Compressing and Decoding Term Statistics Time Series

Jinfeng Rao[1], Xing Niu[1], and Jimmy Lin[2](✉)

[1] University of Maryland, College Park, USA
{jinfeng,xingniu}@cs.umd.edu
[2] University of Waterloo, Waterloo, Canada
jimmylin@uwaterloo.ca

Abstract. There is growing recognition that temporality plays an important role in information retrieval, particularly for timestamped document collections such as tweets. This paper examines the problem of compressing and decoding term statistics time series, or counts of terms within a particular time window across a large document collection. Such data are large—essentially the cross product of the vocabulary and the number of time intervals—but are also sparse, which makes them amenable to compression. We explore various integer compression techniques, starting with a number of coding schemes that are well-known in the information retrieval literature, and build toward a novel compression approach based on Huffman codes over blocks of term counts. We show that our Huffman-based methods are able to substantially reduce storage requirements compared to state-of-the-art compression techniques while still maintaining good decoding performance.

Keywords: Integer compression techniques · Huffman coding

1 Introduction

There is increasing awareness that time plays an important role in many retrieval tasks, for example, searching newswire articles [5], web pages [3], and tweets [2]. It is clear that effective retrieval systems need to model the temporal characteristics of the query, retrieved documents, and the collection as a whole. This paper focuses on the problem of efficiently storing and accessing term statistics time series—specifically, counts of unigrams and bigrams across a moving window over a potentially large text collection. These retrospective term statistics are useful for modeling the temporal dynamics of document collections. On Twitter, for example, term statistics can change rapidly in response to external events (disasters, celebrity deaths, etc.) [6]. Being able to store and access such data is useful for the development of temporal ranking models.

Term statistics time series are large—essentially the cross product of the vocabulary and the number of time intervals—but are also sparse, which makes them amenable to compression. Naturally, we would like to achieve as much

N. Ferro et al. (Eds.): ECIR 2016, LNCS 9626, pp. 675–681, 2016.
DOI: 10.1007/978-3-319-30671-1_53

compression as possible to minimize the storage requirements, but this needs to be balanced with decoding latencies, as the two desiderata are often in tension.[1] Our work explores this tradeoff.

The contribution of this paper is an exploration of compression techniques for term statistics time series. We begin with a number of well-known integer compression techniques and build toward a novel approach based on Huffman codes over blocks of term counts. We show that our Huffman-based techniques are able to substantially reduce storage requirements compared to state-of-the-art compression techniques while still maintaining good decoding performance. Our contribution enables retrieval systems to load large amounts of time series data into main memory and access term statistics with low latency.

2 Background and Related Work

We adopt the standard definition of a time series as a finite sequence of n real numbers, typically generated by some underlying process for a duration of n time units: $x = \{x_0, x_1, x_2, \ldots, x_n\}$, where each x_n corresponds to the value of some attribute at a point in time. In our case, these time series data correspond to counts on a stream of timestamped documents (tweets in our case) at fixed intervals (e.g., hourly). To be precise, these term statistics represent collection frequencies of unigrams and bigrams from a "temporal slice" of the document collection consisting of documents whose timestamps fall within the interval.

There has been much previous work, primarily in the database and data mining communities, on analyzing and searching time series data. We, however, focus on the much narrower problem of compressing and decoding time series data for information retrieval applications. There are a number of well-known integer coding techniques for compressing postings lists in inverted indexes: these include variable-byte encoding, γ codes, interpolative coding, the Simple-9 family [1], and PForDelta [9]. Various compression techniques represent different tradeoffs between degree of compression and decoding speed, which have been well studied in the indexing context. Note that our problem is different from that of postings compression: postings lists only keep track of documents that contain the term, and hence differ in length, whereas in our case we are also interested in intervals where a term *does not* appear.

3 Methods

In this work, we assume that counts are aggregated at five minute intervals, so each unigram or bigram is associated with $24 \times 60/5 = 288$ values per day. Previous work [7] suggests that smaller windows are not necessary for most applications, and coarser-grained statistics can always be derived via aggregation.

We compared five basic integer compression techniques: variable-byte encoding (VB) [8], Simple16 [1], PForDelta (P4D) [9], discrete wavelet transform

[1] We set aside compression speed since we are working with retrospective collections.

(DWT) with Haar wavelets, and variants of Huffman codes [4]. The first three are commonly used in IR applications, and therefore we simply refer readers to previous papers for more details. We discuss the last two in more detail.

Discrete Wavelet Transform (DWT). The discrete wavelet transform enables time-frequency localization to capture both frequency information and when (in time) those frequencies are observed. In this work, we use Haar wavelets. To illustrate how DWT with Haar wavelets work, we start with a simple example. Suppose we have a time series with four values: $X = \{7, 9, 5, 3\}$. We first perform pairwise averaging to obtain a lower resolution signal with the values: $\{8, 4\}$. The first value is obtained by averaging $\{7, 9\}$ and the second by averaging $\{5, 3\}$. To account for information lost in the averaging, we store detail coefficients equal to pairwise differences of $\{7, 9\}$ and $\{5, 3\}$, divided by two. This yields $\{-1, 1\}$, which allows us to reconstruct the original signal perfectly. Assuming a signal with 2^n values, we can recursively apply this transformation until we end up with an average of all values. The final representation of the signal is the final average and all the detail coefficients. This transformation potentially yields a more compact representation since the detail coefficients are often smaller than the original values. We further compress the coefficients using either variable-byte encoding or PForDelta. Since the coefficients may be negative, we need to store the signs (in a separate bit array).

Huffman Coding. A nice property of Huffman coding [4] is that it can find the optimal prefix code for each symbol when the frequency information of all symbols are given. In our case, given a list of counts, we first partition the list into several *blocks*, with each block consisting of eight consecutive integers. After we calculate the frequency counts of all blocks, we are able to construct a Huffman tree over the blocks and obtain a code for each block. We then concatenate the binary Huffman codes of all blocks and convert this long binary representation into a sequence of 32-bit integers. Finally, we can apply any compression method on top of these integer sequences. To decode, we first decompress the integer array into its binary representation. Then, this binary code is checked bit by bit to determine the boundaries of the original Huffman codes. Once the boundary positions are obtained, we can recover the original integer counts by looking up the Huffman code mapping. The decoding time is linear with respect to the length of Huffman codes after concatenation.

Beyond integer compression techniques, we can exploit the sparseness of unigram counts to reduce storage for bigram counts. There is no need to store the bigram count if any unigram of that bigram has a count of zero at that specific interval. For example, suppose we have count arrays for unigram A, B and bigram AB below: A: 00300523, B: 45200103, and AB: 00100002. In this case, we only need to store the 3rd, 6th, and 8th counts for bigram AB (that is, 102), while the other counts can be dropped since at least one of its unigrams has count zero in those intervals. To keep track of these positions we allocate a bit vector 288 bits long (per day) and store this bit vector alongside the compressed data. This truncation technique saves space but at the cost of an additional step during decoding. When recovering the bigram counts, we need to consult the bit vector, which is used to pad zeros in the truncated count array accordingly.

In terms of physical storage, we maintain a global array by concatenating the compressed representations for all terms across all days. To access the compressed array for a term on a specific day, we need its offset and length in the global array. Thus, we keep a separate table of the mapping from (term id, day) to this information. Although in our experiments we assume that all data are held in main memory, our approach can be easily extended to disk-based storage.

As an alternative, instead of placing data for all unigrams and bigrams for all days together, we could partition the global array into several shards with each shard containing term statistics for a particular day. The advantage of this design is apparent: we can select which data to load into memory when the global array is larger than the amount of memory available.

4 Experiments

We evaluated our compression techniques in terms of two metrics: size of the compressed representation and decoding latency. For the decoding latency experiments, we iterated over all unigrams or bigrams in the vocabulary, over all days, and report the average time it takes to decode counts for a single day (i.e., 288 integers). All our algorithms were implemented in Java and available open source.[2] Experiments were conducted on a server with dual Intel Xeon 4-core processors (E5620 2.4 GHz) and 128 GB RAM.

Our algorithms were evaluated over the Tweets2011 and Tweets2013 collections. The Tweets2011 collection consists of an approximately 1 % sample of tweets from January 23, 2011 to February 7, 2011 (inclusive), totaling approximately 16 m tweets. This collection was used in the TREC 2011 and TREC 2012 microblog evaluations. The Tweets2013 collection consists of approximately 243 m tweets crawled from Twitter's public sample stream between February 1 and March 31, 2013 (inclusive). This collection was used in the TREC 2013 and TREC 2014 microblog track evaluations. All non-ASCII characters were removed in the preprocessing phase. We set a threshold (by default, greater than one per day) to filter out all low frequency terms (including unigrams and bigrams). We extracted a total of 0.7 m unigrams and 7.3 m bigrams from the Tweets2011 collection; 2.3 m unigrams and 23.1 m bigrams from the Tweets2013 collection.

Results are shown in Table 1. Each row denotes a compression method. The first row "Raw" is the collection without any compression (i.e., each count is represented by a 32-bit integer). The row "VB" denotes variable-byte encoding; row "P4D" denotes PForDelta. Next comes the wavelet and Huffman-based techniques. The last row "Optimal" shows the optimal storage space with the lowest entropy to represent all Huffman blocks. Given the frequency information of all blocks, the optimal space can be computed by summing over the entropy bits

[2] https://github.com/Jeffyrao/time-series-compression.

consumed by each block (which is also the minimum bits to represent a block). The column "size" represents the compressed size of all data (in base two). To make comparisons fair, instead of comparing with the (uncompressed) raw data, we compared each approach against PForDelta, which is considered state of the art in information retrieval for coding sequences such as postings lists [9]. The column "percentage" shows relative size differences with respect to PForDelta. The column "time" denotes the decompression time for each count array (the integer list for one term in one day).

Table 1. Results on the Tweets2011 (top) and Tweets2013 (bottom) collections.

Tweets2011	Unigrams			Bigrams		
Method	size (MB)	percentage	time (µs)	size (MB)	percentage	time (µs)
Raw	4760			12800		
VB	1200	+442 %	1.9	3200	+318 %	1.1
Simple16	200	−9.50 %	1.1	653	−14.6 %	0.7
P4D	221	–	1.0	764	–	1.2
Wavelet+VB	1300	+488 %	2.3	3700	+384 %	2.3
Wavelet+P4D	352	+59.3 %	2.7	978	+28.0 %	2.3
Huffman	65	−70.6 %	7.8	396	−48.2 %	2.9
Huffman+VB	46	−79.2 %	8	180	−76.4 %	3.2
Optimal	32	−85.5 %	–	108	−85.9 %	-
Tweets2013	Unigrams			Bigrams		
Method	size (GB)	percentage	time (µs)	size (GB)	percentage	time (µs)
Raw	52.5			171.8		
VB	13.1	+446 %	3.8	43.0	+347 %	1.3
Simple16	2.2	−8.33 %	2.2	8.3	−13.5 %	0.8
P4D	2.4	–	1.9	9.6	–	1.2
Wavelet+VB	14.8	+517 %	6.4	49.0	+410 %	2.6
Wavelet+P4D	3.8	+58.3 %	4.7	12.9	+34.4 %	6.2
Huffman	0.71	−70.4 %	14.7	4.9	−49.0 %	6.2
Huffman+VB	0.48	−80.0 %	15.6	3.0	−68.7 %	6.3
Optimal	0.33	−86.2 %	-	0.95	−90.1 %	–

Results show that both Simple16 and PForDelta are effective in compressing the data. Simple16 achieves better compression, but for unigrams is slightly slower to decode. Variable-byte encoding, on the other hand, does not work particularly well: the reason is that our count arrays are aggregated over a relative small temporal window (five minutes) and therefore term counts are generally small. This enables Simple16 and PForDelta to represent the values using very few bits. In contrast, VB cannot represent an integer using fewer than eight bits.

We also noticed that the Wavelet+VB and Wavelet+P4D techniques require more space than just VB and PForDelta alone, which suggests that the wavelet transform is not effective. We believe this increase comes from: (1) DWT requires an additional array to store the sign bits of the coefficients, and (2) since the original counts were already sparse, DWT does not additionally help.

The decoding times for VB, Simple16, PForDelta, and the wavelet methods are all quite small, and it is interesting to note that decoding bigrams can be actually *faster* than decoding unigrams, which suggests that our masking mechanism is effective in reducing the length of the bigram count arrays.

Experiments show that we are able to achieve substantial compression with the Huffman-based techniques, up to 80 % reduction over PForDelta. Overall, our findings hold consistently over both the Tweets2011 and Tweets2013 collections. In fact, Huffman+VB is pretty close to the entropy lower bound. Entropy coding techniques like Huffman coding prefer highly non-uniform frequency distributions, and thus are perfectly suited to our time series data. Although our Huffman+VB technique also increases decoding time, we believe that this tradeoff is worthwhile, but of course, this is application dependent. We did not try to combine Huffman coding with Simple16 or PForDelta as we found that the integer lists transformed from Huffman codes were generally composed of large values, which are not suitable for word-aligned compression methods.

5 Conclusion

The main contribution of our work is an exploration of integer compression techniques for term statistics time series. We demonstrated the effectiveness of our novel techniques based on Huffman codes, which exploit the sparse and highly non-uniform distribution of blocks of counts. Our best technique can reduce storage requirements by a factor of four to five compared to PForDelta encoding. A small footprint means that it is practical to load large amounts of term statistics time series into memory for efficient access.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation under IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor.

References

1. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retrieval **8**(1), 151–166 (2005)
2. Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., Lin, J.: Earlybird: real-time search at Twitter. In: ICDE (2012)
3. Elsas, J.L., Dumais, S.T.: Leveraging temporal dynamics of document content in relevance ranking. In: WSDM (2010)
4. Huffman, D.A., et al.: A method for the construction of minimum redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)

5. Jones, R., Diaz, F.: Temporal profiles of queries. ACM TOIS **25**, Article no. 14 (2007)
6. Lin, J., Mishne, G.: A study of "churn" in tweets and real-time search queries. In: ICWSM (2012)
7. Mishne, G., Dalton, J., Li, Z., Sharma, A., Lin, J.: Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In: SIGMOD (2013)
8. Williams, H.E., Zobel, J.: Compressing integers for fast file access. Comput. J. **42**(3), 193–201 (1999)
9. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: WWW (2008)