TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 2194 – 2204
© TÜBİTAK
doi:10.3906/elk-1402-92

Name spell-check framework for social networks

Burak YILDIZ*, **Fatih EMEKCİ**
Department of Computer Engineering, Turgut Özal University, Ankara, Turkey

Received: 10.02.2014 • **Accepted/Published Online:** 05.07.2014 • **Final Version:** 15.04.2016

Abstract: The problem of finding similar strings is very important in most real life applications including spell-checking, data cleaning, next generation sequencing, and alignment. In order to query and manage string data online, scalable algorithms and frameworks are essential. Scalable frameworks and algorithms have been introduced in the past few years. However, these frameworks mainly deal with caching and querying structured data. They do not deal with fuzzy queries, where we need to search for an approximate string. In this paper, we propose an edit distance aware filtering algorithm for all kinds of approximate string search problems. We also propose a novel name spell-check engine mainly for social networks. Our experiments show that our edit distance aware filtering mechanism alone improves the query processing time and throughput by almost 30%. Additionally, our name spell-check engine improved the name spell-check response time and throughput almost 10 times by using our filtering scheme and some domain specific observations.

Key words: string search, spell check, social networks

1. Introduction

The management of string data and making them available online have been important problems in the last decade. All information is becoming queryable online, which increases the need for scalable systems. There are scalable frameworks and algorithms that were introduced in the past few years such as Hadoop [1], Memcache [2], HBase [3], and Membase. These frameworks mainly deal with caching and querying structured data. They do not deal with fuzzy queries, where users may make mistakes in spelling or do not know the exact phrase. We call systems that can be queried with almost correct phrases approximate string search systems. In other words, given a set of strings, these systems try to find the strings in the set that are similar to the query string. The problem of finding approximately similar strings is a very important aspect in some real life applications, including the following:

- Spell-checking: given an input document or a string, a spell checker needs to find the possible mistyped words. This can be done by finding the strings that are close to the mistyped strings (i.e. with an approximate string search). For each word that is not in the dictionary (misspelled word), we need to find a potential match.
- Data cleaning: data from different information sources often have various conflicts. The same real world entity could be in different forms. There might also be errors in some of the data sources (e.g., missing an apartment number or a street name). It is important to find these types of errors and standardize data while merging. The data cleaning process needs to find similar entities to match. For example, the

*Correspondence: yildizb@turgutozal.edu.tr

address "PO Box 17,State St. Santa Barbara 93117 CA" should be matched with "P.O. Box 17, State St Santa Barbara CA 93117".

- Next-generation sequencing (NGS)/alignment: next-generation sequencing produces short reads or short read pairs. These are short sequences of usually less than 200 DNA bases. One needs to compare and find the matches of these short reads on the reference sequence. However, these reads may differ from person to person (they may differ by several bases from the reference sequence). One needs to perform an approximate string search to find the most similar part of the DNA sequence on the reference sequence, which is called aligning or mapping the reads against the reference sequence [4].

The queries in these applications require a high real-time performance and they are very resource intensive. Some of the applications have strict query response time (QRT) requirements. For example, the name spell-checking in social networks has to be done in 5–10 milliseconds and they also require high query throughput on the machines running these applications. The reason for these requirements is that applications are needed to be invoked many times in every second as there are millions of users all over the world. Users may not feel the difference between 20 ms processing time and 2 ms processing time. However, from the server perspective, the former means 50 queries per second (QPS), while the latter means 500 QPS. Therefore, reduced processing time gives the server more computing power to serve more requests, and thus reduces the need for the extra hardware and cost.

In this paper, we attack the problem of spelling correction of name queries for social networks. Our query string consists of misspelled first names and last names such as "Jonh Nsah" and the proposed system finds the correct spelling of the query (i.e. "John Nash"). Name spelling is especially hard for the human brain as they may be very different than the daily life words. For example 50% of the 6 million monthly Google Play search terms (mainly application names) are misspelled (http://www.pocketgamer.biz/r/PG.Biz/Google+Play/news.asp?c=51017). LinkedIn had 5.7 billion name searches in 2013 (http://blog.linkedin.com/2013/03/25/linkedin-search-just-got-smarter/). Based on these numbers, we can say that name spell-checking is becoming more important. We propose a novel name spelling engine for social networks, which can be used in other applications as well. Given a set of names and an approximate query, we find the names that are similar to the given query. The similarity metric here is the edit distance [5]. We proposed a novel algorithm (called AND of n Filters (AnF)) to index and prune names that are e edit distances away from the given query. In addition, we also proposed a framework specific to names (we call it the first character right (FCR)) to speed up the query processing and throughput drastically. Our experiments show that AnF improves the query processing time and throughput by almost 30% and FRC improves them by almost 10 times. As for future works, we plan to extend our technique to other applications such as DNA alignment and data cleaning.

The rest of the paper is organized as follows: Section 2 introduces preliminaries and defines the problem formally. We discuss related works in Section 3. How to use n-gram filtering to find the strings that lie within the e edit distance away from the query is discussed in Section 4. Then, we introduced our edit distance aware filtering in Section 5. Our novel approach to solve name spelling problem (FCR) and a novel name spell-checking framework is described in Section 6. We evaluate the effectiveness of our proposed solutions with extensive experimental analysis in Section 7. The last section concludes the paper.

2. Preliminaries and problem definition

2.1. Edit distance

When we search the query name q in a set of names N, we need to measure how similar the q and each of the names in the N are. In information theory, there are some techniques to measure the similarity of two strings.

Some of them are the edit distance (a.k.a. Levenshtein distance) [5], the Jaccard coefficient [6], and the cosine similarity [7]. In this paper we use edit distance for the similarity measurement. It is based on finding three types of atomic edits in the comparison of two names. Atomic edits are insertions, deletions, and substitutions. As an example, the edit distance between the two names BRUCE#WILLIS and BRUSE#WILISS is 3. Let q be the first name and n be the second name. The first atomic edit is the substitution of C at position 4 in q with S. The second atomic edit is the deletion of L at position 10. The last atomic edit is the addition of S at the end of n. After the effect of these atomic edits, q transformed into n. The representation of the edit distance between q and n is $ed(q,n) = 3$.

For the rest of the paper, e and $ed(n_1, n_2)$ represent the edit distance between string n_1 and string n_2.

2.2. N-gram

To make the string searching efficient, some of the irrelevant strings would be filtered to reduce the count of candidate strings by using some filtering algorithms. Many of these algorithms rely on the n-gram concept to eliminate the irrelevant strings [7–9]. Thus, an n-gram of a string is a contiguous n base of the string. In other words, an n-gram is a substring of the string and a set of n-grams means all the possible n-grams of the string. For example, the 3-grams of the name BRUCE are {__B, _BR, BRU, RUC, UCE, CE_, E__}. As seen in the example, the beginning of the name was padded with *n – 1* characters, as was the end of the name.

We use $g_1, g_2, g_3, ...$ for representing the n-grams of a name and g_1 starts at position 1 of the padded version of the name and has a length of n; g_2 starts at position 2 and its length is the same as g_1 and so on; and $G(s,n)$ represents the set of n-grams for a given string s and n-gram length n (i.e. 1, 2, 3, ...) for the rest of the paper.

2.3. Inverted index of n-grams

Inverted indexes have been widely used in document searching in Information Retrieval [10,11]. We also used inverted indexes to index names to be able to find the names in given indexed names for a given edit distance away from the query name q. Each name is decomposed into the n-grams and an inverted list for each gram is created. This list holds n-grams and name IDs containing that n-gram.

In Figure 1, an example name database having 4 names is shown. The first names and last names are concatenated with a place holder character (in this example it is #). Let us show how the inverted list is created. If n of the n-gram is chosen as 3, the 3-grams of first name will be {__R, _RU, RUS, USS, SSE, SEL, ELL, LL#, L#C, #CR, CRO, ROW, OWE, WE_, E__}. In the same way, other 3-grams of names are determined; lastly, for every unique 3-gram, a new entry is created and the 3-gram and the IDs of names that contain that 3-gram are added to this entry. For this example, the MOR 3-gram exists in names whose IDs are 2 and 4. Therefore, the new entry for the inverted index contains MOR at its first column [11, 12] and (2.4) at its second column. The inverted index (for an n-gram length of 3) of the example name database in Figure 1 looks like the one in Figure 2.

3. Problem definition

The problem dealt with in this paper is finding misspelled names into the name database in a certain error range with a minimum latency. In other words, for the given query name q and a set of names N, find all the names as quickly as possible in N, which are at most e edit distance away from q. The formal definition of the problem is stated as follows: Find all $n \epsilon N$ such that $ed(q,n) \leq e$.

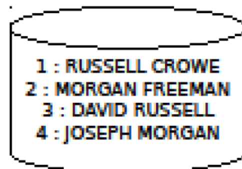

Figure 1. An example of a name database.

ID	NGRAM	NAMEIDs
1	__R	1
2	_RU	1
3	RUS	1, 3
4	USS	1, 3
5	SSE	1, 3
6	SEL	1, 3
7	ELL	1, 3
8	LL#	1, 3
9	L#C	1
10	#CR	1
11	CRO	1
12	ROW	1, 4
13	OWE	1
14	WE_	1
15	E__	1
.		
.		
.		

Figure 2. Inverted index for the name database example from Figure 1.

4. Related works

Searching a key approximately in a haystack is a well-studied problem in the information retrieval (IR) area where the key can be a string, a phoneme, or a k-mer and the haystack can be a digital dictionary, a speech record, or a k-mer database, respectively. If the key is a string and the haystack is a text, searching a key approximately in a haystack specifically refers to an "approximate string matching" problem in the literature and there has been a lot of research on this problem. There is a good review of the literature [13] and many algorithms have been proposed to overcome the problem efficiently [6,8,12,14]. Some of them focus on filtering algorithms to eliminate irrelevant candidates and others focus on merging techniques of the inverted index of n-grams efficiently.

In recent years Kim at al. [15] contributed to the inverted index structure to improve the performance of the inverted index. Li et al. [14,16] proposed a technique called VGRAM to improve the performance of approximate string queries using variable n-grams lengths in 2007 and proposed some filtering and merging algorithms in 2008.

In the current paper, we propose a novel filtering algorithm and a name spell-check framework. Our proposed filtering algorithm aims to improve the time and memory efficiency. The name spell-check framework is a complete spell checker and it is based on a high percentage of correctness of the first characters of names.

5. Name filtering using N-grams

The basic principle of the approximate string search problem is that similar strings share enough common n-grams. This filtering technique is known as gram filtering or count filtering; it was proposed and described in [14,17] in detail. This filtering technique (we call it T-occurrence filtering (Tocc)) was used to compare with the proposed techniques in the current paper.

It is stated that if the edit distance between two strings s_1 and s_2 is e, and n is the number of bases in the n-grams, then they should share at least T of n-grams where $T = \max(|s_1|, |s_2|) + n - 1 - e \times n$ [14,17]. Therefore, we can find a subset of names that are at most e edit distance away from the query by solving the

T-occurrence problem (which would give the subset) [14,17]. The T-occurrence problem is stated as follows: given a query name q, and a set of n-grams $G(q,n)$, find the set of IDs of names that appear at least T of the inverted lists of the n-grams in $G(q,n)$.

For the solution of the name search problem, an inverted index is created for a given name database N. A list called l_g is created for each n-gram in N. For example, in Figure 2, g_1 = __R, $l_{g1} = \{1\}$and g_2 = ROW, $l_{g2} = \{1, 4\}$.

If a query name q has n-grams $g_1, g_2, ...$, the problem can be solved by counting name IDs while traversing the inverted lists of the n-grams that are retrieved from the query name and then filtering the counts equal to or greater than the T value. The found names that satisfy the T-occurrence rule are assigned to the preliminary set called P. Then, the edit distance between the query name and the names in P are calculated and the names in P are selected and added to the result set called R by their edit distance from the query name q, which are equal to or less than the given edit distance e. The algorithm for the name search problem is given in Algorithm 1.

Let us illustrate the Algorithm 1. Let the query name q and edit distance e be RUSEEL#CROVE and 3, respectively, and let us assume that we would execute the query on the inverted index created in Section 2. The first step in the algorithm is finding n-gram set G for q. That is G = {__R, _RU, RUS, USE, SEE, EEL, EL#, L#C, #CR, CRO, ROV, OVE, VE_, E__}. In the end of the process, one would find that names n_1, n_2, n_3 and n_4 share 8 n-grams, 0 n-gram, 4 n-grams, and 0 n-gram, respectively, with q. For name n_1, the given edit distance $e = 3$, $n = 3$, and $|q| = 12$, $|n_1| = 13$, then the value of T is calculated as $T = \max(12, 13) + 3 - 1 - 3 \times 3 = 6$. According to Algorithm 1, we would put only name n_1 sharing at least 6 n-grams with q in subset P. Then, the sorted P, by edit distance between q, becomes $\{n_1\}$. Finally, the result set R consists of only k_1 whose edit distance equals 3, which equals e.

Algorithm 1. Basic filtering algorithm.

1. Inputs: name set N, a query name q, edit distance e.
2. Output: set of IDs of names R that contains names whose edit distances between q are equal to or less than e.
3. Create the n-grams $g_1, g_2, \ldots, g_m$ for query q.
4. Create the inverted lists called $l_{g1}, l_{g2}, \ldots, l_{gm}$ for each n-gram in N.
5. Scan all the lists corresponding to the n-grams of q (i.e., $l_{g1}, l_{g2,...,} l_{gm}$) and find the set of name IDs, P, whose counts are equal to or greater than T in the inverted lists (FILTER-STEP).
6. Sort the names in P by their edit distance from q and append the IDs of names that are less than e to the set R (SORT-STEP).

6. Edit distance aware name filtering

The edit distance aware algorithm that we developed is AnF. In this algorithm, we divided the FILTER-STEP in Algorithm 1 into smaller n subfilters (F_i, i = 1, 2, 3, ..., n) connected to each other with logical AND operator. Each subfilter consists of at most $\lceil |q|/n \rceil$ ($|F_i|$) nonoverlapping n-grams connected with a special logical OR operator (symbolized with ∇), which returns true if at least $|F_i| - e$ of its components

exist in the searched name. As an example, for $n = 3$, we have 3 subfilters consisting of 3-grams and F_1 consists of substrings of query name q such that $substr(q, j, 3)$ for $j = 1, 4, 7, ...$ (i.e. substring of name q starting from the index (starts at 1 not 0) j and length of 3), F_2 consists of substrings of query name q such that $substr(q, j, 3)$ for $j = 2, 5, 8, ...$ and so on. Thus, the generic representation of the filters is $F_i = \sigma_{substr(q,i+0\times n,n)\nabla substr(q,i+1\times n,n)\nabla substr(q,i+2\times n,n)\nabla ...}$, for $i = 1, 2, 3, ..., n$.

The AnF filter creation process is illustrated in a simple example in Figure 3. In this example, we choose the length of n-gram n as 3 and the edit distance e as also 3. At the top of the figure, a query name q is shown; it consists of 12 characters and its padded version has length of two times $n - 1$ added to the original length, which is calculated as 16. Therefore, the n-grams of subfilter F_1 start at positions 1, 4, 7, 10, and 13 of the padded version and have a length of 3. The start points of n-grams of subfilter F_2 are one increased over those of F_1; thus, they are 2, 5, 8, 11, and 14. Similarly, the start points of n-grams of subfilter F_3 are one increased over those of F_2 except the start position of the last n-gram, and they are 3, 6, 9, and 12. The reason for not including the last n-gram is that there is a pad character at the position of 15. The required numbers of common n-grams with q of filters F_1, F_2, and F_3 are $5 - 3 = 2$, $5 - 3 = 2$, and $4 - 3 = 1$, respectively. Hereby, the AnF filter is AND of 3 subfilters, which are F_1, F_2, and F_3.

The intuition behind this algorithm is reducing the query time and decreasing the number of candidate names in the searched database. In other words, by using this algorithm, it is not required to control the other subfilters if one of the subfilters fails. This is known as a binary AND operation shortcut. Because of this fact, it is expected that the query time could be reduced. Another improvement that comes with the AnF filtering algorithm is that once the number of eliminated names (false-positive names) are increased, the filtered subset size would decrease. The reason for the decrease in the subset size is that the AnF filter does not just require the existence of an n-gram in the searched name, but also looks for the position of the n-gram. Let us give an example. Let us have a query name q, and according to T-occurrence filtering, the filter requires to have 6 common 3-grams. On the other side, the AnF filtering divides the filter into 3 subfilters (F_1, F_2, and F_3) and it is possible that each subfilter requires 2 common 3-grams. If one of the names has 6 common 3-grams, it will pass the T-occurrence filter, but it is possible that the 6 common 3-grams can be spread 3, 1, and 2 to AnF subfilters F_1, F_2, and F_3, respectively, and it will not pass the AnF filter. This situation is illustrated in Figure 4. Consequently, while a name passes the T-occurrence filter, may not pass the AnF filter; another name that passes the AnF filter must pass the T-occurrence filter because the total number of required common n-grams is always equal to the calculated T number, as explained in Section 4. Thus, it is obvious that the

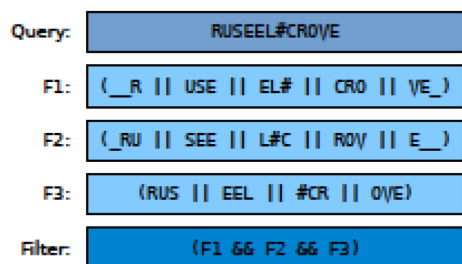

Figure 3. Illustration of the AnF filter creation.

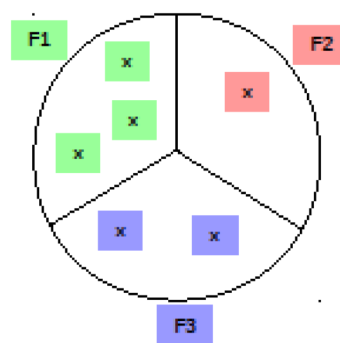

Figure 4. Illustration of how AnF filtering reduces the filtered subset of names.

size of the subset created after AnF filtering is always equal to or less than the size of the subset created after T-occurrence filtering. The important point is that while struggling to decrease the total search time, the time for work of the decreasing process must be less than the decreased time. As a result of this intuition, we expect that the proposed algorithm improves searching time.

Algorithm 2 is the basic usage of the AnF. This algorithm takes the inverted index of name IDs L, the edit distance e, the n-gram length n, and the AnF subfilter list F as input and returns the filtered name ID list. The algorithm counts each name ID repetition for each subfilter in AnF and controls the counts to determine whether one of the name ID repetition is greater than or equal to $|f|- e$ ($|f|$corresponds to the number of n-grams that subfilter f has). If it is, that means that the corresponding name passed that subfilter, so the algorithm increases the count of subfilters passed by the name. The algorithm also controls the counts of subfilters passed by the name to determine whether they are equal to n-gram length n. If they are, the corresponding name will pass the AnF, and thus its ID is inserted to the result list. After finishing one of the subfilter controls, the algorithm loops the counts of the name ID repetitions and deletes all name IDs less than $|f|- e$ (it means that the name could not pass the subfilter) from the inverted index of the name IDs L, because all the subfilters in the AnF are connected to each other with logical AND operator. After all subfilters are visited, the created result name ID list is returned.

Algorithm 2. AnF filtering algorithm.

1. Input: the inverted index of n-grams I, edit distance e, gram length n, and AnF subfilter list F.
2. Output: filtered list of name IDs R.
3. Initialize an empty list R for the result.
4. Initialize a list A for the number of subfilters passed by the names.
5. For each subfilter f in F:
6.   Initialize a list O for the true count of special IDs or for each name ID.
7.   Select grams list as l for f from I.
8.   For each item ID in each list l:
9.   Increment the value O[id].
10.   If $O[\mathrm{id}] \geq |f|- e$:
11.   Increment the value A[id].
12.   If $A[\mathrm{id}] = n$: insert ID to R.
13.   For each name ID as ID and count c in O.
14.   If $c <|f|- e$: delete all IDs from I.
15. Return to R.

7. Name spell-check framework

In Section 5, we proposed a novel filtering algorithm AnF to search names, and in this section we will give an adequate explanation of a newly developed, complete name spell-check framework, especially for social networks. However, before the explanation, we will describe our observation of name writings of people.

It is human nature to give more attention to more outstanding and more noticeable things and these things are more permanent in the mind. One of the noticeable things is the first character of the strings. One example of this (FCR) is that people usually do not misspell the first characters of first names and last names. We observed this behavior and did short tests on online spell checker web sites (e.g., spellcheck.net). The results of our tests showed that people write more than 95% of the first characters of the first names and the last names correctly. These results give us an intuition that if we assume that people write the first characters of the first names and the last names correctly while searching names, and if we use these correctly written characters in the AnF filtering algorithm, the size of the results of filtering will be reduced significantly.

The first thing to do in order to apply the FCR observation to the AnF filtering algorithm is to create a newly structured inverted index. We designed a new FCR inverted index by minimally changing the original inverted index. While creating the inverted index, all found n-grams were indexed with two extra characters, one for the first character of the first name and the other for the first character of the last name. Therefore, the inverted index in Figure 2 turned into the FCR inverted index in Figure 5.

The last thing is to adapt the AnF filter to the new FCR inverted index. In Figure 6, we demonstrate how we easily adapted the AnF filter to the new index: for a given query name q, all subfilters would be created with n-grams (explained in Section 5 in a detailed manner) by adding the first characters of the first name and the last name of q to each n-grams. With this little addition, only names that have the same first characters of first and last name of the query would be in the result set of the AnF filter.

Now we will explain the novel name spell-check framework. The framework consists of two levels. The first level is for finding candidate names for a given query name with the AnF filtering algorithm with FCR, and the second level is for finding candidate names with only AnF filtering. The first level assumes that a user would write the first characters of the first and last names while the second level does not. The framework works as follows: the user sends a query name to the framework. At the beginning, the first level handles the query and creates the result name set. If the result set is not empty, it will be returned. If the result set is empty or the user is not satisfied with the result and sends the same query again, this time the second level of the framework

ID		NGRAM		NAMEIDs
1	:	RC:__R	:	1
2	:	RC:_RU	:	1
3	:	RC:RUS	:	1
4	:	RC:USS	:	1
5	:	RC:SSE	:	1
6	:	RC:SEL	:	1
7	:	RC:ELL	:	1
8	:	RC:LL#	:	1
9	:	RC:L#C	:	1
10	:	RC:#CR	:	1
11	:	RC:CRO	:	1
12	:	RC:ROW	:	1
13	:	RC:OWE	:	1
14	:	RC:WE_	:	1
15	:	RC:E__	:	1
		.		
		.		
		.		

Figure 5. FCR inverted index for the example name database.

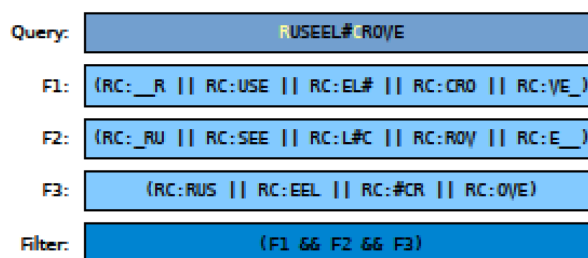

Figure 6. Demonstration of AnF filter creation with FCR.

handles the query and returns the result name set. With this two-level name spell-check framework approach, at the beginning the first more related result set (because it assumes the first characters are correct) is created within a minimum latency, and if it is not satisfactory, the framework can also offer to serve the complete result name set.

8. Experimental results

We wrote a benchmark program to compare the performance of the AnF filtering technique proposed in this paper with the Tocc filtering proposed in [14,17], also known as gram filtering or count filtering in terms of QRT and throughput. The IMDB (http://www.imdb.com) dataset, which consists of 2.5 million actor and actress names downloaded from the IMDB website, was used as the name database. First, we increased the IMDB dataset up to 12 million names by adding random edit distances and created 6 different amounts of name lists that have 1, 3, 5, 8, 10, and 12 million names. We then created 12 different indexes from 6 name lists with 2 different types of indexing (normal and FCR). Indexes were created with an n-gram length of 3.

A query set was created and it had 1000 randomly chosen names from the name lists. We ran 2 tests that used the created query set on each index, one for AnF filtering and one for Tocc. The tests were run with an edit distance of 3.

The test application was implemented using Lucene with Java programming language. All tests were run on a PC that had 4 GB main memory, 2.4 GHz Intel Core i5 CPU, and a Linux Mint operating system.

The QRT and throughput were calculated for each test. The results are provided as graphics. The comparison of QRT for AnF and Tocc is given in Figure 7: both filtering techniques have near linear QRT with different index sizes and AnF filtering has always better QRT than Tocc. As the index size increases, the gap between the QRT of AnF and Tocc also increases.

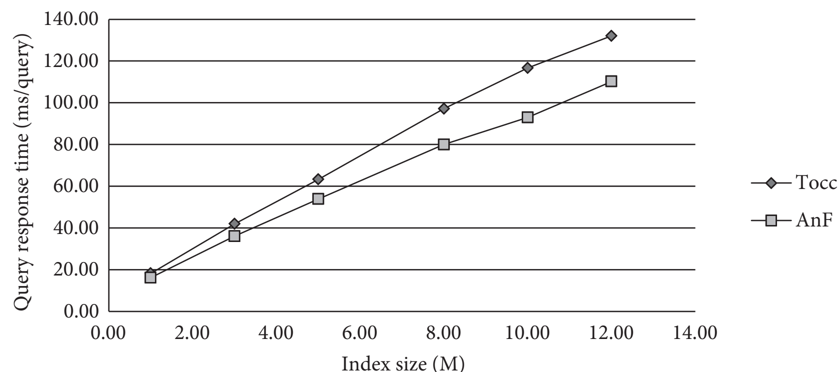

Figure 7. Comparison between the query response times of T-occurrence and AnF filtering.

In Figure 8, all of the QRTs were compared. While the QRT difference between AnF and Tocc was clear in normal indexing, it was not clear in FCR indexing because in FCR indexing the size of the filtered subset of names was already too small.

The best result was achieved by AnF with FCR, according to the results seen in Figures 8 and 9.

The weighted results for QRT and throughput were also calculated. The weights were obtained as 95% of the FCR and 5% of the normal index by looking at the results of the observations discussed in Section 6.

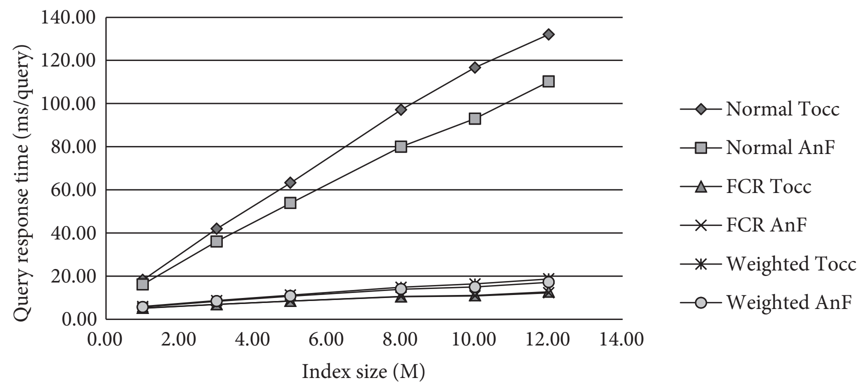

Figure 8. Comparison of query response times.

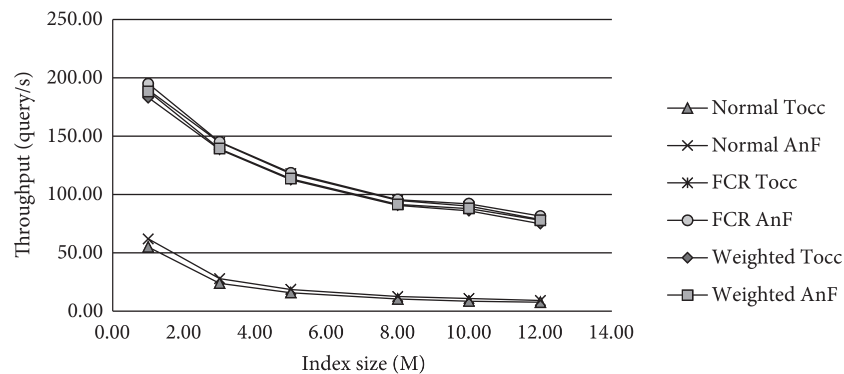

Figure 9. Comparison of throughputs.

9. Conclusion

In the present research work, we developed a novel edit distance aware filtering algorithm for string search problems. We also developed a novel name spell-check engine mainly for social networks. Our experiments showed that our edit distance aware filtering mechanism improved the query processing time and throughput by almost 30%. Moreover, our name spell-check engine improved the name spell-check response time and throughput almost 10 times by using our filtering scheme and some domain specific observations. Based on these promising results, we plan to extend our algorithm to other applications such as data cleaning and next-generation DNA sequencing and alignment.

References

[1] White T. Hadoop: The Definitive Guide. Sebastopol, CA, USA: O'Reilly Media, 2009.

[2] Fitzpatrick B. Distributed caching with memcached. Linux Journal 2004; 2004: 124-129.

[3] George L. HBase: the definitive guide. Sebastopol, CA, USA: O'Reilly Media, 2011.

[4] Xin H, Lee D, Hormozdiari F, Yedkar S, Mutlu O, Alkan C. Accelerating read mapping with FastHASH. BMC Genomics 2013; 14: S13-S14.

[5] Levenstein V. Binary codes capable of correcting spurious insertions and deletions of ones. Probl Inf Transm 1965; 1: 8-17.

[6] Sarawagi S, Kirpal A. Efficient set joins on similarity predicates. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data; 13–18 June 2004; Paris, France. New York, NY, USA: ACM. pp. 743-754.

[7] Bayardo RJ, Ma Y, Srikant R. Scaling up all pairs similarity search. In: WWW 2007 Proceedings of the 16th International Conference on World Wide Web; 8 May 2007; Banff, Alberta, Canada. New York, NY, USA: ACM. pp. 131-140.

[8] Sutinen E, Tarhio J. On using q-gram locations in approximate string matching. In: Spirakis P, editor. Algorithms ESA '95. Berlin, Germany: Springer, 1995. pp. 327-340.

[9] Ukkonen E. Approximate string-matching with q-grams and maximal matches. Theor Comput Sci 1992; 92: 191-211.

[10] Chowdhury G. Introduction to Modern Information Retrieval. London, UK: Facet Publishing, 2010.

[11] Cutting D, Pedersen J. Optimization for dynamic inverted index maintenance. In: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval; 1990; Brussels, Belgium. New York, NY, USA: ACM. pp. 405-411.

[12] Chaudhuri S, Ganjam K, Ganti V, Motwani R. Robust and efficient fuzzy match for online data cleaning. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data; 9–12 June 2003; San Diego, CA, USA. New York, NY, USA: ACM. pp. 313-324.

[13] Navarro G. A guided tour to approximate string matching. ACM Comput Surv 2001; 33: 31-88.

[14] Li C, Lu J, Lu Y. Efficient merging and filtering algorithms for approximate string searches. In: IEEE 2008 Proceedings of the 24th International Conference on Data Engineering; 7–12 April 2008; Cancun, Mexico. New York, NY, USA: IEEE. pp. 257-266.

[15] Kim M, Whang K, Lee J, Lee M. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In: VLDB 2005 Proceedings of the 31st International Conference on Very Large Data Bases; 30 August–2 September 2005; Trondheim, Norway. New York, NY, USA: ACM. pp. 325-336.

[16] Li C, Wang B, Yang X. Vgram: improving performance of approximate queries on string collections using variable-length grams. In: VLDB 2007 Proceedings of the 33rd International Conference on Very Large Data Bases; 23–27 September 2007; Vienna, Austria. New York, NY, USA: ACM. pp. 303-314.

[17] Li H, Homer N. A survey of sequence alignment algorithms for next-generation sequencing. Brief Bioinform 2010; 11: 473-483.