Efficient Instant-Fuzzy Search with Proximity Ranking

Inci Cetindil*, Jamshid Esmaelnezhad†, Taewoo Kim‡ and Chen Li§
Department of Computer Science, University of California, Irvine
{*icetindi, †jesmaeln, ‡taewok2, §chenli}@ics.uci.edu

Abstract—**Instant search is an emerging information-retrieval paradigm 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 paradigm is 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. We adapt existing solutions on proximity ranking to instant-fuzzy search. A naïve 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. We conducted a thorough experimental study on real data sets to show the tradeoffs between time, space, and quality of these solutions.

I. Introduction

Instant Search: As an emerging information-access paradigm, instant search returns the answers immediately based on a partial query a user has typed in. For example, the Internet Movie Database, IMDB, has a search interface that offers instant results to users while they are typing queries[1]. When a user types in "`sere`", the system returns answers such as "`Serena`", "`Serenity`", "`Serendipity`", and "`Serena Williams`". 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. Our recent study showed that this new information-retrieval method helps users find their answers quickly with less effort [1].

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. Figure 1 shows an instant-fuzzy search interface on a people directory using this technology. The system finds an answer to the keyword query "`prefessor wenkatsu`" even though the user mistyped a prefix of the name "`venkatasubramanian`". 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.

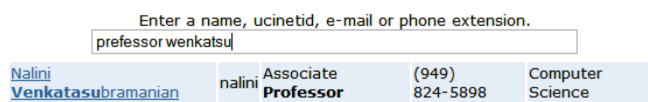


Fig. 1. Instant-fuzzy search on the UC Irvine people directory (http://psearch.ics.uci.edu).

Finding Relevant Answers within Time Limit: A main computational challenge in this search paradigm is its high-speed requirement. 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]. For example, if the search query is "`Michael Jackson`", the user is most likely looking for the records containing information about the singer Michael Jackson, while documents containing "`Peter Jackson`" and "`Michael J Fox`" would be less relevant.

[1] http://www.imdb.com, as of July 2013.

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. For example, the keyword prefix "`cleam`' can have many similar completions such as "`clean`", "`clear`", and "`cream`". 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.

To summarize, we make the following contributions in this paper. (1) We adapt existing solutions for proximity ranking into instant-fuzzy search. (2) We propose a space-efficient indexing approach for utilizing proximity information to rank answers in instant-fuzzy search. (3) We present an incremental-computation method for identifying the indexed phrases in the query. (4) We propose methods to compute segmentations efficiently and decide in which order to execute these segmentations. (5) We conduct a thorough experimental study on real data sets to compare the space, time, and relevancy tradeoffs of the proposed approaches.

A. Related Work

Auto-Completion: In auto-completion, the 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 trie-based 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.

Our work differs from the earlier studies since we focus on how to *instantly* compute relevant answers based on proximity information as the user is typing keywords character by character. The high-efficiency demand requires novel incremental-computation algorithm, which is the main focus of this work. Also notice that our technique does not assume availability of query log, which is needed by many query-suggestion systems.

II. PRELIMINARIES

Data: Let $R = \{r_1, r_2, \ldots, r_n\}$ 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. We will use this sample data set throughout the paper to explain our techniques.

Query: A query q is a string that contains a list of keywords $\langle w_1, w_2, \ldots, w_l\rangle$, 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 $\langle w_1, w_2, \ldots, w_{l-1}\rangle$ 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: $q_1 = \langle \texttt{b}\rangle$, $q_2 = \langle \texttt{br}\rangle$, ..., $q_{10} = \langle \texttt{brain}, \texttt{tumo}\rangle$, $q_{11} = \langle \texttt{brain}, \texttt{tumor}\rangle$.

Answers: A record r from the data set R is an answer to the query q if it satisfies the following conditions: (1) For $1 \leq i \leq l-1$, 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 $q = \langle \texttt{heart}, \texttt{surge}\rangle$, 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, Jaccard similarity, and cosine similarity. In this work we focus on the commonly used function, 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 $ed(w_i, p) \leq \delta$. 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 $q = \langle \texttt{heart}, \texttt{surgery}\rangle$, 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 naïve 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 linear weighted sum of content-based relevancy score and

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	**EGFR Mutations** in Squamous Cell **Lung Cancer** in Never-Smokers.	**Christina S Baik**, ...
r_3	The training of congenital **heart surgeons**.	**Emile A Bacha**
r_4	**Plastic surgery** of the **mitral valve** in patients with **coronary heart disease**.	**I A Borisov**, ...
r_5	Organizing hematoma mimicking **brain tumor**.	**Ahmet Turan Ilica**, ...
r_6	Comment on the "update on **blood conservation** for **cardiac surgery**".	**James Hart**, ...

proximity score that consider the similarity of each matching keyword. For example, we can use a variant of the scoring model proposed by Büttcher 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 $q = \langle t_1, t_2 \rangle$, 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 $q = \langle t_1, t_2 \rangle$, 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. In this section we give the detail of this approach.

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 =

$\langle \texttt{heart}, \texttt{surgery} \rangle$, 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.

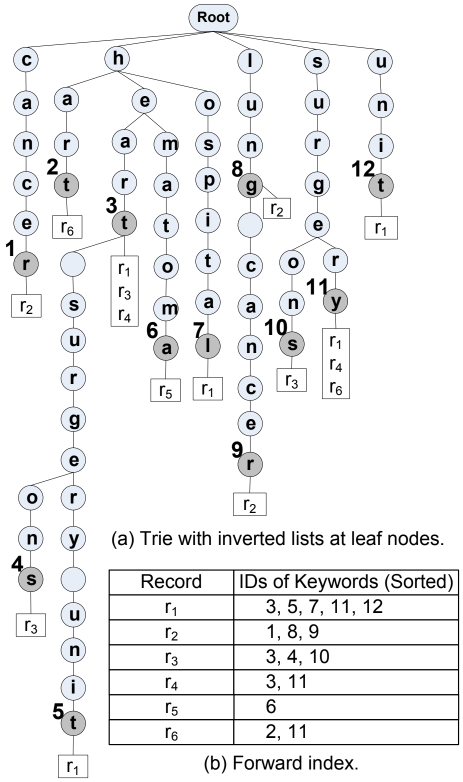
Based on this analysis, we need to index phrases to be able to retrieve the records containing these phrases efficiently. However, the number of phrases up to a certain length in the data set can be much larger than the number of unique words [29]. Therefore, indexing all the possible phrases can require 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 $q = \langle \texttt{heart}, \texttt{surgery} \rangle$, "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., those trie nodes whose string corresponding to the path from the root to this node is similar to the query phrase.



(a) Trie with inverted lists at leaf nodes.

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.

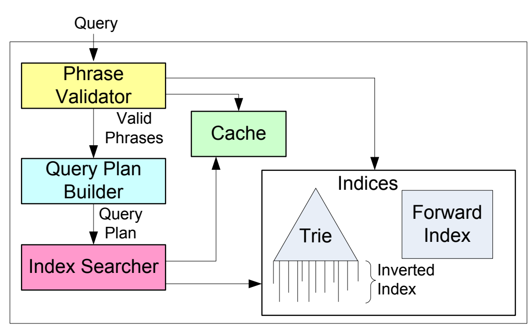


Fig. 3. Server architecture of instant-fuzzy search.

If a query keyword appears in multiple valid phrases, the query can be segmented into phrases in different ways. Let "|" denote a place between two adjacent valid phrases. For instance, "heart | surgery" and "heart surgery" are two different segmentations for q. We will refer the query segmentations that consist of only valid phrases as *valid segmentations*. After identifying the valid phrases, the *Query Plan Builder* generates a *Query Plan* $\mathcal{Q}$, which contains all the possible valid segmentations in a specific order. The ranking of $\mathcal{Q}$ determines the order in which the segmentations will be executed. After $\mathcal{Q}$ 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 $\mathcal{Q}$.

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, we have 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 trie 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.

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. For the query $q = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$, there is no term in D similar to the phrase "`surgery unit`", and the result set for each segmentation containing this phrase is empty. Hence, the segmentation "`heart | surgery unit`" is not valid. Based on this observation, we only need to consider the valid phrases and segmentations that consist of these phrases.

We now show how to compute valid phrases. For a query $q = \langle w_1, w_2, \ldots, w_l \rangle$, there are $C(m) = l - m + 1$ possible phrases having m keywords: $\langle w_1 \ldots w_m \rangle$, $\langle w_2 \ldots w_{m+1} \rangle$, $\ldots$, $\langle w_{l-m+1}, \ldots w_l \rangle$. Therefore, the total number of possible phrases is $\sum_{i=1}^{l} C(i) = \frac{l(l+1)}{2}$. However, not all of these phrases are valid. Using the active-node computation described in [14], we can find a valid phrase by verifying whether the trie has a prefix similar to this phrase. Using a trie for this validation has several advantages. Intuitively, if there is no prefix in the trie similar to a phrase p_1, then a phrase p_2 with p_1 as a prefix will not have any similar prefixes in the trie. For example, consider a query $q = \langle \texttt{heart}, \texttt{failure}, \texttt{patients} \rangle$. If there is no prefix on the trie similar to the phrase "`heart failure`", then there is no prefix similar to the phrase "`heart failure patients`". This property helps us prune some phrases without computing their active nodes.

The trie also allows incremental validation for phrases with the same prefix. For example, the active nodes of the phrase "`heart failure`" can be computed by starting from the active nodes of the phrase "`heart`" and adding a space (" ") and each character in "`failure`". 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. For example, for $q = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$, first we find all the trie prefixes similar to "`heart`". Then, starting from the active-node set of "`heart`", we compute the active-node set for "`heart surgery`" incrementally. The active-node set of "`heart surgery unit`" is computed similarly by using the active-node set of "`heart surgery`". The valid phrases that start with a keyword similar to "`surgery`" and "`unit`" are computed similarly.

B. Incremental Computation of Valid Phrases

The basic approach described above 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. In this section, 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 $q_1 = \langle \texttt{heart}, \texttt{surge} \rangle$, $q_2 = \langle \texttt{heart}, \texttt{surgery} \rangle$, and $q_3 = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$. 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 active-node set of the cached phrase, and traversing the trie for the additional characters to determine if the phrase is still valid.

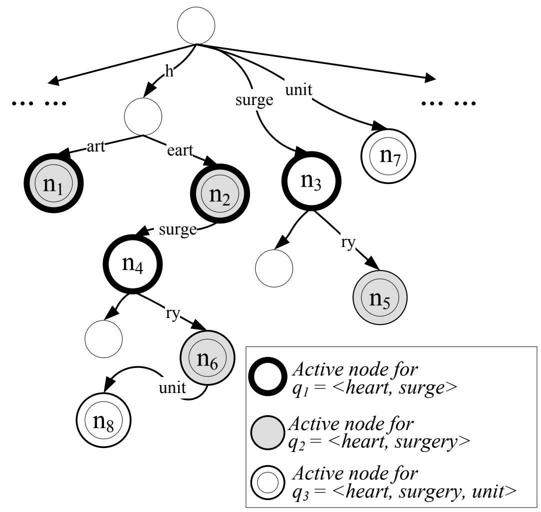


Fig. 4. Active nodes for valid phrases.

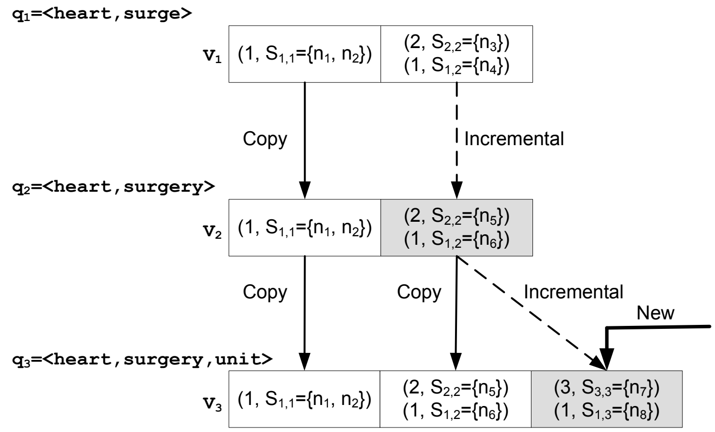


Fig. 5. Computing valid phrases incrementally using cached valid phrases of previous queries.

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 $w_m \dots w_n$ can be computed by using the valid phrase $w_m \dots w_l$ 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. 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 trie 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.

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 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 $(1, S_{1,3} = \{n_8\})$ to the third element of V_3, indicating that there is a valid phrase starting from the 1^{st} keyword and ending at the 3^{rd} keyword. We also add an element $(3, S_{3,3} = \{n_7\})$ for the 3^{rd} 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. For example, for the query $q = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$, "`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 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 = ⟨w_1, w_2, ..., w_l⟩ where w_i is a
         keyword; a cache module C;
Output: a valid-phrase vector V;
(q_c, V_c) ← FindLongestCachedPrefix(q, C)
m ← number of keywords in q_c
if m > 0 then // Cache hit
    for i ← 1 to m − 1 do // Copy the valid-phrase vector
        V[i] ← V_c[i]
    if w_m == q_c[m] then // The last keyword of q_c is a complete keyword in q
        V[m] ← V_c[m]
    else // Incremental computation for the last keyword retrieved from cache
        V[m] ← ∅
        foreach (start, S) in V_c[m] do
            newS ← compute active nodes for w_m
                        incrementally from S
            if newS == ∅ then
                V[m] ← V[m] ∪ (start, newS)
    foreach (start, S) in V[m] do
    // Incremental computation for the phrases partially cached
        for j ← m + 1 to l do
            newS ← compute active nodes
                        from S by appending w_j
            if newS == ∅ then  break
            V[j] ← V[j] ∪ (start, newS)
            S ← newS
for i ← m + 1 to l do // Computation of non-cached phrases
    S ← compute active nodes for w_i
    V[i] ← V[i] ∪ (i, S)
    for j ← i + 1 to l do
        newS ← compute active nodes
                    from S by appending w_j
        if newS == ∅ then break
        V[j] ← V[j] ∪ (i, newS)
        S ← newS
cache (q, V) in C
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 $\langle w_1, \ldots, w_{start-1} \rangle$, we can append the phrase $[start, end]$ to it to obtain a segmentation for the query $\langle w_1, \ldots, w_{end} \rangle$. 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. This analysis helps us reduce the problem of generating segmentations for the query $\langle w_1, \ldots, w_l \rangle$ to solving the subproblems of generating segmentations for each query $\langle w_1, \ldots, w_{i-1} \rangle$, 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 memoizing all the computed results for each end position.

TABLE II. THREE SEGMENTATIONS FOR QUERY $q = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$.

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

Algorithm 2: GenerateSegmentations(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 ≤ l);
Output: a vector P_end of all valid segmentations of
         w_1, w_2, ..., w_end
1 P_end ← ∅
2 foreach (start, S_start,end) in V[end] do
3     if start == 1 then // Base Case
4         P_end ← P_end ∪ ⟨w_start ... w_end⟩
5     else
6         foreach seg in
          GenerateSegmentations(q, V, start - 1)
          do
7             seg ← seg | ⟨w_start ... w_end⟩
8             P_end ← P_end ∪ seg
9 return P_end
```

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 $q = \langle \texttt{hart}, \texttt{surgery} \rangle$, 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, 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. 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 II are ranked based on this feature. If two segmentations have both the same total minimum distance and the number of phrases, then we assume 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 $q = \langle \texttt{hart}, \texttt{surgery} \rangle$ 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, we evaluate the performance of the proposed techniques on real data sets. We implemented the following methods: (1) FindAll ("FA"): We found all the answers and returned the top-k answers after sorting them based on their relevancy score. (2) QuerySegmentation ("QS"): In this approach, we computed a query plan based on valid segmentations, and ran the segmentations one by one until top-k answers were computed. (3) TermPair ("TP"): We implemented the approach described in Section III-C for only one-term and two-term queries.

TABLE III. Data sets.

Data Set	IMDB	Enron	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

We used three data sets in the experiments, namely IMDB, Enron, and Medline. Table III shows the details of the data sets. We obtained the IMDB data set from www.imdb.com/interfaces. We used 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, we extracted the queries from an AOL query log[3], 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, we used the queries provided by [31]. The Medline data set consisted of more than 20 million medical publications, and we used its subsets of different sizes to do experiments. For this data set, we 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, we used $1/3$ as the normalized edit-distance threshold. That is, we 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 Linux server running a Ubuntu 10.04 64-bit operating system, with two 2.93GHz Intel Xeon 6-core processors, 96 GB of memory, and 2 TB of hard disk. To satisfy the high-efficiency requirement of instant search, all index structures were stored in memory.

A. Effect of Indexed Phrases on Index Size

We first compared the index size for different approaches. For FA we only indexed single keywords. For QS choosing what phrases to index is a data-dependent problem, and is orthogonal to the approach we propose. We indexed the movie names, actor names, and character names as phrases for the IMDB data set. For the Enron data set, we extracted popular bigrams and trigrams from the subject and body attributes. For the Medline data set, we used 1.7 million records and indexed the author names, affiliations, and mesh headings as phrases. Additionally, we extracted popular bigrams and trigrams that did not contain stop words from titles and abstracts. We added the n-grams that occurred more than $t = 100$ times in the data set to the index. In TP we chose window size $w = 3$ for building term-pair index.

Figure 6 shows the index sizes for different approaches on these data sets. The results showed that, not surprisingly, the index size increased when the phrases were added to the index. For instance, the total index size of FA that contained only single keywords for Medline was 1.9GB while the total index size of QS was 2.5GB, and the total index size of TP was 5.9GB. This experiment also showed that indexing only selected entities instead of all the term pairs reduced the index size dramatically.

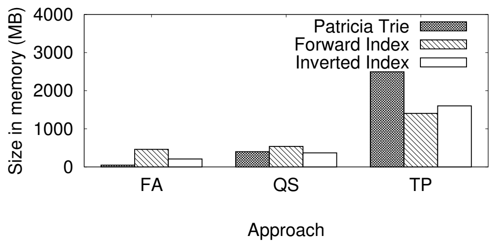
In our experiments the trie size increased drastically when the phrases were added to the index. This is because each phrase was an extension from an existing keyword in the trie and these extensions made the trie very sparse. For this reason, we compressed the trie by implementing a Patricia trie [33], and this optimization reduced the size to approximately one third of its uncompressed version. The similar trends were observed in all the data sets. For simplicity, we reported only the size of the compressed trie.

B. Efficiency of Computing Valid Phrases

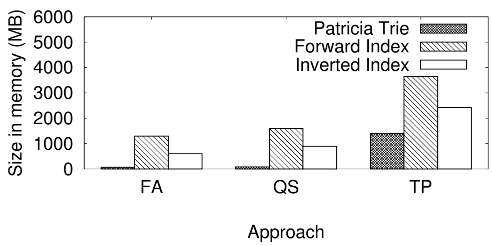
We implemented and compared the algorithms for valid-phrase computation for QS as explained in Sections V-A, V-B: Basic and Incremental. Figure 7 shows the results for the

[2]We used one shard with 1.7 million records for all the experiments except scalability.

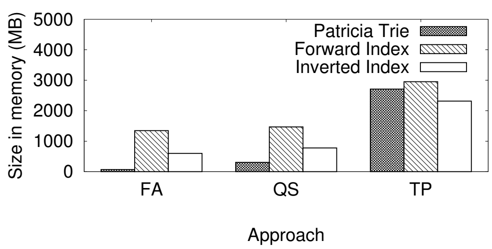
[3]http://www.gregsadetsky.com/aol-data/



(a) IMDB



(b) Enron



(c) Medline

Fig. 6. Index sizes for different approaches.

Medline data set. 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. For example, the computation times of the Basic algorithm for 4-keyword, 5-keyword, and 6-keyword queries were 36, 57, and 64 ms respectively. This observation showed that the Basic algorithm cannot satisfy the high-speed requirement of instant search when the number of keywords in a query increased. At the same time, 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. We observed the similar trends for the other data sets.

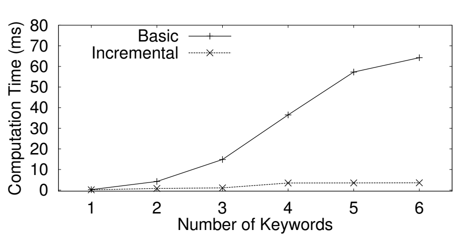
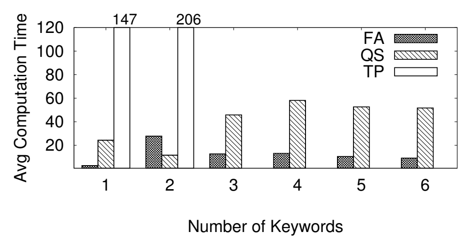


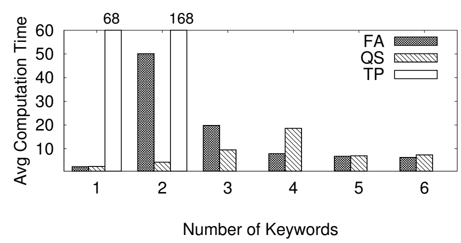
Fig. 7. Performance of computing valid phrases in a query (Medline).

C. Query Time

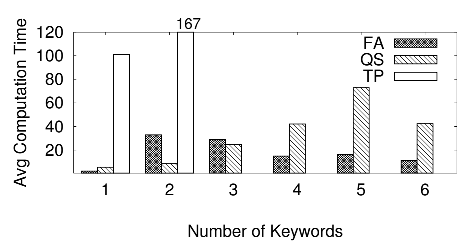
We compared the average computation time of FA, QS, and TP as the number of keywords in query varied. Figure 8 shows the results for the three data sets. Since TP supports at most 2-keyword queries, we reported the query computation time for 1-keyword and 2-keyword queries for TP. The results for TP showed that it did not meet the high-efficiency requirement of instant search. The main reason of its slowness was indexing all the term pairs within a specified window size; because it caused to have too many completions for each query keyword, especially for short prefixes. From this experiment we concluded that TP is not very suitable for instant search. FA outperformed for the queries having more than 3 keywords, while QS outperformed for 2-keyword and 3-keyword queries. For instance, for the Enron data set the average query time for 2-keyword queries was 50 milliseconds in FA and 4 milliseconds in QS, while for 4-keyword queries it was 7 milliseconds in FA and 18 milliseconds in QS. We discussed the reasons of this behavior in detail in the scalability evaluation of these approaches.



(a) IMDB



(b) Enron



(c) Medline

Fig. 8. Performance Evaluation.

The experiments reported in Figure 8 used cache for incremental computation. We also ran the experiments by disabling cache, but we omit the results due to space limitation. We observed similar trends as the experiments reported in Figure 8. However, without cache the response time for long queries exceeded the 100 ms threshold due to lack of incremental computation. We concluded that caching is very crucial in the instant search context to do incremental computation since consequent queries in incremental typing usually differ from each other by one character.

D. Cache Hit Rate

We also compared the cache hit rate of QS and FA approaches for the experiments in Figure 8. The TP approach did not have any incremental computation since it only works for queries with up to 2 keywords. For this experiment, we considered all the queries that retrieved the answers to an earlier query from cache during incremental computation as a cache hit. Figure 9 shows the cache hit rates of QS and FA for the three data sets.

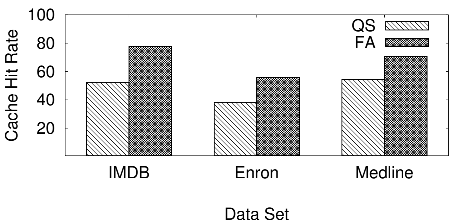


Fig. 9. Cache Hit Rate.

We observed that FA had a better cache hit rate for all the three data sets. We explain the reason with the example query $q_i = \langle \texttt{heart}, \texttt{surgery}, \texttt{unit} \rangle$. FA uses the answers to the query $q_j = \langle \texttt{heart}, \texttt{surgery} \rangle$, and intersects them with the records that contain the keyword "`unit`". However, in QS if we have a valid segmentation "`heart surgery unit`" and this segmentation returns k answers, it does not use any earlier query from cache. Thus, FA always uses the answers to an earlier query if they are cached. QS uses the cache only if the segmentation of an earlier query is the prefix of the segmentation of the current query.

E. Quality of Answers

To evaluate the quality of the answers returned by QS and TP, we conducted a user study with a special user interface that showed the top-10 answers from computer science publications (DBLP[4]) computed and ranked by the FA approach. We assumed FA returned the best answers, since it used a ranking function that fully utilized the proximity information. We asked 12 people to search for publications, authors, venues, or combination of any of these fields, and give feedback by selecting the relevant answers among the returned top-10 answers. With this study we collected 69 queries and 353 relevant answers for these queries. Most of the queries had up to 3 keywords, while there were also a few queries with 4 keywords. We ran the same queries on QS and TP, and measured their precision by computing what percentage of the expected results were returned by these approaches. Based on this analysis, the precision of QS was 81%, and the precision of TP was 69%. We observed that QS did not perform well for two types of queries: (1) the queries that contained rare n-grams from the data since we did not index them, and (2) the queries that was missing the beginning keywords of an entity since we relied on prefix matching when computing a valid phrase. TP mostly failed for the queries with more than 2-keywords. Since a significant portion of the query set was 3-keyword queries, the precision was degraded.

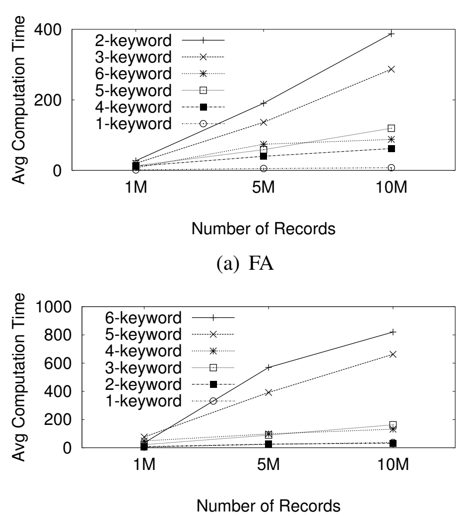
[4]http://dblp.uni-trier.de/xml/

F. Scalability

We evaluated the scalability of FA and QS using the Medline data set. We did not include TP in this experiment because of its prohibitive index size for big data sets. Figure 10 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. We observed that the average computation time increased linearly in both approaches as we increased the number of records.

Another interesting observation was that for 2-keyword and 3-keyword queries QS gave the best results while FA outperformed for the rest. For instance, using 10 million records, the average computation time for 2-keyword queries was 32 milliseconds in QS, and 387 milliseconds in FA. The reason FA outperformed other approaches for queries more than 3 keywords was due to the increase in the selectivity of the query. Adding more keywords to a query resulted in a smaller answer set, which is in favor of FA. On the other hand, 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 improved the performance remarkably. For 1-term queries, all the three approaches returned the top-k elements from the union of the inverted lists of similar complete keywords to the given prefix. FA outperformed in this case since it only considered single keyword completions while the other approaches had also phrase completions.

This experiment also showed that QS approach is indeed very useful since users predominantly use 2 and 3 keyword queries [34].



(a) FA

(b) QS

Fig. 10. Scalability (Medline).

We summarize the experimental results as follows:

- TP requires too much space and is too slow to be practical for instant search.
- FA is better for long queries (with at least four keywords).

- QS works the best for 2-keyword and 3-keyword queries, which are common in search applications.
- For single-keyword queries FA is slightly better due to its compact trie index.
- Benefits of QS increase as the size of the data increases.

VIII. 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. We compared our techniques to the instant-fuzzy adaptations of basic approaches. We conducted a very thorough analysis by considering space, time, and relevancy tradeoffs of these approaches. In particular, our experiments on real data showed the efficiency of the proposed technique for 2-keyword and 3-keyword queries that are common in search applications. We concluded that computing all the answers for the other queries would give the best performance and satisfy the high-efficiency requirement of instant search.

Acknowledgment

The authors Inci Cetindil, Jamshid Esmaelnezhad, and Chen Li have financial interest in SRCH2, a company commercializing techniques related to those described in this paper.

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. Büttcher, 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-belong-to-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=2013-06-01, Jun. 2013.