Using Fingerprints in n-Gram Indices

© Stefan Selbach

Lehrstuhl für Informatik II, Universität Würzburg
selbach@informatik.uni-wuerzburg.de

Abstract

The major advantage of the n-gram inverted index is the possibility to locate any given substring in a document collection. Nevertheless, the n-gram inverted index also has drawbacks: If the collections are getting bigger, this index tends to be very large and the performance drops significantly. We propose a novel technique of enhancing the performance of an n-gram inverted index with the use of additional fingerprints for each n-gram. A fingerprint contains information about the positions of an n-gram. When combining two or more n-grams, these fingerprints also provide information about the positions of the combination. This can be used to reduce the complexity of merging the n-gram postings lists for a given search and improves the performance of the n-gram inverted index. Furthermore it is possible to freely scale the size of the fingerprints in order to adjust the performance of the index. The size of a fingerprint is neither dependent of the size of the document collection nor the number of n-grams.

1 Introduction

Text searching is regarded as one of the core subjects in Information Retrieval [1]. Many search engines (e.g. Lucene [2]) are building word based inverted indices for document retrieval. Thus the users of these search engines can only search for words. If only a substring of a term matches the given query, no results are being returned for this hit. Many word based search engines try to solve these problems by adding fuzziness to the index and to the query. While parsing documents every word is reduced to its radical. This is done by using stemmers [3,4] and by limiting the set of allowed characters. This procedure is also being applied to every query. This improves the recall, however it tends to reduce the precision of the search because words with different meaning can accidentally be mapped to the same radical. Even though this technique may perform well with documents containing conventional natural language it is not suited for terminologies. For example, if you think of the chemical substance "1,3-Cyclooctadiene" it should also be possible to find this occurrence using the query "Cyclooctadiene", because the usage of this specific isomer "1,3-Cyclooctadiene" is not common. So this word would be reduced to its radical "Cyclooctadiene". The prefix "1,3-" will be ignored. But since some users might however search for this specific isomer, this information has to be included in the index. That is where the inverted n-gram index [5,6,7] comes into play. It enables us to perform exact string matching, so that both queries mentioned above return the appropriate documents. An n-gram index divides the text of the document into overlapping substrings of the size 1 to n. For each n-gram the positions of all occurrences are stored in the index. The given query is divided into n-grams as well and the position lists of the affected n-grams are intersected to retrieve the positions of the query. Since the number of positions for an n-gram is always increasing while the document collection gets bigger, the performance of this index gets worse, because longer position lists have to be intersected (cp. [8]). First we tried to apply fingerprints to the n-grams, so that without storing any positions of the n-grams in the index decision could be made, whether the query does not hit a certain document, or might hit a document. In the second case verifications had to be made, if these documents match the query. Since this verification can worsen the performance, especially when many documents are being affected, we added additional information of the positions of the n-grams to the index, so that no verification was needed. With this technique we reduced the costs for intersecting the position lists of the n-grams. Another advantage is that we can adjust the performance of the index by changing the size of the fingerprints. This gives us the potential to scale the index according to the size of the document collection.

2 Related Work

A term based inverted index [9] consists of two major components: terms and posting lists (see Figure 1). Each term is linked to one posting list. A posting contains a document-identifier (*fileID*) and a position inside the document (*offset*). Besides, to get a fast access to the terms, an index such as a binary tree is

Proceedings of the 11th All-Russian Research Conference «Digital Libraries: Advanced Methods and Technologies, Digital Collections» - RCDL'2009, Petrozavodsk, Russia, 2009.

created over the terms [10,11]. A query is processed by the binary tree and then the posting list is being returned. This setup can only be used, if a query contains terms. But if the query contains substrings of terms, an n-gram index should be used.

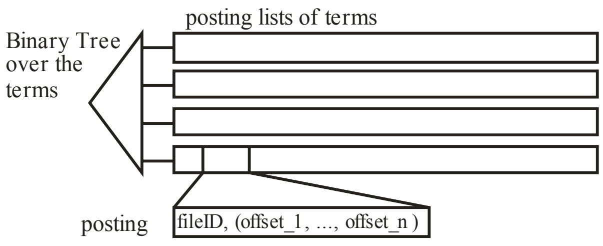
posting lists of terms
Binary Tree over the terms
posting
fileID, (offset_1, ..., offset_n)

Figure 1: inverted index.

An n-gram index (see Figure 2) does not use terms for indexing. The documents are divided into overlapping substrings of the size 1 to n. For each of these n-grams a posting list containing the positions of all occurrences is stored in the index. In order to get a fast access to the n-grams, a trie or hashmap is used [12].

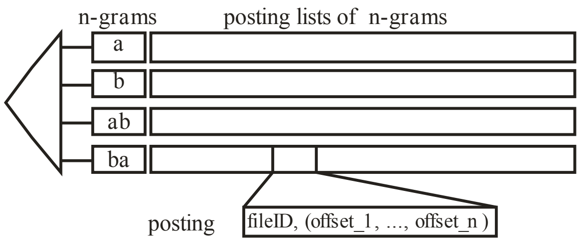

Figure 2: n-gram index.

Since all documents are covered by only including the uno-grams to the index, not all of the n-grams (n>=2) have to be included to the index. Adding more n-grams to the index increases the performance but also increases the index size.
The query is splitted in n-grams in a manner that the combination of these n-grams overlaps the query. After that the posting lists of these n-grams are being intersected. For example the positions for $pos(w \in \Sigma^n)$ the substring w_1w_2 can be calculated with the positions of the 1-grams w_1 and w_2:

$$pos(w_1w_2) = pos(w_1) \cap dec(pos(w_2))$$
$$dec(P) = \{x | x+1 \in P\}$$

If the document collection gets bigger, the posting lists get bigger as well. This significantly reduces the performance. An n-gram index grows multiple times faster compared to a word based inverted index. While in a word based index every additional term occurrence creates one new posting, in a n-gram index multiple postings have to be added because all posting lists of affected n-grams have to be updated. Kim et Al. [13] introduced a technique of adding a second m-gram layer to the index. The documents were at first splittet into m-grams and these m-grams were indexed using n-grams.

Choueka et Al. [14] presented a search index that uses bitmaps which act as an "occurrence" map. To each word a bitvector has been assigned. This vector has the same length as the number of documents and each bit identifies an occurrence of a term in a specific document. With this technique the offsets of the terms within the documents are discarded. Furthermore the size of this index increases rapidly for large document collections since the number of terms and the size of the bitvectors are getting bigger. Choueka et Al. [14] as well as Faloutsos [15] also introduced a hybrid index organization so that infrequent terms were treated differently to reduce the size of the index.

Signature files [1][9] are using a similar approach but the size of the bitvector is independent from the number of documents. Signature files are also word-oriented and are based on hashing. A hash function maps a word to a bit mask. The collection is split into blocks and to each block a bit mask is assigned by ORing the signatures of all the words in the text block. A search is carried out by comparing the signature of the query to the bitmasks of all blocks.

In this paper we want to propose a technique using fingerprints to increase the performance of an n-gram index. The size of these fingerprints can be freely selected like in signature files and each bit in the fingerprint gives information about the position of an n-gram like in the bitmap index. Furthermore these fingerprints can by combined with the approach that uses only posting lists in order to achieve a better performance.

3 n-Gram Fingerprints

Since the sizes of posting lists of n-grams are enormous when indexing large document collections we introduced a so called fingerprint for each n-gram that contains information about the positions of the n-gram and that can easily be compared to other fingerprints in order to give information about the positions of the combination of these n-grams. A fingerprint for an n-gram *w* is a two-dimensional bit-matrix B_w of the size *f* Ч *o*. A bit $b_{i,j}$ is set to 1 if there is an occurrence $p \in pos(w)$ where *fileID(p) mod f = i* and *offset(p) mod o = j*.

$$B_w = \begin{pmatrix} b_{0,0} & \cdots & b_{0,o-1} \\ \vdots & \ddots & \vdots \\ b_{f,0} & \cdots & b_{f-1,o-1} \end{pmatrix}$$

$$b_{i,j} = \begin{cases} 1 : \exists p \in pos(w) : fileid(p) \bmod f = i \wedge \\ \qquad offset(p) \bmod o = j \\ 0 : otherwise \end{cases}$$

Each bit-position is representing a collection of positions that match the above criteria (see Figure 3). Given two 1-grams w_1 and w_2 and their respective fingerprints B_{w1} and B_{w2} we can approximate the fingerprint B_{w1w2}:

$$B_{w1w2} \approx B'_{w1w2} = B_{w1} \wedge B'_{w2}$$

B'_{w2} is constructed by cyclic shifting each column of B_{w2} by one position to the left. This is done in order to compensate the offset of the second 1-gram in relation to the first n-gram. The matrices B_{w1} and B'_{w2} are combined bitwise with AND ($\wedge$) to get B'_{w1w2}. We can guarantee that every bit that is set in B_{w1w2} is also set in B'_{w1w2}. However B'_{w1w2} may contain bits that are set in B'_{w1w2} but not in B_{w1w2}.

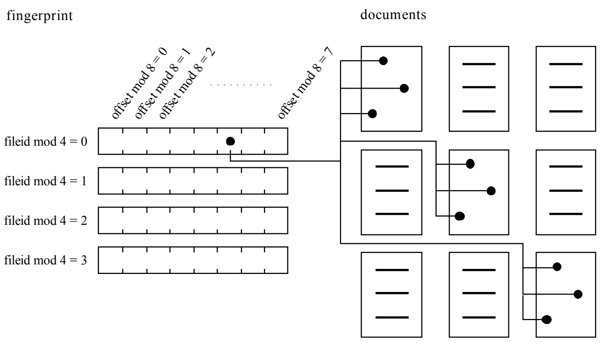

Figure 3: n-gram fingerprint.

To perform a search we have to split the query into n-grams. To get the best results we use as many n-grams as possible. But we can skip n-grams that are included by another n-gram because the bits set in their fingerprints are only supersets of the bits set in the superior n-gram. If we would index for example all 3-grams of a given document collection we would split the query in the same manner, so a query of length *n*, would be split in *n-2* 3-grams. This technique is different to the standard n-gram search where a query is splitted into as few n-grams as possible. After that, the fingerprints of these n-grams are being loaded from the index and the columns of these fingerprints are cyclic-shifted to the left by the relative position of the respective n-gram to the first n-gram of the query. These fingerprints are combined with AND. The resulting fingerprint represents a collection of positions where the query might occur. To verify the results, these positions have to be checked against the query. This is done by opening the involved documents and comparing the local context of the positions to the query. We tested this setup on the "Online Encyclopaedia of Dermatology from P. Altmeyer". This collection contains over 7500 documents. Furthermore we had the access to the querylog and could test the performance of our new setup with real user data. We created an index with f = 1024 and o = 128. Table 1 shows some of the resulting search speeds. The time for the combination of the bit-matrixes is only dependent of the length of the query. But the time for the verification of the positions depends on the number of bits set in the fingerprint of the search result. This can be problematic if a user is searching for a substring, which occurs multiple times in every document. The resulting bit-matrix of this query is very dense and thus many positions have to be verified, which leads to a poor performance.

Table 1: Search speed.

Query	Bit-matrix	Time for verification	Hits
rhinolo	219 ms	94 ms	18
sanfilipo	290 ms	0 ms	0
itracon	266 ms	336 ms	64
oxyuria	197 ms	48 ms	6

Since the size of the fingerprints can be adjusted by changing the parameters *o* or *f* and the number of n-grams can be limited (see section 2) the size of this index is independent from the number of documents in the collection. However it is recommend to increase the size of the fingerprints when indexing large collections. Large collections are leading to dense fingerprints which result in a low filtration ability.

4 n-Gram Fingerprints in Combination with Posting Lists

Analyses of user queries (see Figure 4) showed that most users do not search for very frequent terms. 10% of the queries hit 400 different terms with frequencies between 150 and 2000. The remaining 90% of the queries were almost equally distributed over the remaining 170.000 terms. However the most of the time of a search was needed for the verifications of hits.

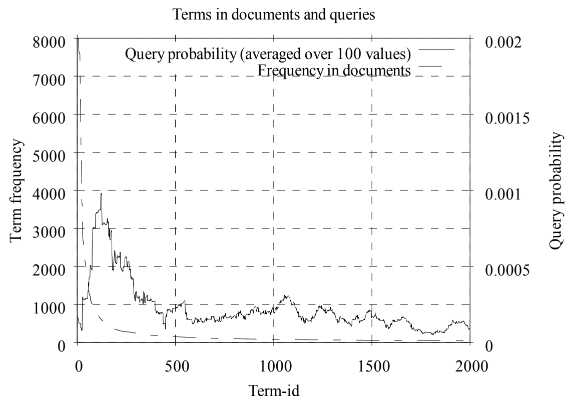

Figure 4: Term frequency in documents and queries.

In the next step we combined the standard n-gram search with our fingerprinting technique. When creating a fingerprint for an n-gram we assign a bit to every posting of an occurrence of the n-gram. Thus we also partitioned the postings into smaller subsets. If we do not discard these lists, we can use them for the verification process. For each n-gram the partitioned posting lists were added to the index (see Figure 5).

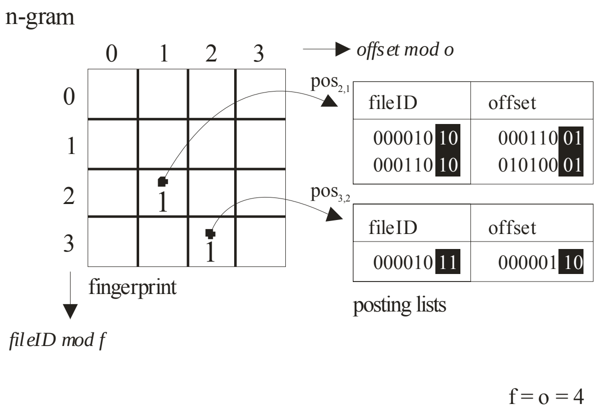

Figure 5: Combination of fingerprints and posting lists.

If *f* and *o* are multiples of 2 we do not have to include the last $log_2(f)$ bits of the *fileID* and the last $log_2(o)$ bits of the *offset* to the posting, because this information is already included by the residue class (see Figure 5).

The bitwise AND combination of the fingerprints of the affected n-grams for a query shows which subsets of posting lists have to be intersected. On the one hand we reduce the cost for the intersection of the postings since only a few of the smaller subsets have to be intersected instead of a complete posting list for an n-gram. On the other hand we have to deal with much more lists. Our test corpus has been indexed with 1024 residue classes for the *fileID* and 128 for the *offset*. As a result we get 131.072 subsets of posting lists for each n-gram. We included 14.000 different (1-3)-grams in our index, which made 18.350.080.000 different lists in total. Most of these lists are empty or have only a few entries. In order to reduce the overhead, which is necessary for handling this large number of posting lists, we used a file based hash table (see Figure 6). The hash value for a given list was computed as a function of the residue classes of *fileID* and *offset*:

$$h(pos_{i,j}) = j + i \cdot o$$

This way the posting lists of n-grams occuring nearby are also stored close to each other in the index. In this case *i* (the residue class for the *fileID*) is constant and *j* (the residue class for the *offset*) only differs by the distance (*mod o*) of the n-grams. Thus the hash value differs also only by the distance.

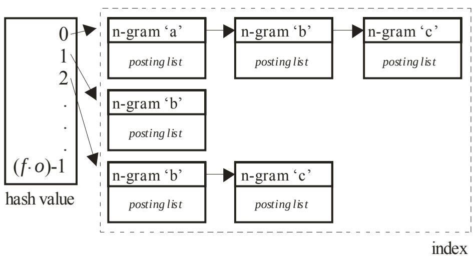

Figure 6: Managing the n-gram posting lists.

The main disadvantage of this technique is that the set of possible hash values in the hash table is rather small and the hashing function produces many collisions. In our index for the "Online Encyclopaedia of Dermatology from P. Altmeyer" we had 25 collisions in average. That means in average 12.5 lists had to be processed until the desired list was found.

If we take a look at the frequency of n-grams in the documents and in the queries of our collection we see, that many n-grams are rarely used in the queries (see Figure 7). We can use this information to define a ranking by which the posting lists sharing the same hash value can be sorted. This way we should get a faster access to the hash table for ordinary queries. In order to measure the performance of this methodology we used 5000 user queries to define the ranking of the n-grams. We then used 5000 different user queries to test the performance of the hash. We came to the result that in average only 5 lists had to be processed

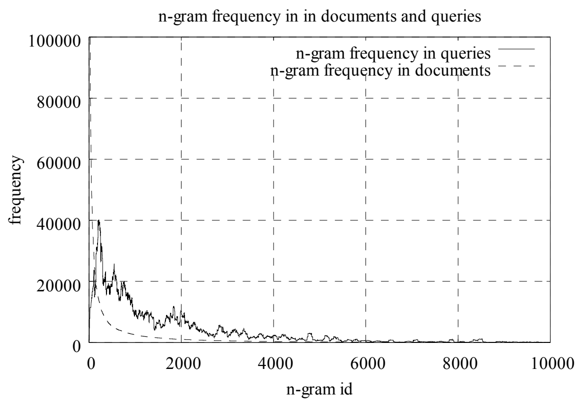

Figure 7: n-gram frequency in documents and queries.

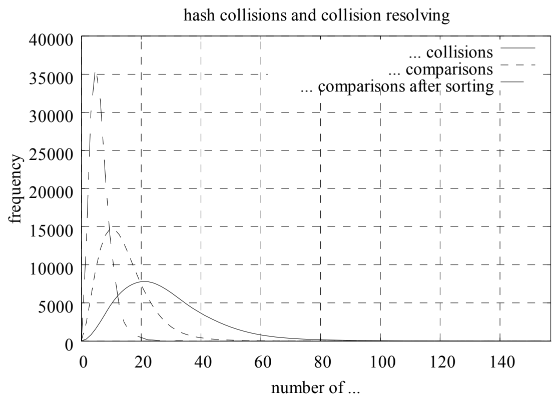

Figure 8: Hash collisions and collision resolution.

in order to retrieve the desired list (see Figure 8). This improved the performance of the hash by a factor of 2.5. This is a great result since we would need 4.6 comparisons in average to retrieve the requested list, if we would have random access to the different lists in one hash entry (which requires more space for the index) and could perform a binary search. This is very close to our result, but we do not need the extra index space for the random access to the lists. We compared this setup with the search engine from section 3. In average the performance improved by 40%. Table 2 shows that the time needed for the verification reduced significantly. Besides it was more balanced.

Table 2: Search speed (fingerprints with position lists).

Query	Bit-matrix	Time for verification	Hits
rhinolo	230 ms	10 ms	18
sanfilipo	271 ms	0 ms	0
itracon	245 ms	15 ms	64
oxyuria	210 ms	12 ms	6

5 Fingerprint Compression

In order to compress the fingerprints we analysed their density. Over 70% of the fingerprints have a density less than 0.1, 20% are between 0.1 and 0.4 and 10% have a density greater than 0.4. Fingerprints, which have an extremely low or high density, do not contain much information. These fingerprints can be compressed by simply reducing their resolution. This equals to a convolution of the original fingerprint. Figure 9 shows an example for a 4 x 4 fingerprint. This fuzziness might in some cases lead to an incorrect fingerprint for a query. But since all bit-positions of the resulting fingerprint are being verified using the position lists of each n-gram no false positive hits are generated. It just slightly slows down the performance, because additional requests for position lists, that do not exist, have to be made.

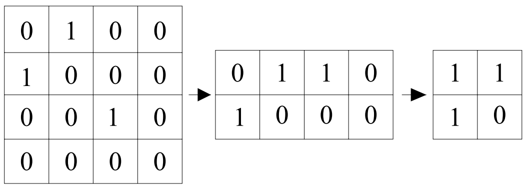

Figure 9: Convolution of fingerprints.

We tried different thresholds of densities for the convolution. If the density of a fingerprint was in the defined range, we reduced its size by factor 2. Table 3 shows the relative performance loss and the relative index reduction.

Furthermore we implemented a dictionary based compression. Each column of a fingerprint was added to a dictionary. The fingerprint now consists of multiple references to this dictionary. This methodology significantly reduced the size of the index. Especially dense and sparse bitvectors had been reused frequently.

Table 3: Fingerprint compression.

Density threshold for convolution	Performance loss	Fingerprint index reduction
no convolution	0 %	0 %
0-0,025 and 0.975-1	3.1 %	23 %
0-0.05 and 0.95-1	3.2 %	27 %
0-0.1 and 0.9-1	10 %	29 %
0-0.2 and 0.8-1	25 %	31 %

In combination with the convolution of fingerprints we were able to reduce the space needed to index the fingerprints by 50-60%. The dictionary based compression reduced the performance only by 1-3%.

6 Conclusion and Future Work

Our experiments have shown that n-gram fingerprints can be used to improve the performance and the scalability of n-gram inverted indices. We introduced techniques to optimize the index in a manner so that common user queries can be processed more efficiently. In standard n-gram indices a query is splitted into n-grams that overlap the query. The posting lists of all these n-grams have to be intersected to get the final search result. With our technique we split these posting lists into smaller lists. After combining the fingerprints of the involved n-grams we know which of these lists have to be intersected. Thereby we reduce the complexity of the intersection and gain more performance. We compressed the fingerprints that do not contain a certain level of information and we were able to reduce the fingerprint index size by 60% without major loss of performance.

In the future we would like to combine our indexing techniques with a word based inverted index in order to profit from the advantages of using both n-grams and words as indexing terms. We intend to split the text into terms and save the posting lists in an index like in the case of an inverted index. However we do not create a binary tree over the terms. The terms will be indexed in a second n-gram inverted index described in this work. This way we do not lose the ability of searching for any kind of substring in the document collection. Another advantage is that stop words, which tend to flood the posting lists of some n-grams, only occur once in the term list of the first inverted index and thus fewer postings for the affected n-grams have to be dealt with in the second index. Using a word based inverted index gives us further benefits. In n-gram inverted indices ranking the results of a search is a difficult task. Since n-grams generally do not hold semantic information, only TF/IDF or field based ranking methods can be used. Having words as index terms it is possible to define a precomputed rank for each tupel (*term*, *document*). Moreover index terms can be linked to each other in order to map thesaurus information to the index.

References

[1] Baeza-Yates R. and Ribeiro-Neto B.: Modern Information Retrieval. ACM Press (1999)

[2] The Lucene search engine, http://jakarta.apache.org/lucene/ (2005)

[3] Porter M. F.: An algorithm for suffix stripping. Readings in Information Retrieval **14**(3) (1980) 130–137.

[4] Snowball stemmers, http://snowball.tartarus.org/ (2003)

[5] Yasushi O. and Toru M.: Optimizing query evaluation in n-gram indexing. In: Proceedings of International Conference on Information Retrieval, ACM SIGIR, Melbourne, Australia (1998) 367–368

[6] Brown M. K., Kellner A., and Ragget D.: Stochastic Language Models (N-Gram) Specification. W3C Working Draft (2001)

[7] Miller E., Shen D., Liu J., and Nicholas C.: Performance and Scalability of a Large-Scale N-gram Based Information Retrieval System. In Journal of Digital Information **1**(5) (2000)

[8] Mayfield J. and McNamee P.: Single N-gram Stemming. In: Proceedings of International Conference on Information Retrieval, ACM SIGIR, Toronto, Canada (2003) 415–416

[9] Witten I. H., Moffat A., and Bell T. C.: Managing Gigabytes, second edition. Morgan Kaufmann. (1999)

[10] Zobel J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys **38**(2) (2006)

[11] Zobel J., Moffat A., and Ramamohanarao K.: Inverted Files versus Signature Files for Text Indexing. ACM Trans. on Database Systems **23**(4) (1998) 453–490

[12] Cohen J. D.: Recursive Hashing Functions for n-Grams. ACM Trans. on Information Systems. **15**(3) (1997) 291–320

[13] Kim M.-S., Whang K.-Y., Lee J.-G., and Lee M.-J.: n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In: Proceedings of VLDB. (2005) 325–336

[14] Choueka Y., Fraenkel A., Klein S., and Segal E.: Improved techniques for processing queries in full-text systems. In: Proceedings of the 10th ACM SIGIR Conference. ACM Press. (1987) 306–315

[15] Faloutsos C., and Jagadish H. V.: Hybrid index organizations for text databases. In: Proceedings of the International Conference on Extending Database Technology. LNCS, vol. 580, Springer. (1992) 310–327