The VLDB Journal (2011) 20:617–640
DOI 10.1007/s00778-011-0218-x

REGULAR PAPER

Efficient fuzzy full-text type-ahead search

Guoliang Li · Shengyue Ji · Chen Li · Jianhua Feng

Received: 29 January 2010 / Revised: 22 January 2011 / Accepted: 2 February 2011 / Published online: 24 February 2011
© Springer-Verlag 2011

Abstract Traditional information systems return answers after a user submits a complete query. Users often feel "left in the dark" when they have limited knowledge about the underlying data and have to use a try-and-see approach for finding information. A recent trend of supporting autocomplete in these systems is a first step toward solving this problem. In this paper, we study a new information-access paradigm, called "type-ahead search" in which the system searches the underlying data "on the fly" as the user types in query keywords. It extends autocomplete interfaces by allowing keywords to appear at different places in the underlying data. This framework allows users to explore data as they type, even in the presence of minor errors. We study research challenges in this framework for large amounts of data. Since each keystroke of the user could invoke a query on the backend, we need efficient algorithms to process each query within milliseconds. We develop various incremental-search algorithms for both single-keyword queries and multi-keyword queries, using previously computed and cached results in order to achieve a high interactive speed. We develop novel techniques to support fuzzy search by allowing mismatches between query keywords and answers. We have deployed several real prototypes using these techniques. One of them has been deployed to support type-ahead search on the UC Irvine people directory, which has been used regularly and well received by users due to its friendly interface and high efficiency.

Keywords Auto complete · Full-text search · Type-ahead search · Fuzzy search

G. Li (✉) · J. Feng
Department of Computer Science, Tsinghua University,
Room 10-204, East Main Building, 100084 Beijing, China
e-mail: liguoliang@tsinghua.edu.cn

J. Feng
e-mail: fengjh@tsinghua.edu.cn

S. Ji · C. Li
Department of Computer Science, University of California,
Irvine, CA, USA
e-mail: shengyuj@ics.uci.edu

C. Li
e-mail: chenli@ics.uci.edu

1 Introduction

In a traditional information system, a user composes a query, submits it to the system, and the system retrieves relevant answers. This information-access paradigm requires the user to have certain knowledge about the structure and content of the underlying data repository. In the case where the user has limited knowledge about the data, often the user feels "left in the dark" when issuing queries and has to use a try-and-see approach for finding information, as illustrated by the following example.

At a conference venue, an attendee named John met a person from a university. After the conference, John wanted to get more information about this person, such as his research projects.John knows about all the person is that he is a professor from that university, and John only remembers the name roughly. In order to search for this person, John goes to the directory page of the university. Figure 1 shows such an interface. John needs to fill in the form by providing information for some attributes, such as name, phone, department, and title. Given his limited information about the person, especially since he does not know the exact spelling of the person's name, John needs to try a few possible keywords, go through the returned results, modify the keywords, and re-issue a new query. He needs to repeat this step multiple

Name:
Phone:
Department:
Title:
Search Clear Form

Fig. 1 A typical directory-search form

times to find the person, if lucky enough. This search interface is neither efficient nor user friendly.

Many systems are introducing various features to solve this problem. One of the commonly used methods is *autocomplete*, which suggests a word or phrase that the user may type next based on the query the user has entered. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in keywords. More and more Web sites have this feature, due to recent advances in high-speed networks and browser-based programming languages and tools such as JavaScript and AJAX.

In this paper, we study a new computing paradigm, called "type-ahead search." Our method searches for the best answers "on the fly" as users type in query keywords. When searching for relevant records, we also extend our method to find those records that include words *similar* to the keywords in the query, even if they do not match exactly.

We have developed several prototypes using this paradigm. The first one supports type-ahead search on the UC Irvine people directory. A screenshot is shown in Fig. 2. In the figure, a user has typed in a query string "`professor smyt`." Even though the user has not typed in the second keyword completely, the system can already find person records that might be of interest to the user. Notice that the two keywords in the query string (including a keyword "`smyt`") can appear in different attributes of the records. In particular, in the first record, the keyword "`professor`" appears in the "title" attribute, and the keyword "`smyt`" appears in the "name" attribute. The matched prefixes are highlighted.

The system can also find records with keywords that are similar to the query keywords, such as a person name "`smith`." The feature of supporting fuzzy search is important especially when the user has limited knowledge about the underlying data or the entities she is looking for. In addition, as the user types in more letters, the system interactively searches on the data and updates the list of relevant records. The system also utilizes a priori knowledge such as synonyms. For instance, given the fact that "`william`" and "`bill`" are synonyms, the system can find a person called "`William Kropp`" when the user has typed in "`bill crop`." This search prototype has been used regularly by many people at UCI and received positive feedback due to the friendly user interface and high efficiency.

In this paper, we study research challenges that arise naturally in this computing paradigm. The main challenge is the requirement of a high efficiency. To make search interactive, for each keystroke on the client browser, from the time the user presses the key to the time the results computed from the server are displayed on the browser, the delay should be as small as possible. An interactive speed requires this delay be within milliseconds. Notice that this time includes the network transfer delay, execution time on the server, and the time for the browser to execute its JavaScript (which tends to be slow). Providing a high efficiency on a large amount of data is especially challenging because of two reasons. First, we allow the query keywords to appear in different attributes with an arbitrary order, and the "on-the-fly join" nature of the problem can be computationally expensive. Second, we want to support fuzzy search by finding records with keywords that match query keywords approximately.

We develop novel solutions to these problems. We present several incremental-search algorithms for answering a query by using cached results of earlier queries. In this way, the computation of the answers to a query can essentially spread across multiple keystrokes of the user, thus we can achieve a high speed. Specifically, we make the following contributions. (1) We present an incremental algorithm for computing keyword prefixes similar to a prefix keyword (Sect. 4). (2) For queries with multiple keywords, we study various techniques for computing the intersection of the inverted lists of query keywords and develop a novel algorithm for computing the results efficiently (Sect. 5.1). (3) We develop an on-demand caching technique for incrementally computing first-k (any-k) answers.[1] Its idea is to cache only part of the results of a query. For subsequent queries, the unfinished computation will be resumed if the previously cached results are not sufficient. In this way, we can efficiently compute and cache a small amount of results (Sect. 5.2). (4) We study various features in this paradigm, such as how to highlight keywords in the results, how to utilize domain-specific information (e.g., synonyms) to improve search, and how to support updates (Sect. 6). (5) In addition to deploying several real prototypes, we conducted a thorough experimental evaluation of the developed techniques on real data sets and show the practicality of this new computing paradigm. All experiments were done using a single desktop machine, which can achieve a response time of milliseconds on millions of records.

[1] In our deployed system of UCI people search, we return the top-k answers for each query. In this paper, we focus on any-k queries and will study how to answer top-k queries efficiently in the future.

Fig. 2 Type-ahead search on the UC Irvine people directory (http://psearch.ics.uci.edu)

UCI People Search

Search by Name, Nick Name, UCInetID, Title, Department, Major, Office Address, Phone Number, EMail Address and Zot Code

professor smyt

Patrick J. **SMYTH**	pjsmyth	Padhraic SMYTH	**Professor**	Computer Science-Computing	(949) 824-2558	4062...
Janellen **Smith**	jesmith7		**Professor**	Dermatology	(949) 824-5515	C252...
Clyde W **SMITH**	smithcw		Clinical **Professor**	Radiological Sciences	(714) 456-5033	Bldg...
John H. **SMITH**	jhsmith		**Professor** and Chair	German	(949) 824-6406, 6107	400G...

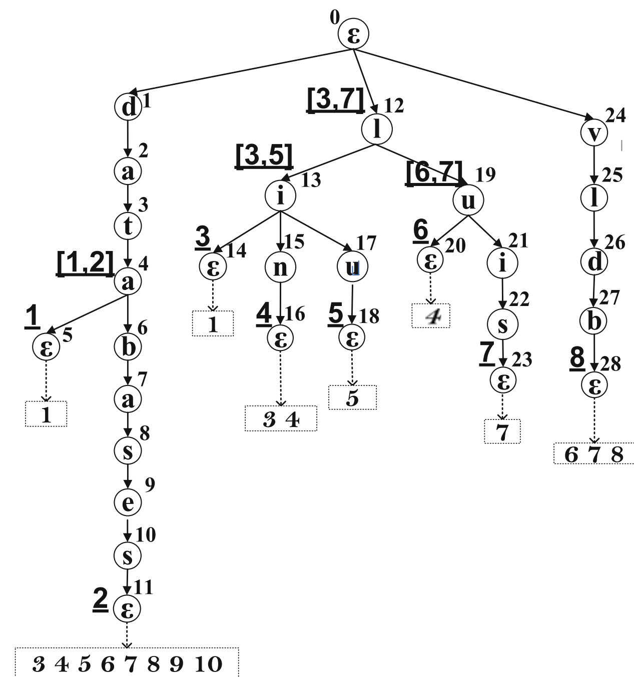

Fig. 3 Trie with inverted lists at leaf nodes (The *numbers in rectangles* are record IDs, the *underlined numbers* are word IDs, and *other numbers* are node IDs.)

1.1 Related work

Bast et al. proposed techniques to support "CompleteSearch," in which a user types in keywords letter by letter, and the system finds records that include these keywords (possibly at different places) [4–7,10]. Different from CompleteSearch, we propose an alternative index structure to achieve the goal. Chaudhuri et al. [15] also studied how to extend autocompletion to tolerate errors. The algorithms for computing similar prefixes for a query presented in the two papers are similar, while our algorithm was published earlier in [26]. In addition, our work also studied type-ahead search of multiple keywords, which was not studied in their paper.

2 Preliminaries

2.1 Problem formulation

We formalize the problem of *type-ahead search* on a set of records, and our method can be adapted to textual documents [57], XML documents [41], and relational databases [42] as well.

Exact type-ahead search: Consider a set of records $R = \{r_1, r_2, \ldots, r_n\}$. Each record is a sequence of words (tokens). A query consists of a set of keywords $Q = \{p_1, p_2, \ldots, p_\ell\}$. The query answer is a set of records r in R such that for each query keyword p_i, record r contains a word with p_i as a prefix. For example, consider the data in Table 1, which has ten records. For a query {"vldb", "l"}, record 7 is an answer, since it contains word "vldb" and a word "luis" with a prefix "l".

Fuzzy type-ahead search: Different from exact type-ahead search, the query answer of fuzzy type-ahead search is a set of records r in R such that for each query keyword p_i, record r contains a word with a prefix *similar to* p_i. In this work, we use edit distance to measure the similarity between two strings. We can transform a string to another using four operations: match, insertion, deletion, and substitution. The *edit distance* between two strings s_1 and s_2, denoted by $\mathsf{ed}(s_1, s_2)$, is the minimum number of insertion, deletion, and substitution operations of single characters needed to transform the first one to the second. We say a word in a record r has a prefix w similar to the query keyword p_i if the edit distance between w and p_i is within a given threshold τ.[2] For example, suppose the edit-distance threshold $\tau = 1$. For a query {"vldb", "lvi"}, record 7 is an answer, since it contains a word "vldb" (matching the query keyword "vldb" exactly) and a word "luis" with a prefix "lui" similar to query keyword "lvi"

[2] For simplicity, we assume a threshold τ on the edit distance between similar strings is given. Our solution can be extended to the case where we want to increase the threshold τ for longer prefixes.

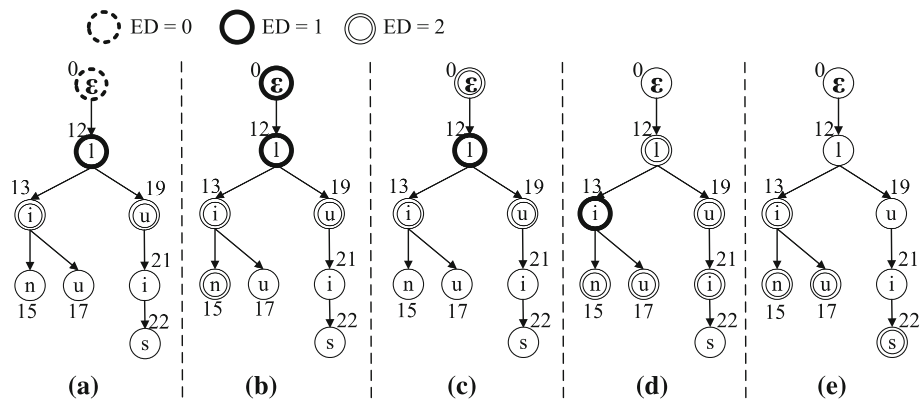

Fig. 4 Fuzzy search of prefix queries of "nlis" on a partial trie (edit-distance threshold $\tau = 2$). **a** Initialize, **b** query 'n', **c** query 'nl', **d** query 'nli', **e** query 'nlis'

Table 1 A set of records

ID	Record
1	EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, Lizhu Zhou. SIGMOD, 2008
2	BLINKS: ranked keyword searches on graphs. Hao He, Haixun Wang, Jun Yang, Philip S. Yu. SIGMOD, 2007
3	Spark: top-*k* keyword query in relational databases. Yi Luo, Xuemin Lin, Wei Wang, Xiaofang Zhou. SIGMOD, 2007
4	Finding top-*k* min-cost connected trees in databases. Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, Xuemin Lin. ICDE, 2007
5	Effective keyword search in relational databases. Fang Liu, Clement T. Yu, Weiyi Meng, Abdur Chowdhury. SIGMOD, 2006
6	Bidirectional expansion for keyword search on graph databases. Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, Hrishikesh Karambelkar. VLDB, 2005
7	Efficient IR-style keyword search over relational databases. Vagelis Hristidis, Luis Gravano, Yannis Papakonstantinou. VLDB, 2003
8	DISCOVER: keyword search in relational databases. Vagelis Hristidis, Yannis Papakonstantinou. VLDB, 2002
9	DBXplorer: a system for keyword-based search over relational databases. Sanjay Agrawal, Surajit Chaudhuri, Gautam Das. ICDE, 2002
10	Keyword searching and browsing in databases using BANKS. Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, S. Sudarshan. ICDE, 2002

(i.e., their edit distance is 1, which is within the threshold $\tau = 1$).

2.2 Indexing

We use a trie to index the words in the records. Each word w in the table corresponds to a unique path from the root of the trie to a leaf node. The label of the root node is ϵ, where ϵ is a special mark and denotes the empty string. The label of each leaf node is also ϵ. Each of other nodes on the path has a label of a character in w. For simplicity, a node is mentioned interchangeably with its corresponding string in the remainder of the paper. Each leaf node has an inverted list of IDs of records that contain the corresponding word. For instance, Fig. 3 shows a partial index structure for publication records. The word "vldb" has a trie node ID of 28, and its inverted list includes record IDs 6, 7, and 8.

In this paper, we focus on in-memory index structures for applications that need to support a high query throughout for a large number of users.

3 Exact search for single keyword

In this section, we discuss how to answer a query with a single keyword using the trie structure.

3.1 Non-incremental method

One naive way to process a query with a single keyword on the server is to answer the query from scratch as follows. We first find the trie node corresponding to the query keyword by traversing the trie from the root. Then, we locate the leaf descendants of this node and retrieve the corresponding words and the records on the inverted lists.

For example, suppose a user types in a query with a single keyword "luis" letter by letter. When the user types in the character "l", the client sends the query "l" to the server. The server finds the trie node corresponding to this keyword, i.e., node 12. Then, it locates the leaf descendants of node 12, i.e., nodes 14, 16, 18, 20, and 23, and retrieves the corresponding words, i.e., "li", "lin", "liu", "lu", and "luis", and the records, i.e., 1, 3, 4, 5, and 7. When the user types in the character "u", the client sends a query string "lu" to the server. The server answers the query from

Table 2 Computing active-node sets for queries of “nlis” (edit-distance threshold $\tau = 2$)

Φ_ϵ	$\langle n_0, 0\rangle$	$\langle n_{12}, 1\rangle$	$\langle n_{13}, 2\rangle$	$\langle n_{19}, 2\rangle$		
(a) Query “n”						
Deletion	$\langle n_0, 1\rangle$	$\langle n_{12}, 2\rangle$	–	–		
Substitution	$\langle n_{12}, 1\rangle$	$\langle n_{13}, 2\rangle$; $\langle n_{19}, 2\rangle$	–	–		
Match	–	–	$\langle n_{15}, 2\rangle$	–		
Insertion	–	–	–	–		
Φ_n	$\langle n_0, 1\rangle$; $\langle n_{12}, 1\rangle$; $\langle n_{13}, 2\rangle$; $\langle n_{19}, 2\rangle$; $\langle n_{15}, 2\rangle$					
Φ_n	$\langle n_0, 1\rangle$	$\langle n_{12}, 1\rangle$	$\langle n_{13}, 2\rangle$	$\langle n_{15}, 2\rangle$	$\langle n_{19}, 2\rangle$	
(b) Query “nl”						
Deletion	$\langle n_0, 2\rangle$	$\langle n_{12}, 2\rangle$	–	–	–	
Substitution	–	$\langle n_{13}, 2\rangle$; $\langle n_{19}, 2\rangle$	–	–	–	
Match	$\langle n_{12}, 1\rangle$	–	–	–	–	
Insertion	$\langle n_{13}, 2\rangle$; $\langle n_{19}, 2\rangle$	–	–	–	–	
Φ_{nl}	$\langle n_{12}, 1\rangle$; $\langle n_0, 2\rangle$; $\langle n_{13}, 2\rangle$; $\langle n_{19}, 2\rangle$					
Φ_{nl}	$\langle n_{12}, 1\rangle$	$\langle n_0, 2\rangle$	$\langle n_{13}, 2\rangle$	$\langle n_{19}, 2\rangle$		
(c) Query “nli”						
Deletion	$\langle n_{12}, 2\rangle$	–	–	–		
Substitution	$\langle n_{19}, 2\rangle$	–	–	–		
Match	$\langle n_{13}, 1\rangle$	–	–	$\langle n_{21}, 2\rangle$		
Insertion	$\langle n_{15}, 2\rangle$; $\langle n_{17}, 2\rangle$	–	–	–		
Φ_{nli}	$\langle n_{13}, 1\rangle$; $\langle n_{12}, 2\rangle$; $\langle n_{15}, 2\rangle$; $\langle n_{17}, 2\rangle$; $\langle n_{19}, 2\rangle$; $\langle n_{21}, 2\rangle$					
Φ_{nli}	$\langle n_{13}, 1\rangle$	$\langle n_{12}, 2\rangle$	$\langle n_{15}, 2\rangle$	$\langle n_{17}, 2\rangle$	$\langle n_{19}, 2\rangle$	$\langle n_{21}, 2\rangle$
(d) Query “nlis”						
Deletion	$\langle n_{13}, 2\rangle$	–	–	–	–	–
Substitution	$\langle n_{15}, 2\rangle$; $\langle n_{17}, 2\rangle$	–	–	–	–	–
Match	–	–	–	–	–	$\langle n_{22}, 2\rangle$
Insertion	–	–	–	–	–	–
Φ_{nlis}	$\langle n_{13}, 2\rangle$; $\langle n_{15}, 2\rangle$; $\langle n_{17}, 2\rangle$; $\langle n_{22}, 2\rangle$					

Table 3 Computing pivotal active-node sets for queries of “nlis”, where the edit-distance threshold is $\tau = 2$

Ψ_ϵ	$\langle n_0, 0, \epsilon, 0\rangle$		
(*a*) Query “n”			
Deletion	$\langle n_0, 1, \epsilon, 0\rangle$		
Match	$\langle n_{15}, 2, \text{“n”}, 2\rangle$		
Ψ_n	$\langle n_0, 1, \epsilon, 0\rangle$; $\langle n_{15}, 2, \text{“n”}, 2\rangle$		
Ψ_n	$\langle n_0, 1, \epsilon, 0\rangle$	$\langle n_{15}, 2, \text{“n”}, 2\rangle$	
(b) Query“nl”			
Deletion	$\langle n_0, 2, \epsilon, 0\rangle$	–	
Match	$\langle n_{12}, 1, \text{“nl”}, 1\rangle$	–	
Ψ_{nl}	$\langle n_{12}, 1, \text{“nl”}, 1\rangle$; $\langle n_0, 2, \epsilon, 0\rangle$		
Ψ_{nl}	$\langle n_{12}, 1, \text{“nl”}, 1\rangle$	$\langle n_0, 2, \epsilon, 0\rangle$	
(c) Query “nli”			
Deletion	$\langle n_{12}, 2, \text{“nl”}, 1\rangle$	–	
Match	$\langle n_{13}, 1, \text{“nli”}, 1\rangle$; $\langle n_{21}, 2, \text{“nli”}, 2\rangle$	–	
Insert	⟨15,2⟩	–	
Ψ_{nli}	$\langle n_{13}, 1, \text{“nli”}, 1\rangle$; $\langle n_{12}, 2, \text{“nl”}, 1\rangle$; $\langle n_{21}, 2, \text{“nli”}, 2\rangle$		
Ψ_{nli}	$\langle n_{13}, 1, \text{“nli”}, 1\rangle$	$\langle n_{12}, 2, \text{“nl”}, 1\rangle$	$\langle n_{21}, 2, \text{“nli”}, 2\rangle$
(d) Query “nlis”			
Deletion	$\langle n_{13}, 2, \text{“nli”}, 1\rangle$	–	–
Match	–	–	$\langle n_{22}, 2, \text{“nlis”}, 2\rangle$
Ψ_{nlis}	$\langle n_{13}, 2, \text{“nli”}, 1\rangle$; $\langle n_{22}, 2, \text{“nlis”}, 2\rangle$		

scratch as follows. It first finds node 19 for this string and then locates the leaf descendants of node 19 (nodes 20 and 23). It retrieves the corresponding words ("`lu`" and "`luis`") and computes the records (4 and 7). Other queries invoked by keystrokes are processed in a similar way. One main limitation of this method is that it involves a lot of re-computation without using the results of earlier queries.

3.2 Incremental algorithm

We can incrementally compute answers as follows. We maintain a session for each user.[3] Each session keeps the keywords that the user has typed in so far and the corresponding trie nodes. The goal is to use such information to answer subsequent queries incrementally.

Assume a user has typed in a query with a single keyword $p_x = c_1c_2 \ldots c_x$ letter by letter. Suppose n_x is the trie node corresponding to p_x. After the user types in a prefix query p_x, we store node n_x for p_x and its relevant records. For example, suppose a user has typed in "`lui`". After this query is submitted, the server has stored node 12 and records 1, 3, 4, 5, and 7 for the prefix query "`l`", node 19 and records 4 and 7 for the prefix query "`lu`", and node 21 and record 7 for "`lui`". For each keystroke from the user, for simplicity, we first assume that the user types in a new character c_{x+1} at the end of the previous query. To incrementally answer the new query, we first check whether node n_x kept for p_x has a child with a character of c_{x+1}. If so, we locate the leaf descendants of node n_{x+1} and retrieve the corresponding words and records. Otherwise, there is no word with a prefix of p_{x+1}, and we can just return an empty answer. For example, if the user types in a letter "`s`" after "`lui`", we check whether node 21 kept for "`lui`" has a child with character "`s`". Here, we find node 22 and retrieve the word "`luis`" and record 7.

It is possible that the user modifies the previous query arbitrarily or copies and pastes a completely different string. In this case, for the new query, among all the keywords typed by the user, we identify the cached keyword that has the *longest* prefix with the new query. Formally, consider a cached query with a single keyword $c_1c_2 \ldots c_x$. Suppose the user submits a new query with a single keyword $p = c_1c_2 \ldots c_i d_{i+1} \ldots d_y$. We find $p_i = c_1c_2 \ldots c_i$ that has a longest prefix with p. Then, we use the node n_i of p_i to incrementally answer the new query p, by inserting the characters after the longest prefix of the new query (i.e., $d_{i+1} \ldots d_y$) one by one. In particular, if there exists a cached keyword $p_i = p$, we use the cached records of p_i to directly answer the query p. If there is no such a cached keyword, we answer the query from scratch.

[3] Different sessions can share their cached results.

4 Fuzzy search for single keyword

In this section, we study fuzzy search for a single keyword. For the case of exact prefix search using a trie index, there can be at most one trie node corresponding to each keyword. We can access the inverted lists of its descendant leaf nodes to retrieve candidate records. The solution to the problem of fuzzy search is more challenging since a keyword can have multiple similar prefixes, and we need to compute them efficiently. In this section, we focus on computing these similar prefixes efficiently.

4.1 Active nodes

Let p be a query keyword and τ be an edit-distance threshold. We call a trie node n an *active node of p with respect to τ*, or simply an *active node of p* when τ is clear in the context, if the edit distance between the string of n and p is within τ, i.e., $\mathsf{ed}(n, p) \leq \tau$. The leaf descendants of the active nodes are called the *similar words* of p. For example, consider the trie in Fig. 4. Suppose the edit-distance threshold $\tau = 2$, and a user types in a prefix $p =$"`nlis`". As illustrated by Fig. 4e, each of the prefixes "`li`", "`lin`", "`liu`", and "`luis`" has an edit distance to p within τ. Thus, nodes 13, 15, 17, and 22 are active nodes of p. The similar words for the prefix are "`li`", "`lin`", "`liu`", and "`luis`". We can retrieve the records on the inverted lists of the similar words to compute answers to the prefix query.

4.2 Incrementally computing active nodes

We now study how to compute active nodes efficiently for a keyword as the user types in a keyword character by character. We develop a caching-based algorithm, called "ICAN", which stands for incrementally computing active nodes. Given a query keyword p, we want to compute and store a set of tuples $\Phi_p = \{\langle n, \xi_n \rangle\}$, such that (1) each n is an active node of p with $\xi_n = \mathsf{ed}(n, p) \leq \tau$ and (2) every active node of p appears in a tuple in Φ_p. We call Φ_p the *set of active nodes of p*. The idea of the ICAN algorithm is the following: when the user types in one more letter after p, the active nodes of p can be used to compute the active nodes of the new query. For example, assume a user types in a query "`nlis`" letter by letter and the threshold τ is 2. Figure 2 illustrates how the algorithm processes the queries invoked by keystrokes. Table 2 shows the details of how to compute the active node sets incrementally.

4.2.1 Algorithm description

Now we describe the details of the ICAN algorithm. Initially, before the user types in any character, the query keyword is the empty string ϵ, and its corresponding active node

set Φ_ϵ is initialized as: $\Phi_\epsilon = \{\langle n, \xi_n \rangle \mid \xi_n = |n| \leq \tau\}$. That is, it includes all trie nodes n whose corresponding string has a length $|n|$ within the edit-distance threshold τ. Clearly, these nodes are all the active nodes for ϵ. In our running example, the first step is to initialize $\Phi_\epsilon = \{\langle n_0, 0\rangle, \langle n_{12}, 1\rangle, \langle n_{13}, 2\rangle, \langle n_{19}, 2\rangle\}$, as shown in Fig. 4a and Table 2a, where n_0 denotes node 0.

Suppose after the user types in a query string $p_x = c_1 c_2 \ldots c_x$, we have computed the active node set Φ_{p_x} for p_x. Now, the user types in a new character c_{x+1} and submits a new query p_{x+1}. The algorithm computes the active node set $\Phi_{p_{x+1}}$ for p_{x+1} by using Φ_{p_x} as follows. We initialize $\Phi_{p_{x+1}}$ to be empty. Before we present the details of adding more tuples to $\Phi_{p_{x+1}}$, we first introduce a notation. Given two strings s_1 and s_2, consider a transformation from s_1 to s_2. We define the *transformation distance with respect to this transformation* is the number of single-character edit operations in the transformation. We introduce this notation instead of using the standard definition of edit distance because there can be different numbers of edit operations transforming one string to another. Two strings can have multiple transformation distances with respect to different transformations, and their edit distance is referring to the minimum one.

For each $\langle n, \xi_n \rangle$ in Φ_{p_x}, only the descendants of n are examined as active node candidates for p_{x+1}, as illustrated in Fig. 5. We consider the following cases.

Considering node n: Consider the active node n of p_x. We can transform n to p_{x+1} with $\xi_n + 1$ edit operations by first transforming n to p_x (with ξ_n edit operations) and then deleting the last character c_{x+1}. If $\xi_n + 1 \leq \tau$, we add $\langle n, \xi_n + 1\rangle$ to $\Phi_{p_{x+1}}$. In our running example, consider the case where the user types in the first character "n". For $\langle n_0, 0\rangle \in \Phi_\epsilon$, since we can apply a deletion operation on letter "n" with 1 edit operation, we add $\langle n_0, 1\rangle$ to $\Phi_{\mathtt{n}}$.

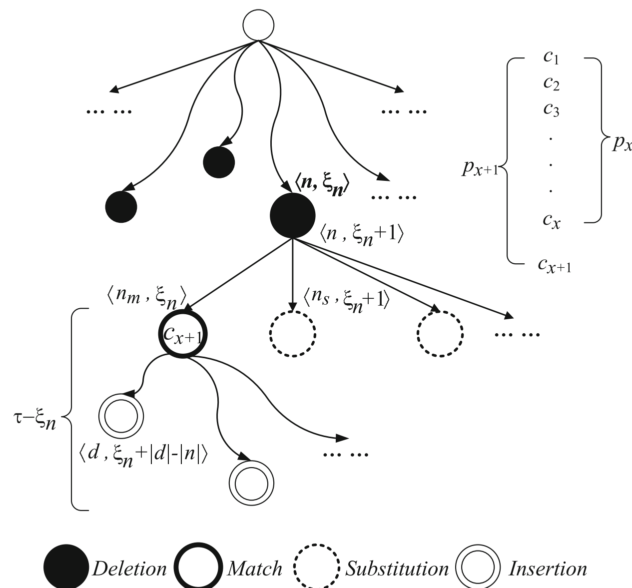

Fig. 5 Incrementally computing the active-node set $\Phi_{p_{x+1}}$ from the active-node set Φ_{p_x}. We consider an active node $\langle n, \xi_n\rangle$ in Φ_{p_x}

Considering children of node n: For each child n_c of node n, there are two possible cases.

Case 1 The child node n_c has a character different from c_{x+1}. Figure 5 shows a node n_s for such a child node, where "*s*" stands for *substitution*. We can transform n_s to p_{x+1} with $\xi_n + 1$ operations by first transforming n to p_x (with ξ_n operations) and then substituting the character of n_s for the character c_{x+1}. If $\xi_n + 1 \leq \tau$, then we add $\langle n_s, \xi_n + 1\rangle$ to $\Phi_{p_{x+1}}$. This case corresponds to substituting the character of n_s for the character c_{x+1}. In the running example, consider the case where the user types in the first character "n". For $\langle n_0, 0\rangle \in \Phi_\epsilon$, node 12 is a child of node 0 with a letter "l". Since we can apply a substitution operation on character "l" for"n" with 1 edit operation, $\langle n_{12}, 1\rangle$ is added to $\Phi_{\mathtt{n}}$.

Case 2 The child node n_c has a character c_{x+1}, i.e., a character matching the new character in the query. Figure 5 shows the node n_m for such a child node, where "*m*" stands for *matching*. In this case, we can transform n_m to p_{x+1} with ξ_n edit operations by first transforming n to p_x (with ξ_n operations) and then matching the character of n_m with the character c_{x+1}. We add $\langle n_m, \xi_n\rangle$ to $\Phi_{p_{x+1}}$. This case corresponds to the match between the character of n_m and the character c_{x+1}. One subtlety here is that if $\xi_n < \tau$, the following operations are also required: for each n_m's descendant d that is at most $\tau - \xi_n$ letters away from n_m, we need to add $\langle d, \xi_d\rangle$ to $\Phi_{p_{x+1}}$, where $\xi_d = \xi_n + |d| - |n_m|$. The operations for each descendant correspond to inserting several letters after node n_m. In our running example, suppose the user types in the first character "l". For $\langle n_0, 0\rangle \in \Phi_\epsilon$, node 12 is a child of node 0 with a letter "l". As the character of node 12 ("l") matches letter "l", $\langle n_{12}, 0\rangle$ is added to $\Phi_{\mathtt{l}}$. Node 13 is a child of node 12. As we can insert the character of n_{13} (character "i") after node 12 with 1 edit operation, $\langle n_{13}, 1\rangle$ is added to $\Phi_{\mathtt{l}}$.

Keeping minimum distances: During the computation of the set $\Phi_{p_{x+1}}$, whenever we add a tuple $\langle v, \xi_1\rangle$ to the set, there can be already a tuple $\langle v, \xi_2\rangle$ for the same trie node v in the set. If $\xi_1 \geq \xi_2$, then the new tuple is not added. If $\xi_1 < \xi_2$, then the new tuple will replace the original tuple $\langle v, \xi_2\rangle$. In other words, for the same trie node v, in the new set, we only keep its minimum transformation distance to the query string p_{x+1}.

Complexity analysis: Given a query keyword p, we analyze the complexity of the algorithm ICAN for computing its active node set from that of its prefix p', which does not have the last character of p. Consider the set of active

nodes of p, Φ_p. We first consider each active node in Φ_p, which is computed from its ancestors within τ steps. Such an active node is inserted into Φ_p at most τ times. The total time to compute them is $\mathcal{O}(\tau * |\Phi_p|)$. In addition, we need to consider those nodes that are considered by the algorithm but are not inserted into Φ_p since their edit distance to p is greater than τ. Such nodes can be detected and avoided by checking whether an active node n in Φ'_p has an edit distance to p strictly less than τ. These checking steps take $\mathcal{O}(|\Phi'_p|)$. Thus, the total time complexity is $\mathcal{O}\left(\tau * (|\Phi_p| + |\Phi'_p|)\right)$.

The space complexity is $\mathcal{O}(|\Phi_p| + |\Phi_{p'}|)$, since we only need to keep these two active node sets.

4.2.2 Proof of correctness

We now prove that the set $\Phi_{p_{x+1}}$ computed by the ICAN algorithm above is indeed the set of active nodes with their edit distances for the new prefix p_{x+1}. We prove the claim by providing two lemmas corresponding to the completeness and soundness, respectively.

Lemma 1 (Completeness) *Let p be a query keyword. For each active node n of p, the tuple $\langle n, \mathsf{ed}(n, p)\rangle$ must be in the set Φ_p computed by the* ICAN *algorithm.*

Proof We prove this lemma by induction. This claim is obviously true when $p = p_0 = \epsilon$. Suppose the claim is true for p_x with x characters. We want to prove this claim is also true for a new query string p_{x+1}, where $p_{x+1} = p_x c$, i.e., p_{x+1} is a concatenated string of the string p_x and a character c.

Suppose v is an active node of p_{x+1}. If $v = \epsilon$, then by definition, $\mathsf{ed}(v, p_{x+1}) = \mathsf{ed}(\epsilon, p_{x+1}) = x + 1 \le \tau$, and $x \le \tau - 1 < \tau$. Thus, $\mathsf{ed}(v, p_x) = \mathsf{ed}(\epsilon, p_x) = x \le \tau$, and v is also an active node of p_x. When the ICAN algorithm considers this node v, it adds the pair $\langle v, x+1\rangle$ (i.e., $\langle v, \mathsf{ed}(v, p_{x+1})\rangle$) to the set $\Phi_{p_{x+1}}$.

Now consider the case where the active node v of p_{x+1} is not the empty string. Let $v = n_{y+1} = n_y d$, i.e., it has $y+1$ characters, and is concatenated from a string n_y and a character d. By definition, $\mathsf{ed}(n_{y+1}, p_{x+1}) \le \tau$. We want to prove that $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$ will be added to $\Phi_{p_{x+1}}$ in the ICAN algorithm.

Based on the idea in the classic dynamic programming algorithm [30], we have

$$\mathsf{ed}(n_{y+1}, p_{x+1}) = \min \begin{cases} \mathsf{ed}(n_{y+1}, p_x) + 1 & \textit{deletion} \\ \begin{cases} \mathsf{ed}(n_y, p_x) + 1 & \text{if } c \neq d \\ \mathsf{ed}(n_y, p_x) & \text{if } c = d \end{cases} \\ \mathsf{ed}(n_y, p_{x+1}) + 1 & \textit{insertion} \end{cases}$$

Correspondingly, we consider the following four cases in the minimum number of edit operations to transform n_{y+1} to p_{x+1}.

Case 1 Deleting the last character c from p_{x+1}, and transforming n_{y+1} to p_x. Since $\mathsf{ed}(n_{y+1}, p_{x+1}) = \mathsf{ed}(n_{y+1}, p_x) + 1 \le \tau$, we have $\mathsf{ed}(n_{y+1}, p_x) \le \tau - 1 < \tau$. Thus, n_{y+1} is an active node of p_x. Based on the induction assumption, $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_x)\right\rangle$ must be in Φ_{p_x}. From the node n_{y+1}, the ICAN algorithm considers the deletion case when it considers the node n_y and adds $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_x) + 1\right\rangle$ to $\Phi_{p_{x+1}}$, which is exactly $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$.

Case 2 Substituting the character d of n_{y+1} for the last character c of p_{x+1}. Since $\mathsf{ed}(n_{y+1}, p_{x+1}) = \mathsf{ed}(n_y, p_x) + 1 \le \tau$, we have $\mathsf{ed}(n_y, p_x) \le \tau - 1 < \tau$. Thus, n_y is an active node of p_x. Based on the induction assumption, $\left\langle n_y, \mathsf{ed}(n_y, p_x)\right\rangle$ must be in Φ_{p_x}. From node n_y, the ICAN algorithm considers the substitution case when it considers this child node (n_{y+1}) of the node n_y and adds $\left\langle n_{y+1}, \mathsf{ed}(n_y, p_x) + 1\right\rangle$ to $\Phi_{p_{x+1}}$, which is exactly $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$.

Case 3 The last character c of p_{x+1} matching the character d of n_{y+1}. Since $\mathsf{ed}(n_{y+1}, p_{x+1}) = \mathsf{ed}(n_y, p_x) \le \tau$, then n_y is active node of n_x. Based on the induction assumption, $\left\langle n_y, \mathsf{ed}(n_y, p_x)\right\rangle$ must be in Φ_{p_x}. From node n_y, the ICAN algorithm considers the match case when it considers this child node (n_{y+1}) of the node n_y and adds $\left\langle n_{y+1}, \mathsf{ed}(n_y, p_x)\right\rangle$ to $\Phi_{p_{x+1}}$, which is exactly $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$.

Case 4 Transforming n_y to p_{x+1} and inserting character d of n_{y+1}. For each transformation from n_y to p_{x+1}, we consider the last character c of p_{x+1}. First, we can show that this transformation cannot delete the character c, since otherwise we can combine this deletion of c and the insertion of d into one substitution, yielding another transformation with a smaller number of edit operations, contradicting to the minimality of edit distance.

Thus, we can just consider two possible operations on the character c in this transformation. (1) Matching c for the character of an ancestor n_a of n_{y+1}: In this case, since $\mathsf{ed}(n_{y+1}, p_{x+1}) = \mathsf{ed}(n_{a-1}, p_x) + y - a + 1 \le \tau$, we have $\mathsf{ed}(n_{a-1}, p_x) \le \tau$, and n_{a-1} is an active node of p_x. Based on the induction assumption, $\langle n_{a-1}, \mathsf{ed}(n_{a-1}, p_x)\rangle$ must be in Φ_{p_x}. From node n_{a-1}, the algorithm considers the matching case and adds $\left\langle n_{y+1}, \mathsf{ed}(n_{a-1}, p_x) + y - a + 1\right\rangle$ to $\Phi_{p_{x+1}}$, which is $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$. (2) Substituting c for the character of an ancestor n_a of n_{y+1}: In this case, instead of substituting c for the character of n_a and inserting the character d, we can insert the character of n_a and substitute c for the last character d. Then, we get another transformation with the same number of edit operations. (Characters c and d cannot be the same, since otherwise the new transformation could have fewer edit operations, contradicting to the minimality of edit distance.) We use the same argument in "Case 2" to show that the ICAN algorithm adds $\left\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\right\rangle$ to $\Phi_{p_{x+1}}$.

In summary, for all cases, the algorithm adds $\langle n_{y+1}, \mathsf{ed}(n_{y+1}, p_{x+1})\rangle$ to $\Phi_{p_{x+1}}$.

Lemma 2 (Soundness) *For each tuple $\langle n, \xi_n\rangle$ in the final set Φ_p computed in the* ICAN *algorithm, the node n is an active node of p, and $\xi_n = \mathsf{ed}(n, p)$.*

Proof By definition, a transformation distance of two strings in each added tuple by the algorithm is no less than their edit distance. That is, $\mathsf{ed}(n, p) \le \xi_n \le \tau$. Therefore, n must be an active node of p. Based on the completeness claim in Lemma 1, $\langle n, \mathsf{ed}(n, p)\rangle$ must be in Φ_p. In addition, for each node, the ICAN algorithm only keeps the minimum transformation distance, and the edit distance is the minimum transformation distance. Therefore, there can be only one tuple $\langle n, \xi_n\rangle$ for node n in Φ_p, and $\xi_n = \mathsf{ed}(n, p)$. □

Theorem 1 *Given a query keyword p, the* ICAN *algorithm computes all the active nodes of p with their edit distances.*

Proof It is a corollary of the two lemmas above. □

4.3 Improvement by pruning active nodes

Our end goal of computing active nodes of a query keyword is to use them to identify the similar words of the keyword. There may be many active nodes for a given query keyword. In this section, we discuss how to prune unnecessary active nodes while we can still compute all the similar words of the keyword. This method has two advantages: (1) We can reduce the space to store active nodes; (2) We can improve search performance since we do not need to scan all the active nodes for incremental computation.

The intuition of our approach can be illustrated as follows. In our running example, consider a query keyword "`nl`" with a threshold $\tau = 2$, where $\Phi_{\mathtt{nl}} = \{\langle n_{12}, 1\rangle; \langle n_0, 2\rangle; \langle n_{13}, 2\rangle; \langle n_{19}, 2\rangle\}$. Although n_{13} ("`li`") and n_{19} ("`lu`") are active nodes, we do not need to keep them, since we can use the active node n_{12} ("`l`") to compute the similar words of "`li`" and "`lu`" using "`l`". In other words, we only need to keep the active node "`l`" to compute the same set of similar words for the query keyword.

4.3.1 Pivotal active nodes

When defining a subset of active nodes that can be used to compute all the similar words for the query keyword, we also want to be able to compute them incrementally and efficiently. To define such a subset, we have the following observation. Given an active node n of p, if for any transformation from n to p with $\mathsf{ed}(n, p)$ operations, the operation on the last character of n is not a match operation and we can delete the last character of n. We keep n's parent and do not keep n. For example, consider the query keyword "`nl`" in our running example. The node of "`l`" is an active node. "`li`" and "`lu`" are also active nodes, and their edit distances to "`nl`" are 2 and their last characters do not match characters in the query keyword "`nl`". Obviously, we can derive the two active nodes from the node "`l`" by appending characters "`i`" and "`u`", respectively. In addition, we can compute the similar words "`li`" and "`lu`" by visiting the leaf descendants of "`l`". Thus, we do not want to keep these two nodes. To this end, we propose the concept of *pivotal active node*.

Definition 1 (*Pivotal Active Node*) Given a query keyword p, a trie node n is a pivotal active node of p with respect to an edit-distance threshold τ, if and only if (1) n is an active node of p and (2) there exists a transformation from p to n with $\mathsf{ed}(n, p)$ edit operations, and the operation on the last character of n is a "match" operation. That is the operation on character of n is neither deletion $\mathsf{ed}(n, p) \ne \mathsf{ed}(n', p) + 1$ nor substitution $\mathsf{ed}(n, p) \ne \mathsf{ed}(n', p') + 1$, where n' and p' are respectively the prefixes of n and p which do not contain the last character.

Next, we prove that the set of pivotal active nodes can be used to compute all the similar words for a query keyword. We will propose an incremental algorithm to efficiently compute pivotal active nodes in Sect. 4.3.2.

Lemma 3 *For a query keyword p, let A and P respectively denote the active node set and pivotal active node set of the query string, $L(A)$ and $L(P)$ respectively denote the set of leaf nodes of nodes in A and P. We have $L(A) = L(P)$.*

Proof It is obvious that $L(P) \subseteq L(A)$ as $P \subseteq A$. We only need to prove $L(A) \subseteq L(P)$. Consider a leaf node $l \in L(A)$, which is a descendant of an active node n. If n is a pivotal active node, i.e., $n \in P$, then $l \in L(P)$. If n is not a pivotal active node, we can find one of its ancestors n_a, which is a pivotal active node. As l is a descendant of n_a, $l \in L(P)$. Hence, $L(A) \subseteq L(P)$.

Next, we present how to find pivotal active node n_a. Let $n = n_y = d_1 d_2 \cdots d_y$ and $p = p_x = c_1 c_2 \cdots c_x$, where d_s and c_t are characters for $1 \le s \le y$ and $1 \le t \le x$. We first check whether n_y is a pivotal active node based on the definition. If not, n_{y-1} must be an active node, since (1) if $\mathsf{ed}(n_y, p_x) = \mathsf{ed}(n_{y-1}, p_x) + 1$, we have $\mathsf{ed}(n_{y-1}, p_x) \le \mathsf{ed}(n_y, p_x) - 1 \le \tau$; (2) if $\mathsf{ed}(n_y, p_x) = \mathsf{ed}(n_{y-1}, p_{x-1}) + 1$, for each transformation ρ from n_{y-1} to p_{x-1}, we can construct a transformation from n_{y-1} to p_x, which is the same as ρ except that we delete character c_x, and thus we have $\mathsf{ed}(n_{y-1}, p_x) \le \mathsf{ed}(n_{y-1}, p_{x-1}) + 1 \le \mathsf{ed}(n_y, p_x) \le \tau$. Then we check whether n_{y-1} is a pivotal active node based on the definition. Iteratively, we will find a pivotal active node. Note that if each n_i (for $y \ge i \ge 1$) is not a pivotal active node, the root node must be. □

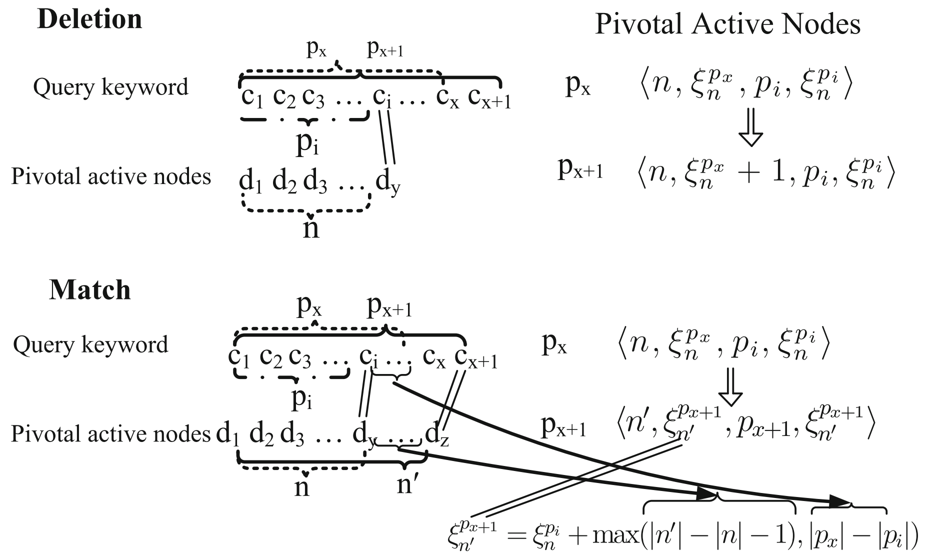

Fig. 6 Incrementally computing pivotal active nodes ($c_i = d_y$ and $c_{x+1} = d_z$)

4.3.2 Incrementally computing pivotal active nodes

We now study how to compute pivotal active nodes efficiently for a keyword as the user types in a keyword character by character. We develop a caching-based algorithm, called "ICPAN", which stands for "Incrementally Computing Pivotal Active Nodes," which extends the previous ICAN algorithm. Given an query keyword p_x, we want to compute and store a set of quadruples $\Psi_{p_x} = \{\langle n, \xi_n^{p_x}, p_i, \xi_n^{p_i} \rangle\}$. In each quadruple, n is a pivotal active node of p_x. p_i is a prefix of p_x such that the last characters of n and p_i match. If no such prefix exists, $p_i = \epsilon$; if there are multiple such prefixes, we select the one with the shortest length. $\xi_n^{p_i} \leq \tau$ is a transformation distance from node n to p_i with a match operation on the last characters of n and p_i. $\xi_n^{p_x} \leq \tau$ is a transformation distance from node n to p_x by first transforming node n to p_i and then appending the characters after p_i, that is $\xi_n^{p_x} = \xi_n^{p_i} + |p_x| - |p_i|$. We devise an algorithm to compute Ψ_{p_x} and guarantee that every pivotal active node of p_x appears as the "n" node in a quadruple in Ψ_{p_x}.

Algorithm description: Initially, before the user types in characters, the query keyword is the empty string ϵ, and we initialize its corresponding set $\Psi_\epsilon = \{\langle r, 0, \epsilon, 0 \rangle\}$, where r is the root node, since obviously the root is the only pivotal active node for ϵ. In the running example, assume a user types in a query "nlis" letter by letter, and the threshold τ is 2. Table 3 shows the details of how to compute the pivotal active node sets incrementally. The first step is to initialize $\Psi_\epsilon = \{\langle n_0, 0, \epsilon, 0 \rangle\}$, where n_0 denotes the root node 0.

After the user types in a query string $p_x = c_1 c_2 \ldots c_x$, we have computed the pivotal active node set Ψ_{p_x} for p_x. Now the user types in a new character c_{x+1} and submits a new query p_{x+1}. The ICPAN algorithm computes the pivotal active node set $\Psi_{p_{x+1}}$ for p_{x+1} by using Ψ_{p_x} as follows. The set $\Psi_{p_{x+1}}$ is initialized to be empty. For each quadruples $\langle n, \xi_n^{p_x}, p_i, \xi_n^{p_i} \rangle$ in Ψ_{p_x}, only the descendants of n are examined as pivotal active node candidates for p_{x+1} as shown in Fig. 6.

Considering node n: Consider the pivotal active node n of p_x. We can transform n to p_{x+1} with $\xi_n^{p_x} + 1$ operations by first transforming n to p_x (with $\xi_n^{p_x}$ operations) and then deleting the last character c_{x+1}. If $\xi_n^{p_x} + 1 \leq \tau$, we add $\langle n, \xi_n^{p_x} + 1, p_i, \xi_n^{p_i} \rangle$ to $\Psi_{p_{x+1}}$. For example, assume the user types in the first character "n". For $\langle n_0, 0, \epsilon, 0 \rangle \in \Psi_\epsilon$, since we can apply a deletion operation on the character "n" with 1 edit operation, we add $\langle n_0, 1, \epsilon, 0 \rangle$ into $\Psi_{\mathtt{n}}$.

Considering descendants of node n: Consider n's descendants that have character c_{x+1} and are within $\tau - \xi_n^{p_i} + 1$ steps from node n. For each such node n', we can transform n' to p_{x+1} as follows: (1) transforming n to p_i; (2) transforming the characters after n and before n' to the characters $c_{i+1} \cdots c_x$; and (3) matching the character of n' with the character c_{x+1}. Thus, we can transform n' to p_{x+1} with $\xi_{n'}^{p_{x+1}} = \xi_n^{p_i} + \max(|n'| - |n| - 1, |p_x| - |p_i|)$ operations. If $\xi_{n'}^{p_{x+1}} \leq \tau$, we add $\langle n', \xi_{n'}^{p_{x+1}}, p_{x+1}, \xi_{n'}^{p_{x+1}} \rangle$ into $\Psi_{p_{x+1}}$. For example, assume the user types in the first character "n". For $\langle n_0, 0, \epsilon, 0 \rangle \in \Psi_\epsilon$, since the character of node n_{15}("lin") matches "n", we add $\langle n_{15}, 2, \text{"lin"}, 2 \rangle$ into $\Psi_{\mathtt{n}}$.

Keeping the minimum transformation distance: During the computation of the new set $\Psi_{p_{x+1}}$, for the same trie node n, we only keep the minimum transformation distance between the node n and the query string p_{x+1}. In particular, whenever we add a quadruple $\langle n, \xi_n^{p_{x+1}}, p_i, \xi_n^{p_i} \rangle$ to the set, there might be already a quadruple $\langle n, {\xi'}_n^{p_{x+1}}, p_j, {\xi'}_n^{p_j} \rangle$ for the same trie

node n in the set. If $\xi'^{p_{x+1}}_n > \xi^{p_{x+1}}_n$, then the new quadruple is not added. If $\xi'^{p_{x+1}}_n = \xi^{p_{x+1}}_n$ and $|p_j| < |p_i|$, then the new quadruple will replace the original quadruple (keeping the prefix with the shortest length). If $\xi'^{p_{x+1}}_n < \xi^{p_{x+1}}_n$, then the new quadruple will replace the original quadruple.

Removing non pivotal active nodes: For each quadruple $\langle n, \xi^{p_{x+1}}_n, p_i, \xi^{p_i}_n \rangle$, if $p_i \neq p_{x+1}$, for each ancestor node $n_a \neq n$ of n, if $\langle n_a, \xi^{p_{x+1}}_{n_a}, p_a, \xi^{p_a}_{n_a} \rangle \in \Psi_{p_{x+1}}$ and $\xi^{p_a}_{n_a} + \max(|p_{x+1}| - |p_a|, |n| - |n_a|) < \xi^{p_{x+1}}_n$, we remove $\langle n, \xi^{p_{x+1}}_n, p_i, \xi^{p_i}_n \rangle$ from the set. The reason is that there is a transformation from n to p_{x+1} with smaller transformation distance than $\xi^{p_{x+1}}_n$ and the last character of node n does not match, i.e., n is not a pivotal active node of p_{x+1}.

Complexity analysis: Given a query keyword p, we analyze the complexity of the algorithm ICPAN for computing its pivotal active node set from that of its prefix p', which does not have the last character of p. Consider the set of pivotal active nodes of p, Ψ_p. Different from ICAN, ICPAN needs to remove non-pivotal active nodes from Ψ_p. Considering the removed tuples, all such tuples must be added into Ψ_p in the deletion case (see Fig. 6). For each node in $\Psi_{p'}$, considering the deletion case, ICPAN adds a tuple into Ψ_p. Thus, ICPAN adds at most $|\Psi_{p'}|$ tuples for the deletion case, that is ICPAN removes at most $|\Psi_{p'}|$ tuples from Ψ_p in the removal step. As there are $|\Psi_p|$ tuples in Ψ_p, ICPAN inserts at most $|\Psi_p| + |\Psi_{p'}|$ tuples into Ψ_p totally. Similar to ICAN, each tuple is added into Ψ_p at most τ times. Thus, the time complexity of ICPAN is $\mathcal{O}\left(\tau * (|\Psi_p| + |\Psi_{p'}|)\right)$.

As ICPAN needs to maintain the pivotal active node sets Ψ_p and $\Psi_{p'}$, the space complexity is $\mathcal{O}(|\Psi_p| + |\Psi_{p'}|)$.

In the running example where a user types in a query "nlis" letter by letter, and the threshold τ is 2. Table 3 shows the details of how to compute the pivotal active node sets incrementally. The first step is to initialize $\Psi_\epsilon = \{\langle n_0, 0, \epsilon, 0\rangle\}$ (Table 2a). When the user types in the first character "n", we apply edit operations on the character until reaching the threshold. Its active node set $\Psi_{\mathtt{n}}$ can be computed based on Ψ_ϵ as follows. For $\langle n_0, 0, \epsilon, 0\rangle \in \Psi_\epsilon$, as we can apply deletion on character "n" with 1 edit operation, we add $\langle n_0, 1, \epsilon, 0\rangle$ into $\Psi_{\mathtt{n}}$. As the character of node n_{15} ("lin") matches the last character of the query keyword, we can apply a match operation and insert $\langle n_{15}, 2, \text{“lin”}, 2\rangle$ into $\Psi_{\mathtt{n}}$. Then, the user types in a character "l". For $\langle n_0, 1, \epsilon, 0\rangle \in \Psi_{\mathtt{n}}$, as we can apply a deletion operation on character "l" with 1 edit operation, we add $\langle n_0, 2, \epsilon, 0\rangle$ into $\Psi_{\mathtt{nl}}$. As the character of node n_{12} ("l") matches the new character in the query keyword, we can apply a match operation and insert $\langle n_{12}, 1, \text{“l”}, 1\rangle$ into $\Psi_{\mathtt{nl}}$. For $\langle n_{15}, 2, \text{“lin”}, 2\rangle \in \Phi_{\mathtt{nl}}$, as it has no descendant, we will not insert any node. Similarly, we can compute the sets for the prefix queries of "nlis" incrementally.

For each active node, the words corresponding to its leaf descendants are similar words. For the query "nli", there are six active nodes as shown in Table 2c, and there are only three pivotal active nodes as shown in Table 3c. We can get the similar words of "nli" using the pivotal active nodes, e.g., "li".

4.3.3 Proof of correctness

We now prove that the set $\Psi_{p_{x+1}}$ computed by the ICPAN algorithm is indeed the set of pivotal active nodes for the new keyword p_{x+1}. For ease of presentation, we first prove that, given a query keyword p and an active node n, if their last characters are the same, n must be a pivotal active node of p as follows.

Lemma 4 *Given a query keyword p and an active node n, if their last characters are the same, n must be a pivotal active node of p.*

Proof Let $n = n_j = d_1 d_2 \cdots d_j$ and $p = p_k = c_1 c_2 \cdots c_k$, where d_s and c_t are characters for $1 \le s \le j$ and $1 \le t \le k$. Based on the classic dynamic programming algorithm [30], $\mathsf{ed}(n_j, p_k) = \min\left(\mathsf{ed}(n_{j-1}, p_{k-1}), \mathsf{ed}(n_{j-1}, p_k) + 1, \mathsf{ed}(n_j, p_{k-1}) + 1\right)$. If $\mathsf{ed}(n_j, p_k) = \mathsf{ed}(n_{j-1}, p_{k-1})$, there exists a transformation from n to p_k with $\mathsf{ed}(n, p_k)$ operations: first transforming n_{j-1} to p_{k-1} and then matching d_j with c_k. Hence, n is a pivotal active node of p_k.

Next, we prove $\mathsf{ed}(n_j, p_k) = \mathsf{ed}(n_{j-1}, p_{k-1})$. Based on the dynamic-programming equation, we prove $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1}) + 1$, and $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_{j-1}, p_k) + 1$. Firstly, we prove $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1}) + 1$. Suppose there is a transformation ρ from n_j to p_{k-1} with $\mathsf{ed}(n_j, p_{k-1})$ operations. We consider the operation on d_j in ρ. If d_j is deleted, we can construct a transformation from n_{j-1} to p_{k-1} with the same operations in ρ except that we do not delete n_j. Thus, $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1}) - 1$. If d_j matches c_s, we can construct a transformation from n_{j-1} to p_{k-1} with the same operations in ρ except that we delete c_s. Thus, $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1}) + 1$. If d_j substitutes for c_s, we can construct a transformation from n_{j-1} to p_{k-1} with the same operations in ρ except that we delete c_s and do not do the substitution. Thus, $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1})$. Based on the above three cases, $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_j, p_{k-1}) + 1$.

Similarly, we can prove that $\mathsf{ed}(n_{j-1}, p_{k-1}) \le \mathsf{ed}(n_{j-1}, p_k) + 1$. Thus, n must be a pivotal active node of p. □

Next, based on Lemma 4, we prove the claim by providing two lemmas corresponding to the completeness and soundness, respectively.

Lemma 5 (Completeness) *Let p be a query keyword. For each pivotal active node n of p, the quadruple* $\langle n, \mathsf{ed}(n, p) =$

$\mathsf{ed}(n, p') + |p| - |p'|, p', \mathsf{ed}(n, p')\rangle$ *must be in the set* Ψ_p *computed by the* ICPAN *algorithm described above, where* p' *is a prefix of* p. *If there are multiple such prefixes,* p' *is the shortest one.*

Proof We prove this lemma by induction. This claim is obviously true when $p = p_0 = \epsilon$. Suppose the claim is true for $p_0, p_1, \ldots, p_x$, and we want to prove this claim is also true for p_{x+1}.

Suppose n' is a pivotal active node of $p_{x+1} = c_1c_2 \cdots c_{x+1}$. By definition, $\mathsf{ed}(n', p_{x+1}) \le \tau$. We want to prove $\langle n', \mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n', p_i) + |p_{x+1}| - |p_i|, p_i, \mathsf{ed}(n', p_i)\rangle$ will be added to $\Psi_{p_{x+1}}$, where p_i is a prefix of p_{x+1} to n' in the ICPAN algorithm. As n' is a pivotal active node of p_{x+1}, there must exist a transformation ρ from n' to p_{x+1} with $\mathsf{ed}(n', p_{x+1})$ edit operations, such that the character d of node n' matches a character in p_{x+1} in the transformation. There are two cases in the transformation.

Case 1 d matches the last character c_{x+1} of p_{x+1}. Suppose n_p is the parent of n'. We have $\mathsf{ed}(n_p, p_x) = \mathsf{ed}(n', p_{x+1})$ based on Lemma 4. Thus, n_p is an active node of p_x. We find the nearest ancestor node n of n_p, which is a pivotal active node of p_x based on Lemma 3. Note that when finding n, the letters after n before n_p can be substituted and deleted (otherwise, a descendant of n could be a pivotal active node). Thus, there exists a transformation ρ from n' to p_{x+1} with $\mathsf{ed}(n', p_{x+1})$ operations, in which the character of n matches a character in p_{x+1}. Suppose the character of n matches c_k. (If there are multiple p_k's, we select the shortest one.) Thus, the transformation ρ from n' to p_{x+1} includes the following: (1) transforming n to p_k with $\mathsf{ed}(n, p_k)$ operations; (2) transforming the characters after n and before n' to the letters $c_{k+1} \cdots c_x$ with $\max(|n'| - |n| - 1, |p_x| - |p_k|)$ operations; and (3) matching the character of n' with the character c_{x+1}. Thus, $\mathsf{ed}(n, p_k) + \max(|n'| - |n| - 1, |p_x| - |p_k|) = \mathsf{ed}(n', p_{x+1})$.

As $\mathsf{ed}(n, p_k) \le \mathsf{ed}(n', p_{x+1}) - \max(|n'| - |n| - 1, |p_x| - |p_k|) \le \tau$, node n is an active node of p_k. As the character of n matches c_k, based on Lemma 4, n must be a pivotal active node of p_k. Based on the induction assumption, $\langle n, \mathsf{ed}(n, p_k), p_k, \mathsf{ed}(n, p_k)\rangle$ must be in Ψ_{p_k}. As n is a pivotal active node of p_x and the character of n matches c_k, we have $\mathsf{ed}(n, p_x) = \mathsf{ed}(n, p_k) + |p_x| - |p_k| \le \mathsf{ed}(n', p_{x+1}) \le \tau$. From node n, the ICPAN algorithm considers the deletion case and adds $\langle n, \mathsf{ed}(n, p_x), p_k, \mathsf{ed}(n, p_k)\rangle$ into Ψ_{p_x}. As n is a pivotal active node of p_x, the distance $\mathsf{ed}(n, p_x)$ is the minimum transformation distance, thus this quadruple cannot be deleted in the ICPAN algorithm. As p_k is the shortest one that matches n, the quadruple $\langle n, \mathsf{ed}(n, p_x), p_k, \mathsf{ed}(n, p_k)\rangle$ must be in Ψ_{p_x}. From node n, the ICPAN algorithm considers the match case and adds $\langle n', \mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n, p_k) + \max(|n'| - |n| - 1, |p_x| - |p_k|), p_{x+1}, \mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n, p_k) + \max(|n'| - |n| - 1, |p_x| - |p_k|)\rangle$ into $\Psi_{p_{x+1}}$, which is exactly $\langle n', \mathsf{ed}(n', p_{x+1}), p_{x+1}, \mathsf{ed}(n', p_{x+1})\rangle$.

Case 2 d matches character c_i $(i < x + 1)$. If there are multiple such c_i characters, we select the smallest one. In this case, we have $\mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n', p_i) + |p_{x+1}| - |p_i|$. As $\mathsf{ed}(n', p_x) = \mathsf{ed}(n', p_i) + |p_x| - |p_i| \le \tau - 1$, n' is an active node of p_x. By definition, n' is a pivotal active node of p_x. Based on the induction assumption, $\langle n', \mathsf{ed}(n', p_x) = \mathsf{ed}(n', p_i) + (|p_x| - |p_i|), p_i, \mathsf{ed}(n', p_i)\rangle$ must be in Ψ_{p_x}. As n' is a pivotal active node of p_x, the distance $\mathsf{ed}(n', p_x)$ is the minimum transformation distance, thus this quadruple cannot be deleted in the ICPAN algorithm. As p_i is the shortest one that matches n', the quadruple $\langle n', \mathsf{ed}(n', p_x), p_i, \mathsf{ed}(n', p_i)\rangle$ must be in Ψ_{p_x}. From node n', as $\mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n', p_x) + 1 = \mathsf{ed}(n', p_i) + |p_{x+1}| - |p_i| \le \tau$, the ICPAN algorithm considers the deletion case, and adds $\langle n', \mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n', p_i) + |p_{x+1}| - |p_i|, p_i, \mathsf{ed}(n', p_i)\rangle$ into $\Psi_{p_{x+1}}$. As n' is a pivotal active node of p_{x+1}, $\mathsf{ed}(n', p_{x+1})$ is the minimum transformation distance, thus this quadruple cannot be deleted in the ICPAN algorithm. As p_i is the shortest one that matches n', the quadruple $\langle n', \mathsf{ed}(n', p_{x+1}), p_i, \mathsf{ed}(n', p_i)\rangle$ must be in $\Psi_{p_{x+1}}$. □

Lemma 6 (Soundness) *For each quadruple* $\left\langle n, \xi_n^p, p', \xi_n^{p'}\right\rangle$ *in* Ψ_p, *the node* n *is a pivotal active node of* p, $\xi_n^p = \mathsf{ed}(n, p)$, $\xi_n^{p'} = \mathsf{ed}(n, p')$, *and* $\mathsf{ed}(n, p) = \mathsf{ed}(n, p') + |p| - |p'|$, *where* p' *is a prefix of* p.

Proof By definition, a transformation distance of two strings in each added quadruple by the ICPAN algorithm is no less than their edit distance. That is, $\mathsf{ed}(n, p) \le \xi_n^p \le \tau$ and $\mathsf{ed}(n, p') \le \xi_n^{p'} \le \tau$. Therefore, n must be an active node of p.

Next we prove that n must be a pivotal active node of p. We prove it by induction. This claim is obviously true when $p = p_0 = \epsilon$. Suppose the claim is true for $p_0, p_1, \ldots, p_x$, and we want to prove this claim is also true for p_{x+1}. Suppose n is a pivotal active node of p_x. Based on the completeness claim in Lemma 5, $\langle n, \mathsf{ed}(n, p_x), p_i, \mathsf{ed}(n, p_i)\rangle$ is in Ψ_{p_x}, where p_i is a prefix of p and $\mathsf{ed}(n, p_x) = \mathsf{ed}(n, p_i) + |p_x| - |p_i|$. The ICPAN algorithm considers the following two cases to add nodes.

Considering node n. In this case, the ICPAN algorithm considers the deletion case and adds. $\langle n, \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|, p_i, \mathsf{ed}(n, p_i)\rangle$ into $\Psi_{p_{x+1}}$. For each $\langle n, \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|, p_i, \mathsf{ed}(n, p_i)\rangle \in \Psi_{p_{x+1}}$ after removing non-pivotal active nodes, we first prove that $\mathsf{ed}(n, p_{x+1}) = \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$. As $\mathsf{ed}(n, p_{x+1}) \le \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$, we only need to prove $\mathsf{ed}(n, p_{x+1}) \ge \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$. We prove it by contradiction.

Suppose $\mathsf{ed}(n, p_{x+1}) < \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$. Without loss of generality, in the transformation ρ from n to p_{x+1} with $\mathsf{ed}(n, p_{x+1})$ operations, let node n_a denote the nearest ancestor node of n, such that the character of n_a matches a character c_j in p_{x+1}. We have $\mathsf{ed}(n, p_{x+1}) = \mathsf{ed}(n_a, p_j) + \max(|n| - |n_a|, |p_{x+1}| - |p_j|)$. Based on the proof in Lemma 5, n_a is a pivotal active node of p_j. Based on the completeness claim, $\langle n_a, \mathsf{ed}(n_a, p_j), p_j, \mathsf{ed}(n_a, p_j)\rangle$ is in Ψ_{p_j}. The ICPAN algorithm considers the deletion case and adds $\langle n_a, \mathsf{ed}(n_a, p_{x+1}) = \mathsf{ed}(n_a, p_j) + |p_{x+1}| - |p_j|, p_j, \mathsf{ed}(n_a, p_j)\rangle$ into $\Psi_{p_{x+1}}$. As $\mathsf{ed}(n_a, p_j) + \max(|n| - |n_a|, |p_{x+1}| - |p_j|) = \mathsf{ed}(n, p_{x+1}) < \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$, the ICPAN algorithm will remove the quadruple $\langle n, \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|, p_i, \mathsf{ed}(n, p_i)\rangle$, which contradicts to the fact that $\langle n, \mathsf{ed}(n, p_i)+|p_{x+1}|-|p_i|, p_i, \mathsf{ed}(n, p_i)\rangle$ is in $\Psi_{p_{x+1}}$. Thus, $\mathsf{ed}(n, p_{x+1}) = \mathsf{ed}(n, p_i)+|p_{x+1}|-|p_i|$.

As the character of n matches c_i, there exists a transformation from n to p_{x+1} with $\mathsf{ed}(n, p_{x+1})$ operations, and the last character of node n matches a character in p_{x+1}. Thus, node n is a pivotal active node of p_{x+1}. For each node, as the ICPAN algorithm only keeps the minimum transformation distance, there is only one quadruple $\langle n, \mathsf{ed}(n, p_{x+1}), p_i, \mathsf{ed}(n, p_i)\rangle$ for node n in $\Psi_{p_{x+1}}$. Based on the completeness claim, we have $\xi_n^{p_{x+1}} = \mathsf{ed}(n, p_{x+1}), \xi_n^{p_i} = \mathsf{ed}(n, p_i)$, and $\mathsf{ed}(n, p_{x+1}) = \mathsf{ed}(n, p_i) + |p_{x+1}| - |p_i|$.

Considering descendants of node n: In this case, the ICPAN algorithm considers the match case. For a node n' with character c_{x+1}, the ICPAN algorithm adds $\left\langle n', \xi_n^{p_i} + \max(|n'|-|n|-1, |p_x|-|p_i|), p_{x+1}, \xi_n^{p_i} + \max(|n'|-|n|-1, |p_x|-|p_i|)\right\rangle$ into $\Psi_{p_{x+1}}$. As $\xi_{n'}^{p_{x+1}} = \xi_n^{p_i} + \max(|n'|-|n|-1, |p_x| - |p_i|) \leq \tau$, node n' must be an active node of p_{x+1}. As the character of n' matches c_{x+1}, node n' must be a pivotal active node based on Lemma 4. Based on the completeness claim, n' must be in $\Psi_{p_{x+1}}$. For each node, as the ICPAN algorithm only keeps the minimum transformation distance, there is only one quadruple $\left\langle n', \mathsf{ed}(n', p_{x+1}), p_{x+1}, \mathsf{ed}(n', p_{x+1})\right\rangle$ for node n' in $\Psi_{p_{x+1}}$ Based on the completeness claim, $\xi_{n'}^{p_{x+1}} = \mathsf{ed}(n', p_{x+1})$,and $\mathsf{ed}(n', p_{x+1}) = \mathsf{ed}(n', p_{x+1}) + \max(|n'| - |n'|, |p_{x+1}| - |p_{x+1}|)$. □

Theorem 2 *Given a query keyword p, the* ICPAN *algorithm computes all the pivotal active nodes of p with their edit distances.*

Proof It is a corollary of the two lemmas above. □

In general, the user may modify the previous query string by deleting multiple characters at the end or changing some characters in the middle of the string. The user may also copy and paste a completely different string to the search interface. In this case, we can first identify the cached keyword that has the longest prefix of the new query. We then use the cached pivotal active nodes to incrementally answer the new query by inserting the characters after the longest prefix one by one.

5 Type-ahead search using multiple keywords

In this section, we study how to support type-ahead search for a query with multiple keywords (tokenized from the query string by white space). Given a query consisting of a set of keywords $Q = \{p_1, p_2, \ldots, p_\ell\}$, the query answer of type-ahead search is a set of records r in R such that for each query keyword p_i, record r contains a word with p_i as a prefix. The query answer of fuzzy type-ahead search is a set of records r in R such that for each query keyword p_i, record r contains a word with a prefix *similar to* p_i. There are scenarios where we want to use the semantics of multi-keyword completions. For example, a user wants to search for a person, but only vaguely remembers the first few letters of the name and the first few letters of the telephone number. In this case, the user needs multi-keyword completions. Our solutions also work for the case where only the last keyword in a query is treated as a prefix condition.

Our goal is to efficiently and incrementally compute the relevant records. Given a query, each query keyword (treated as a prefix) has multiple similar words as shown in Fig. 7. To find the answers, a straightforward method first computes the *union list* of each keyword, which is the union of inverted lists of this keyword's similar words. Then, it intersects the union lists of every keyword and generates the final answers. These operations can be computationally costly, especially when each query keyword can have multiple similar prefixes. In Sect. 5.1, we study various algorithms for computing the answers efficiently. Note that in most cases, the user types the query letter by letter, and subsequent queries append additional letters to previous ones. Based on this observation, we study how to use the cached results of earlier queries to answer a query incrementally (Sect. 5.2).

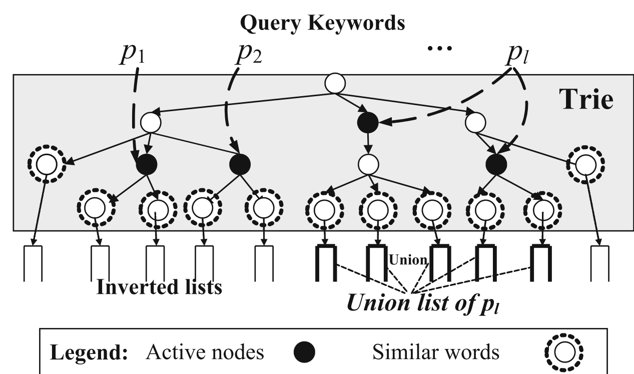

Fig. 7 Notations used in the paper

5.1 Intersecting union lists

For simplicity, we first consider exact search and then extend the results to fuzzy search. Given a query $Q = \{p_1, p_2, \ldots, p_\ell\}$, suppose $K_i = \{k_{i_1}, k_{i_2}, \ldots\}$ is the set of words that share the prefix p_i. Let $L_{i_j} = L(k_{i_j})$ denotes the inverted list of k_{i_j}, which is sorted-based record IDs, and $\mathcal{U}_i = \bigcup_{k_{i_j} \in K_i} L(k_{i_j})$ be the union of the lists for p_i. We study how to compute the answer to the query, i.e., $\bigcap_{1 \le i \le \ell} \mathcal{U}_i$.

Simple methods: One method is the following. For each keyword p_i, we compute the corresponding union list $\mathcal{U}_i$ on-the-fly and intersect the union lists of different keywords. The time complexity for computing the unions is $O(\sum_{i,j} |L_{i_j}|)$, where $|L_{i_j}|$ is the size of L_{i_j}. The shorter the keyword is, the lower the performance could be, as inverted lists of more similar words need to be traversed to generate the union list. This approach requires the inverted lists of trie leaf nodes. The space complexity of the inverted lists is $O(|R| \times r_{avg})$, where $|R|$ is the number of records and r_{avg} is the average number of distinct words of each record.

Alternatively, we can pre-compute and store the union list of each keyword, and intersect the union lists of query keywords when a query comes. The main issue of this approach is that the pre-computed union lists require a large amount of space, especially since each record occurrence on an inverted list needs to be stored many times. The space complexity of all the union lists is $O(|R| \times r_{avg} \times k_{avg})$, where k_{avg} is the average word length (i.e., the average number of letters of each word).

There have been other approaches for answering keyword intersection. For instance, Bast et al. [6] proposed a method that groups ranges of keywords and builds document lists separately for each range. Intersection is performed between an existing document list and several ranges called "HYB blocks." The limitation of this approach is that, for most queries, the ranges can include many irrelevant documents, which require a lot of time to do a post-processing. We will show experimental results in Sect. 7.

Efficient intersection using forward lists: We develop a new solution based on the following ideas. Among the union lists $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_\ell$, we identify the shortest union list. Each record ID on the shortest list is verified by checking if it appears on all the other union lists (following the ascending order of their lengths). Notice that these union lists are not materialized in the computation. We can enumerate the record IDs on the shortest union list by accessing the leaf nodes of the corresponding keyword and visiting the record IDs in their inverted lists. The length of each union list can be pre-computed and stored in the trie or estimated on-the-fly. To verify record occurrences efficiently, we maintain a forward list for each record r, which is a sorted list of IDs of words in r, denoted as F_r. A unique property of the word IDs is that they are encoded using the alphabetical order of the words. Therefore, each keyword p_i has a range of word IDs $[MinId_i, MaxId_i]$. Moreover, if p_i is a prefix of word w, then the ID of w must be within this range.

An interesting observation is that, for a record r on the shortest union list, the problem of verifying whether r appears on (non-materialized) union list $\mathcal{U}_k$ of a query keyword p_k is equivalent to testing if p_k appears on the forward list F_r as a prefix. We can do a binary search for $MinId_k$ on the forward list F_r to get a lower bound Id_{lb} and check if Id_{lb} is no larger than $MaxId_k$. The probing succeeds if the condition holds, and fails otherwise. The time complexity for processing each single record r is $O\left((\ell - 1) * \log(r_{avg})\right)$, where ℓ is the number of keywords in the query, and r_{avg} is the average number of distinct words in each record. A good property of this approach is that the time complexity of each probing does not depend on the lengths of inverted lists, but on the number of unique words in a record.

Figure 8 shows an example when a user types in a query {"vldb", "l"}. The words for "l" are "li", "lin", "liu", "lu", and "luis". The keyword-ID range of each query keyword is shown in brackets. For instance, the keyword-ID range for "l" is [3, 7] (Fig. 3), which covers the ranges of "li", "lin", "liu", "lu", and "luis". To intersect the union list of "vldb" with that of "l", we first identify "vldb" as the one with the shorter union list. The record IDs (6, 7, 8) on the list are probed one by one. Take record 7 as an example. Its forward list contains word IDs {2, 7, 8, ...}. We use the range of "l" to probe the forward list. By doing a binary search for the word ID 3, we find word with ID 7 on the forward list, which is then verified to be no larger than MaxID = 7. Thus, record 7 is an answer to the query, and the word with ID 7 ("luis") has "l" as a prefix.

Extension to fuzzy search: The algorithm described earlier naturally extends to the case of fuzzy search. Since each query keyword has multiple active nodes of similar prefixes, instead of considering the union of the leaf nodes of one pre-

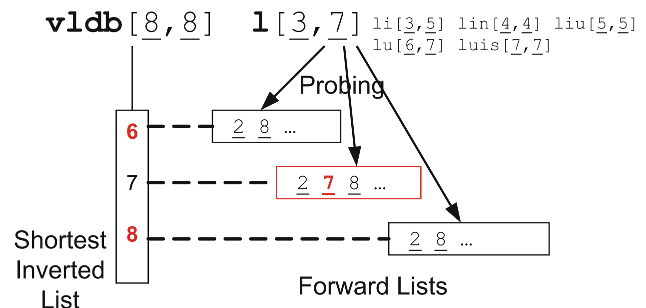

Fig. 8 Intersection using forward lists. (*Numbers with underlines* are word IDs, and *numbers without underlines* are record IDs.)

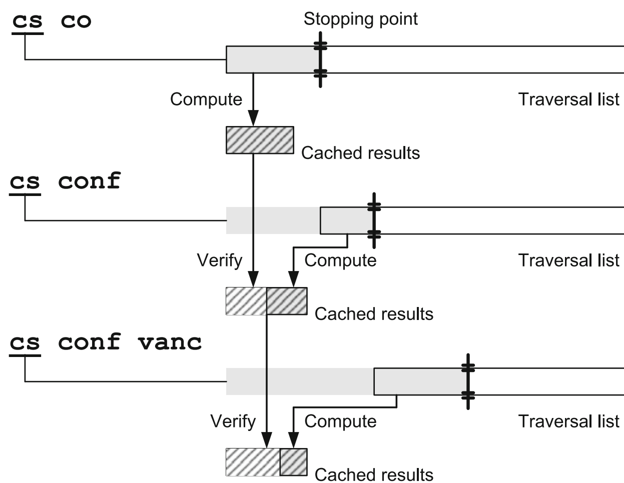

Fig. 9 Computing k results using cached answers and resuming unfinished traversal on a list

fix node, now we need to consider the unions of the leaf nodes for all active nodes of a keyword. For each record r on the shortest union list, we use r's forward list to test whether the record contains a word having a prefix of an active node (or a pivotal active node) for every other query keyword. Note that the lengths of these union lists can be estimated in order to find the shortest one as follows. Given a trie node, we can store the number of distinct records with words having a prefix of the trie node, i.e., the length of the union list of the trie node. Given a query keyword, for fuzzy search, we can take the sum of lengths of union lists of its active nodes as an estimation of its union-list length.

5.2 Cache-based intersection

In Sects. 3 and 4, we presented algorithms for incrementally computing similar prefixes for a query keyword as the user types the keyword letter by letter. Now, we show that intersection can also be performed incrementally using previously cached results. Here, we use an example to illustrate how to cache query results and use them to answer subsequent queries. Suppose a user types in a keyword query Q_1 = {"`cs`", "`co`"}. All the records in the answers to Q_1 are computed and cached. For a new query Q_2 = {"`cs, conf`"} that appends two letters to the end of Q_1, we can use the cached results of Q_1 to answer Q_2, because the second keyword "`conf`" in Q_2 is more restrictive than the corresponding keyword "`co`" in Q_1. Each record in the cached results of Q_1 is verified to check whether "`conf`" can appear in the record as a prefix. In this way, Q_2 does not need to be answered from scratch, and this observation was also made in [6]. As in this example, in the following discussion, we use "Q_1" to refer to a query whose results have been cached and "Q_2" to refer to a new query whose results we want to compute using those of Q_1.

Cache miss: Often the more keywords the user types in, the more typos and mismatches the query could have. Thus, we dynamically increase the edit-distance threshold τ as the query string is getting longer. Then, it is possible that the threshold for the new query Q_2 is strictly larger than that of the original query Q_1. In this case, the active nodes of keywords in Q_1 might not include all those of keywords in Q_2. Thus, we cannot use the cached results of Q_1 (active nodes and answers) to compute those of Q_2. This case is a cache miss, and we compute the answers of Q_2 from scratch.

Reducing cached results: The cached results of query Q_1 could be large, which could require a large amount of time to compute and space to store. There are several cases where we can reduce the size. The first case is when we want to use pagination, i.e., we show the results in different pages. In this case, we can traverse the shortest list partially, until we have enough results for the first page. As the user browses the results by clicking "Previous" and "Next" links, we can continue traversing the shortest list to compute more results and cache them. The second case is when the user is only interested in k records. We can compute the answers to the query Q_1 without traversing the entire shortest list. When using k results of Q_1 to compute k results of Q_2, it is possible that the cached results do not provide enough answers, since Q_2 contains a more restrictive keyword. In this case, we can continue the unfinished traversal on the shortest list, assuming that we have remembered the place where the traversal stopped on the shortest list when answering query Q_1, and stop the traversal when we get k records.

Figure 9 shows an example of incrementally computing k answers using cached results. A user types in a query string "`cs conf vanc`" letter by letter, and the server receives queries {"`cs`", "`co`"}, {"`cs`", "`conf`"}, and {"`cs`", "`conf`", "`vanc`"} in order. (Notice that the client does not need to send a query to the server for each keystroke of the user.) The answer to the first query {"`cs`", "`co`"} is computed. Assuming the union list of keyword "`cs`" is the shorter one. The traversal stops at the first vertical bar. Each record accessed in the traversal is verified by probing the word range of {"`cs`", "`co`"} using the forward list of the record. Records that pass the verification are cached. When we want to answer the query {"`cs`", "`conf`"} incrementally, we first verify each record in the cached result of the previous query by probing the word range of "`conf`". Some of these results will become results of the new query. If the results from the cache are not enough to compute k results of the new query, we resume the traversal on the list of "`cs`", starting from the stopping position for answering the previous query, until we have enough k results for the new query. The next query {"`cs`", "`conf`", "`vanc`"} is answered similarly.

In the case of cache miss, i.e., earlier cached results cannot be used to compute the answers of a new query, we need to answer the new query from scratch. We may choose a different list as the shortest one, and subsequent queries can be answered similarly.

6 Improving search by supporting additional features

In this section, we discuss various features to improve the results and user interface in type-ahead search.

6.1 Highlighting best prefixes

When displaying records to the user, we want to highlight the most similar prefixes for an input prefix. This highlighting feature is straightforward to implement for the exact-match case. For fuzzy search, a query keyword could be similar to several prefixes of the same similar word. Thus, there could be multiple ways to highlight the similar word. For example, suppose a user types in "`lus`," and there is a similar word "`luis`." Both prefixes "`lui`" and "`luis`" are similar to "`lus`." There are several ways to highlight them, such as "**lui**s" or "**luis**," where underlined characters are highlighted. To address this issue, we use the concept of *normalized edit distance*. Formally, given two prefixes p_i and p_j, their normalized edit distance is as follows:

$$\mathsf{ned}(p_i, p_j) = \frac{\mathsf{ed}(p_i, p_j)}{\max(|p_i|, |p_j|)}, \tag{1}$$

where $|p_i|$ denotes the length of p_i. Given a query keyword and one of its similar word, the prefix of the similar word with the minimum ned to the query is highlighted. We call such a prefix a *best matched prefix* and call the corresponding normalized edit distance the "minimal normalized edit distance," denoted as "mned." This prefix is considered to be most similar to the query keyword. For example, for the keyword "`lus`" and its similar word "`luis`," we have ned("`lus`", "`l`") = $\frac{2}{3}$, ned("`lus`", "`lu`") = $\frac{1}{3}$, ned("`lus`", "`lui`") = $\frac{1}{3}$, and ned("`lus`", "`luis`") = $\frac{1}{4}$. Since mned("`lus`", "`luis`") = ned("`lus`", "`luis`"), "**luis**" will be highlighted.

6.2 Using synonyms

We can utilize a priori knowledge about synonyms to find relevant records. For example, "`William = Bill`" is a common synonym in the domain of person names. Suppose in the underlying data, there is a person called "`Bill Gates`." If a user types in a query string "`William Gates`," we can also find this person. One way to support this feature is the following. On the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`" and vice versa. When a user types in a keyword "`Bill`", in addition to retrieving the records for "`Bill`", we also identify those of "`William`" following the link.

6.3 Supporting updates

We discuss how to deal with data updates, specifically insertions, deletions, and updates of records.

Insertion: Assume a record is inserted. We first assign it a new record ID. For each word in the record, we insert the word into the trie as follows. For each prefix of the word, if the prefix is not in the trie, we add a trie node for the prefix. For the leaf node corresponding to the word, we append the record ID on the inverted list of this leaf node. In addition, if we use forward indexes, we create a forward list for the record. For the word-range encoding of the trie structure, we can reserve extra space for word ids to accommodate future insertions [45]. We only need to do global reordering when all the reserved spaces are consumed.

Deletion: Assume a record is deleted. For each word in the record, on the inverted list of the word, we use a bit to denote whether a record is deleted. Here, we use the bit to mark the record to be deleted. We do not update the trie structure until we need to rebuild the index. For the range encoding of the trie, we can use the deleted word ids for future insertions.

Update: Assume a record is updated. We first delete (using a bit to mark the record to be deleted) and insert a new record based on the above methods.

7 Experiments

We deployed several prototypes in different domains to support type-ahead search. In addition, we conducted a thorough experimental evaluation of the developed techniques on real data sets, such as publications and people directories. Here, we report the results on the following two data sets. (1) *DBLP*: It included about 1.1 million computer science publication records, with six attributes: authors, title, conference or journal name, year, page numbers, and URL. (2) *MEDLINE*: It had about 4 million latest publication records related to life sciences and biomedical information. We used five attributes: authors, their affiliations, article title, journal name, and journal issue. Table 4 shows the data sizes, index sizes, and index-construction times.

For each data set, we set up a Web server using Apache2 on a Linux machine with an Intel Core 2 Quad processor Q6600 (2.40 GHz) and 8 GB memory. We implemented the

Table 4 Data sets and index costs

Data set	DBLP	MEDLINE
Record number	1.1 million	4 million
Dataset size	190 MB	1.25 GB
# of distinct words	392 K	1.79 million
# of words	21.9 million	136.7 million
Avg. length of records	20.1	34.34
Index-construction time	50 s	588 s
Trie size	36 MB	165 MB
Inverted-list size	52 MB	445 MB
Forward-list size	54 MB	454 MB

backend as a FastCGI server process, which was written in C++ compiled with a GNU compiler.

7.1 Efficiency of single-keyword queries

7.1.1 Exact search

We first evaluated the efficiency of exact search. For the DBLP data set, we randomly selected 1,000 real queries from the logs of our deployed systems. For the MEDLINE data set, we generated 1,000 single-keyword queries by randomly selecting keywords in the data set. We implemented two methods to find the trie node for a query keyword using methods discussed in Sect. 3. (1) Incremental: We incrementally found the trie node. (2) Non-Incremental: We found the trie node from scratch. For each query, for each of its prefixes, we computed its running time and evaluated the average running time for prefixes with the same length. Figure 10 shows the results. As the prefix length increased, the running time of the Incremental method decreased, while that of the Non-Incremental method increased. This is because the Incremental method can use previously computed results.

7.1.2 Fuzzy search

We evaluated the efficiency of computing the prefixes on the trie that are similar to a query keyword. For the DBLP data set, we selected the same 1,000 real queries from the logs of our deployed systems. For the MEDLINE data set, we generated 1,000 single-keyword queries by randomly selecting keywords in the data set and applying a random number of edit changes (0–2) on the keyword. The average length (number of letters) of keywords was 9.9 for the DBLP data set and 10.2 for the MEDLINE data set. For each prefix of a query, we measured the time to find similar prefixes within an edit distance of 2, not including the time to retrieve records. We computed the average time for the prefix queries with the same length.

Computing active nodes: We implemented three methods to compute similar prefixes. (1) Incremental: We computed the active nodes of a query using the cached active nodes of previous prefix queries, using the incremental algorithm presented in Sect. 4. This algorithm is applicable when the user types a query letter by letter. (2) Non-Incremental: We computed active nodes from scratch. This case happens when a user copies and pastes a long query, and none of the active nodes of any prefix queries has been computed. It also corresponds to the traditional search case, where a user submits a query and clicks the "Search" button. (3) Gram-Based: We built gram inverted lists on all prefixes with at least three letters using the method described in [36]. We used the implementation in the Flamingo release,[4] using a gram length of 3 and a length filter. The total number of such prefixes was 1.1 millions for the DBLP data set and 4 millions for the MEDLINE data set. The index structure can be used to compute similar prefixes for keywords with at least four letters. Figure 11 shows the performance results of these three methods.

The method Incremental was most efficient. As the user types in letters, its response time first increased slightly (less than 5 ms) and then started decreasing quickly after the fourth letter. The main reason is that the number of active nodes decreased, and the cached results made the computation efficient. The method Non-Incremental required longer time since each query needed to be answered from scratch, without using any cached active nodes. The method Gram-Based performed efficiently when the query keyword had at least seven letters. But it had a very poor performance for shorter keywords, since the count filter had a weak power to prune false positives.

In addition, we evaluated the memory used for the three methods. Table 5 gives the results. We see that the Gram-Based method involved large index size as it needs to store large numbers of grams and the corresponding gram inverted lists. The Incremental method involved a bit larger memory space than that of the Non-Incremental method, as it needs to cache active nodes. The Gram-Based method also consumed memory space for maintaining heap structures and larger numbers of candidates that need to be verified. In summary, the Gram-Based method involved the largest memory, and the Incremental method used a bit larger memory space than the Non-Incremental method.

Improving performance using pivotal active nodes: We evaluated the ICPAN algorithm for computing pivotal active nodes. We compared the number of active nodes and that of pivotal active nodes. Figure 12 shows the results. The number of pivotal active nodes was much smaller than that of active nodes. For example, on the MEDLINE data set, when the prefix length was 3, the number of active nodes was 140,000,

[4] http://flamingo.ics.uci.edu/releases/2.0.

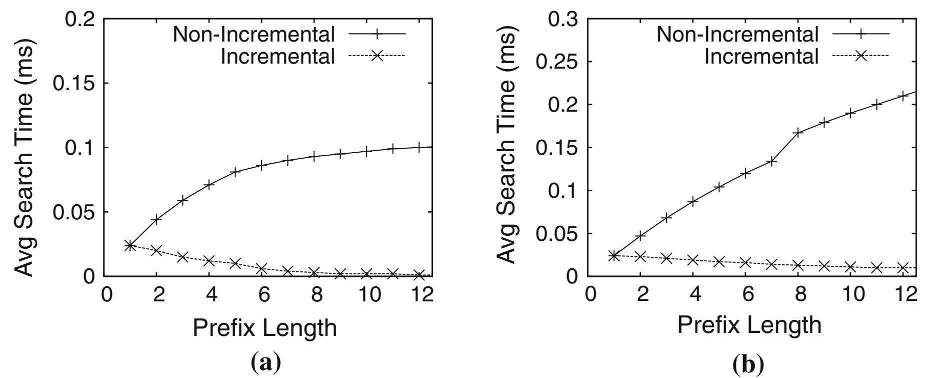

Fig. 10 Efficiency of exact search. **a** DBLP, **b** MEDLINE

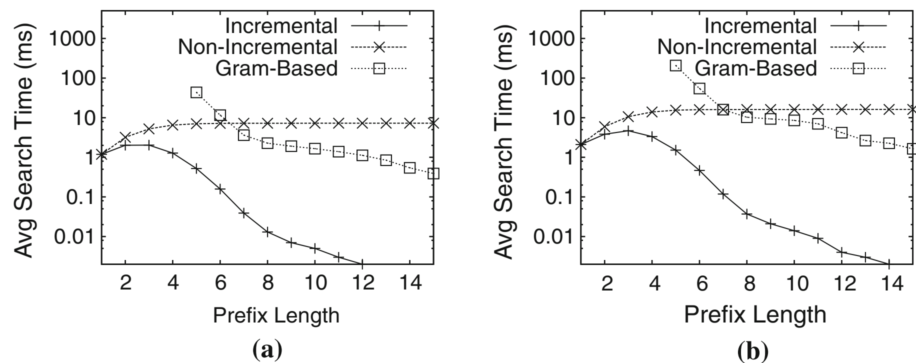

Fig. 11 Computing prefixes similar to a keyword ($\tau = 2$). **a** DBLP, **b** MEDLINE

Table 5 Memory size for computing prefixes similar to a keyword ($\tau = 2$)

	Memory for storing indexes	Memory for computing similar words	Total memory usage
(a) DBLP dataset			
Incremental	36	4	40
Non-Incremental	36	0.2	36.2
Gram-Based	187	5	192
(b) MEDLINE dataset			
Incremental	165	12	177
Non-Incremental	165	0.5	165.5
Gram-Based	713	21	733

All numbers are in MBs

while the number of pivotal active nodes was only 20,000. Thus, using pivotal active nodes can reduce the storage space, especially for short query strings.

We then evaluated the efficiency of the ICPAN algorithm for computing pivotal active nodes as described in Sect. 4.3. We also compared it with the ICAN algorithm in Sect. 4.2 and the similar method in [15]. (We did not used the pre-computation techniques and took examining the impact of pre-computation as a future work.) Figure 13 shows the results. As we used edit-distance threshold 3, the performance was lower than that on Fig. 11. We can see that the ICPAN algorithm achieved the best performance and was 2–4 times faster than the other two methods. For example, on the MEDLINE data set, when the prefix length was 3, the running time of ICPAN was 40 ms, while the other two methods needed more than 80 ms. Thus, the pivotal active node-based method ICPAN not only reduced the space, but also improved the efficiency, especially for short queries.

7.1.3 Performance of single-keyword queries

We evaluated the performance of single-keyword queries by varying the edit-distance threshold τ. We implemented our best algorithms and computed the answers in two steps: (1) computing similar prefixes and (2) computing answers based on similar prefixes. Figure 14 shows the results. Our methods could answer a query within 50 ms.

7.2 Efficiency of multi-keyword queries

In this section, we evaluated the performance of answering keyword queries with multiple keywords.

List intersection: We evaluated several methods for intersecting union lists of multiple keywords, as described in Sect. 5.1. For the DBLP data set, we selected the same 1,000 real queries from the logs of our deployed systems. For the MEDLINE data set, we generated 1,000 queries by randomly

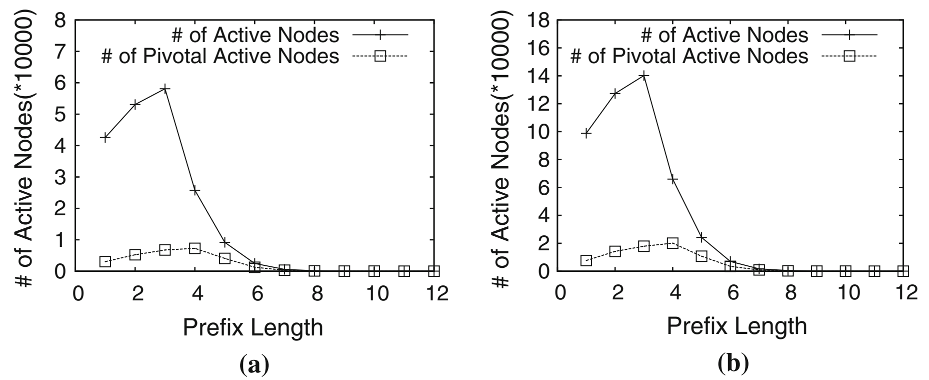

Fig. 12 Number of active nodes for a keyword ($\tau = 3$). **a** DBLP, **b** MEDLINE

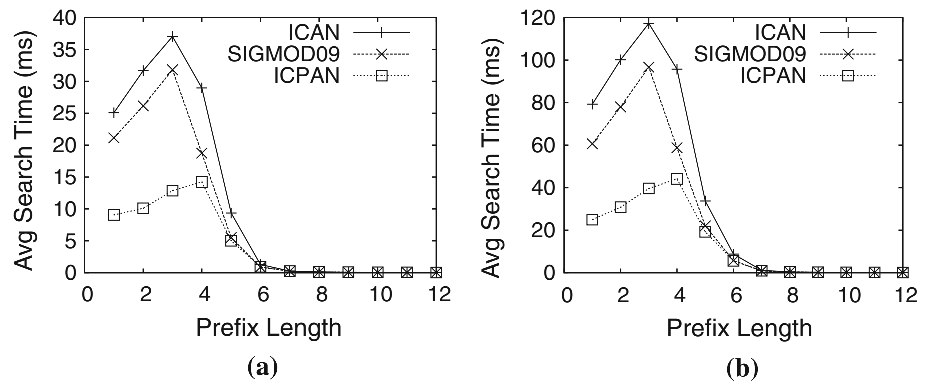

Fig. 13 Time for computing active nodes for a keyword ($\tau = 3$). **a** DBLP, **b** MEDLINE

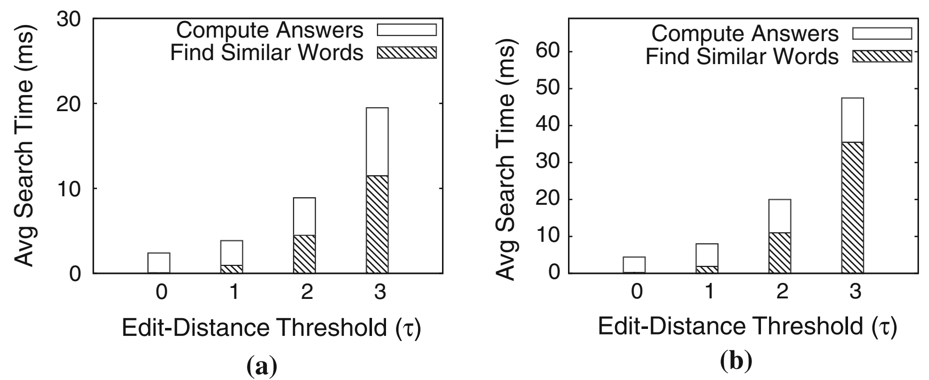

Fig. 14 Efficiency of single-keyword queries (varying τ). **a** DBLP, **b** MEDLINE

selecting records from the data set and choosing keywords from each record. We implemented the following methods to intersect the union lists of keywords. (1) ForwardLists: It traverses the shortest union list and uses the other query keywords to probe the forward lists of records on the shortest list as discussed in Sect. 5.1. The union list was "generated" on the fly without being materialized. (2) HashProbe: The shortest union list was materialized as a hash table at query time. Each record ID on the other union lists was searched on the hash table. (3) MaterializedUnions: We materialized the union lists of all the query keywords and their prefixes, and computed an intersection by using the record IDs of the shortest list to probe the other union lists. We measured the intersection time only. (4) MergeJoin: It used the merge-sort algorithm to on-the-fly generate the union list (by getting the next record of each union list of query keywords) and computed an intersection using the merge-join algorithm on top of the generated records. (5) HYB: We implemented a structure called "HYB" as described in [6]. We used an in-memory implementation, and all IDs were stored without any encoding and compression. The number of HYB blocks was 47 for the DBLP data set and 285 for the MEDLINE data set, using the parameters recommended in [6].

We evaluated these methods on queries with two keywords, assuming no previous query results were cached. Figure 15 shows the average time of each method as the length of the second keyword increased.

The intersection operation was very time consuming when the second keyword had no more than two letters, since the union lists of the prefixes were too long. The

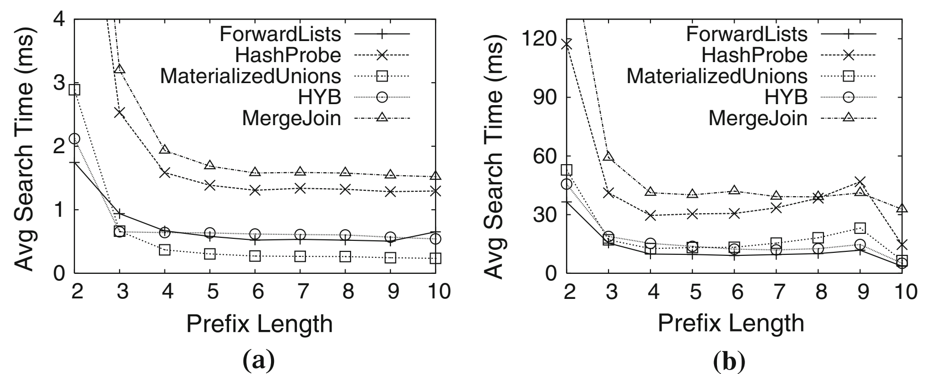

Fig. 15 List intersection of multiple keywords. **a** DBLP, **b** MEDLINE

HashProbe method performed relatively poorly due to the cost of building the hash table for the shorter list and traversing the other list. The MaterializedUnions method performed well, but with a high memory cost as discussed in Sect. 5.1. The MergeJoin algorithm performed worse than the MaterializedUnions method and the HashProbe method, as it needs to generate the union list on-the-fly. The ForwardLists algorithm achieved an excellent performance, at the cost of storing the forward lists. The HYB method also achieved high performance, and our method and the HYB method provide different ways to improve the performance of type-ahead search. An interesting finding in the results is that ForwardLists even outperformed MaterializedUnions on the MEDLINE data set. This is because as the data set became larger, the average time of each binary search on the union lists increased, while that of each binary probe on the forward lists did not change much.

Cache-based intersection: We evaluated the performance of different methods of cache-based prefix intersection, as described in Sect. 5.2. We allowed at most one typo for each prefix with at most five letters and two errors for prefixes with more than five letters. As a consequence, for a query with two keywords, when the sixth letter of the second keyword was typed in, a cache miss occurred. We implemented the following methods. (1) NoCache: No query results are cached. A query is computed without using any cached query results. (2) Complete Traversal: It traverses the shortest union list completely to compute the results of the current query. (3) Partial Traversal: It traverses the shortest union list partially until it finds the first k results for the current query as follows. For each record on the shortest union list, it first uses the active nodes with smaller edit distances to probe the forward list of the record. The algorithm terminates if it gets k records. Figure 16 shows the query time of the methods on the DBLP data set. Complete Traversal outperformed the No Cache method for relatively long prefixes (with more than six letters) mainly due to the smaller set of cached results. The Partial Traversal method was the most efficient one in most cases, since it can stop early during the traversal of the list, and a new query can be incrementally computed using earlier results. All these methods required a relative long time when the prefix had six letters due to the cache miss.

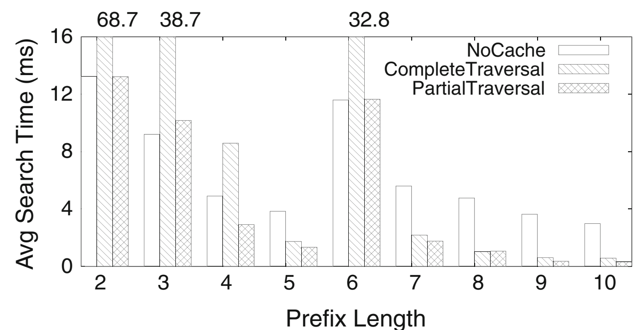

Fig. 16 Performance of prefix intersection (DBLP)

7.3 Evaluation on per-query time

We evaluated the per-query time of our algorithms on the DBLP data set. We randomly selected 1,000 queries with 1 keyword and 1,000 queries with 2–6 keywords. Each keyword has at leat 2 characters. We used the forward list-based method for multi-keyword intersection and ICPAN algorithm for incrementally computing similar words. Table 6 shows the results. We see that for single-keyword queries, our method can answer a query about 2 ms in terms of exact search and 8 ms in terms of fuzzy search ($\tau = 1$). For multi-keyword queries, the elapsed time respectively increased to 4 and 19 ms as we needed to do multi-keyword intersection.

7.4 Space and time scalability

We evaluated the scalability of our algorithms. As an example, we used the MEDLINE data set. Figure 17a shows how the index size increased as the number of records increased. It shows that all the sizes of the trie structures, inverted lists, and forward lists increased linearly.

We measured the query performance of computing the first 10 answers (as described in Sect. 7.2) as the data size

Table 6 Evaluation on per-query time on the DBLP dataset

	Exact search		Fuzzy search ($\tau = 1$)	
	1 keyword	Multi-keyword	1 keyword	Multi-keyword
Find trie nodes (ms)	0.01	0.01	1.02	1.13
Find answers (ms)	1.84	3.65	7.25	18.34
Total (ms)	1.85	3.66	8.27	19.47

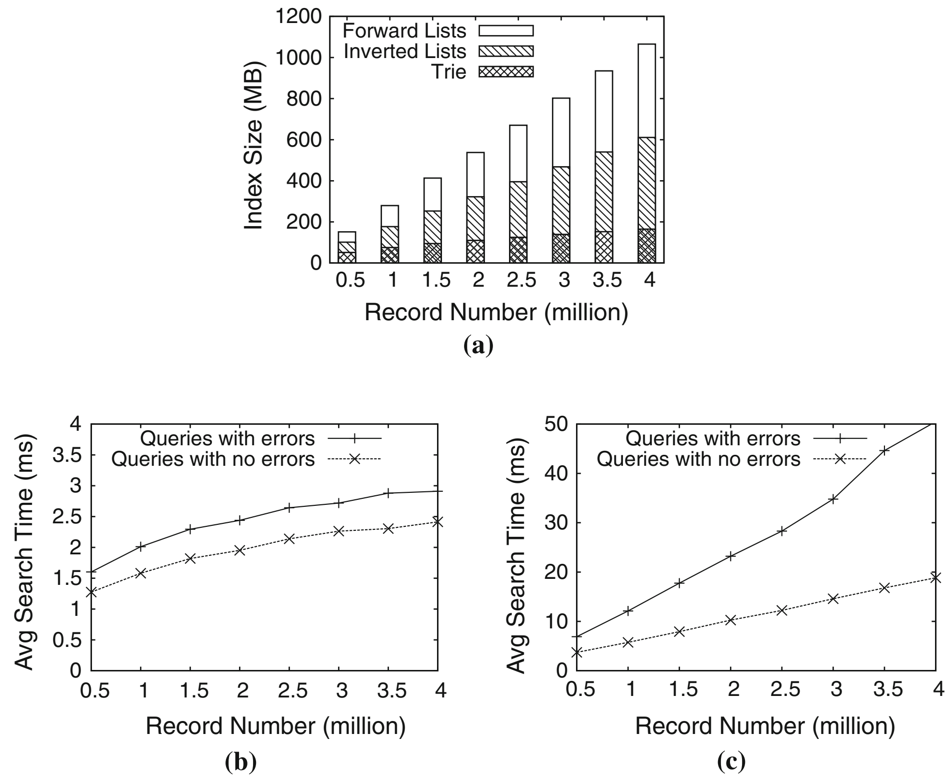

Fig. 17 Scalability (MEDLINE). **a** Index size, **b** single keyword (return 10 answers), **c** multiple keywords (return 10 answers)

increased. We first evaluated queries with a single keyword and tested the scalability of our incremental algorithm. We considered two types of queries: the first type was generated by randomly selecting keywords in the data set; the second type was obtained by modifying the first type by adding edit errors (0–2). Figure 17b shows the results for the MEDLINE data set as we increased the number of records. It shows that the algorithms can answer a single-keyword query very efficiently (within 3 ms), for both types of queries.

We next evaluated the algorithms for queries with multiple keywords and tested the scalability of our forward list-based algorithm. We measured the average query time of each keystroke on the last keyword. Figure 17c shows that our algorithms can process such queries very efficiently. For instance, when the data set had 4 million records, the average search time for queries without errors was 20 ms and the variance was 5 ms, while the average search time for queries with errors was 55 ms and the variance was 9 ms. The variance was similar on other data sets.

Table 7 Queries and saved typing effort

ID	Query	Saved typing effort (%)
Q_1	*sunta sarawgi*	42
Q_2	surajit *chuardhuri*	50
Q_3	*nick kodas approxmate*	41
Q_4	*flostsos* icde similarity	38
Q_5	*similarty* search icde	55
Q_6	*divsh srivstava* search	41

7.5 Round-trip time

The round-trip time of type-ahead search consists of three components: server processing time, network time, and JavaScript running time. Different locations in the world could have different network delays to our servers in southern California. We measured the time cost for a sample query

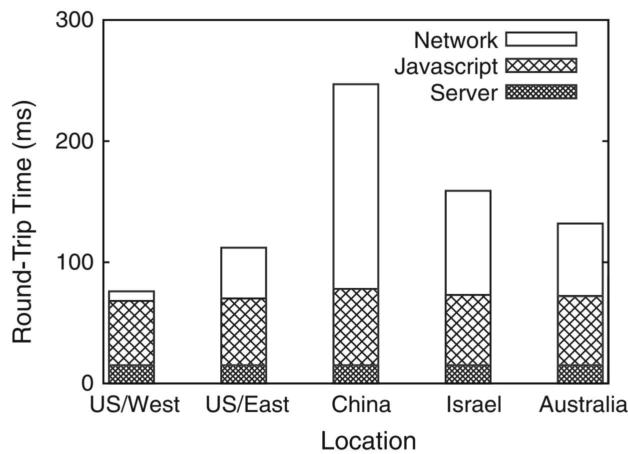

Fig. 18 Round-trip time for different locations (return 10 answers)

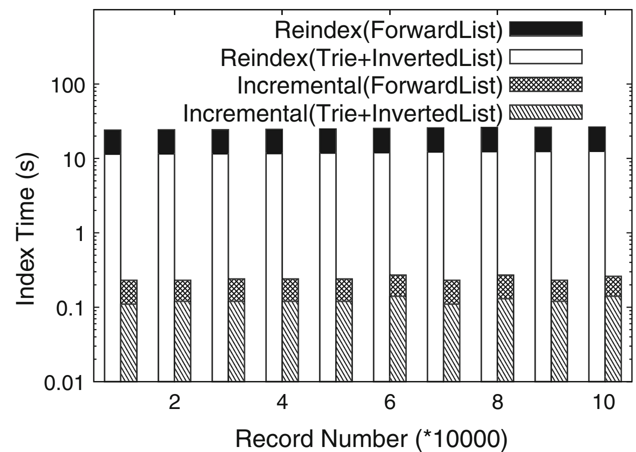

Fig. 19 Update

"`divsh srivstava search`" on the DBLP prototype from five locations around the world: US west coast, US east coast, China, Israel, and Australia. Figure 18 shows the results for finding first 10 answers. We can see that the server running time was less than 1/5 of the total round-trip time. JavaScript took around 40–60 ms. The relative low speed at some locations was mainly due to the network delay. For example, it took about 4/5 of the total round-trip time when our system was tested from China. For all the users from different locations, the total round-trip time for a query was always below 250 ms, and all of them experienced an interactive interface. For large-scale systems processing queries from different countries, we can solve the possible network-delay problem by using distributed servers.

7.6 Saved typing effort

Type-ahead search can also save users' typing effort, since results can be found before a user types in complete words. To evaluate the saving of typing effort, we constructed six queries on the DBLP data set as shown in Table 7. The keywords in italic font are mistyped keywords. Each query was typed in letter by letter, until the system found the expected records in first 10 answers. We measured how much letter-typing effort the system can save for the user. For each query Q_i, let $N(Q_i)$ be the number of letters the user typed before the relevant answers are found. We use $1 - \frac{N(Q_i)}{|Q_i|}$ to quantify the relative saved effort. For example, for query Q_6, the user could find relevant answers right after typing in "`divsh sri sea`". The saved effort of Q_6 is $1 - \frac{13}{22} = 41\%$, as the user only needed to type in 13 letters, instead of 22 letters in the full query. Table 7 shows that this paradigm can save the user 40–60% typing effort on average.

7.7 Data updates

We tested the cost of updates on the DBLP data set. We first built indexes for 1 million records and then inserted 10,000 records at each time. We compared the time of re-indexing the data from scratch and incrementally indexing using the data. Figure 19 shows the results. We see that the incremental method only took 0.1 s to update the index while it took more than 20 s to re-index the data. This result shows the advantage of the incremental-computation indexing method.

8 Other related work

Prediction and autocomplete: There have been many studies on predicting queries and user actions [18,33,49,50,59] in information search. With these techniques, a system predicts a word or a phrase the user may type in next based on the sequence of partial input the user has already typed. Many prediction and autocomplete systems[5] treat a query with multiple keywords as a single string, thus they do not allow these keywords to appear at different places in the answers. For instance, consider the search box on the home page of Apple.com, which allows autocomplete search on Apple products. Although a keyword query "`itunes`" can find a record "`itunes wi-fi music store`," a query with keywords "`itunes music`" cannot find this record (as of January 2010), simply because these two keywords appear at different places in the record. The techniques presented in this paper focus on "search on the fly," and they allow query keywords to appear at different places in the answers. As a consequence, we cannot answer a query by

[5] The word "autocomplete" could have different meanings. Here we use it to refer to the case where a query (possibly with multiple keywords) is treated as a *single* prefix.

simply traversing a trie index. Instead, the backend intersection (or "join") operation of multiple lists requires more efficient indexes and algorithms.

Approximate string search and similarity join: There have been recent studies to support efficient fuzzy string search [2, 11, 12, 14, 21, 22, 29, 32, 36, 37, 62, 65]. Many algorithms use grams for efficient fuzzy string search. A gram of a string is a substring that can be used as a signature for efficient search. These algorithms answer a fuzzy query on a collection of strings using the following observation: if a string r in the collection is similar to the query string, then r should share a certain number of common grams with the query string. This "count filter" can be used to construct gram inverted lists for string ids to support efficient search. In Sect. 7, we evaluated some of the representative algorithms. The results showed that, not surprisingly, they are not as efficient as trie-based incremental-search algorithms, mainly because it is not easy to do incremental computation on gram lists, especially when a user types in a relatively short prefix, and count filtering does not give enough pruning power to eliminate false positives. In addition, there have been many studies for similarity join [2, 8, 13, 19, 52, 60, 61], estimating selectivity of approximate string queries [23, 27, 34, 35], and approximate entity extraction [1, 11, 58].

Keyword search in databases and XML data: There have many studies on keyword search over XML data [3, 16, 20, 31, 38, 39, 43, 44, 46–48, 53, 55, 56, 63, 64] and relational databases [9, 17, 24, 25, 28, 40, 51, 54]. Our work is orthogonal to these studies since it focuses on type-ahead search.

Compared to our previous work in [26], this article includes the following additional materials. (1) In Sect. 4.3, we presented new optimization techniques based on the concept of pivotal active node and conducted additional experiments to evaluate the techniques. (2) We included formal proofs of the claims. (3) We discussed how to support updates in Sect. 6.3 and conducted additional experiments in Sect. 7.7.

9 Conclusion and future work

We studied a new information-access paradigm, called type-ahead search, which finds answers to queries as a user types in keywords character by character, even allowing minor errors. We proposed an efficient incremental algorithm to answer single-keyword queries that are treated as fuzzy prefix conditions. We studied various algorithms for computing the answers to a query with multiple keywords. We developed efficient algorithms for incrementally computing answers to queries by using cached results of previous queries in order to achieve a high interactive speed on large data sets. We studied several useful features such as highlighting results, utilizing synonyms, and supporting data updates. We deployed several real systems to test the techniques and conducted an thorough experimental study of the algorithms. The results proved the practicality of our techniques to enable this new computing paradigm.

There are several open problems for type-ahead search. One is about how to support ranking queries efficiently. Another one is how to deal with large amounts of data when the index structures cannot fit into the memory.

Acknowledgments This work is partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, National S&T Major Project of China, the Scientific Research in Inner Mongolia under Grant No. NJzy08152, the US NSF awards IIS-0742960 and IIS-1030002, and a Google research award to UC Irvine.

References

1. Agrawal, S., Chakrabarti, K., Chaudhuri, S., Ganti, V.: Scalable ad-hoc entity extraction from text collections. PVLDB **1**(1), 945–957 (2008)
2. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)
3. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML keyword search with relevance oriented ranking. In: ICDE, pp. 517–528 (2009)
4. Bast, H., Chitea, A., Suchanek, F.M., Weber, I.: Ester: efficient search on text, entities, and relations. In: SIGIR, pp. 671–678 (2007)
5. Bast, H., Mortensen, C.W., Weber, I.: Output-sensitive autocompletion search. In: SPIRE, pp. 150–162 (2006)
6. Bast, H., Weber, I.: Type less, find more: fast autocompletion search with a succinct index. In: SIGIR, pp. 364–371 (2006)
7. Bast, H., Weber I.: The completesearch engine: interactive, efficient, and towards IR & DB integration. In: CIDR, pp. 88–95 (2007)
8. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)
9. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: ICDE, pp. 431–440 (2002)
10. Celikik, M., Bast, H.: Fast error-tolerant search on very large texts. In: SAC, pp. 1724–1731 (2009)
11. Chakrabarti, K., Chaudhuri, S., Ganti, V., Xin, D.: An efficient filter for approximate membership checking. In: SIGMOD Conference, pp. 805–818 (2008)
12. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: SIGMOD Conference, pp. 313–324 (2003)
13. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE, pp. 5–16 (2006)
14. Chaudhuri, S., Ganti, V., Motwani, R.: Robust identification of fuzzy duplicates. In: ICDE, pp. 865–876 (2005)
15. Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: SIGMOD Conference, pp. 707–718 (2009)
16. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: Xsearch: a semantic search engine for XML. In: VLDB, pp. 45–56 (2003)

17. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: ICDE, pp. 836–845 (2007)
18. Grabski, K., Scheffer, T.: Sentence completion. In: SIGIR, pp. 433–439 (2004)
19. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
20. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: Xrank: ranked keyword search over XML documents. In: SIGMOD Conference, pp. 16–27 (2003)
21. Hadjieleftheriou, M., Chandel, A., Koudas, N., Srivastava, D.: Fast indexes and algorithms for set similarity selection queries. In: ICDE, pp. 267–276 (2008)
22. Hadjieleftheriou, M., Koudas, N., Srivastava, D.: Incremental maintenance of length normalized indexes for approximate string matching. In: SIGMOD Conference, pp. 429–440 (2009)
23. Hadjieleftheriou, M., Yu, X., Koudas, N., Srivastava, D.: Hashed samples: selectivity estimators for set similarity selection queries. In: VLDB (2008)
24. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB, pp. 850–861 (2003)
25. Hristidis, V., Papakonstantinou, Y.: Discover: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
26. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: WWW, pp. 371–380 (2009)
27. Jin, L., Li, C., Vernica, R.: Sepia: estimating selectivities of approximate string predicates in large databases. VLDB J. **17**(5), 1213–1229 (2008)
28. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)
29. Kim, M.-S., Whang, K.-Y., Lee, J.-G., Lee, M.-J.: n-gram/2l: a space and time efficient two-level n-gram inverted index structure. In: VLDB, pp. 325–336 (2005)
30. Knuth D., The Art of Computer Programming, Sorting and Searching, third edition, Addison-Wesley (1998)
31. Kong, L., Gilleron, R., Lemay, A.: Retrieving meaningful relaxed tightest fragments for XML keyword search. In: EDBT, pp. 815–826 (2009)
32. Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing join and selection queries. In: VLDB, pp. 199–210 (2006)
33. Kukich, K.: Techniques for automatically correcting words in text. ACM Comput. Surv. **24**(4), 377–439 (1992)
34. Lee, H., Ng, R.T., Shim, K.: Extending q-grams to estimate selectivity of string matching with low edit distance. In: VLDB, pp. 195–206 (2007)
35. Lee, H., Ng, R.T., Shim, K.: Power-law based estimation of set similarity join size. PVLDB **2**(1), 658–669 (2009)
36. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
37. Li, C., Wang, B., Yang, X.: VGRAM: improving performance of approximate queries on string collections using variable-length grams. In: VLDB, pp. 303–314 (2007)
38. Li, G., Feng, J., Wang, J., Zhou, L. Effective keyword search for valuable LCAs over XML documents. In: CIKM, pp. 31–40 (2007)
39. Li, G., Feng, J., Wang, J., Zhou, L.: KEMB: a keyword-based XML message broker. In: TKDE (2011)
40. Li, G., Feng, J., Zhou, X., Wang, J.: Providing built-in keyword search capabilities in RDBMS. VLDB J. **20**(1), 1–19 (2011)
41. Li, G., Feng, J., Zhou, L.: Interactive search in XML data. In: WWW, pp. 1063–1064 (2009)
42. Li, G., Ji, S., Li, C., Feng, J.: Efficient type-ahead search on relational data: a tastier approach. In: SIGMOD Conference, pp. 695–706. (2009)
43. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: SIGMOD Conference, pp. 903–914 (2008)
44. Li, G., Zhou, X., Feng, J., Wang, J.: Progressive keyword search in relational databases. In: ICDE (2009)
45. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: VLDB, pp. 361–370 (2001)
46. Li, Y., Yu, C., Jagadish, H.V.: Schema-free xquery. In: VLDB, pp. 72–83. (2004)
47. Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: SIGMOD Conference, pp. 329–340 (2007)
48. Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for XML keyword search. PVLDB **1**(1), 921–932 (2008)
49. Motoda, H., Yoshida, K.: Machine learning techniques to make computers easier to use. Artif. Intell. **103**(1–2), 295–321 (1998)
50. Nandi, A., Jagadish, H.V.: Effective phrase prediction. In: VLDB, pp. 219–230. (2007)
51. Qin, L., Yu, J., Chang, L.: Scalable keyword search on large data streams. VLDB J. **20**(1), 35–57 (2011)
52. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD Conference, pp. 743–754 (2004)
53. Shao, F., Guo, L., Botev, C., Bhaskar, A., Chettiar, M., Yang, F., Shanmugasundaram, J.: Efficient keyword search over virtual XML views. VLDB J. **18**(2), 543–570 (2009)
54. Simitsis, A., Koutrika, G., Ioannidis, Y.E.: Précis: from unstructured keywords as queries to structured databases as answers. VLDB J. **17**(1), 117–149 (2008)
55. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway sLCA-based keyword search in XML data. In: WWW, pp. 1043–1052 (2007)
56. Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: Topx: efficient and versatile top-*k* query processing for semistructured data. VLDB J. **17**(1), 81–115 (2008)
57. Wang, J., Li, G., Feng, J.: Automatic URL completion and prediction using fuzzy type-ahead search. In: SIGIR, pp. 634–635 (2009)
58. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: SIGMOD Conference, pp. 759–770 (2009)
59. Williams, H.E., Zobel, J., Bahle, D.: Fast phrase querying with combined indexes. ACM Trans. Inf. Syst. **22**(4), 573–594 (2004)
60. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
61. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: ICDE, pp. 916–927 (2009)
62. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW (2008)
63. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD Conference, pp. 537–538 (2005)
64. Xu, Y., Papakonstantinou, Y.: Efficient LCA based keyword search in XML data. In: EDBT, pp. 535–546 (2008)
65. Yang, X., Wang, B., Li, C.: Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In: SIGMOD Conference (2008)