International Journal of Engineering Research and General Science Volume 2, Issue 6, October-November, 2014
ISSN 2091-2730

IMPROVED FUZZY TECHNOLOGY FOR EFFICIENT SEARCHING

Mr. R. P. Sabale
Department of Computer Engineering.
G.H. Raisoni College of Engineering and Management Ahmednagar

Abstract—Instant search is an information-retrieval in which a system finds answers to a query instantly while a user types in keywords character-by-character. Fuzzy search further improves user search experiences by finding relevant answers with keywords similar to query keywords. A main computational challenge in this the high-speed requirement, i.e., each query needs to be answered within milliseconds to achieve an instant response and a high query throughput. At the same time, we also need good ranking functions that consider the proximity of keywords to compute relevance scores.

In this paper, we study how to integrate proximity information into ranking in instant-fuzzy search while achieving efficient time and space complexities. A naıve solution is computing all answers then ranking them, but it cannot meet this high-speed requirement on large data sets when there are too many answers, so there are studies of early-termination techniques to efficiently compute relevant answers. To overcome the space and time limitations of these solutions, we propose an approach that focuses on common phrases in the data and queries, assuming records with these phrases are ranked higher. We study how to index these phrases and develop an incremental-computation algorithm for efficiently segmenting a query into phrases and computing relevant answers.

I. INTRODUCTION

Instant Search: As an emerging information-access it returns the answers immediately based on a partial query a user has typed in. Many users prefer the experience of seeing the search results instantly and formulating their queries accordingly instead of being left in the dark until they hit the search button

Fuzzy Search: Users often make typographical mistakes in their search queries. Meanwhile, small keyboards on mobile devices, lack of caution, or limited knowledge about the data can also cause mistakes. In this case we cannot find relevant answers by finding records with keywords matching the query exactly. This problem can be solved by supporting fuzzy search, in which we find answers with keywords similar to the query keywords. Combining fuzzy search with instant search can provide an even better search experiences, especially for mobile-phone users, who often have the "fat fingers" problem, i.e., each keystroke or tap is time consuming and error prone.

Finding Relevant Answers within Time Limit It is known that to achieve an instant speed for humans (i.e., users do not feel delay), from the time a user types in a character to the time the results are shown on the device, the total time should be within 100 milliseconds [2]. The time includes the network delay, the time on the search server, and the time of running code on the device of the user (such as JavaScript in browsers). Thus the amount of time the server can spend is even less. At the same time, compared to traditional search systems, instant search can result in more queries on the server since each keystroke can invoke a query, thus it requires a higher speed of the search process to meet the requirement of a high query throughput. What makes the computation even more challenging is that the server also needs to retrieve *high-quality* answers to a query given a limited amount of time to meet the information need of the user.

Problem Statement: In this paper, we study the following problem: *how to integrate proximity information into ranking in instant-fuzzy search to compute relevant answers efficiently?* The proximity of matching keywords in answers is an important metric to determine the relevance of the answers. Search queries typically contain correlated keywords, and answers that have these keywords together are more likely what the user is looking for [3].

Our Contributions: We study various solutions to this important problem and show the insights on the tradeoffs of space, time, and answer quality. One approach is to first find all the answers, compute the score of each answer based on a ranking function, sort them using the score, and return the top results. However, enumerating all these answers can be computationally expensive when these answers are too many. This case is more likely to happen compared to a traditional search system since query keywords in instant search are treated as prefixes and can have many completions. In addition, fuzzy search makes the situation even more challenging

since there can be many keywords with a prefix similar to a query prefix. As a consequence, the number of answers in instant fuzzy search is much larger than that in traditional search.

An efficient way to address the problem is to use early termination techniques that allow the engine to find top answers without generating all the answers of the query [4]. The main idea is to traverse the inverted index of the data following a certain order, and stop the traversal once we are sure that the most relevant results are among those records we have visited. The traversal order of the inverted index is critical to be able terminate the traversal sooner. However, using a proximity aware ranking in early termination is challenging, because the document order in the inverted index is typically based on individual keywords. At the same time, proximity information is between different keywords and does not depend on the order of an inverted list.

There are studies on building an additional index for each term pair that appears close to each other in the data, or for phrases [5], [6], [7]. However, building an index for the term pairs will consume a significant amount of space. For instance, the approach in [5] reported an index of 1.3 TB for a collection of 25 million documents, and reduced the size to 343.5 GB by pruning the lists horizontally. In addition, these studies focus on two-keyword queries only, and do not consider queries with more keywords.

Studies show that users often include entities such as people names, companies, and locations in their queries [8]. These entities can contain multiple keywords, and the user wants these keywords to appear in the answers as they are, i.e., the keywords are adjacent and in the same order in the answers as in the query. Users sometimes enter keywords enclosed by quotation marks to express that they want those keywords to be treated as phrases [3]. Based on this observation, we propose a technique that focuses on the important case where we rank highly those answers containing the query keywords as they are, in addition to adapting existing solutions to instant-fuzzy search. To overcome the known limitations of existing solutions, we propose an approach that indexes additional common phrases in addition to indexing single terms. This method can not only avoid the space overhead of indexing all the term pairs or phrases, but also improve ranking significantly by efficiently finding relevant answers that contain these common phrases. To find relevant answers, we identify the indexed phrases in the query, then access their inverted lists before accessing single-keyword lists. If the query has different ways to be segmented into phrases, we consider all these segmentations and rank them. Each segmentation corresponds to a unique index-access strategy to execute the query. We execute the ranked segmentations one by one until we compute the most relevant answers or enough time is spent. We focus on a main challenge in this approach, which is how to do incremental computation to answer a query so that we do not need to compute the results from scratch for each keystroke.

A. Related Work

Auto-Completion: It system suggests several possible queries the user may type in next. There have been many studies on predicting queries (e.g., [9], [10]). Many systems do prediction by treating a query with multiple keywords as a single prefix string. Therefore, if a related suggestion has the query keywords but not consecutively, then this suggestion cannot be found.

Instant Search: Many recent studies have been focused on instant search, also known as *type-ahead search*. The studies in [11], [12], [13] proposed indexing and query techniques to support instant search. The studies in [14], [15] presented triebased techniques to tackle this problem. Li et al. [16] studied instant search on relational data modeled as a graph.

Fuzzy Search: The studies on fuzzy search can be classified into two categories, gram-based approaches and trie-based approaches. In the former approach, sub-strings of the data are used for fuzzy string matching [17], [18], [19], [20]. The second class of approaches index the keywords as a trie, and rely on a traversal on the trie to find similar keywords [14] , [15]. This approach is especially suitable for instant and fuzzy search [14] since each query is a prefix and trie can support incremental computation efficiently.

Early Termination: Early-termination techniques have been studied extensively to support top-*k* queries efficiently [21], [22], [23], [5], [6], [7]. Li et al. [4] adopted existing top-*k* algorithms to do instant-fuzzy search. Most of these studies reorganize an inverted index to evaluate more relevant documents first. Persin et al. [23] proposed using inverted lists sorted by decreasing document frequency. Zhang et al. [22] studied the effect of term-independent features in index reorganization.

Proximity Ranking: Recent studies show proximity is highly correlated with document relevancy, and proximity aware ranking improves the precision of top results significantly [24], [25]. However, there are only a few studies that improve the query efficiency of proximity-aware search by using early-termination techniques [26], [5], [6], [7]. Zhu et al. [26] exploited document structure to

build a multi-tiered index to terminate the search process without processing all the tiers. The techniques proposed in [5], [6] create an additional inverted index for all term pairs, resulting in a large space. To reduce the index size, Zhu et al. [7] proposed to build a compact phrase index for a subset of the phrases. However, both [6] and [7] studied the problem for two-keyword queries only.

II. PRELIMINARIES

Data: Let $R = \{r_1, r_2, ..., r_n\}$ be a set of records with text attributes, such as the tuples in a relational table or a collection of documents. Let D be the dictionary that includes all the distinct words of R. Table I shows an example data set of medical publication records. Each record has text attributes such as title and authors.

Query: A query q is a string that contains a list of keywords $\langle w_1, w_2, ..., w_l \rangle$, separated by space. In an instant-search system, a query is submitted for each keystroke of a user. When a user types in a string character by character, each query is constructed by appending one character at the end of the previous query. The last keyword in the query represents the word currently being typed, and is treated as prefix, while the first $l-1$ keywords $\langle w_1, w_2, ..., w_{l-1} \rangle$ are complete keywords. (Our techniques can be extended to the case where each keyword in the query is treated as a prefix.) For instance, when a user types in "brain tumor" character by character, the system receives the following queries one by one: q_1 = hbi, q_2 = hbri, ..., q_{10} = hbrain,tumoi, q_{11} = hbrain,tumori.

Answers: A record r from the data set R is an answer to the query q if it satisfies the following conditions: (1) for $1 \le i \le l - 1$, it has a word *similar to* w_i, and (2) it has a keyword with a prefix *similar to* w_l. The meaning of "*similar to*" will be explained shortly. For instance, r_1, r_3, and r_4 are answers to q = hheart, surgei, because all of them contain the keyword "heart". In addition, they have words "surgery", "surgeons", and "surgery", respectively, each of which has a prefix similar to "surge". Record r_6 is also an answer since it has an author named "hart" similar to the keyword "heart", and also contains "surgery" with a prefix "surge" matching the last keyword in the query.

The similarity between two keywords can be measured using various metrics such as edit distance.The edit distance between two strings is the minimum number of single-character operations (insertion, deletion, and substitution) to transform one string to the other. For example, the edit distance between the keywords "Kristina" and "Christina" is 2, because the former can be transformed to the latter by substituting the character "K" with "C", and inserting the character "h" after that. Let $ed(w_i, p)$ be the edit distance between a query keyword w_i and a prefix p from a record, and δ be a threshold. We say p is similar to w_i if $ed(w_i, p) \le \delta$. Our techniques can be extended to other variants of the edit distance function, such as a function that allows a swap operation between two characters, a function that uses different costs for different edit operations, and a function that considers a normalized threshold based on the string lengths.

Ranking: Each answer to a query is ranked based on its relevance to the query, which is defined based on various pieces of information such as the frequencies of query keywords in the record, and co-occurrence of some query keywords as a phrase in the record. Domain-specific features can also play an important role in ranking. For example, for a publication, its number of citations is a good indicator of its impact, and can be used as a signal in ranking. In this paper, we mainly focus on the effect of phrase matching in ranking. For example, for the query q = hheart,surgeryi, record r_1 in Table I containing the phrase "heart surgery" is more relevant than the record r_4 containing the keywords "heart" and "surgery" separately.

Basic Indexing: As the techniques described in Ji et al. [14] that combines fuzzy and instant search, we use three indexes to answer queries efficiently, a trie, an inverted index, and a forward index. In particular, we build a trie for the terms in the dictionary D. Each path from the root to a leaf node in the trie corresponds to a unique term in D. Each leaf node stores an inverted list of its term. We also build a forward index, which includes a forward list that contains encoded integers of the terms for each record. We can use this index to verify if a record contains a keyword matching a prefix condition.

Top-k Query Answering: Given a positive integer k, we compute the k most relevant answers to a query. One way to compute these results is to first find all the results matching the query conditions, then rank them based on their score. An alternative solution is to

utilize certain properties of the ranking function, and compute the *k* most relevant results using early termination techniques without computing all the results.

III. BASIC ALGORITHMS FOR TOP-k QUERIES

A. Computing All Answers

A naive solution is to first compute all the answers matching the keywords as follows. For each query keyword, we find the documents containing a similar keyword by computing the union of the inverted lists of these similar keywords. For the last query keyword, we consider the union of the inverted lists for the completions of each prefix similar to it. We intersect these union lists to find all the candidate answers. Then we compute the score of each answer using a ranking function, sort them based on the score, and return the top-*k* answers.

A main advantage of this approach is that it supports all kinds of ranking functions. An example ranking function is a

TABLE I. EXAMPLE DATA OF MEDICAL PUBLICATIONS. THE TEXT IN BOLD REPRESENTS THE INDEXED PHRASES

.

Record ID	Title	Authors
r_1	**Royal Brompton Hospital** challenges decision to close its **heart surgery unit**.	**Clare Dyer**
r_2		, ...
r_3	**EGFR Mutations** in Squamous Cell **Lung Cancer** in Never-Smokers. — **Christina S Baik** The training of congenital **heart surgeons** — **Emile A Bacha** **Plastic surgery** of the **mitral valve** in patients with **coronary heart disease** — **I A Borisov** Organizing hematoma mimicking **brain tumor** — **Ahmet Turan Ilica**	
	.	
r_4	.	, ...
r_5	.	, ...
r_6	Comment on the "update on **blood conservation** for **cardiac surgery**".	**James Hart**, ...

linear weighted sum of content-based relevancy score and proximity score that consider the similarity of each matching keyword. For example, we can use a variant of the scoring model proposed by Buttcher¨ et al. [27], which can be enhanced by considering similarity based on edit distance. This ranking function uses Okapi BM25F [28] as content-based relevancy score, and computes the proximity score between each pair of adjacent query term occurrences as inversely proportional to the square of their distance. We can adapt this ranking function by multiplying each term-related computation with a weight based on the similarity between the matching term and its corresponding query keyword.

A main disadvantage of this approach is that its performance can be low if there are many results matching the query keywords, which may take a lot of time to compute, rank, and sort. Thus it may not meet the high-performance requirement in an instant-search system.

B. Using Early Termination

To solve this problem, Li et al. [4] developed a technique that can find the most relevant answers without generating all the candidate answers. In this approach, the inverted list of a keyword is ordered based on the relevancy of the keyword to the records on the list. This order guarantees that more relevant records for a keyword are processed earlier. This technique maintains a heap for each keyword *w* to partially compute the union of the inverted lists for *w*'s similar keywords ordered by relevancy. By processing one record at a time, it aggregates the relevancy score of each keyword with respect to the record using a monotonic ranking function. For example, we can use a variant of Okapi BM25F as a monotonic ranking function, which is enhanced by considering a similarity based on edit distance. This technique works for many top-*k* algorithms. For instance, we can use the well-known top-*k* query processing

algorithm called the Threshold Algorithm [21] to determine when to terminate the computation. In particular, we can traverse the inverted lists and terminate the traversal once we are guaranteed that the top-*k* answers are among those records we have visited. The way the lists are sorted and the monotonicity property of the ranking function allow us to do this early termination, which can significantly improve the Search performance and allow us to meet the high-speed requirement in instant search. However, this approach does not consider the proximity in ranking due to the monotonicity requirement of the ranking function.

C. Using Term-Pair Index

In order to support term proximity ranking in top-*k* query processing, [6] introduces an additional term-pair index, which contains all the term pairs within a window size *w* in a document along with their proximity information. For example, for $w = 2$, the term-pair (t_1,t_2) is indexed if a document contains "$t_1\, t_2$", "$t_1\, t_x\, t_2$", or "$t_1\, t_x\, t_y\, t_2$". It is clear that the number of term pairs for the window size *w* can be $\binom{w+2}{2}$ $= \frac{(w+2)(w+1)}{2}$. Therefore, as the window size increases, the number of additional term pairs will increase quadratically. The authors also propose techniques to reduce index size while not affecting retrieval performance much. One of the proposed techniques is not creating a term-pair list for a pair if both terms are very rare. The intuition behind this strategy is that the search engine does not need too much time to process both terms even if there is no term-pair list since inverted lists of these terms are relatively short compared to those of other terms.

Given a query $q = \mathrm{h}t_1,t_2\mathrm{i}$, if the index contains the pairs (t_1,t_2) or (t_2,t_1), their inverted lists are processed, their relevancy scores are computed based on the linear combination of content-based score and the proximity score, and the temporary top-*k* answer list is maintained. Then the top*k* answer computation continues with the inverted lists of single keywords t_1 and t_2. Since the answers computed in the first step have high proximity scores, the early termination condition can be quickly satisfied in the second step.

We can adapt the approach in [6] into instant-fuzzy search, specifically to the approach described in III-B as follows. First, we insert the term pairs based on the specified window size *w* to the index as phrases. Therefore, the trie structure contains the phrase "$t_1\, t_2$" for the term pair (t_1,t_2). When computing top-*k* results for a query $q = \mathrm{h}t_1,t_2\mathrm{i}$, first we find the phrases similar to "$t_1\, t_2$" and "$t_2\, t_1$", and retrieve their inverted lists. Then we continue with the normal top-*k* computation for separate keywords t_1 and t_2. The main limitation of this approach is that it only support two-keyword queries, and does not work if the query has more than two keywords.

IV. PHRASE-BASED INDEXING AND LIFE-CYCLE OF A QUERY

To overcome the limitations of the basic approaches, we develop a technique based on phrase-based indexing.

A. Phrase-Based Indexing

Intuitively, a *phrase* is a sequence of keywords that has high probability to appear in the records and queries. We study how to utilize phrase matching to improve ranking in this top-*k* computation framework. We assume 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, we want to access the records containing phrases first. For instance, for the query $q = \mathrm{h}$heart,surgery$\mathrm{i}$, we want to access the records containing the phrase "heart surgery" before the records containing "heart" and "surgery" separately. Notice that the framework sorts the inverted list of a keyword based on relevancy of its records to the keyword. If we 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, we 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. We consider 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.

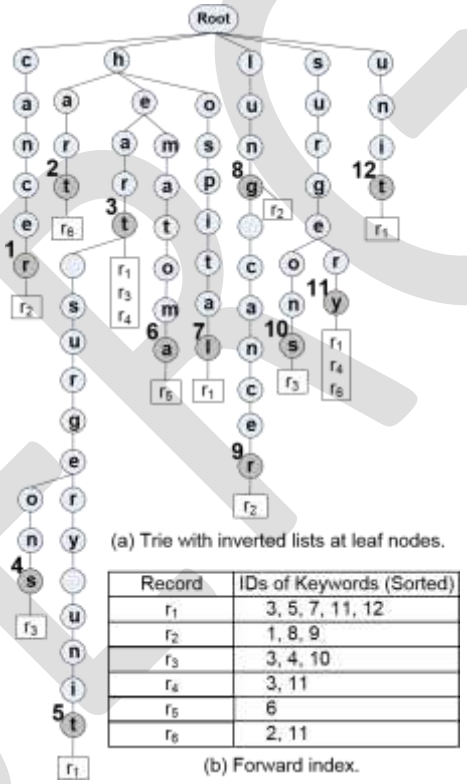
Figure 2 shows the index structures for the sample data in Table I. For instance, the phrase "heart surgery unit" is indexed in the trie in Figure 2(a), in addition to the keywords "heart", "surgery", and "unit". The leaf nodes corresponding to these terms are numbered as

5, 3, 11, and 12, 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 2(b) shows the forward index, where the keyword id 3 for the term "heart" is stored for these records.

B. Life Cycle of a Query

To deal with a large data set that cannot be indexed by a single machine, we assume 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 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 3 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 are similar to a term in the dictionary *D*. For example, for the query *q* = hheart,surgeryi, "heart" is a valid phrase for the data set in Table I, since the dictionary contains 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 trie-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.,



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

(b) Forward index.

Fig. 2. Index structures.

those trie nodes whose string corresponding to the path from the root to this node is similar to the query phrase.

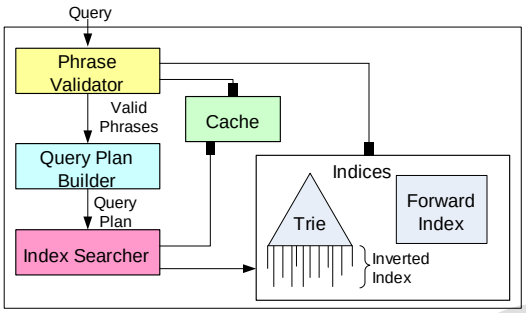


Fig. 3. Server architecture of instant-fuzzy search.

If a query keyword appears in multiple valid phrases, the query can be segmented into phrases in different ways 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 a segmentation. A result set is then created by combining the result sets of the segmentations of Q.

The rest of the paper is organized as follows. In Section V we study how to identify valid phrases in a query, and present an algorithm to do the computation incrementally. In Section VI we explain how a query is segmented based on the computed valid phrases and how these segmentations are ranked to generate a query plan. We present our experimental results in Section VII and conclude in Section VIII.

V. COMPUTING VALID PHRASES IN A QUERY

In this section we study how to efficiently compute the valid phrases in an instant-search query, i.e., those phrases that match the terms in the dictionary *D* extracted from the data set. We first give a basic approach that computes the valid phrases from scratch, then develop an efficient algorithm for doing incremental computation using the valid phrases of previous queries.

A. Basic Approach

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, $\sum_{i=1}^{l}\binom{l-1}{i-1}=2^{l-1}$, 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]. Moreover, we do not need to consider all possible segmentations since some of them are not valid. A segmentation can produce an answer to a query only if each phrase of the segmentation is a valid phrase, i.e., it is similar (possibly as a prefix) to a term in *D*, we only need to consider the valid phrases and segmentations that consist of these phrases.

The trie also allows incremental validation for phrases with the same prefix. To exploit this property, we need to validate the phrases in a specific order. Specifically, for a query $q = \langle w_1, w_2, \ldots, w_l \rangle$, for each keyword w_i, we traverse the trie to find the prefixes similar to a phrase starting with w_i. To check all the phrases starting with w_i, the keywords $w_{i+1}, w_{i+2}, \ldots, w_l$ are added incrementally during the traversal. The traversal is stopped either when all the keywords after w_i are added or when the obtained active-node set is empty. In the latter case, the phrases with more keywords will also have an empty active-node set.

B. Incremental Computation of Valid Phrases

we study how to incrementally compute the valid phrases of a query q_j using the cached valid phrases of a previous query q_i. The valid phrases of q_i are cached to be used for later queries that start with the keywords of q_i.

Figure 4 shows the active nodes of the valid phrases in the queries $q_1 = \langle$heart,surgei, $q_2 = \langle$heart,surgeryi, and $q_3 = \langle$heart,surgery,uniti. 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 n_4 of phrase "heart surge" in q_1. Hence, we can use the active nodes n_3 and n_4 to compute n_5 and n_6 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 activenode set of the cached phrase, and traversing the trie for the additional characters to determine if the phrase is still valid.

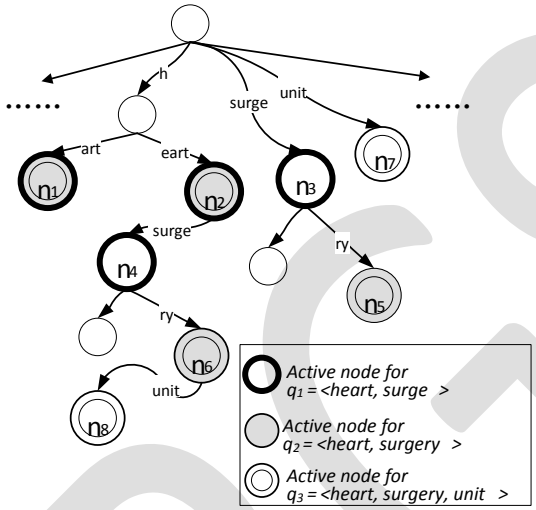


Fig. 4. Active nodes for valid phrases.

Another case where we 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 m^{th} keyword of q_i, w_m ($m \leq l$), can be extended to a phrase containing the n^{th} keyword of q_j, w_n ($l < n$), the phrase $w_m ... w_n$ can be computed by using the valid phrase $w_m ... w_l$ of q_i.

Based on these observations, we cache a vector of valid phrases V_i for a query q_i with the following properties: (1) V_i has an element for each keyword in q_i, i.e., $|V_i| = l$; (2) The n^{th} element in V_i is a set of starting points of the valid phrases that end with the keyword w_n 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.

We develop an algorithm for computing the valid phrases of a query incrementally using previously cached vector of valid phrases. The pseudo code is shown in Algorithm 1. As

q_1=< heart,surge >

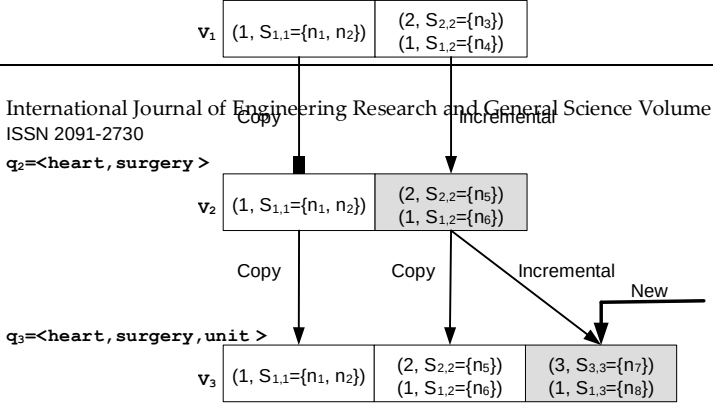


Fig. 5. Computing valid phrases incrementally using cached valid phrases of previous queries.

an example, 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, we copy 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, we traverse the trie 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 we add the pair (1, *S1*, *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, *S3*, *3*= {n_7}) for the 3rd keyword "unit" since it is also a valid phrase with an active node n_7 (lines 22–30).

VI. GENERATING EFFICIENT QUERY PLANS

As explained in Section IV, the Phrase Validator computes the valid phrases in a query using the techniques described in Section V, and passes the valid-phrase vector to the Query Plan Builder. In this section, we study how the Query Plan Builder generates and ranks valid segmentations.

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.. Table II shows all possible segmentations that can be generated from the valid phrases vector V_3 in Figure 5.

Algorithm 1: ComputeValidPhrases(q,C)

Input : query q = hw_1,w_2,...,w_li where w_i is a keyword; a cache module C; **Output**: a valid-phrase vector V ;

1 (q_c,V_c) ← FindLongestCachedPrefix(q, C)
2 m ← number of keywords in q_c

```
3  if m > 0 then // Cache hit
4      for i ← 1 to m − 1 do // Copy the
       valid-phrase vector
5          V[i] ← V_c[i]
6      if w_m == q_c[m] then // The last
       keyword of q_c is a complete
       keyword in q
7          V[m] ← V_c[m]
8      else // Incremental computation for
       the last keyword retrieved from
       cache
9          V[m] ← ∅
10         foreach (start, S) in V_c[m] do
11             newS ← compute active nodes for w_m
12                    incrementally from S
13             if newS == ∅ then
14                 V[m] ← V[m] ∪ (start, newS)
15     foreach (start, S) in V[m] do
       // Incremental computation for
       the phrases partially cached
16         for j ← m + 1 to l do
17             newS ← compute active nodes
18                    from S by appending w_j
19             if newS == ∅ then break
20             V[j] ← V[j] ∪ (start, newS)
21             S ← newS
22 for i ← m + 1 to l do // Computation of
   non-cached phrases
23     S ← compute active nodes for w_i
24     V[i] ← V[i] ∪ (i, S)
25     for j ← i + 1 to l do
26         newS ← compute active nodes
27                from S by appending w_j
28         if newS == ∅ then break
29         V[j] ← V[j] ∪ (i, newS)
30         S ← newS
31 cache (q, V) in C
32 return V
```

We develop a divide-and-conquer algorithm 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 hw_1,...,$w_{start-1}$i, we can append the phrase [*start*, *end*] to it to obtain a segmentation for the query hw_1,...,w_{end}i. 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

TABLE II. THREE SEGMENTATIONS FOR QUERY q = hheart,surgery,uniti.

1. "heart surgery unit"
2. "heart surgery | unit"
3. "heart | surgery | unit"

phrase $[i,j]$ to each of these segmentations to form new segmentations. This analysis helps us reduce the problem of generating segmentations for the query $hw_1,...,w_li$ to solving the sub problems of generating segmentations for each query $hw_1,...,w_{i-1}i$, 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. We can convert this recursive algorithm into a top down dynamic programming algorithm by memorizing all the computed results for each end position.

Algorithm 2: Generate Segmentations (q,V,end)

Input : a query with a list of keywords $q = \{w_1,w_2,...,w_l\}$; its valid-phrase vector V ; a keyword position end ($end \leq l$) ;

Output: a vector P_{end} of all valid segmentations of $w1,w2,...,wend$

```
1  Pend ← ∅
2  foreach(start,S start,end ) in V[end] do
3      if start == 1 then // Base Case
4          Pend ← Pend ∪ hwstart ...w end i
5      else
6          foreach seg in GenerateSegmentations(q, V, start - 1) do
7              seg ← seg | hwstart ...w end i
8              Pend ← Pend ∪ seg
9  return Pend
```

B. Ranking Segmentations

Each generated segmentation 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. We can run these segmentations one by one until we find enough answers (i.e., k results). Thus, the ranking needs to guarantee that the answers to a high-rank segmentation are more relevant than the answers to a low-rank segmentation. There are different methods to rank a 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 distance 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 = hhart,surgeryi, 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.. Using this method, we would rank the first segmentation higher due to its small total edit distance. If two segmentations have the same total minimum edit distance, then we can rank the segmentation with fewer segments higher. When there are fewer phrases in a segmentation, the number of keywords in a phrase increases.

VII. CONCLUSIONS

In this paper we studied how to improve ranking of an instant-fuzzy search system by considering proximity information when we need to compute top-k answers. We studied how to adapt existing solutions to solve this problem, including computing all answers, doing early termination, and indexing term pairs. We proposed a technique to index important phrases to avoid the large space overhead of indexing all word grams. We presented an incremental-computation algorithm for finding the indexed phrases in a query efficiently, and studied how to compute and rank the segmentations consisting of the indexed phrases.

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: http://doi.acm.org/10.1145/1476589.1476628

[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.

[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. Buttcher,¨ 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-ngram-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/keyword-stats.html?date=201306-01, Jun. 2013