CHALLENGES OF INDEX EXCHANGE FOR SEARCH ENGINE INTEROPERABILITY

Djoerd Hiemstra, Gijs Hendriksen, Chris Kamphuis, and Arjen P. de Vries*
Radboud University, The Netherlands

Abstract

We discuss tokenization challenges that arise when sharing inverted file indexes to support interoperability between search engines, in particular: How to tokenize queries such that the tokens are consistent with the tokens in the shared index? We discuss various solutions and present preliminary experimental results that show when the problem occurs and how it can be mitigated by standardizing on a simple, generic tokenizer for all shared indexes.

INTRODUCTION

Web search is dominated by a small number of giant corporations that effectively hold a monopoly on web search. Quite worryingly, these corporations control almost every aspect of web search: They crawl the Web, they build the index, they provide the actual search results given a query, they sell advertisements, they provide free web analytics to get usage statistics, they even own the web browsers and operating systems that we need to use their search engines.

In our opinion, a single corporation should not control a large share of these aspects of search. For instance, incentives for providing high quality search results do not align with incentives to sell advertisements, or to quote Brin and Page [2]: *"The goals of the advertising business model do not always correspond to providing quality search to users (...) we expect that advertising funded search engines will be inherently biased towards the advertisers and away from the needs of the consumers."*

Two important solutions may help break up these monopolies. One is regulation: It should not be legal to run an advertisement company and a search engine, nor should it be legal to own large web sites as well as the web browser that renders them. The second solution (that might help enable the first) is technical: We should create tools that enable collaboration between multiple organizations, so they can develop web search engines together. This paper focuses on a solution of the second, technical, kind. Specifically, we discuss the challenges of defining open standards that support interoperability between search engines to enable organizations to build web search engines collaboratively.

Building a web-scale search engine is a challenging task. Crawling the web takes a lot of resources, as does building the inverted index. Once the index is ready, however, running queries on the index can be done with relatively little compute power. We envision a future where organizations collaboratively build a search engine by using open standards that define the results of each step [6]. We show these steps in Figure 1. The first step is to (collaboratively) crawl the Web and provide it in a standard format, such as the Internet Archive's Web Archive (WARC) format [13]; In Step 2, others may build an inverted index to be provided in the standard Common Index File Format (CIFF) [11]; In Step 3, yet others take the index and build the search engine (backend) provided as an API based on the OpenSearch standard [5]; which is finally used in Step 4 by the organization that builds the search application (frontend).

This paper discusses the challenges of using the common index file format CIFF in Step 2. We discuss CIFF and why it is currently underspecified for the use in a production search engine in the following section. In the final section, we present preliminary experiments that demonstrate the problem in practice, provide a generic solution and report on experiments showing its adequacy. The code used to run our experiments is available via a public Git repository[1].

THE COMMON INDEX FILE FORMAT

The Common Index File Format (CIFF) was defined in 2020 by researchers and developers of the following open source retrieval research systems: Terrier, Anserini (which uses Lucene), PISA, JASSv2, and OldDog [11]. CIFF's goal is to improve the reproducibility of information retrieval experiments by allowing a search system to export the inverted index and import it into another system. This way, researchers can rule out differences in retrieval performance that are caused by building the index, such as text preprocessing and boilerplate removal, focusing solely on other aspects, such as the ranking algorithm.

Tokenization challenges of CIFF

To build an inverted index, the document texts need to be tokenized. When the CIFF index is used in another system, the queries that are put to the system, should use the tokens in the index. One of the main challenges of using CIFF in practice is the following: How do we tokenize the query such that the tokens are consistent with the tokens in the index? To ensure consistent tokenization between the indexer and searcher, Lin et al. [11] exchanged pre-tokenized versions of the test queries (also known as topics). However, in a production search engine, there is no way of knowing all possible queries beforehand, let alone pre-tokenizing them. Consistent tokenization between index and queries remains an unsolved problem of index exchange, at least outside the narrow scope of information retrieval experiments that use benchmark test collections with a small set of test queries. We discuss tokenization in information retrieval and three

* {djoerd.hiemstra, gijs.hendriksen, chris.kamphuis, arjen.devries}@ru.nl

[1] https://opencode.it4i.eu/openwebsearcheu-public/index-sharing

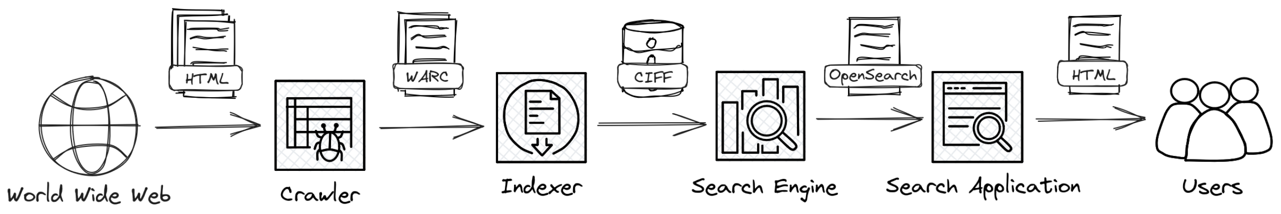

Figure 1: Steps in building and running a search engine. In between each step, data is exchanged conform to a specific open standard: HTML for web content; WARC for web archives; CIFF for inverted files; and OpenSearch for search results.

possible approaches to achieve tokenization consistency for CIFF below.

Tokenization in information retrieval

There is surprisingly little research done into tokenization for information retrieval. A good overview is given by Büttcher et al. [3, Chapter 3]. For English and some other western languages, a simple tokenizer that splits on space and punctuation usually suffices. Sometimes a stop word list is used to remove common words. Often several surface forms are mapped to the same token, for instance acronyms might be written as *EU* or as *E.U.* and the tokenizer may map them to the same token. For inflective languages, a stemmer such as Porter's stemmer for English [17] could map many different inflections of the same word to a common root, for instance *indexing*, *indexation* and *indexes* will all be mapped to a common root: *index*. Instead of a stemmer, the use of letter n-grams has been shown to be surprisingly effective for inflective languages [12], but an n-gram index is less efficient as queries will have more tokens for which then (longer) posting lists need to be fetched and merged.

Non-Western languages like Chinese and Japanese use many more characters than Western languages. Chinese has thousands of distinct characters. Even though each character has a meaning by its own, lots of words consist of multiple characters and those words are not separated by spaces. Indexing Chinese documents therefore requires a non-trivial word segmentation algorithm [22]. Consistent tokenization for an imported index may therefore be a bigger problem for Chinese than for English. The Unicode consortium provides extensive guidelines for text segmentation for many other languages [4].

Our solution to the tokenization problem should support all human languages and at least general approaches like stop words and mapping multiple surface terms to the same token (including stemmers).

Possible tokenization solutions

How do we make sure that query tokenization is done in a way that is consistent with the imported index? In this Section, we discuss possible solutions, including: shipping the tokenization source code with the CIFF index; providing a declarative specification of the tokenizer with the CIFF index; and defining a generic tokenizer that works with any CIFF index.

Including the tokenizer source code Providing the tokenizer code inside (or with) the CIFF index would solve the tokenizer inconsistency problem, but it also creates several new questions and problems. One is: What programming language should be used? Terrier, Lucene and Anserini use Java. PISA and JASSv2 use C++. Another problem is that each index may come with their own tokenizer, so the number of tokenizers that need to be shared would possibly increase with every CIFF index. This would make CIFF an easy target of software supply chain attacks, where malicious code is injected into tokenizers. To conclude, including the tokenizer source code into CIFF would require the CIFF developers to agree on a programming language for tokenizers, and it would require a high level of trust into the shared code.

Tokenizers in embeddable scripts or bytecode Security concerns of including the code of tokenizers can be partly met by using an embeddable scripting language like Lua that is designed to run inside applications in a carefully guarded sandbox. Other options would be to use JavaScript or WebAssembly, that are both used in web browsers and therefore heavily guarded against malicious use.

Declarative tokenizers Another option may be to use the lexers of parser generators like ANTLR [16] as a tool for specifying a tokenizer and generating the tokenizer for many programming languages, including the ones mentioned above. Parser generators are used to parse programming languages and possibly structured query languages. They are used in search engines that require complex structured queries, such as Lemur [14] and PF/Tijah [8] (both are research systems that are no longer maintained). Specifying tokenizers this way seems to be a non-trivial, tedious job. We have not investigated this option further.

A single, generic tokenizer CIFF comes with the complete dictionary containing all possible tokens as part of the inverted file. The query tokenizer may therefore adapt to the dictionary, ensuring that the query is tokenized in a way that best fits the imported index. We will pursue this solution in the next section.

A generic CIFF tokenizer

Recent advances in neural machine translation and large language models come with interesting developments for tokenization. Their tokenizers use a relatively limited vocabulary by splitting uncommon words into word pieces. This is done for two reasons: *1)* to speed up processing and decrease the number of parameters to be trained; and *2)* to gracefully handle out-of-vocabulary words, which will occur in unseen data no matter how big of a vocabulary the model uses. Word piece models are trained on the data to find the best word piece tokenizer for that data [10, 18]. So, these tokenizers are generic tokenizers, that are trained or finetuned on data. The trained vocabulary, possibly with additional frequency information, and a generic tokenization algorithm define the tokenizer.

We will use this idea of trained tokenizers to define a generic tokenizer for CIFF. In this analogy, the indexing step is the training step. The index, encoded in CIFF, contains the possible tokens to be used by the generic tokenizer algorithm. This way, every CIFF index will use its own custom tokenizer, without the need to share the tokenizer source code or bytecode. Unlike the word piece tokenizers mentioned above, the CIFF tokenizer will typically use a much larger vocabulary, although it could as easily use word pieces or, alternatively, use multi-word units for phrases or named entities. The generic tokenizer is completely language-agnostic: It works on any unicode string and does not need to know about (English or Chinese) character sets. It does not even know about spaces or punctuation. The generic tokenizer may be implemented in about 20 lines of code. Example code for the generic tokenizer is included in the appendix. Its efficiency may be further improved by building a trie from the vocabulary [19].

PRELIMINARY EVALUATION OF TOKENIZATION FOR CIFF

In this section we show preliminary evaluation results using the TREC Robust 2004 dataset [20]. We made inverted indexes for two search systems: GeeseDB [9] and Terrier [15]. Each index was exported to CIFF and imported in the other system. We then evaluate *1)* the original system, *2)* the other system, tokenizing queries with its standard tokenizer, and *3)* the other system with the generic tokenizer. We show that performance degrades when a mismatch in tokenization occurs, but that this can be mitigated by using the generic tokenizer.

For both GeeseDB and Terrier, we rank documents with BM25, using the parameters $b = 0.4$ and $k_1 = 0.9$. Stop words are *not* removed from the corpus, and no stemming is applied. We discuss both techniques, and how they can be handled in CIFF, in more detail in our section below on future plans.

GeeseDB is configured to use the NLTK tokenizer [1], and Terrier uses its standard internal tokenizer. These tokenizers mostly differ in how they handle certain types of punctuation. For instance, NLTK leaves tokens intact when they contain hyphens or periods (like *on-line* or *U.S.*), while Terrier will split these into multiple tokens. To highlight these differences, we additionally run our experiments on the subset of only those Robust04 topics that contain a hyphen or period (19 topics in total).

In all our experiments, we test whether differences are statistically significant by applying a two-tailed paired t-test. We use a significance level of $\alpha = 0.01$.

Results on the Terrier index

Table 1a shows our results on the Terrier index.[2] We see that performance significantly degrades if we use the Terrier inverted file in GeeseDB out of the box. However, once we apply the generic CIFF tokenizer, we are able to correctly adapt GeeseDB to the tokenization used by Terrier. In fact, using the Terrier index in GeeseDB with the generic tokenizer seems to match (or even slightly surpass) the performance of Terrier itself.

These results are even more apparent when looking at Table 1b, where we zoom in on the topics that contain hyphens or periods. There is a very large performance drop when we use GeeseDB with the NLTK tokenizer for query preprocessing, but this drop disappears when using the generic tokenizer.

Table 1: Performance of different systems using an inverted file generated with Terrier. Best results are marked in bold. Configurations that perform significantly ($p < 0.01$) better than the Terrier index imported into GeeseDB (the middle row) are marked with †.

(a) All Robust04 topics

System	Tokenizer	MAP	nDCG
Terrier	Terrier	0.221†	0.480†
GeeseDB	NLTK	0.208	0.457
	CIFF	**0.224**†	**0.482**†

(b) Robust04 topics that contain a hyphen or period

System	Tokenizer	MAP	nDCG
Terrier	Terrier	**0.234**†	**0.541**†
GeeseDB	NLTK	0.081	0.292
	CIFF	**0.234**†	**0.541**†

Results on the GeeseDB index

Table 2 shows the results of our experiments with the GeeseDB inverted file. The differences are not as large (or significant) as they were with the Terrier index, but we still notice a drop in performance for queries with hyphens or periods (Table 2b). Again, this drop can be mitigated by using the generic tokenizer.

Discussion

Our preliminary experiments on Robust04 indicate that our proposed generic tokenizer could be a useful addition

[2] Systems that use a similar tokenizer, like Anserini/Lucene, give similar experimental results.

Table 2: Performance of different systems using an inverted file generated with GeeseDB. Best results are marked in bold. Configurations that perform significantly ($p < 0.01$) better than the GeeseDB index imported into Terrier (the middle row) are marked with †.

(a) All Robust04 topics

System	Tokenizer	MAP	nDCG
GeeseDB	NLTK	0.207	0.460
Terrier	Terrier	0.208	0.460
	CIFF	**0.209**	**0.462**

(b) Robust04 topics that contain a hyphen or period

System	Tokenizer	MAP	nDCG
GeeseDB	NLTK	0.155	0.433
Terrier	Terrier	0.145	0.392
	CIFF	**0.183**	**0.474**

to the CIFF standard for inverted files. The results show us that a retrieval system that is tokenized by using a greedy matching approach on the inverted file's dictionary is able to match the performance of the system in which the inverted file was created.

We also see that the tokenizer used to build the inverted file matters in terms of retrieval effectiveness. The Terrier index seems to consistently result in higher performance than the GeeseDB index. To optimize the effectiveness of a system using CIFF files, we would need to look at which tokenizers produce the most useful dictionaries and inverted files – knowing that the downstream system will use the generic tokenizer.

CONCLUSION AND FUTURE PLANS

We discuss challenges of using the Common Index File Format (CIFF) as an open standard for index exchange between search engines. We propose a generic CIFF tokenizer that ensures that the tokenization of queries is consistent with the tokenization that was used to make the CIFF index, without the need to exchange the tokenizers themselves.

The CIFF generic tokenizer

Preliminary experimental results with two search systems confirm that exchanging a CIFF index without properly handling tokenization may result in a significant drop in search quality. This drop happens for instance when a token is split in multiple pieces at index time (like when *on-line* is indexed as *on* and *line*) but in one piece at query time (*on-line*, which then cannot be found in the index). Experimental results show that our our generic CIFF indexer fixes this without explicit knowledge of the pre-processing pipeline of the source system used to create the index. Together with developers of search engines, we hope to define a new version of CIFF that includes the generic tokenizer and that will work for all languages and a large range of search applications. We plan to work on the issues discussed below to make this happen.

Towards CIFF version 2

For a version 2 of the CIFF standard, we need more experimentation and several other adaptations for stop words, stemming and, possibly, incremental indexing.

Indexing for non-western languages We plan to do experiments for non-western datasets such as Chinese and Japanese. Tokenization problems may be much more common for these languages, because they do not distinguish words using spaces.

Stop words To allow the proper handling of stop words and other tokens that are not indexed, CIFF indexes should include them in the index with an empty posting list. If the stop words themselves are not included, they will not be properly tokenized by the CIFF tokenizer (but split up in shorter tokens that are included in the index).

Using stemmers Supporting stemming in CIFF is not entirely trivial. The CIFF tokenizer only works on surface forms of the tokens, not on derived tokens like stems. One way for CIFF to support stemming, is to include all surface tokens that are found in the data into the CIFF index, grouping them for each stem. So, every posting list could come with multiple tokens. For instance, the posting list for the stem *pickl* (Stemmers do not always produce linguistically correct stems) will in CIFF contain a set of tokens: *pickle*, *pickled*, *pickles*, and any other word that stems to *pickl*. This approach may be used for many other tokenization challenges too, like conflating the acronyms *EU* and *E.U.* as mentioned above. Furthermore, it allows for more flexible stemmers, including stemmers that use lookup tables and corpus-based stemmers [21]. The approach will make it harder to export any index to CIFF, because the surface tokens need to be retained somewhere. It will also give different results for query terms that do not occur once in the entire data, but of which the stem does occur. Nevertheless, we believe including sets of surface terms is an elegant solution for derived tokens, including the use of stemmers.

Open indexes for large-scale web search Large web indexes contain multiple languages and several kinds of data. This opens up the possibility to use different tokenizers depending on the language of the document, or even depending on the language of paragraphs in multilingual documents. This setup would introduce new challenges: For instance, if multiple surface terms are grouped for one language, as described above, they will have to be grouped for all languages. Experiments will have to show how the generic CIFF tokenizer would handle such a setup.

Index updates A large CIFF web index may be many gigabytes or even some terabytes in size. Additionally, the Web changes all the time, so it makes sense to share a large

CIFF index once and share updates for the index regularly afterwards. We think CIFF should include some rudimentary way to indicate updates of the index.

Open source implementations A standard is more resilient if there are multiple implementations. We implemented several tools for CIFF, including a CIFF importer for Lucene [7], and tools for merging multiple smaller indexes into one bigger index.

ACKNOWLEDGMENTS

This work is funded by the EU project OpenWebSearch.eu under GA 101070014. We are grateful for the EU's support.

REFERENCES

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, 2009.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30:107–117, 1998.

[3] Stefan Büttcher, Charles Clarke, and Gordon Cormack. *Information retrieval: Implementing and evaluating search engines*. MIT Press, 2010.

[4] Mark Davis. Unicode standard annex 29: Unicode text segmentation (revision 41). Technical report, Unicode Consortium, 2022. http://www.unicode.org/reports/tr29/.

[5] Clinton DeWitt. *What is the OpenSearch protocol?* A9.com, 2005. https://github.com/dewitt/opensearch.

[6] Michael Granitzer, Stefan Voigt, et al. Impact and development of an open web index for open web search. *Journal of the Association for Information Science and Technology*, 2023.

[7] Gijs Hendriksen, Djoerd Hiemstra, and Arjen de Vries. Lucene CIFF importer. Zenodo, 2023. https://doi.org/10.5281/zenodo.8261333.

[8] Djoerd Hiemstra, Henning Rode, Roel van Os, and Jan Flokstra. PF/Tijah: text search in an XML database system. In *Proceedings of the 2nd International Workshop on Open Source Information Retrieval (OSIR)*, pages 12–17, 2006.

[9] Chris Kamphuis and Arjen de Vries. GeeseDB: A Python graph engine for exploration and search. In *Proceedings of the 2nd International Conference on Design of Experimental Search and Information REtrieval Systems (DESIRES)*, 2021.

[10] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[11] Jimmy Lin, Joel Mackenzie, Chris Kamphuis, Craig Macdonald, Antonio Mallia, Michał Siedlaczek, Andrew Trotman, and Arjen de Vries. Supporting interoperability between open-source search engines with the common index file format. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2149–2152, 2020.

[12] Paul McNamee and James Mayfield. Character n-gram tokenization for European language text retrieval. *Information retrieval*, 7:73–97, 2004.

[13] Gordon Mohr, John Kunze, and Michael Stack. The WARC file format 1.0 (ISO 28500). Technical report, International Organization for Standardization, 2008. https://escholarship.org/uc/item/9nh616wd.

[14] Paul Ogilvie and Jamie Callan. Experiments using the Lemur toolkit. In *Proceedings of the 10th Text Retrieval Conference (TREC)*, pages 103–108, 2001.

[15] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Douglas Johnson. Terrier information retrieval platform. In *Proceedings of the 27th European Conference on IR Research (ECIR)*, pages 517–519, 2005.

[16] Terence Parr and Russell Quong. ANTLR: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[17] Martin Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[18] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

[19] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast wordpiece tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2089–2103, 2021.

[20] Ellen Voorhees. The TREC robust retrieval track. *ACM SIGIR Forum*, 39(1):11–20, 2005.

[21] Jinxi Xu and Bruce Croft. Corpus-based stemming using cooccurrence of word variants. *ACM Transactions on Information Systems*, 16(1):61–81, 1998.

[22] Nianwen Xue. Chinese word segmentation as character tagging. *International Journal of Computational Linguistics & Chinese Language Processing*, 8(1):29–48, 2003.

APPENDIX

Example Python code for the generic tokenizer.

```
def tokenize_generic_greedy(self, query):
    """ Tokenize a query generically.
        self.dictionary contains all tokens from
        the inverted file.
        self.max_token_length contains the length
        of the longest token.
    """
    tokens = []
    begin = 0
    query_length = len(query)
    while begin < query_length:
        end = begin + self.max_token_length
        if end > query_length:
            end = query_length
        token_found = None
        while begin < end:
            token = query[begin:end]
            if token in self.dictionary:
                token_found = token
                break
            end -= 1
        if token_found:
            tokens.append(token)
            begin = end
        else:
            begin += 1
    return tokens
```