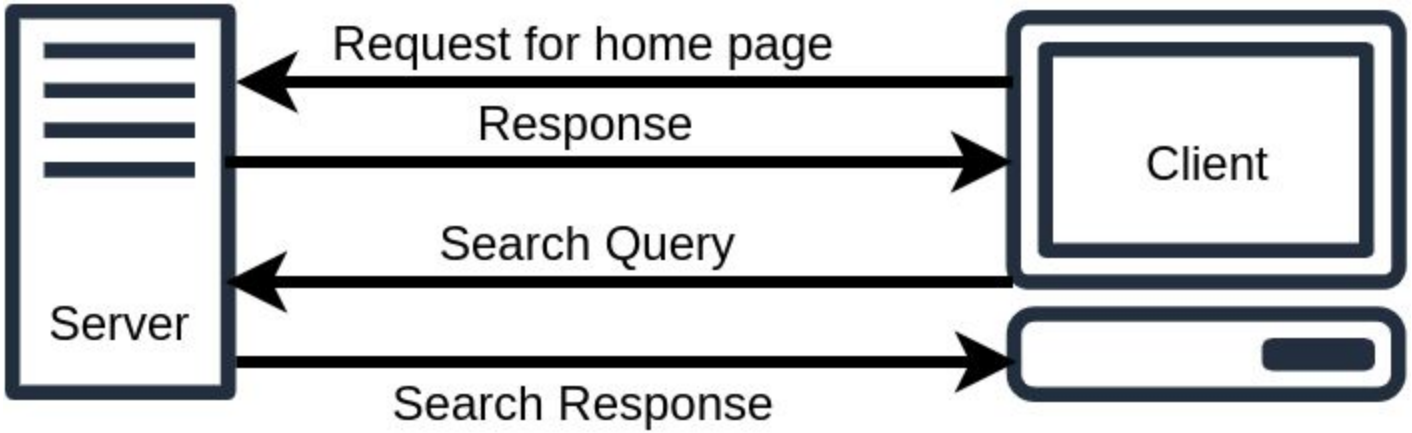
Spectral Bloom Filters for Client Side Search

Parth Parikh, Mrunank Mistry, Dhruvam Kothari, Sunil Khachane

Table Of Contents

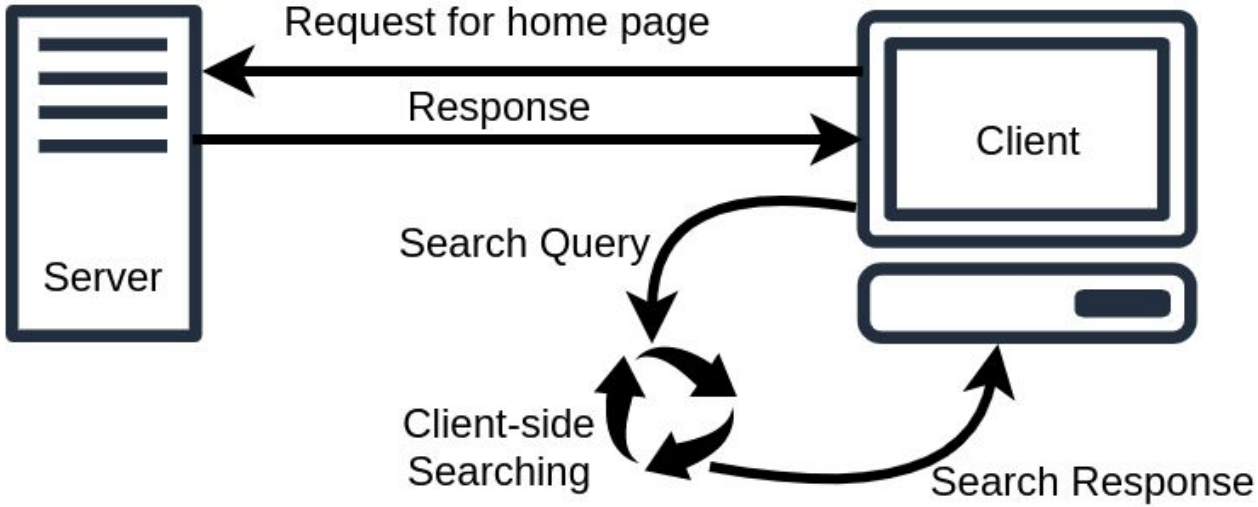
Introduction

Traditional Server Side Search



Client types his/her search query on the search box of a website and the search request is sent to the server. The server performs the search in a database, ranks the results and provides the relevant results to the client in the form of a response.

What is Client Side Search ?



Client Side Searching completely eliminates the need for a request-response trip to the server for search queries. The search query is processed entirely on the client side and the results are sorted by relevance.

How does Client Side Search work?

- Client-side search is most relevant to ***static websites***, wherein a search query is entirely processed without sending any request to a server (unlike server-side websites). This is particularly useful for documentation, blogs, online portfolios, etc.
- Along with the actual markup of the search page of the website, the entire search index has to be sent to the client-side which in turn enables searching.
- *For example*, libraries like Lunr.js employ **Inverted Index** based solutions, wherein, an inverted index is developed and sent to the client-side. In order to serve a search query, the words in the query are looked-up in this index and the links of the most relevant documents are then returned.

The Memory-Accuracy Tradeoff

- Minimizing the size of the search index/page is extremely vital.
- Inverted index solution does guarantee 100% accurate term frequencies. However, the memory usage blows up quite quickly with increase in document size and the number of documents itself.
- Large search pages can potentially over-burden the client-side browser. Also, it consumes larger bandwidth during its transfer from server to client.
- Consequently, user experience is affected. Inverted Index clearly causes major issues for large scale applications.

The Memory-Accuracy Tradeoff

- Using Cuckoo filters is an alternative solution to reduce the memory requirements. They are probabilistic data structures which can be used for storing term frequencies.
- They consume lesser space as compared to Inverted Index.
- However, this space reduction comes at an expense of accuracy, i.e. frequency estimates provided by Cuckoo filters are subject to false-positive errors.
- The library - **Tinysearch** uses Cuckoo filters for performing client side search.

Problem Statement

What are we aiming for?

- Given a number of documents (static web-pages), we aim to build a system which is capable of providing a ***client side search facility*** for the user in a ***memory efficient*** way.
- When queried, the system should be able to ***rank the documents*** according to the term frequencies of the words present in the search query.
- The system should also allow the user, the ability to trade minute accuracy for considerable reduction in memory consumption. This is desirable when memory efficiency is most essential and a few false positive errors are permissible.

Proposed Approach

Preprocessing

- We need to keep in mind that preprocessing done on the server side (generating search page) must also be done at client side.
- We tokenize each document by removing all punctuations and HTML tags involved to get a list of words or tokens.
- From these tokens, we filter out stopwords. Stopwords are words like *the, that, it*, etc. which are common across all English language and contribute little to the semantic structure of text.
- Eg: *<h1> Wow! What a goal. <h1> → {wow, what, goal}*

Bit Arrays

- For performance reasons, many computer languages implement boolean as 1 or more bytes.
- Hence, for every bit, we are wasting 7 bits of storage.
- Bit array is a data structure which solves this problem. We store 8 bits as 1 character and apply bitwise/arithmetic operation to manipulate the bit values underneath.
- This reduces the space required by a factor of 8 at the cost of some additional overhead.

Paper: Burton H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: Communications of the ACM (1970).

Bloom Filters

- A Bloom Filter is a probabilistic data structure used to test set membership in situations where we don’t want to explicitly store the elements of the set.
- The idea is to hash the tokens k times and set the corresponding bits as pointed by the hash function.
- Bloom filters can have false positive and no false negatives.
- Given a set ***S*** with ***k*** independent and uniform hash functions $\boldsymbol{h_1, \ldots, h_k}$, for each element $e \in S$, Bloom filter constructs a ***m*** bit array (initialized with zeros) by setting k bits $(h_1(e) \bmod m), \ldots, (h_k(e) \bmod m)$ to 1.
- To test if an element $x \in U$ is a member of the set S, the positions at $h_1(e),\ldots,h_k(e)$ are checked. If at any position, the bit is 0, x is not a member of set S. However, if all the bits are set, x may or may not be a member of set S. This is to say that Bloom Filters can have false positives.
- $$k = \frac{m}{n} \ln 2 \qquad m = -\frac{n \ln E}{(\ln 2)^2} \qquad E = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

Paper: John Byers et al. “Informed Content Delivery Across Adaptive Overlay Networks”. In: In Proc. of ACM SIGCOMM. 2002.

Counting Bloom Filters

- The basic bloom filters mentioned in the previous slide can only be used for performing set membership tests. They cannot store term frequencies of the words present in the documents.
- Hence, the documents cannot be weighted by relevance.
- Instead of allocating a single bit for every element of the array of size m, counting bloom filters use an array of m counters $C_0, C_1, \ldots, C_{m-1}$ where every counter has a width of w bits.
- To insert an item $e \in S$ in the Spectral Bloom Filters we simply increment the value of counters $C_{h1(e)}, C_{h2(e)}, \ldots, C_{hk(e)}$ by 1.

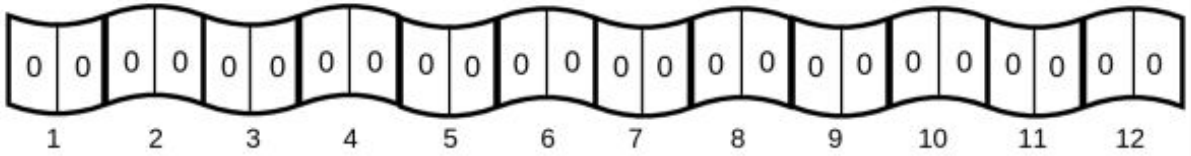
Paper: Saar Cohen and Yossi Matias. “Spectral Bloom Filters”. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. SIGMOD ’03.

Spectral Bloom Filters

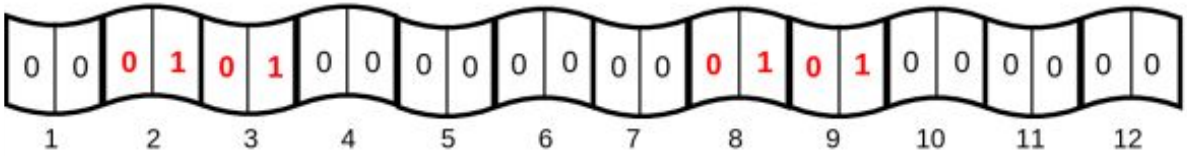
- Spectral Bloom Filter (SBF) is space efficient probabilistic data structure that can be used to estimate the multiplicity of an element in a multiset. It does this without explicitly storing the elements at the cost of some false positives.
- We use SBF as an improvement over the Inverted Index method to store the term frequencies of a document in a memory efficient manner.
- For each document, we index the tokens into a SBF and coalesce all such SBF’s to generate a search index which is embedded in the search page.
- For performing searches, we query a document’s SBF to get frequency of the tokens and rank them.

Example of Minimum Selection and Minimal Increase

Document: "apples are apples; lemons are apples"
Tokens: apples, are, apples, lemons, are, apples



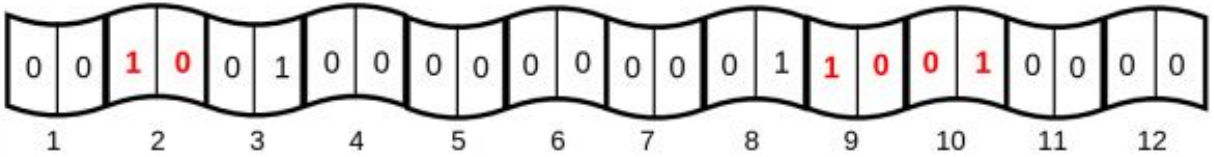
Using *k=2*, with seeds *10, 13* and hash algorithm as *Murmurhash3*
Inserting first two tokens - *apples* and *are*
$C(h_1(apples))$ = 9, $C(h_2(apples))$ = 2
$C(h_1(are))$ = 3, $C(h_2(are))$ = 8



Currently, the above filter can be interpreted as - {0,1,1,0,0,0,0,1,1,0,0,0}
Inserting next two tokens - *apples* and *lemons*
$C(h_1(apples))$ = 9, $C(h_2(apples))$ = 2
$C(h_1(lemons))$ = 2, $C(h_2(lemons))$ = 10
Will there be a Collision?

Example of Minimum Selection and Minimal Increase

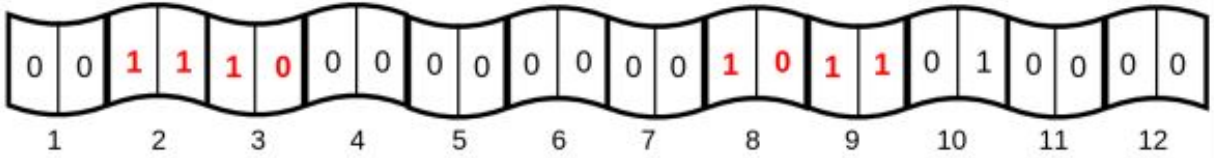
No! As we are using the **minimal increase** algorithm, C(2) will be 2 (after *apple* token's insertion), whereas C(10) will be 0. Therefore, we increment C(10) (the minimum value). Had C(2) been 0, we would have incremented both C(2) and C(10).



Inserting last two tokens - *are* and *apples*

$C(h_1(are)) = 3$, $C(h_2(are)) = 8$

$C(h_1(apples)) = 9$, $C(h_2(apples)) = 2$



Done! Now lets perform a **query** operation using **minimum selection** algorithm.

Frequency(*lemons*) = min[$C(h_1(lemons))$, $C(h_2(lemons))$]

= min[*C(2)*, *C(10)*] = min[3, 1] = **1**

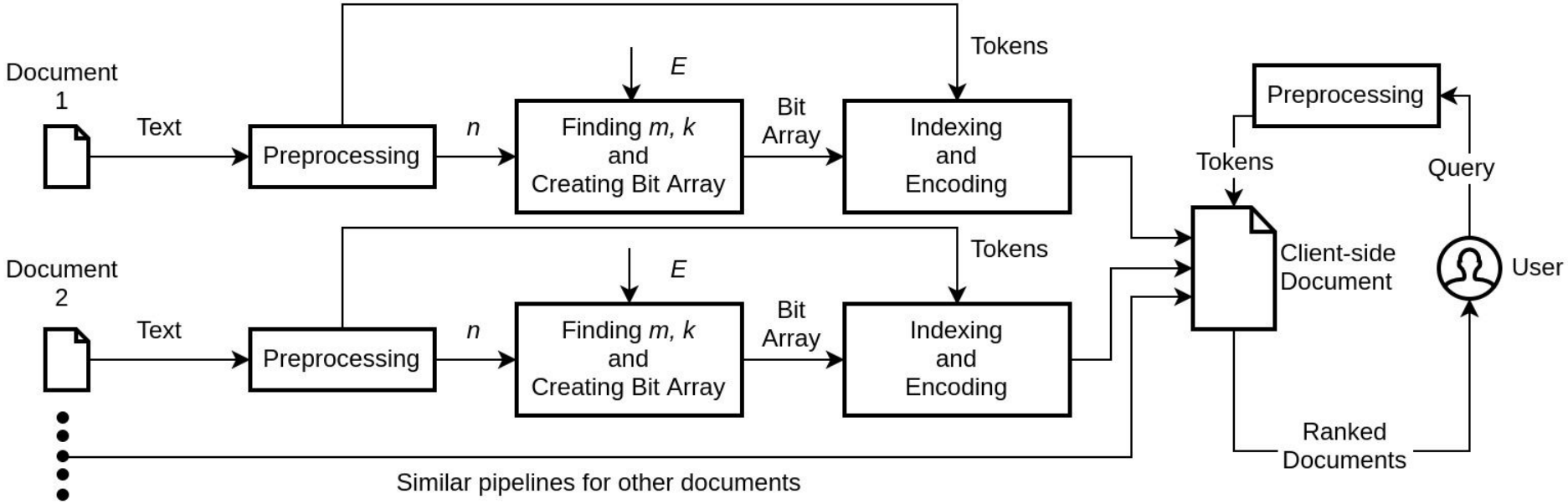
Encoding and Decoding

- The task here is to encode the given bit array into an HTML document.
- We iterate over chunks of *15* bits, and map them to a bijection of UTF-16 unicode values.
- If the input is not a multiple of 15 bits, we pad it with zeroes and store a *HEX* value as the first character, which denotes the number of padded bits.
- This encoded unicode string *encodes* a SBF. Think of it as *base32768* encoding.
- This encoding/decoding scheme is very efficient and gives about **94%** efficiency (calculated as ratio of *input bits* to *output bits*).
- Additionally, the string doesn't need to be *decoded* we can read any bit by working with its encoded representation.

Processing Queries at Client Side

- We apply the same preprocessing we applied on server-side to our search query for obtaining tokens.
- For a given token, we can get its frequency estimate in a document from the corresponding SBF
- We do this for all tokens in the search query for all documents to get the frequencies.
- Once we have the frequencies, we can apply an appropriate ranking algorithm such as *TF-merge*, *TF-IDF*, *Okapi BM25*, etc. to rank the documents.

Pipeline


Document 1
Text
Preprocessing
n
E
Finding m, k and Creating Bit Array
Bit Array
Indexing and Encoding
Tokens
Document 2
Text
Preprocessing
n
E
Finding m, k and Creating Bit Array
Bit Array
Indexing and Encoding
Tokens
Similar pipelines for other documents
Tokens
Client-side Document
Preprocessing
Query
User
Ranked Documents

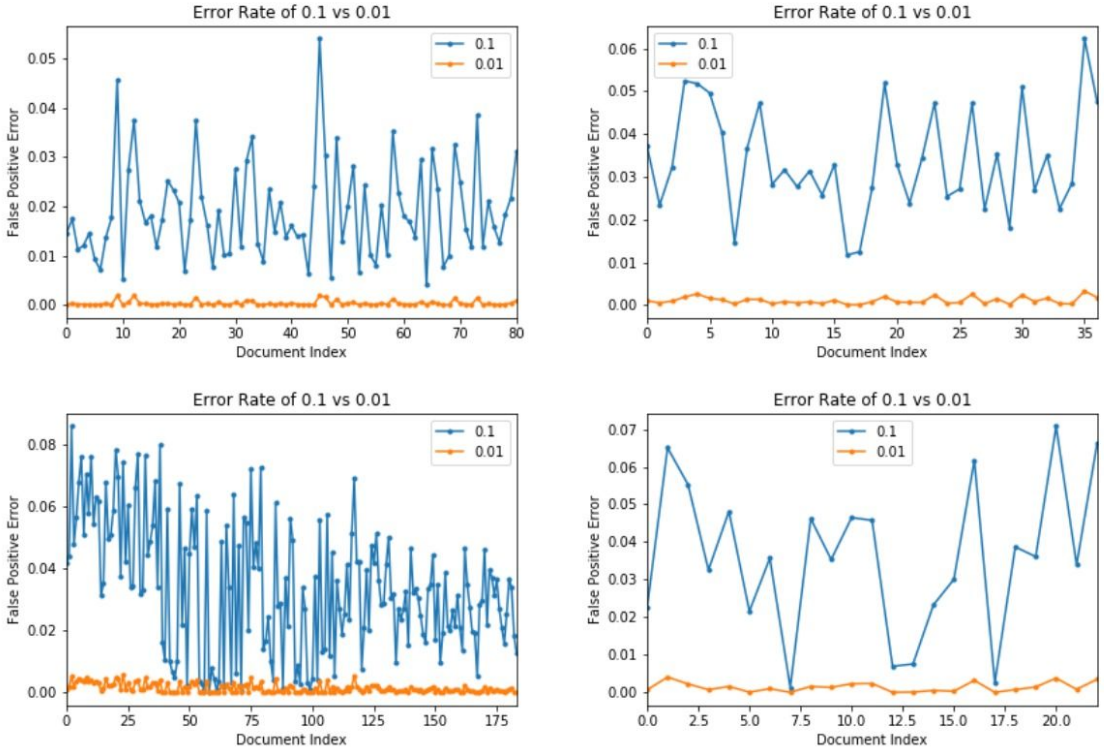
Results

Background

The testing was primarily conducted using blog-posts from four different blogs:

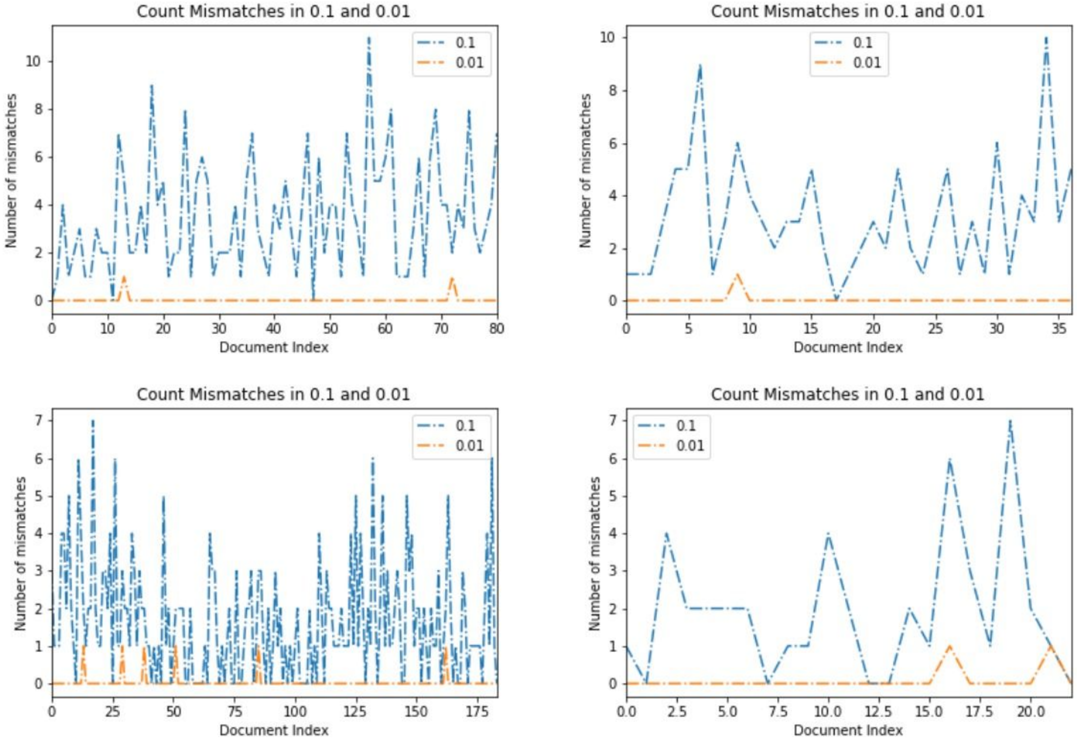
- ***danluu***: This blog contains 81 blog-posts with total blog size of 2.1 MB.
- ***endler***: This blog contains 37 blog-posts with total blog size of 1.4 MB. The author of this blog conceived *Tinysearch*, which is a Cuckoo Hashing based library to search static sites on the client-side.
- ***parth***: Maintained by the co-author of this paper, this blog contains 23 blog-posts with total blog size of 1.3 MB.
- ***matthewdaly***: This blog it the largest of all the four blogs and contains 184 blog-posts with a total blog size of 6.1 MB.
- To reflect SBF's performance on large datasets, **4000 short stories dataset** has been added. It contains 4071 short stories with a total size of 80.3 MB. The mean length is 4377 words with a SD of 3821 words.

Performance



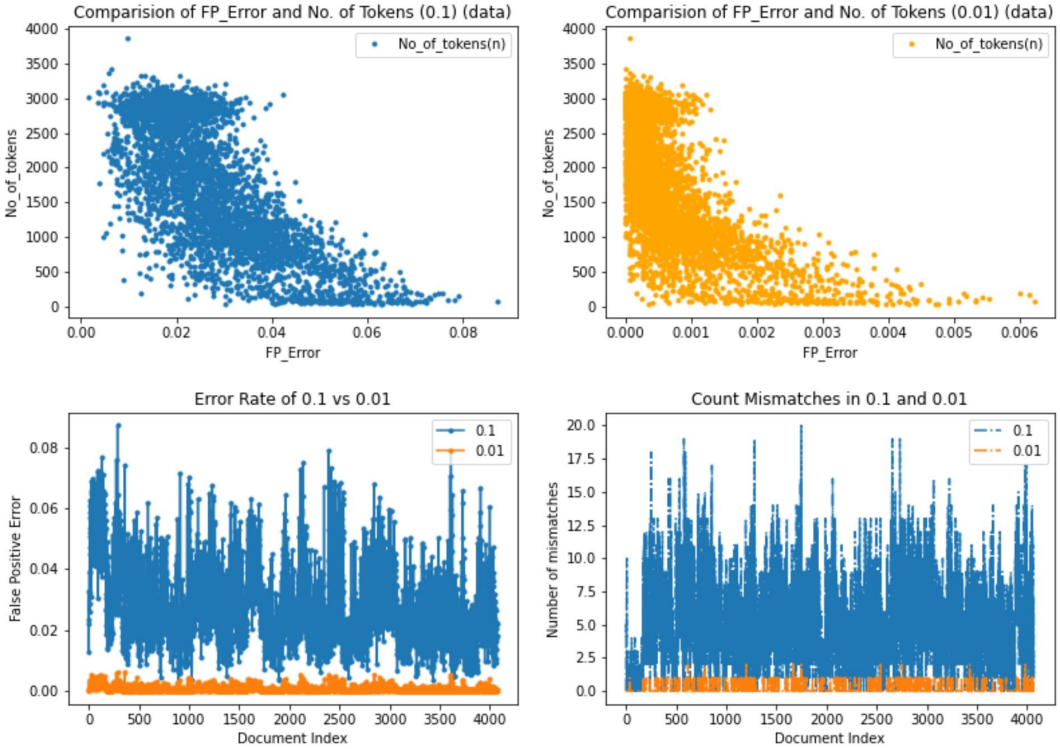
Estimated Error Rate of four unique websites when False Positive Error Rate (**E**) was set to **0.1** and **0.01**. In clockwise order from Top-Left, the figures are from *Danluu, Endler, Parth, and Matthew Daly's* blogs. Counter size of **w=4** was chosen all the calculations.

Performance



Count mismatches of four unique websites when False Positive Error Rate (**E**) was set to **0.1** and **0.01**. In clockwise order from Top-Left, the figures are from *Danluu, Endler, Parth, and Matthew Daly*'s blogs. Counter size of **w=4** was chosen all the calculations.

Performance



The performance of the SBF model on ***4000 short stories dataset***. The results are similar in nature to those obtained from the four blogs, with ***E=0.01*** significantly reducing the number of count mismatches and the estimated false-positive error rate.

Size Comparison

Model	Compressed	Uncompressed	Method
SBF (0.01)	388.7 KB	861.2 KB	TF-Merge
SBF (0.1)	183.8 KB	433.4 KB	TF-Merge
Tinysearch	105.8 KB	177.1 KB	Cuckoo Filter
Lunr	174.3 KB	3400 KB	Inverted Index

TABLE I

DANLUU

Model	Compressed	Uncompressed	Method
SBF (0.1)	173.3 KB	468.2 KB	TF-Merge
SBF (0.1)	173.5 KB	469.2 KB	TF-IDF
SBF (0.01)	356.7 KB	902.9 KB	TF-Merge
SBF (0.01)	356.9 KB	903.8 KB	TF-IDF
Lunr	347.7 KB	5900 KB	Inverted Index

TABLE II

MATTHEWDALY

Model	Compressed	Uncompressed	Method
SBF (0.1)	35.9 KB	83.5 KB	TF-Merge
SBF (0.01)	69.8 KB	152.4 KB	TF-Merge
Tinysearch	58 KB	119.14 KB	Cuckoo Filter
Lunr	53 KB	904 KB	Inverted Index

TABLE III

ENDLER

Model	Compressed	Uncompressed	Method
SBF (0.1)	9.5 MB	21.7 MB	TF-IDF
SBF (0.01)	20.2 MB	43.6 MB	TF-IDF
Lunr	56.4 MB	349.4 MB	Inverted Index

TABLE V

4000 SHORT STORIES

Search Result Comparison

Words	Website: Matthew Daly				
	τ	σ	Common Results in A and B	No. of Results in A	No. of Results in B
Python	-0.333	4	3	10	10
Database	1	0	5	10	10
Type	0.429	14	8	10	10
Task	0.149	12	7	10	10
Url	0.429	10	7	10	10
Models	0.333	4	4	10	10
Google	0.5	10	8	10	10
Language	0.867	2	6	10	10
Testing	0.667	2	4	10	10
Javascript	0.167	24	9	10	10

TABLE IX

COMPARING LUNR (A) WITH SBF (B) USING KENDALL'S TAU τ AND SPEARMAN'S FOOTRULE σ METHODS

Words	Website: Endler				
	τ	σ	Common Results in A and B	No. of Results in A	No. of Results in B
Python	0.278	18	9	10	10
Database	-1	2	2	3	3
Type	0.2	10	6	10	10
Task	-0.214	26	8	10	9
Url	1	0	2	4	2
Models	None	None	1	2	1
Google	0	6	4	5	5
Language	0	6	4	10	10
Testing	0.333	2	3	10	5
Javascript	0.666	2	4	4	5

TABLE X

COMPARING LUNR (A) WITH TINYSEARCH (B) USING KENDALL'S TAU τ AND SPEARMAN'S FOOTRULE σ METHODS

Hardware/Software Requirements

Hardware/Software Requirements

Software:

- Languages used - *Python, Javascript*
- Web Technologies - *HTML, CSS*
- Libraries - *nltk, bitarray, bs4, newspaper3k, lxml, requests*
- Hosted on *PyPi* and documentation generated using *Sphinx*

Hardware:

- This project has no specific hardware constraints.
- The library has been tested on mobile devices, laptops and desktops and requires a modern web-browser (we have primarily tested on *Google Chrome* and *Mozilla Firefox*).

Conclusion

Conclusion

- As observed in *Results* section, Spectral Bloom Filter-based client-side searching is a highly efficient approach to generate an Inverted Index like search results, without consuming significant memory.
- As Lunr and Tinysearch deal on the extreme ends of the spectrum, our approach was to aim at balancing the memory-search efficiency trade-off.
- An interesting direction to extend this research would be to explore various other encoding-decoding schemes to further reduce the client-side document sizes.
- Furthermore, Spectral Bloom Filters can prove to be a good prospect for searching in mobile applications where reducing memory requirement and computational cost is extremely crucial.

End

References

[1] Alan Agresti. Analysis of ordinal categorical data. Vol. 656. John Wiley & Sons, 2010.
[2] A. Appleby. Murmurhash. URL : https://sites.google.com/site/murmurhash/.
[3] Judit Bar-Ilan, Mazlita Mat-Hassan, and Mark Levene. “Methods for comparing rankings of search engine results”. In: CoRR abs/cs/0505039 (2005). arXiv: cs/0505039. URL:http://arxiv.org/abs/cs/0505039.
[4] Burton H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: Communications of the ACM (1970).
[5] Ian Boyd. Which hashing algorithm is best for uniqueness and speed? Software Engineering. URL:https://softwareengineering.stackexchange.com/a/145633.
[6] John Byers et al. “Informed Content Delivery Across Adaptive Overlay Networks”. In: In Proc. of ACM SIGCOMM. 2002.
[7] James Carney and Cole Robertson. “4000 stories with sentiment analysis dataset”. In: (Mar. 2019). DOI: 10.17633/rd.brunel.7712540.v1.
[8] Saar Cohen and Yossi Matias. “Spectral Bloom Filters”. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. SIGMOD ’03. San Diego, California: Association for Computing Machinery, 2003, pp. 241–252. ISBN: 158113634X. DOI: 10.1145/872757.872787.
[9] S. Korokithakis. Writing a full-text search engine using bloom filters. 2013. URL:https://www.stavros.io/posts/bloom-filter-search-engine/.
[10] Olivernn. lunr.js. URL: https://github.com/olivernn/lunr.js/.
[11] Peter Norvig. “Natural Language Corpus Data”. In: Beautiful Data. 2009, pp. 219–242.

Demonstration