Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Efficient Search Engine Using Proximity Ranking & Instant Fuzzy Logic

Ratnesh Kumar Choudhary
M.Tech, Scholar, CSE Dept,
NCET Nagpur, India

Prof. Shyam Prakash Dubey
Assistant Prof. & Head CSE Dept,
NCET Nagpur, India

Prof. Anil Warbhe
Assistant Professor,
MPIET Gondia, India

Abstract— Efficient Search is an immerging information retrieval technology used to retrieve information from the World Wide Web or say internet. It uses various techniques together to achieve good quality results in minimum time. It uses mainly fuzzy logic, proximity ranking, top k-query algorithm and term-pair indexing techniques. When user searches then sometimes they make typographical mistakes which are corrected by fuzzy technique. Also it allows system to rank the result pages so that it can show the most desired result page first then less desired one. This is done by proximity ranking technique. The top-k query algorithm is used to limit the search result pages so that it shows only most relevant pages only, the value of k decides how many pages will be shown in output. The main challenge in this work is to minimize the search time so it also implements instant search technique. A new algorithm has been implemented for indexing of pages which is called term-pair indexing, also it uses new ranking algorithm called proximity ranking. This paper includes implementation of all above techniques together to create efficient search engine. It also shows various experiments carried out on this system. It also presents deep analysis of efficiency & performance of this new system and old one.

Keywords— Efficient, Search Engine, User, Keyword, Ranking, fuzzy logic, Instant, top-k query, term-pair Indexing.

I. INTRODUCTION

This document is a User often makes typographical mistakes in their search queries. Also small keywords on mobile device, lack of knowledge & lack of caution cause mistakes in search keywords. Due to these mistakes user do not get relevant answers in the output results. This problem can be easily solved by fuzzy system. In the fuzzy search described in [17][18][19][20], user can find answers based on similar keywords [1]. For example Figure 1 shows instant fuzzy search interface & its result on people directory. The system finds the answer to the query *"facebk"* even though the user mistyped a prefix of the name *"Facebook"*. Combining fuzzy search with instant search can provide an even better search experience, especially for mobile phone users. Instant search is an emerging search system which provides the answers immediately based on partial query typed by the user in the search interface. For example when the user type *"fa"* , it immediately returns the answers such as *"face"*, *"facebook"*, *"fault"*, *"famous"* etc. Many users like this kind of experience of searching where results appear before user type the complete keywords i.e. type-ahead search which is described in [11][13][15][16].

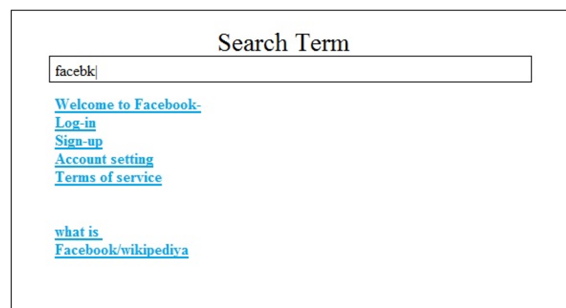


Fig: 1(Search System UI with Search Example)

Finding relevant answer within time limit is a challenge for the search engine developers. The total time should be less than 100 milliseconds [2]. This time includes the network delay, the time on the search server and the time of running the code on the user device. In instant search the system needs to find the answers with partial query as user progress typing, therefore it becomes more challenging task to meet the time requirement [3]. To mitigate above problem solution has been implemented in this dissertation.

II. SEARCH SYSTEM OPERATION

To deal with a large data set that cannot be indexed by a single machine, It is assumed that the data is partitioned into multiple shards to ensure the scalability. Each server builds the index structures on its own data shard, and is responsible

© 2015, IJARCSSE All Rights Reserved

for finding the answers to a query in its shard. The Broker on the Web server receives a query for each keystroke of a user. The Broker is responsible for sending the requests to multiple search servers, retrieving and combining the results from them, and returning the answers back to the user. Figure 2 shows the query flow in a server for one shard. When a search server receives a request, it first identifies all the phrases in the query that are in the dictionary D, and intersects their inverted lists. For this purpose, we have a module called Phrase Validator that identifies the phrases (called "valid phrases") in the query q that is similar to a term in the dictionary D. For example, for the query q = <heart, surgery>, "heart" is a valid phrase for the data set in Table I, since the dictionary contains

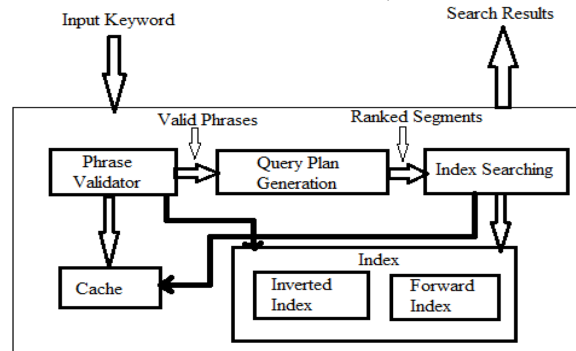
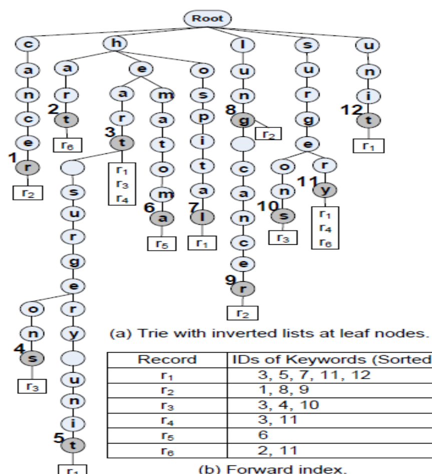


Fig: 2 (Single shard Single server architecture of search engine)

the similar terms "heart" and "hart". In addition, "surgery" and "heart surgery" are also valid phrases. To identify all the valid phrases in a query, the Phrase Validator uses the tree-based algorithm in [14], which can compute all the similar terms to a complete or prefix term efficiently. The Phrase Validator computes and returns the active nodes for all these terms, i.e., those tree nodes whose string corresponding to the path from the root to this node is similar to the query phrase. If a query keyword appears in multiple valid phrases, the query can be segmented into phrases in different ways. Let "|" denote a place between two adjacent valid phrases. For instance, "heart | surgery" and "heart surgery" are two different segmentations for q. The query segmentations that consist of only valid phrases as valid segmentations has been referred. After identifying the valid phrases, the Query Plan Builder generates a Query Plan Q, which contains all the possible valid segmentations in a specific order. The ranking of Q determines the order in which the segmentations will be executed. After Q is generated, the segmentations are passed into the Index Searcher one by one until the top-k answers are computed, or all the segmentations in the plan are used. The Index Searcher uses the algorithm described in [4] to compute the answers to segmentation. A result set is then created by combining the result sets of the segmentations of Q. Since the subsequent queries of the user typically share many keywords with previous queries due to incremental typing, it is very important to do the computation incrementally and distribute the computational cost of a query between its preceding queries. For this reason, a Cache module that stores some of the computed results of early queries that can be used to expedite the computation of later queries. The Phrase Validator uses the Cache module to validate a phrase without traversing the tree from scratch, while the Index Searcher benefits from the Cache by being able to retrieve the answers to an earlier query to reduce the computational cost.

III. PHRASE-BASED INDEXING FOR TOP K-QUERY

Intuitively, a phrase is a sequence of keywords that has high probability to appear in the records and queries [5][6]. This technique has been used to utilize phrase matching to improve ranking in this top-k computation framework [7]. It assumes an answer having a matching phrase in the query has a higher score than an answer without such a matching phrase. To be able to still do early termination [4], it accesses the records containing phrases first. For instance, for the query q=<heart, surgery>, it accesses the records containing the phrase "heart surgery" before the records containing "heart" and "surgery" separately.



Record	IDs of Keywords (Sorted)
r_1	3, 5, 7, 11, 12
r_2	1, 8, 9
r_3	3, 4, 10
r_4	3, 11
r_5	6
r_6	2, 11

Fig: 3(Inverted Index Tree & Forward Index)

© 2015, IJARCSSE All Rights Reserved

Notice that the framework sorts the inverted list of a keyword based on relevancy of its records to the keyword. If it order the inverted list of the keyword "surgery" based on the relevancy to the phrase "heart surgery", the best processing order for another phrase, say, "plastic surgery", may be different. Based on this analysis, it need to index phrases to be able to retrieve the records containing these phrases efficiently. However, the number of phrases up to a certain length in the data set can be much larger than the number of unique words [29]. Therefore, indexing all the possible phrases can require a large amount of space [5]. To reduce the space overhead we need to identify and index those phrases that are more likely to be searched. It considers a set of important phrases E that are likely to be searched for indexing, where each phrase appears in records of R. The set E can be determined in various ways such as person names, points of interest, and popular n-grams in R. Examples include Michael Jackson, New York City, and Hewlett Packard. Let W be the set of all distinct words in R. We will refer the set $W \cup E$ as the dictionary D, and call each item $t \in D$ a term. In Table I, the indexed phrases are shown in bold. Figure 3 shows the index structures for the sample data. For instance, the phrase "heart surgery unit" is indexed in the tree in Figure 3(a), in addition to the keywords "heart", "surgery", and "unit". The leaf nodes corresponding to these terms are numbered as 5, 3, 1_1, and 1_2, respectively. The leaf node for the term "heart" points to its inverted list that contains the records r_1, r_3, and r_4. In addition, Figure 3(b) shows the forward index, where the keyword id 3 for the term "heart" is stored for these records. Early-termination techniques can also be used for top k-query computation which is given in [21], [22], [23], [5], [6], [7].

IV. VALID PHRASES GENERATION

A. Incremental Computation of Valid Phrases

A query with l keywords can be segmented into m phrases in $\binom{l-1}{m-1}$ different ways, because there are l − 1 places to choose for m − 1 separators to obtain m phrases. Therefore, the total number of possible segmentations, 2l−1, grows exponentially as the number of query keywords increases. Fortunately, the typical number of keywords in a search query is not large. For instance, in Web search it is between 2 and 4 [30].The basic approach describe in [14] does not utilize the fact that the subsequent queries of a user typically differ from each other by one character, and their valid-phrase computations have a lot of overlap [30][14]. The valid phrases of q_i are cached to be used for later queries that start with the keywords of qi. Figure 4 shows the active nodes of the valid phrases in the queries q_1=<heart,surge>, q_2=<heart, surgery>, and q_3=<heart, surgery, unit>. In the figure, q_1 and q_2 have the same active nodes n_1 and n_2 for the phrase "heart". Moreover, the phrase "surgery" in q_2 has an active node n_5, which is close to the active node n_3 of phrase "surge" in q_1. Similarly, the phrase "heart surgery" in q_2 has an active node n_6, which is close to the active node n4 of phrase "heart surge" in q_1. Hence, it can use the active nodes n_3 and n_4 to compute n_5 and n6 efficiently. The key observation in this example is that the computation is needed only for the phrases containing the last query keyword. If a query q_j extends a query q_i by appending additional characters to the last keyword w_l of q_i, then each valid phrase of q_i that ends with a keyword other than w_l is also a valid phrase of q_j . The valid phrases of q_i that end with the keyword w_l have to be extended to be valid phrases of q_j. The new active-node set can be computed by starting from the active node set of the cached phrase, and traversing the tree for the additional characters to determine if the phrase is still valid.

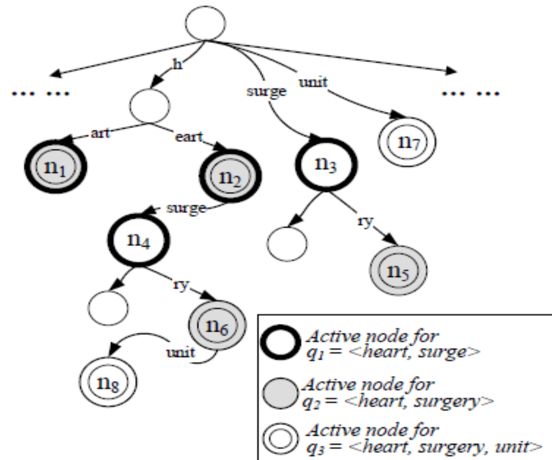


Fig: 4(Active node computation for valid phrases)

Another case where it can use the cached results of the query q_i is when the query q_j has additional keywords after the last keyword w_l of q_i. The queries q_2 and q_3 in Figure 4 are an example of this case. In this example, all the active nodes of q_2 (i.e., n_1, n_2, n_5, and n_6) are also active nodes for q_3. In addition to these active nodes, q_3 has the active nodes n_7 and n_8 for the phrases that contain the additional keyword "unit" (i.e., "unit" and "heart surgery unit"). The phrase "unit" is a new phrase, and its active node (n_7) is computed from scratch. However, the phrase "heart surgery unit" has a phrase from q_2 as a prefix, and its active node n_8 can be computed incrementally starting from n_6. As seen in the example, if the query q_j has additional keywords after the last keyword w_l of q_i, then all of the valid phrases of q_i are also valid in q_j . Moreover, some of the valid phrases of q_i that end at w_l can be extended to become valid phrases of q_j . If a phrase starting with the mth keyword of q_i, w_m ($m \leq l$), can be extended to a phrase containing the nth keyword of q_j , w_n ($l < n$), the phrase $w_m \ldots w_n$ can be computed by using the valid phrase $w_m \ldots w_l$ of q_i. Based on these observations, it cache a vector of valid phrases V_i for a query qi with the following properties: (1) V_i has an element for each keyword in qi, i.e.,

© 2015, IJARCSSE All Rights Reserved

$|V_i| = l$; (2) The nth element in V_i is a set of starting points of the valid phrases that end with the keyword wn and their corresponding active-node sets. Figure 5 shows the vectors of valid phrases V_1, V_2, and V_3 for the queries q_1, q_2, and q_3, respectively. For example, the third element of V_3 (shaded) shows the starting points of all the valid phrases ending with a prefix similar to "unit" and their active-node sets. In other words, the pair "(1, $S_{1,3} = \{n_8\}$)" means that the tree node n_8 in $S_{1,3}$ represents a term prefix in the dictionary that is similar to the phrase "heart surgery unit". Notice that all the end points in V_i also have themselves as the starting point, since each keyword can be a phrase by itself. Therefore an algorithm for computing the valid phrases of a query incrementally using previously cached vector of valid phrases has been developed. The pseudo code is shown in Algorithm 1. As an example,

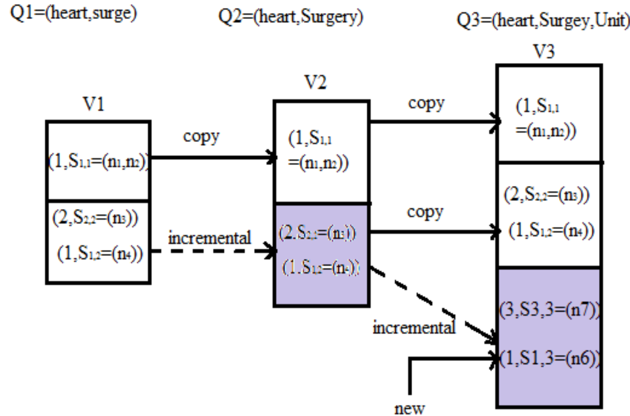


Fig: 5(Incremental computation of valid phrase using cache valid phrase)

Figure 5 shows how a cached valid-phrase vector is used for incremental computation. Assuming V_1 in the figure is stored in the cache, vector V_2 can be incrementally computed using V_1 as follows. First, the first element of V_1 is copied to V_2, because q_1 and q_2 share the same first keyword (lines 4–5). Then, the second element of V_2 is computed incrementally starting from the active-node sets $S_{2,2}$ and $S_{1,2}$ in the second element of V_1 (lines 8–14). The incremental computation from V_2 to V_3 is an example case where there are additional keywords in the new query. In this case, it copies the first two elements of V_2 to V_3 since the queries share their first two keywords. We compute the third element of V_3 based on the active-node sets of the second element of V_2 (lines 15–21). In particular, it traverses the tree starting from nodes n_5 and n_6 to see if it contains a term prefix similar to "surgery unit" or "heart surgery unit', respectively. The traversal results in no active node for n_5 and the active node n_8 for n_6. Thus it add the pair (1, $S_{1,3} = \{n_8\}$) to the third element of V_3, indicating that there is a valid phrase starting from the 1st keyword and ending at the 3rd keyword. We also add an element (3, $S_{3,3} = \{n_7\}$) for the 3rd keyword "unit" since it is also a valid phrase with an active node n_7 (lines 22–30).

Algo. 1: CompValPhrase(q, C)

Input: keywords $q=(w_1,w_2,\ldots w_t)$ where w_i is a keyword,
& C is Cache module
Output: a valid-Phrase vector **V**;

```
1.  (q_c, V_c)← FindLongCachePrefix(q, C)
2.  M←no. of keywords in q_c
3.  if m>0 then // cache hit
4.      for i←1 to m-1 do //copy valid phrase vector
5.              V[i]←V_c[i]
6.      if W_m==q_c[m] then // q_c[m] last word is complete keyword
7.              V[m]←V_c[m]
8.      else  // computation for last keyword retrieved from C
9.              V[m]←Null
10.             foreach (start, S) in V_c[m] do
11.                     newS←find active nodes for w_m from S
12.                     if newS== Null then
13.                       V[m]←V[m]U(start,newS)
14.     foreach(start, S) in V[m] do // compute for partially cached phrase
15.                 for j←m+1 to l do
16.                     newS←compute active nodes from S by appending w_j
17.                     if  newS==Null then break
18.                     V[j]←V[j]U(start, newS)
19.                     S← newS
20. for  i←m+1 to l do  // computes uncached phrase
21.         S←compute active nodes for Wi
22.         V[i]←V[i]U(i,S)
23.         for j←i+1 to l do
```

24. **newS**← compute active nodes from S by appending w_j
25. **if newS**==Null **then** break
26. V[j]←V[j]U(i,**newS**)
27. S←**newS**
28. Cache(q,V) in C
29. **Return** V

V. EFFICIENT QUERY PLANS COMPUTATION

A. Generating Valid Segmentations

After receiving a list of valid phrases, the Query Plan Builder computes the valid segmentations. The basic segmentation is the one where each keyword is treated as a phrase. For example, for the query q = <heart, surgery, unit>, “heart | surgery | unit” is the basic segmentation. If there are multi keyword phrases in the query, then there will be other segmentations as well. In the running example, “heart surgery” is a valid phrase, and “heart surgery | unit” is a segmentation. Table I shows all possible segmentations that can be generated from the valid phrases vector V_3 in Figure 5.

A divide-and-conquer algorithm is developed for generating all the segmentations from the valid-phrase vector V. Each phrase has a start position and an end position in the query. The start position is stored in V [end] along with its computed active-node set. If there is a segmentation for the query $<w_1,\ldots,w_{start-1}>$, we can append the phrase [start, end] to it to obtain a segmentation for the query $<w_1,\ldots,w_{end}>$. Therefore, to compute all the segmentations for the first j keywords, we can compute all the segmentations for the first $i-1$ keywords, where $(i, S_{i,j}) \in V[j]$, and append the phrase [i,j] to each of these segmentations to form new segmentations.

Table- I. Segmentations for Query
q = <heart, surgery, unit>

1. “heart surgery unit”
2. “heart surgery | unit”
3. “heart | surgery | unit”

This analysis helps us reduce the problem of generating segmentations for the query $<w_1,\ldots,w_l>$ to solving the sub-problems of generating segmentations for each query $<w_1,\ldots,w_{i-1}>$, where $(i, S_{i,l}) \in V_{[l]}$. Hence, the final segmentations can be computed by starting the computation from the last element of V. Algorithm 2 shows the recursive algorithm. Line 3 is the base case for the recursion, where the start position of the current phrase is the beginning of the query. The recursive algorithm can be converted into a top down dynamic programming algorithm by memorizing all the computed results for each end position.

Algo. 2: GenSeg(q, V, end)

Input: any query $q=(w_1,w_2,\ldots,w_t)$;
Its valid phrase vector V;
a keyword position *end* (*end*<=l);
Output: a vector P_{end} having valid segmentation of w_i;
1. P_{end}← Null
2. **foreach**(start, S_{start},end) **in** V[end] **do**
3. **if** start = =1 **then** //base case
4. P_{end}←P_{end} U ($w_{start}\ldots\ldots w_{end}$)
5. **else**
6. **foreach** sed **in**
7. GenSeg(q,V,start-1)
8. **do**
9. seg←seg|($w_{start}\ldots w_{end}$)
10. P_{end}←P_{end} U seg
11. **return** P_{end}

VI. RANKING SEGMENTATIONS

Each generated segments corresponds to a way of accessing the indexes to compute its answers. The Query Plan Builder needs to rank these segmentations to decide the final query plan, which is an order of segmentations to be executed. These segmentations can be run one by one until it finds enough answers (i.e., k no. of results). Thus, the ranking needs to guarantee that the answers to high-rank segmentation are more relevant than the answers to low-rank segmentation. There are different methods to rank segmentation. Our segmentation ranking relies on a segmentation comparator to decide the final order of the segmentations. This comparator compares two segmentations at a time based on the following Features and decides which segmentation has a higher ranking:

(1)The summation of the minimum edit distances between each valid phrase in the segmentation and its active nodes; (2) the number of phrases in the segmentation. The comparator ranks the segmentation that has the smaller minimum edit distance summation higher. If two segmentations have the same total minimum edit distance, then it ranks the segmentation with fewer segments higher. As an example, for the query q = <hart, surgery>, consider the segmentation

"hart | surgery" with two valid phrases. Each of them has an exact match in the dictionary D, so its summation of minimum edit distances is 0. Consider another segmentation "hart surgery" with one valid phrase. This phrase has an edit distance 1 to the term "heart surgery", which is minimum. Using this method, it would rank the first segmentation higher due to its small total edit distance. If two segmentations have the same total minimum edit distance, then it will rank the segmentation with fewer segments higher. When there are fewer phrases in segmentation, the number of keywords in a phrase increases. Having more keywords in a phrase can result in better answers because more keywords appear next to each other in the answers. The segmentations in Table-I are ranked based on this feature. If two segmentations have both the same total minimum distance and the number of phrases, then it assumes that they have the same rank. Notice that the answers to the segmentation where each keyword is a separate phrase include the answers to all the other segmentations. Therefore, once this segmentation is executed, there is no need to execute the rest of the segmentations in the plan. In the q = <hart, surgery> example, the segmentation "hart surgery" is discarded from the query plan since the segmentation "hart | surgery" is ranked higher due to its smaller edit distance.

VII. EXPERIMENTS

In this section, the performance of the proposed techniques on real data sets has been evaluated. In Query Segmentation ("QS") approach, it computes a query plan based on valid segmentations, and rum the segmentations one by one until top-k answers were computed.

Table- II. Datasets.

Data Set	*IMDB*	*Eron*	*Medline*[2]
# of records(millions)	0.7	0.5	10
# of distinct Keywords (millions)	0.76	1	4.6
Average record length	40	294	132
Data Size	238 MB	969 MB	20 GB

Three data sets in the experiments, namely IMDB, Enron, and Medline have been used. Table II shows the details of the data sets. The IMDB data set has been obtained from www.imdb.com/interfaces. It uses the data in the movies, actors, and characters tables, and constructed a table in which each record was a movie with a list of actors and a list of characters. For this data set, it has extracted the queries from an AOL query log3, and selected those queries whose clicked domain was IMDB.com. The Enron data set consisted of email records with attributes such as date, sender, receiver, subject, and body. For this data set, it has used the queries provided by [31]. The Medline data set consisted of more than 20 million medical publications, and it has used its subsets of different sizes to do experiments. For this data set, it has used a single-day query log from PubMed as analyzed in [32]. Based on user-behavior statistics in instant search reported in [1], we assumed 12% of the queries were copied and pasted while the other queries were typed character by character. In the experiments of fuzzy search, it has used 1/3 as the normalized edit-distance threshold. That is, it allowed no typographical errors if a query prefix is within 3 characters, up to 1 error if the length is between 4 and 6, up to 2 errors if the length is between 7 and 9, and so on. All the experiments were conducted on a Wamp server running a Windows 7 64-bit operating system, with two 2.2GHz Intel core 2 duo processors, 3000 GB of RAM memory, and 2 TB of hard disk. To satisfy the high-efficiency requirement of instant search, all index structures were stored in memory.

A. *Efficiency of Computing Valid Phrases*

When the number of keywords increased, the computation time also increased. This is because when the query had more keywords, the number of phrases that needed to be validated also increased. The Incremental algorithm improved the efficiency tremendously. For instance, for 6-keyword queries in Medline, the computation time was reduced from 64 ms to 3 ms . The reason for this improvement is that the Incremental algorithm avoided computation of previously computed phrases by using cached result. It is observed for the similar trends for the other data sets.

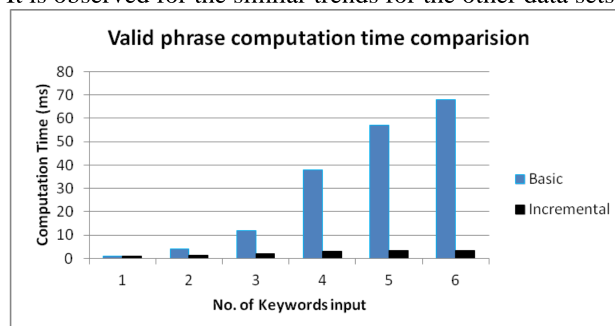


Fig: 6 (Comparison of basic & Incremental approach for valid phrase computation)

B. *Query Time & Scalability*

The query time & scalability of QS using the Medline data set has been evaluated. Figure 7 shows the average computation time for varying number of keywords in the query as we increased the number of records from 1 million to

10 millions. It has been observed that the average computation time increased linearly in this approach as we increased the number of records. Another interesting observation was that for 2-keyword and 3-keyword queries QS gave the best results. For instance, using 10 million records, the average computation time for 2-keyword queries was 32 milliseconds in QS. The QS can generate a lot of valid segmentations with a few answers due to high selectivity of the query, and running those segmentations until the top-k answers were computed degraded the performance significantly. For 2-keyword and 3-keyword queries, the answer sets can be very large, thus early-termination [4] improved the performance remarkably. For 1-term queries, this approach returned the top-k elements from the union of the inverted lists of similar complete keywords to the given prefix. This experiment also showed that QS approach is indeed very useful since users predominantly use 2 and 3 keyword queries [34].

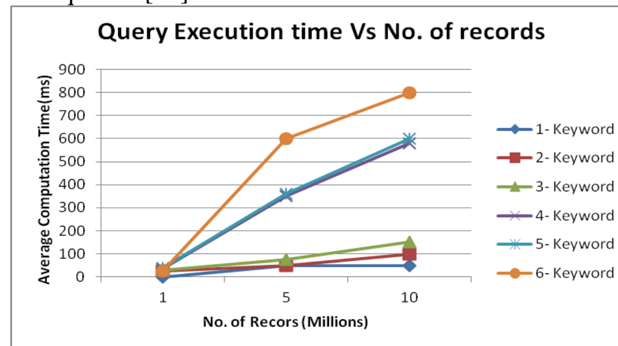


Fig: 7 (Query execution time for various no. of keywords)

We summarize the experimental results as follows:

- QS Works the best for 2-keyword and 3-keyword queries, which are common in search applications.
- Benefits of QS increase as the size of the data increases.

VIII. CONCLUSION

In this paper the improve ranking of an instant-fuzzy search system by considering proximity information is used where it needs to compute top-k answers. The existing solutions have been adopted to solve timing problem. A technique to index important phrases to avoid the large space overhead of indexing all word grams has been implemented. It presents an incremental-computation algorithm for finding the indexed phrases in a query efficiently, to compute and rank the segmentations consisting of the indexed phrases. An experiment is conducted & a very thorough analysis by considering space, time, and relevancy tradeoffs of this approach has been done. In particular, the experiments on real data showed the efficiency of the proposed technique for 2-keyword and 3-keyword queries that are common in search applications. It has been concluded that computing all the answers for the other queries would give the best performance and satisfy the high- efficiency requirement of instant search.

REFERENCES

[1] I. Cetindil, J. Esmaelnezhad, C. Li, and D. Newman, "Analysis of instant search query logs," in WebDB, 2012, pp. 7–12.

[2] R. B. Miller, "Response time in man-computer conversational transactions," in Proceedings of the December 9-11, 1968, fall joint computer conference, part I, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. [Online]. Available:

[3] C. Silverstein, M. R. Henzinger, H. Marais, and M. Moricz, "Analysis of a very large web search engine query log," SIGIR Forum, vol. 33,no. 1, pp. 6–12, 1999.

[4] G. Li, J. Wang, C. Li, and J. Feng, "Supporting efficient top-k Queries in type-ahead search," in SIGIR, 2012, pp. 355–364.

[5] R. Schenkel, A. Broschart, S. won Hwang, M. Theobald, and G. Weikum, "Efficient text proximity search," in SPIRE, 2007, pp. 287–299.

[6] H. Yan, S. Shi, F. Zhang, T. Suel, and J.-R. Wen, "Efficient term proximity search with term-pair indexes," in CIKM, 2010, pp. 1229– 1238.

[7] M. Zhu, S. Shi, N. Yu, and J.-R. Wen, "Can phrase indexing help to process non-phrase queries?" in CIKM, 2008, pp. 679–688.

[8] A. Jain and M. Pennacchiotti, "Open entity extraction from web Search query logs," in COLING, 2010, pp. 510–518.

[9] K. Grabski and T. Scheffer, "Sentence completion," in SIGIR, 2004, pp.433–439.

[10] A. Nandi and H. V. Jagadish, "Effective phrase prediction," in VLDB,2007, pp. 219–230.

[11] H. Bast and I. Weber, "Type less, find more: fast autocompletion Search with a succinct index," in SIGIR, 2006, pp. 364–371.

[12] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber, "Ester: efficient search on text, entities, and relations," in SIGIR, 2007, pp. 671– 678.

[13] H. Bast and I. Weber, "The completesearch engine: Interactive, efficient, and towards ir& db integration," in CIDR, 2007, pp. 88– 95.

© 2015, IJARCSSE All Rights Reserved

[14] S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," in WWW, 2009, pp. 371–380.
[15] S. Chaudhuri and R. Kaushik, "Extending autocompletion to Tolerate errors," in SIGMOD Conference, 2009, pp. 707–718.
[16] G. Li, S. Ji, C. Li, and J. Feng, "Efficient type-ahead search on relational data: a tastier approach," in SIGMOD Conference, 2009, pp. 695–706.
[17] M. Hadjieleftheriou and C. Li, "Efficient approximate search on String collections," PVLDB, vol. 2, no. 2, pp. 1660–1661, 2009.
[18] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin, "An efficient Filter for approximate membership checking," in SIGMOD Conference, 2008,pp. 805–818.
[19] S. Chaudhuri, V. Ganti, and R. Motwani, "Robust identification of fuzzy duplicates," in ICDE, 2005, pp. 865–876.
[20] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in ICDE, 2009, pp.604–615.
[21] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in PODS, 2001.
[22] F. Zhang, S. Shi, H. Yan, and J.-R. Wen, "Revisiting globally sorted indexes for efficient document retrieval," in WSDM, 2010, pp. 371–380.
[23] M. Persin, J. Zobel, and R. Sacks-Davis, "Filtered document retrieval with frequency-sorted indexes," JASIS, vol. 47, no. 10, pp. 749–764,1996.
[24] R. Song, M. J. Taylor, J.-R. Wen, H.-W. Hon, and Y. Yu, "Viewing term proximity from a different perspective," in ECIR, 2008, pp. 346–357.
[25] T. Tao and C. Zhai, "An exploration of proximity measures in information retrieval," in SIGIR, 2007, pp. 295–302.
[26] M. Zhu, S. Shi, M. Li, and J.-R. Wen, "Effective top-k computation in retrieving structured documents with term-proximity support," in CIKM, 2007, pp. 771–780.
[27] S. B¨uttcher, C. L. A. Clarke, and B. Lushman, "Term proximity scoring for ad-hoc retrieval on very large text collections," in SIGIR, 2006, pp.621–622.
[28] H. Zaragoza, N. Craswell, M. J. Taylor, S. Saria, and S. E. robertson,"Microsoft cambridge at trec 13: Web and hard tracks," in TREC, 2004.
[29] A. Franz and T. Brants, "All our n-gram are belong to you," http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are- belongto-you.html, Aug. 2006.
[30] A. Arampatzis and J. Kamps, "A study of query length," in SIGIR, 2008, pp. 811–812.
[31] Z. Bao, B. Kimelfeld, and Y. Li, "A graph approach to spelling correction in domain-centric search," in ACL, 2011.
[32] J. R. Herskovic, L. Y. Tanaka, W. R. Hersh, and E. V. Bernstam, "Research paper: A day in the life of pubmed: Analysis of a typical day's query log," JAMIA, vol. 14, no. 2, pp. 212–220, 2007.
[33] D. R. Morrison, "Patricia - practical algorithm to retrieve information coded in alphanumeric," J. ACM, vol. 15, no. 4, pp. 514–534, 1968.
[34] "Keyword and search engines statistics," http://www.keyworddiscovery.com/keywordstats.html?date=2013-06-01, Jun. 2013.

© 2015, IJARCSSE All Rights Reserved