terms account for a large proportion of the unique terms in the vocabulary and that they do occur in queries. In this section we examine a method of adapting the index to reducing the space necessary storing these terms.

Figure 1 presents the index used in our search engine, we present it in full for the purpose of reproducibility. The vocabulary root is similar to that described by Jia *et al.* [6]. a 4-byte value (*Prefix_n*) stores the number of prefixes then

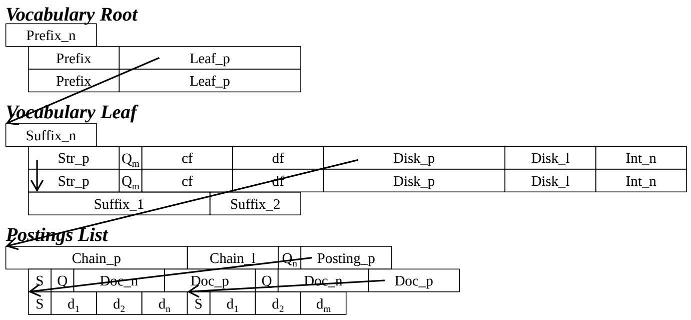

Figure 1: The index is divided into three parts including the vocabulary root and leaves, and the postings lists

each prefix is stored in 4-bytes (*Prefix*) along with an 8-byte pointer to a vocabulary leaf (*Leaf_p*).

The search engine must store a number of term-specific variables for postings list management and ranking. In our search engine these are stored in the vocabulary leaf. It stores a count of the number of terms in the leaf (*Suffix_n*) in 4 bytes followed by a pointer (*Str_p*) to each variable-length suffix (*Suffix_n*). The collection frequency (*cf*) and document frequency (*df*) are stored in 4 each. The location of the postings list (on disk or in memory) is stored in 8 bytes (*Disk_p*), its length is stored in 4 bytes (*Disk_l*). The number of integers in the postings list, used for decompression, is stored in 4 bytes (*Int_n*). Finally we store in 1 byte the largest impact value in the postings list (Q_m) which is used for early termination by query term pruning [9].

We observe that the amount of space taken to store the location and length of compressed postings list (8 + 4 bytes) is larger than the amount of space necessary to store an uncompressed singleton (4 byte document id + 1 byte impact) or a doubleton ($2 * (4 + 1)$ bytes). A `union` could therefore be used to store singletons and doubletons in the space otherwise used for postings list management. It is possible to distinguish the two cases by checking the document frequency.

The general case of this technique is that any compressed postings list that is short enough to store in the vocabulary could be so stored. Only a single bit is needed to determine whether the `union` stores offset / length data or a compressed postings list. In practice additional management information is stored in the postings list, such as skip list pointers, which could render the general case impractical. For example, our postings lists (Figure 1) have a management header that includes an 8 byte *Chain_p* and 4 byte *Chain_l* used for dynamic update, Q_n the number of unique impact scores (1 byte), and a pointer (*Posting_p*) to the start of the postings list (4 bytes). Then each postings list contains a (compressed) postings header where the compression strategy is stored in 1 byte (S) along with the 1 byte impact score (Q), the number of documents with that Q score (*Doc_n*) and a pointer to the document ID list (*Doc_p*). Finally each impact stores its compression strategy in 1 byte (S) and the list of document id delta encoded (d_d) and then further compressed. Compression schemes, S, in Figure 1 can differ from each other.

For query evaluation, the vocabulary root is binary searched then the leaf is loaded and binary searched to find the term details. If the document frequency is less than or equal to 2, the postings are read directly from the vocabulary, otherwise the postings are read from disk.

Collection	In Postings	In Vocabulary	No Singles	No Sporadics
WSJ	64	61 (4.7%)	58 (9.4%)	57 (10.9%)
WT10G	899	804 (10.6%)	727 (19.1%)	672 (25.3%)
WT100G	7,124	6,808 (4.4%)	6,574 (7.7%)	6,328 (11.2%)
.GOV2	11,711	10,977 (6.3%)	10,475 (10.6)	10,020 (14.4%)

Table 3: Index size (MB) and reduction (in parentheses). An average saving of about 6.5% is seen

Our search engine supports loading the entire index into memory at start-up. In this case *Disk_p* is an 8-byte pointer storing a location in memory. There is no effectual difference between the on-disk and in-memory representations. When the index is stored on disk this technique can be thought of a as a form of mass pre-fetching [7] of the short postings lists. All postings of all highly sporadic terms seen in a vocabulary leaf are loaded when the leaf is loaded. When seen in a query a disk seek and a disk read are saved.

4.1 Experiment and Results

In this experiment we examine the storage savings obtained by storing short postings lists in the vocabulary. It uses the same document collections as in Section 3. The structure of the index is discussed in Section 4. Additionally a comparison is made to stopping highly sporadic terms, despite the (likely) precision consequence of doing so.

First, an index storing the postings of highly sporadic terms as postings lists was created. Then an index storing these short postings lists in the vocabulary was created. Finally two additional indexes were created, one with singletons stopped and the other with highly sporadic terms stopped. Variable byte compression was used throughout.

The results are shown in Table 3 where the first column gives the name of the collection, the second column gives the size of the index with short postings lists stored as postings, the third column gives the index size when short postings lists are stored in the vocabulary, the fourth column gives the index size when singletons are stopped and the fifth column gives the index size when singletons and doubletons are stopped. Savings are shown in parentheses. For example, for .GOV2 (426GB) the full index is 11.7GB (2.7% of collection size), with highly sporadic term management the index is 11.0GB (2.5%), when singletons are stopped it is 10.5GB (2.4%) and when highly sporadic terms are stopped it is 10.0GB (2.3%). We observe that our index is already small as a consequence of impact ordering and compression, but further savings are seen by placing short postings lists in the vocabulary.

Overall the indexes were reduced by 4.7%, 10.6%, 4.4% and 6.4% (averaging 6.5%) when highly sporadic terms were stored in the vocabulary. The savings from stopping singletons ranged from 7.7% to 19.1% and from stopping singletons and doubletons it was 10.9% to 25.3%; with possible consequential effects on precision.

5. CONCLUSION

In this investigation we defined highly sporadic terms as those terms that occur in one (singleton) or two (doubleton) documents in the collection. We showed that such terms are a large proportion of the unique terms in the collection, as expected. We showed that as the collection size increased so to did the number of highly sporadic terms, as expected. We took the queries from the TREC Million Query Tracks and showed that highly sporadic terms occur in those queries.

A method of reducing index size was presented. In that method short postings lists were stored in the vocabulary using a `union`. The meaning of the elements in the `union` was determined by examining the document frequency of the term which was also stored in the vocabulary. Doing this saved on average about 6.5% of the index size, and in the case where these terms were seen in queries (and an on-disk index) it additionally saves a seek and a read.

We also examined stopping highly sporadic terms. This resulted in a larger saving (as much as 25.3%). However, our observation that these terms occur in Million Query Track queries suggests that there could be a consequential effect on precision (albeit small when averaged over many queries). In this work we have not examined techniques to stop terms or to increase precision, but rather have concentrated on lossless space savings in the index. We observe that in the case of highly sporadic terms, throughput and space savings can be made with no effect on precision.

6. REFERENCES

[1] J. Allan, B. Carterette, J. Aslam, V. Pavlu, B. Dachev, and E. Kanoulas. Million Query Track 2007 Overview. In *TREC 2007*.

[2] V. N. Anh, O. de Kretser, and A. Moffat. Vector-Space Ranking With Effective Early Termination. In *SIGIR 2001*, pages 35–42.

[3] V. N. Anh and A. Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, 2005.

[4] V. N. Anh and A. Moffat. Improved Word-Aligned Binary Compression for Text Indexing. *TKDE*, 18(6):857–861, 2006.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient Query Evaluation Using A Two-Level Retrieval Process. In *CIKM 2003*, pages 426–434.

[6] X.-F. Jia, A. Trotman, and J. Holdsworth. Fast Search Engine Vocabulary Lookup. In *ADCS 2011*.

[7] X.-F. Jia, A. Trotman, R. O'Keefe, and Z. Huang. Application-Specific Disk I/O Optimisation for a Search Engine. In *PDCAT 2008*, pages 399–404.

[8] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory Efficient Ranking. *IP&M*, 30(6):733–744, 1994.

[9] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered Document Retrieval With Frequency-Sorted Indexes. *JASIS*, 47(10):749–764, 1996.

[10] A. Trotman. Compressing Inverted Files. *Information Retrieval*, 6(1):5–19, 2003.

[11] A. Trotman, X. Jia, and M. Crane. Towards an efficient and effective search engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, pages 40–47.

[12] A. Trotman and V. Subramanya. Sigma Encoded Inverted Files. In *CIKM 2007*, pages 983–986.

[13] H. Turtle and J. Flood. Query Evaluation: Strategies And Optimizations. *IP&M*, 31(6):831–850, 1995.

[14] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE 2006*.