Blooming Onion: Efficient Deduplication through Approximate Membership Testing

Ondřej Herman[1,2]

[1] Faculty of Informatics, Masaryk University
Botanická 68a, 612 00 Brno, Czech Republic
xherman1@fi.muni.cz
[2] Lexical Computing s.r.o.
Botanická 68a, 612 00 Brno, Czech Republic
ondrej.herman@sketchengine.eu

Abstract. Deduplication of source text is an important step in corpus building. Maximum corpus sizes have been grown significantly, along with the requirements for computing resources required for processing them. This article explores reducing the cost of deduplication by applying approximate membership testing using Bloom filtering.

Keywords: deduplication, text corpora, Bloom filter

1 Introduction

Deduplication is an essential step in the preparation of text corpora for many downstream tasks in natural language processing. For example, in the process of training unsupervised machine learning models, repeated instances of the same data can cause significant biases. In fulltext search applications, the end users do not want to see repeated results for a single query, but a balanced representation of the source corpus. In linguistic applications, repeated text in the source data influences the statistics derived from it and reduces the quality and representativeness of the result.

Duplicates appear for many reasons in text data. Humans like to copy. This happens on many levels. Citations, boilerplate, or outright spam are common reasons. Data obtained from the Web is rife with repeated text in the main content, but also advertisments, context management system artifacts and links to the same, repeated content. The repeated content is not always an exact copy, but is sometimes changed slightly, to escape detection, or simply due to errors, so detection of exact instances is not enough for practical use, near duplicates also need to be considered.

1.1 Onion

The tool we use for text deduplication for the building of corpora at Sketch Engine ([6]) is Onion ([7]). Onion (ONe Instance ONly) works on the vertical text

A. Horák, P. Rychlý, A. Rambousek (eds.): Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2022, pp. 91–95, 2022.

format. At this stage, the text has already been tokenized and is represented as a single line per token, possibly with additional data for the same token separated by TAB characters on the same line, such as lemmata or part-of-speech tags. The text is segmented at least into documents and paragraphs, delimited by <doc>, </doc> and , markers. Other markers, such as <s> for describing sentence boundaries, may be present. For example, the beginning of the Susanne corpus in the vertical text format:

```
<doc file="A01" n="1">
<p>
<s>
The     the     AT
Fulton  Fulton  NP1s
County  county  NNL1cb
Grand   grand   JJ
Jury    jury    NN1c
said    say     VVDv
Friday  Friday  NPD1
an      an      AT1
investigation   investigation   NN1n
```

Onion works by detecting and discarding duplicate or near-duplicate paragraphs. Paragraph is split into overlapping sequence of tuples of words, called shingles. For example, shingles of length 3 in the above text would be *(The, Fulton, County)*, *(Fulton, County, Grand)*, *(County, Grand, Jury)* and so on. A paragraph is considered to be a duplicate if the proportion of already seen shingles contained within it is larger than a specific threshold. For our purposes, we set this threshold to **50 %** and the shingle length to **7**.

Onion works by storing the hashes of all already seen shingles in a hash table, and therefore the memory requirements can be quite significant for large corpora. In the following, I explore the possibility of replacing the hash table, which stores the exact hashes for ecery shingle, by an approximate data structure. This can have a significant effect on memory requirements, but only a small and predictable effect on the precision of the threshold check.

2 Approximate membership testing

Membership testing is the problem of checking whether an element is present as a member of a set. Our elements do not have any special mathematical properties and can be arbitrary, so storing some information about them is unavoidable.

The straightforward approach to this problem is storing the elements or their fingerprints in a collection and then searching the collection to see whether the elements are present or not.

For n-element collections, simple tree structures such as Binary search trees allow for average insertion and retrieval complexity in $O(\log n)$ per element,

while Hashtable based data structures approach $O(1)$. Nevertheless, the space requirement is $O(n)$, so the memory requirements increase linearly as new, unseen data points arrive.

A method first described in [2], the Bloom filter, allows for a significantly reduced memory footprint for the creation of a data structure, at the cost of possible false positives. That is, an element which has not been inserted, can be deemed present in the set with a non-zero probability. The Bloom filter consists of a zero-initialized m-bit array and k hash functions. Each of the hash functions takes the set element and hashes it to a number between 0 and m.

During the **insertion** into the Bloom filter, the element is hashed by each of the hash functions and the bits at the corresponding positions in the bit array are set to 1.

During the **retrieval**, the element is hashed in the same way and the bit positions are examined. If any of them is 0, the element has certainly not been inserted into the array.

Bloom filter can trade off the memory requirements against the false positive rate. The rate is approximately 1 % for a Bloom filter which uses 10 bits per element.

The main drawback is that the Bloom filter does not allow for resizing and that the parameters need to be known in advance. An extension, the Scalable Bloom filter ([1]), allows for indefinitely growable approximate membership structure. First, a single Bloom filter is created. As elements are added and the false positive rate raises above a specified threshold, another larger filter is allocated and new elements are inserted into it. This procedure is repeated as required. Membership is then checked in every Bloom filter in sequence.

Many other data structures for approximate membership testing have been devised over the years with reduced memory requirements, better cache locality or throughput. Unfortunately, all of them seem to have properties which disqualify them for the use case at hand.

For example, the Cuckoo filter ([4]), which uses Cuckoo hashing, is more efficient in terms of space required and exhibits good cache locality, but resizing requires rehashing all the elements which have already been inserted.

The XOR filter ([5]) and Ribbon filter ([3]) are even better in terms of memory requirements, but do not support dynamic insertion and require a distinct build step before they can be used.

3 Blooming Onion

It is written in the Rust[3], which is a modern programming language, designed with performance and safety in mind.

The program uses the Growable Bloom filter[4] library, which implementats the Scalable Bloom filter data structure. The Scalable Bloom filter. The structure is initialized with the false positive rate set to 1 %.

[3] `https://www.rust.org`

[4] `https://crates.io/crates/growable-bloom-filter`

Table 1: Susanne corpus

	runtime	max RSS
Blooming Onion	1.94 s	3608 kB
Onion	1.71 s	30616 kB

Table 2: JSI Newsfeed

	runtime	max RSS
Blooming Onion	720.6 s	271.6 MB
Onion	491.3 s	2367.2 MB

Only the most essential features have been implemented at this point and the program serves as a proof of concept. The whole implementation fits into 160 lines of code.

4 Evaluation

Blooming Onion was evaluated against Onion on two datasets:

1. Susanne corpus, repeated 20 times (100 MB, 190 k lines, 97.5 % duplicate, see Table 1)
2. 7 days of the JSI Newsfeed Corpus (13 GB, 876 k lines, 64 % duplicate, see Table 2)

The time required for the deduplication and the maximal resident set size (RSS) have been measured.

While Blooming Onion is about 25 % slower, it uses only 10 % of the memory compared to Onion. The slowdown can be attributed to two major causes. Scalable Bloom filter has worse cache-related behavior compared to the hashtable used by Onion. The backing bits of the filter are scattered around in memory, and therefore require multiple random accesses. This problem could be improved by using a different data structure, perhaps some type of Quotient filter, which orients the accesses for a single elements into a smaller memory are. No implementation of such data structure seems to be available. Of interest could be the fact that the evaluation has been carried out on a server with slow DDR3 memory. A cursory check on a modern laptop with a smaller amount of faster DDR4 memory swaps the order of performance and Onion is slower than Blooming Onion.

The second reason is that Onion is written in a highly optimized way, which avoids many copies of the data at the expense of readability, while Blooming

Onion aims to be simple and readable, and the input text is being copied multiple times. This can be explored in a future version of Blooming Onion.

5 Conclusion

The problem of text deduplication and a current approach we use has been described. The Blooming Onion deduplicator was presented and compared against Onion. Blooming onion is approximately 25 % slower, but only requires 10 % of the compared to Onion. With the proposed improvements, Blooming Onion could be both faster and use more memory compared to Onion.

Acknowledgements This work has been partly supported by the Ministry of Education of CR within the LINDAT-CLARIAH-CZ project LM2018101.

References

1. Almeida, P.S., Baquero, C., Preguiça, N., Hutchison, D.: Scalable bloom filters. Information Processing Letters **101**(6), 255–261 (2007)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
3. Dillinger, P.C., Walzer, S.: Ribbon filter: practically smaller than bloom and xor. arXiv preprint arXiv:2103.02515 (2021)
4. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: Practically better than bloom. In: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. pp. 75–88 (2014)
5. Graf, T.M., Lemire, D.: Xor filters: Faster and smaller than bloom and cuckoo filters. Journal of Experimental Algorithmics (JEA) **25**, 1–16 (2020)
6. Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlỳ, P., Suchomel, V.: The sketch engine: ten years on. Lexicography **1**(1), 7–36 (2014)
7. Pomikálek, J.: Removing boilerplate and duplicate content from web corpora. Ph.D. thesis, Masaryk university, Faculty of informatics, Brno, Czech Republic (2011)