On the application of Bloom Filter Hierarchies representing Sub-word Token Bigram Occurrence to Probabilistic Full Text Search

Sarah Jamie Lewis *†

May 29, 2024

1 Motivating Problem

This note is rooted in the problem of efficient and decentralized document search (where the indexing and/or search operations are conducted by a number of distinct distributed entities).

1.1 This Work

We present a method for building a searchable index of documents through a combination of subword tokenization and bloom filters, this structure permits efficient (probabilistic) full text search where both indexing and search can be highly parallelized (and, potentially, distributed).

We quantify performance through a CUDA-based prototype that can retrieve sorted (ranked) search results over a recent Wikipedia dataset ($|N| = 6325358$) at around **19ms** per query token .

2 Related Work

Bloom filters[2] have a long history in document search, with the use case motivating much of the early research (e.g. [12]). The application of bloom filters to detecting and removing duplicate documents (e.g. [5]), likewise has a similar history.

Hierarchical Bloom Filters are less common in the literature, and where they do appear, are mostly constrained to considering structures where the filters in consideration represent documents, with the root bloom filters representing a superset of documents contained by their child nodes (e.g. see [8][9]).

More recently, Hierarchical Interleaved Bloom Filters[10] have been proposed for approximate matching of genome samples. This technique is analogous to the method outlined above where multiple samples are coalesced into a hierarchy of bloom filters suitable for matching.

Counting Bloom Filters are a generalization of bloom filters that permit an efficient removal operation[3], at the cost of false negatives[4].

3 Our Approach

Given a document we first tokenize it using a subword tokenizer[1]. Once complete, we proceed sequentially through the list of tokens and make a note of each token-pair (bigrams).

These sequential token-pairs can represent individual terms, common subterms, and very common multi-word terms.

Next we derive a count of each bigram, and finally instantiate a set of bloom filters representing the occurrence of each bigram in the document:

- Every bigram is inserted into the top-level bloom filter and thus this bloom filter can be used as a probabilistic indication of the **presence** of a bigram in the document.
- At each subsequent level (l) of bloom filter we only insert a bigram if it has at least $l = \ln(n)$ occurrences in the document. As such the

*Open Privacy Research Society (sarah@openprivacy.ca)
†Blodeuwedd Labs (sarah@blodeuweddlabs.com)

[1]For our prototype we used an implementation of the open source Mistral[6] v1 BPE/SentencePiece[7] tokenizer with a vocab size of 32k.

more common a term in a document the more levels it is represented in.

Each bloom filter can be parameterized based on the documents contents to ensure a consistent false positive rate. These parameterization impact both the size of the bloom filters themselves and the number of bloom filters representing the document.

4 Searching the Index

The approach that we take for search is analogous to the long-standing term-frequency/inverse-document-frequency[13] (tf-idf) but applied to token-pairs instead of word terms.

Conceptually, we tokenize and construct a set of overlapping bigrams for the search term, and then use the bloom filters to estimate the occurrence of each bigram in each document. We can then calculate the tf-idf score by multiplying the bigram frequency of the document by the idf score calculated over all documents. Finally, to arrive at a document score we calculate the cosine-similarity of resulting vector of term-level tf-idf scores with the query vector itself - arriving at a score between 0 and 1.

We can then sort the document pool by this score.

Breaking this down for our CUDA-based prototype:

1. The query is tokenized and split into a sequence of overlapping bigram terms (e.g. "quick brown fox jumped" might become "quick brown" and "brown fox" etc.) (though in practice we use a modified sentencepiece for tokenization) and sent to the GPU

2. (document-parallel) For each n-gram term, a probabilsitic term frequency score is computed for each document in parallel using the filters defined above (and a global frequency term is also calculated)

3. (document-parallel) Each document computes an aggregate tf-idf score and if that score is above a certain threshold it adds itself (atomically) to a resulsts list

4. (core-parallel) The results list is sorted (using a parallel bitonic sort)

Because of the method of indexing, the efficacy of search improves with longer search terms (see 5.3.1 for experimental validation).

5 Discussion

The presented index structure has a few interesting features regardless of the methods used to tokenize the source documents and queries:

1. **We can definitively identify the occurrence of a sentence in the source document** (e.g. an index that contains token bigrams representing "The quick" AND "quick brown" AND "brown fox" AND "fox jumps" and so on, likely represents a document that contains "The quick brown fox jumps over the lazy dog")

2. **The probabilities of each bloom filter are tuneable** when indexing, so each document can be represented by a **variable sized index**. (See 5.4.1 for a possible application of this property)

3. **Each index can be (re-)computed independent of each other** (and be be included in a searches dynamically - i.e. this approach permits sub-index queries). This allows for some flexibility when searching, e.g. trivially restricting to a set of documents, or expanding to others.

And a few critical considerations:

1. There is little to no semantic content in the document indexes themselves, they only "unlock" information with respect to a particular query (and any information they do unlock is probablistic in nature)

2. There is an explicit tradeoff between false positives and the size of a document index i.e. the smaller a document index the more likely it is to match against a query.

3. Shorter queries are more likely to match a larger number of documents (as true negatives from one part of the query are less likely to cancel out false positives from another).

5.1 False Positives

Because bloom filters are a probabilistic data structure the rate of false positives is tightly determined by the parameterization of each filter.

The hierarchical construction is designed to minimize the impact of individual false positives and it is important that every level be independent from each other (e.g. by using different hash algorithms at each level).

As noted above, the efficacy of search improves with longer search terms. This happens because the rate of false positives is spread out over the document pool, but a single document which matches the query will always positively match the given input.

5.2 Tokenization

While there is nothing inherent in the index structure which prevents using an alternative tokenization scheme (e.g. space-delineated / full word), by leveraging recent advances in subword document tokenization we are able to reduce the total number of unique bigrams we must possibly consider - with modern tokenizers providing significantly reduced representations of common language terms e.g. the word "Hello" is so common in many corpora that it represented by a single 2 byte token.

This has significant implications when considering the choice of false positive rates for our bloom filters - with our choice of sub-word tokenization vocabulary we must **at-most** be able to deal with $1024000000 = 32000^2$ bigrams (of which many are very unlikely to occur in any source document). This strong upper bound makes reasonable false positive rates (between 1 and 5%) tractable for document sets containing millions of documents.

Using such a tokenization scheme also means we are able to apply our search scheme over documents containing programming language code, documents containing emojis, foreign language documents etc. without changing the indexing structure, or the query interface.

5.3 Performance

Each document can be tokenized, and indexed separately from every other document. As such the process of indexing can be done in parallel, and each document can be parameterized separately.

The size of the index depends on the parameterization. For our prototype we indexed a recent Wikipedia dump, a total of over 6 million articles (for clarity we filter out category, disambiguation, and other meta pages from the indexing process), with an average word count of around 700 words. The resulting index would be between 4GB to 12GB in size depending on the parameters used (an average of between 500 bytes and 5kb per document - including auxillary information like an encoded representation of the document title and URL - useful when displaying results).

For search there are multiple ways of optimizing the runtime; we can parallelize the process of determining term frequency for each document, and once complete we can concurrently calculate the td-idf score calculation. Finally, the sorting stage can also be performed in parallel using bitonic search[1].

5.3.1 Experimental Validation

Implementing the above procedure in CUDA[2], and pre-loading an efficient representation of the dataset bloom filters into the VRAM prior to searching, we investigated the performance characteristics of this scheme over different lengths of query tokens over the Wikipedia dataset (see Figure 1).

For this experiment we limited consideration to a filter hierarchy constructed with fixed false positive rates ($p = 0.05$ for the presence filter, and $p = 0.01$ for each subsequent occurrence level) i.e. for every document the index consisted of a token bigram presence filter with a 5% false positive rate, and 3 additional filters representing at least 1 (level 1), at least 2 (level 2), and at least 8 (level 3) occurrences of the token bigram. This resulted in an index size of 9GB (both on-disk and in VRAM)[3]. Tuning these parameters for storage efficiency and search efficacy is likely an interesting avenue for future research.

To source test queries we randomly selected between 2 and 16 consecutive tokens from $10,000$ of

[2]All experiments were done on a low end NVIDIA GeForce RTX 3060 with 12 GB of VRAM.

[3]We note that, at these parameters, the size of the index is approximately the size of the uncompressed, formatting-stripped text representation of the page contents.

Query Token Length	**2**	**4**	**8**	**16**
Mean Query Time	38ms	72ms	152ms	320 ms
Mean Query Time per Query Length	19ms	18ms	19ms	20ms
Mean Ranking of Found Documents	22	10	3	1
Miss Rate	0.774	0.320	0.056	0.015

Figure 1: Performance characteristics of our CUDA prototype searching over $|N| = 6325358$ documents for various lengths of input queries ($|Q| = 10000$) . A **miss** is defined as any search where the target document was not returned in the first 100 ranked results.

the source documents (wiki pages), and used the selected text as a query; we would typically expect the source document to appear high in the rankings for such a query [4]

As the query grows we observe the total time of a search increases (but the mean time per query token stays constant at around 19 milliseconds / query); additionally as query length grows we observe that the efficacy of the search improves, by the time we reach query lengths of 16 tokens, the source document of the query tokens is typically always the first ranked result.

5.4 Possible Extensions / Future Work

In this section we present a select number of idea that require further investigation and/or experimental validation.

5.4.1 Mip Map Search

As stated above, there is an explicit tradeoff between false positives and the size of a document index. Further, the smaller a document index is, the more likely it is to match against any given query.

This does however suggests a "mipmap" approach where a tiny index is first computed against to fast-reject from the complete corpus, and then a more expensive index is used to filter the remaining documents.

[4]This is not always the case as there is a non-zero chance of randomly selecting a set of tokens that match a generic phrase appearing across many documents e.g. "is a species of" or "was born in" etc. We make no attempt to filter out such generic terms.

5.4.2 Page-Rank Post-Analysis

For a full application to web search, we need to take into account more factors than full text matching e.g. if searching for "Alan Turing" (which in token bigrams may be represented by something like "Al-ant-uring") - the results returned lean towards documents that mention Alan Turing (e.g. the Alan Turing Memorial, Turing award winners etc.) rather than articles about Alan Turing (which may very rarely use their full name).

One approach that may help is to apply a page-rank[11] like analysis to documents *after* the initial filtering (on the assumption that documents *mentioning* Alan Turing are more likely to link to documents *about* Alan Turing).

5.4.3 Tuning Parameters

In our experimental validation we only considered one possible parameterization of the search index. It is unlikely that this parameterization is the most efficient in any measure. More work is needed to understand the best structure for the bloom filter levels (the number levels of occurance filters / the occurrence threshold be for each level etc.) and the false positive rate assigned to each level (and whether such rates should be static for each level, or dynamically assigned per document).

6 Conclusion

We present a method for building a searchable index of documents through a combination of subword tokenization and bloom filters, and a method for efficiently (and probablistically) providing full text search over this index in a way that is significantly parallelizable.

Additional research is needed to understand how to best parameterize this structure.

References

[1] Ranieri Baraglia, Gabriele Capannini, Franco Maria Nardini, Fabrizio Silvestri, et al. Sorting using bitonic network with cuda. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, 2009.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*, pages 684–695. Springer, 2006.

[4] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False negative problem of counting bloom filter. *IEEE transactions on knowledge and data engineering*, 22(5):651–664, 2010.

[5] Navendu Jain, Mike Dahlin, and Renu Tewar. Using bloom filters to refine web search results. In *Eighth International Workshop on the Web and Databases (WebDB'05)*, 2005.

[6] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[7] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[8] David Lillis, Frank Breitinger, and Mark Scanlon. Hierarchical bloom filter trees for approximate matching. *arXiv preprint arXiv:1712.04544*, 2017.

[9] David Lillis, Frank Breitinger, and Mark Scanlon. Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees. In *Digital Forensics and Cyber Crime: 9th International Conference, ICDF2C 2017, Prague, Czech Republic, October 9-11, 2017, Proceedings 9*, pages 144–157. Springer, 2018.

[10] Svenja Mehringer, Enrico Seiler, Felix Droop, Mitra Darvish, René Rahn, Martin Vingron, and Knut Reinert. Hierarchical interleaved bloom filter: enabling ultrafast, approximate sequence queries. *Genome Biology*, 24(1):131, 2023.

[11] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bring order to the web. Technical report, Technical report, stanford University, 1998.

[12] Michael A. Shepherd, William J Phillips, and C-K Chu. A fixed-size bloom filter for searching textual documents. *The Computer Journal*, 32(3):212–219, 1989.

[13] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.