BIOINFORMATICS ORIGINAL PAPER Vol. 26 no. 18 2010, pages 2321–2327
doi:10.1093/bioinformatics/btq414

Data and text mining

Advance Access publication July 12, 2010

Interactive and fuzzy search: a dynamic way to explore MEDLINE

Jiannan Wang[1,†], Inci Cetindil[2,†], Shengyue Ji[2], Chen Li[2,*], Xiaohui Xie[2,3,*], Guoliang Li[1] and Jianhua Feng[1]

[1]Department of Computer Science, Tsinghua University, Beijing 100084, China, [2]Department of Computer Science and [3]Institute for Genomics and Bioinformatics, University of California, Irvine, CA 92697, USA

Associate Editor: Alex Bateman

ABSTRACT

Motivation: The MEDLINE database, consisting of over 19 million publication records, is the primary source of information for biomedicine and health questions. Although the database itself has been growing rapidly, the search paradigm of MEDLINE has remained largely unchanged.

Results: Here, we propose a new system for exploring the entire MEDLINE collection, represented by two unique features: (i) *interactive*: providing instant feedback to users' query letter by letter, and (ii) *fuzzy*: allowing approximate search. We develop novel index structures and search algorithms to make such a search model possible. We also develop incremental-update techniques to keep the data up to date.

Availability: Interactive and fuzzy searching algorithms for exploring MEDLINE are implemented in a system called iPubMed, freely accessible over the web at http://ipubmed.ics.uci.edu/ and http://tastier.cs.tsinghua.edu.cn/ipubmed/

Contact: chenli@ics.uci.edu; xhx@ics.uci.edu

Received on April 27, 2010; revised on July 7, 2010; accepted on July 8, 2010

1 INTRODUCTION

The PubMed service provided by NCBI is the most widely used system for accessing the MEDLINE database, which contains more than 19 million (as of April 2010) records from approximately 5000 selected publications covering biomedicine and health from 1950 onwards. It handles over 2 million searches per day, has become an essential part of every biomedical scientist's research effort, and is increasingly employed by physicians and patients as an indispensable tool to answer clinical questions.

PubMed uses keywords and Boolean operators to retrieve documents from MEDLINE. To perform a search, users need to first compose a keyword query, submit it to the server, wait and finally review the returned search results. If the returned results are too many or not pertinent, the users need to modify or refine the query, and resubmit it to the server. This type of *try-and-see* search paradigm requires the users to have certain knowledge to choose wisely the appropriate keywords, and often requires numerous iterations to reach the desired documents (Lewis *et al.*, 2006; Wildemuth and Moore, 1995), creating significant delay between the initial query and the final results. Even though there are several systems supporting search in the medical domain such as CiteXplore and HubMed, all of these systems use this traditional search paradigm. Recently, PubMed has started to give automatic suggestions as typing the query; but these suggestions are not based on the entire dataset. The suggestions are obtained by performing prefix search on the popular queries made by other users. For instance, if we type '*Weinberg oncogene*' to search for publications written by 'Weinberg' related to 'oncogene', PubMed does not give any suggestions, while there are a lot of documents containing these terms. In addition, PubMed cannot automatically handle approximate query search. Instead, it provides a list of candidate terms close to the query string and relies on users to pick up the right one, based on which it then performs exact search. This limitation is problematic for searching biomedical literatures, for which user queries frequently contain difficult-to-spell author names, non-standard gene symbols or specialized medical terms.

We propose an *interactive* and *dynamic* model of information retrieval and implement it to explore MEDLINE. The new model incorporates two unique features: (i) *interactive*: providing instant feedback as the query is being typed, and (ii) *fuzzy*: allowing approximate search (Gusfield, 1997; Navarro, 2001). Under this model, the system updates search results online invoked by every keystroke from the users. This type of *search-as-you-type* paradigm allows the users to find results 'on the fly' and enables them to dynamically modify or refine queries, removing the major barrier between queries and search results. The existing PubMed system has several similar features, such as 'browsing the index of terms', 'automatic term mapping' and 'truncating search terms'. The main difference between these features and iPubMed's features is that we do prefix-based search on the fly as the user types in a query, and we allow minor errors.

The new search paradigm poses significant computational challenges, due to the requirement of high interactive speed and the capability of relaxing keyword conditions. The total round-trip time between the client browser and the backend server includes the network delay and data-transfer time, query-execution time on the server, and the javascript-execution time on the client browser. To achieve an interactive speed, this total time should not exceed milliseconds (typically within 100 ms); the query-execution time on the server should be even shorter. This high speed is challenging to achieve especially since we allow keywords to appear at different places and to match approximately, both of which are not permitted by the popular autocompletion method implemented in major search engines (Bast and Weber, 2006) and more recently by PubMed.

*To whom correspondence should be addressed.

†The authors wish it to be known that, in their opinion, the first two authors should be regarded as joint First authors.

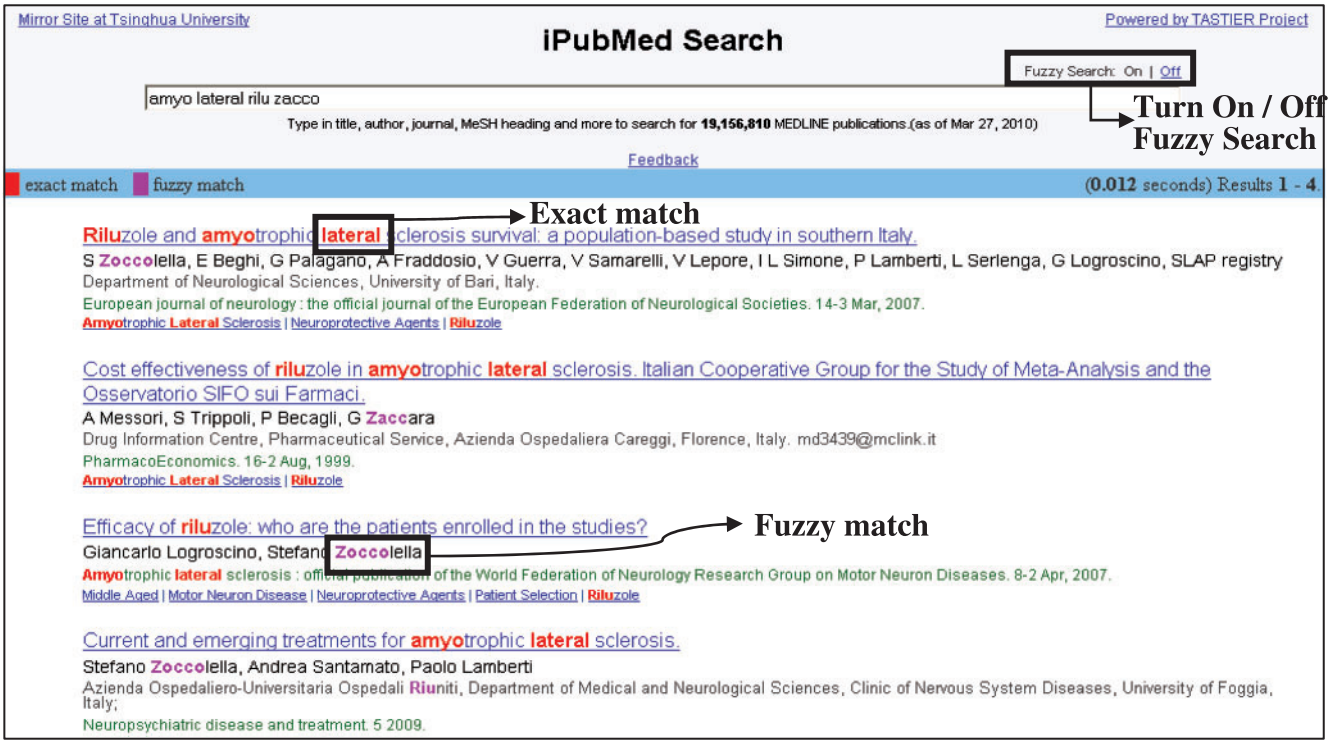

Fig. 1. Screenshot of iPubmed system (http://ipubmed.ics.uci.edu/ and http://tastier.cs.tsinghua.edu.cn/ipubmed/).

In this article, we show that the goal of high speed for interactive and fuzzy search is achievable by employing novel index structures, caching techniques and search algorithms. We implemented these algorithms and techniques in a system called iPubMed (stands for Interactive PubMed), which is currently able to search the entire MEDLINE. The preliminary algorithmic aspect of this work was presented previously in a conference proceeding (Ji *et al.*, 2009). Here, we provide a full description of the algorithms used and deploy these techniques specifically for MEDLINE search, incorporating additional methods such as incremental update, article ranking and parallel computing.

Figure 1 shows a screenshot of the system as a user typed in four keywords—'*amyo lateral rilu zacco*'. The user intended to find the publications describing the treatment of amyotrophic lateral sclerosis with drug riluzole authored by Zoccolella. PubMed at NCBI failed to return any publication record for this query as it contains a misspelling of the author name and two incomplete query keywords. In contrast, iPubMed was able to retrieve the right publications. More importantly, because the search results are returned in real time as query strings are being typed, users can adaptively change queries until desired results are reached.

The iPubMed interface has several important features that make it powerful and user friendly. It allows users to specify whether the system should do 'fuzzy' search by clicking the 'On' or 'Off' links. In addition, keywords in returned results are highlighted in the client's browser, with different colors depending on whether it is a fuzzy or an exact match. The system has also a pagination feature that helps users easily navigate through the results by using the provided links for the previous and next pages.

2 METHODS

2.1 System architecture of iPubMed

The overall architecture of iPubMed is shown in Figure 2. The client accepts a query through the user interface, and checks whether the cached results are enough to answer the query. If not, the client sends the query to the web server. The server has several components. The web server has a *Broker* that receives a query from a user, and sends the query to the FastCgi servers in the cluster. Each *FastCgi Server* waits for queries from the broker, and caches query results. The *Cache* component checks whether the query can be answered using the cached results. If not, the FastCgi server incrementally answers the query. For each query keyword, the *Fuzzy Prefix Finder* computes the predicted words and the lists of records that contain a predicted word. Next, the FastCgi server computes the intersection of the lists to compute the predicted records of the query and ranks the predicted records to identify the best answers. Finally, the broker collects all these local best answers from the FastCgi servers, aggregates these results and returns the best answers to the client. The *Indexer* component indexes the data as a trie structure with inverted lists of keywords and creates a forward index. It keeps the data and all these structures are in memory. For data changes, we download the update files from an NLM FTP server on a daily basis. We pre-process the files to extract the six most commonly searched attributes: authors, their affiliations, article title, journal name, journal issue and MESH terms, and keep this data in a relational table. Then, we partition the data into several machines horizontally and keep in a data shard. The *Updater* component reads the updates from the corresponding data shard and loads it into memory. Then it incrementally updates the index in memory.

2.2 Problem formulation

We formalize the problem of interactive, fuzzy search on a structured table, although our method can be easily adapted to textual documents, XML data and relational databases. Consider a relational table T with m attributes and n

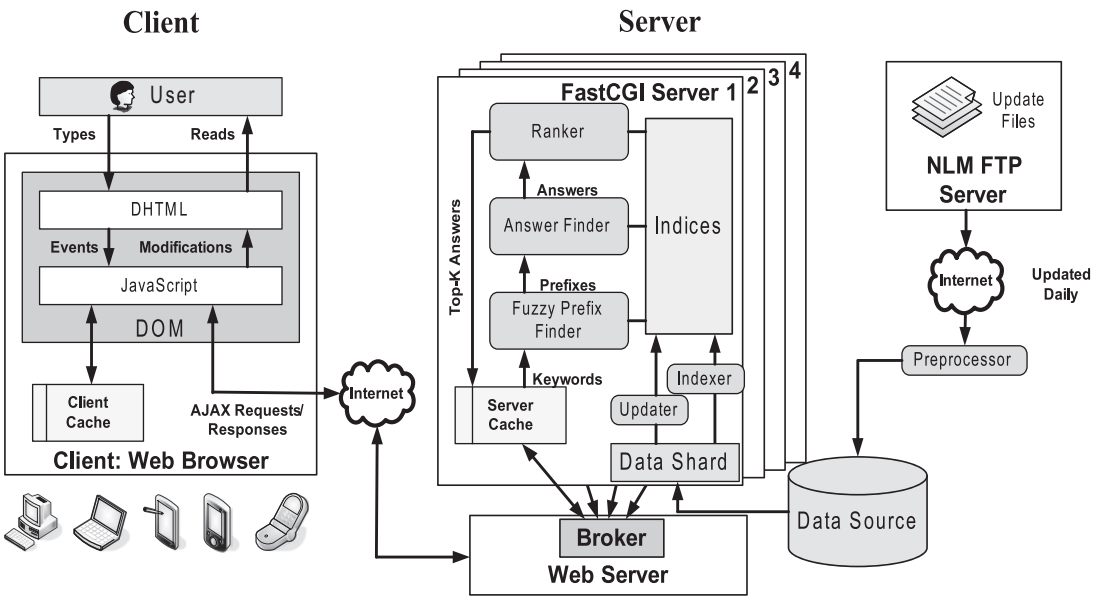

Fig. 2. The system architecture of iPubMed.

Table 1. A sample publication relational table

ID	Title	Authors	Journal name	Year
*r*1	Biopsy findings after breast conservation therapy for early-stage invasive breast cancer	Vapiwala,N., Starzykm,J., Harris,E.E., Tchou,J.C., Boraas,M.C., Czerniecki,B.J., Rosato,E.F., Orel,S.G. and Solin,L.J.	*Int. J. Radiat. Oncol. Biol. Phys.*	2007
*r*2	Fine-needle aspiration biopsy findings in patients with small lymphocytic lymphoma transformed to hodgkin lymphoma	Catrina Reading,F., Schlette,E.J., Stewart,J.M., Keating,M.J., Katz,R.L., Caraway,N.P.	*Am. J. Clin. Pathol.*	2007
*r*3	Histopathology reporting of prostate needle biopsies	Montironi,R., Vela Navarrete,R., Lopez-Beltran,A., Mazzucchelli,R., Mikuz,G., Bono,A.V.	*Virchows Arch.*	2006
*r*4	Ultrasound-guided prostate biopsy in 2005	Clements,R. and Luis,T.	*Int. Am. J.*	2006
*r*5	Epidemiology of biopsy proven giant cell arteritis in northwestern Spain: trend over an 18 year period	Gonzalez-Gay,M.A., Garcia-Porrua,C., Rivas,M.J. Rodriguez-Ledo,P., Llorca,J.	*Ann. Rheum. Dis.*	2007
*r*6	The optimal diet for women with polycystic ovary syndrome?	Marsh,K. and Brand-Miller,J.	*Br. J. Nutr.*	2007
*r*7	Bile duct dysplasia and congenital hepatic fibrosis associated with polycystic kidney (Caroli syndrome) in a rat	Bettini,G., Mandrioli,L., Morini,M.	*Vet. Pathol.*	2007
*r*8	Open-heart operations in patients with a spinal cord injury	Lin,D., Bakaeen,F.G., Shenaq,S.A., Ribati,M., Atluri,P.V., Holmes,S.A., Berger,D.H., Huh,J.	*Am. J. Surgery*	2007
*r*9	Effects of zinc coadministration on lead toxicities in rats	Piao,F., Cheng,F., Chen,H., Li,G., Lu,X., Liu,S., Yamauchi,T., Yokoyama,K.	*Ind. Health*	2007
*r*10	Dye-guided and radio-guided sentinel node biopsy in breast cancer	Imoto,S. and Ito,H.	*J. Surgery*	2007

records. Let $A=\{a_1, a_2, \ldots, a_m\}$ denote the attribute set, $R=\{r_1, r_2, \ldots, r_n\}$ denote the record set and $W=\{w_1, w_2, \ldots, w_p\}$ denote the distinct word set in T. Given two words w_i and w_j, $w_i \leq w_j$ denotes that w_i is a prefix string of w_j. An example relational table is shown in Table 1, which has 10 records and 4 attributes.

Each keyword in a given query is treated as a *partial* keyword. For each query keyword, we first identify the words in W (called *predicted words*) that contain a prefix matching the query keyword exactly or approximately (in the case of fuzzy search). Then, we find the records in R (called *predicted records*) that contain at least one of the predicted words of every query keyword. Finally, we rank the returned records.

More precisely, the search problem is formulated as follows. Given a query consisting of a set of prefixes $Q=\{p_1, p_2, \ldots, p_l\}$, we first identify the predicted-word set of each prefix, that is, for prefix p_i, $P_i=\{p'_i | \exists w \in W, p'_i \leq w \text{ and } \mathrm{ed}(p'_i, p_i) \leq \delta\}$, where $\mathrm{ed}(p'_i, p_i)$ is the edit distance between two strings and δ is the edit-distance threshold. Next, we identify the predicted-record set of the query, $R_Q=\{r | \exists p'_i \in P_i \ \& \ w_i \text{ in } r \in Rs.t.\ p'_i \leq w_i, \forall i \in [1,l]\}$. Finally, we rank the records in R_Q according to their relevance to Q.

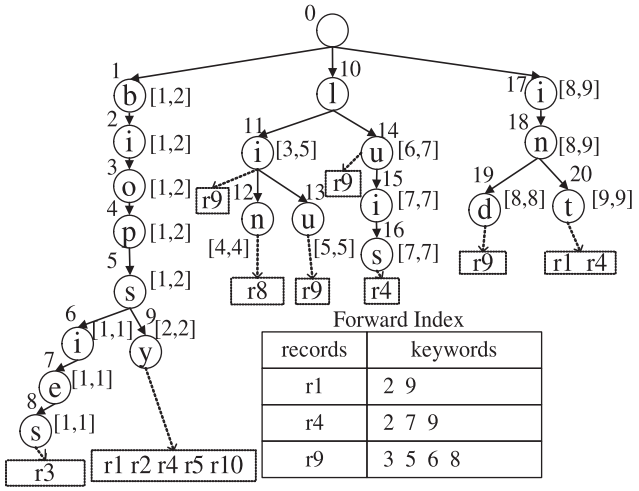

Fig. 3. An example index structure (partial) for the publication records shown in Table 1.

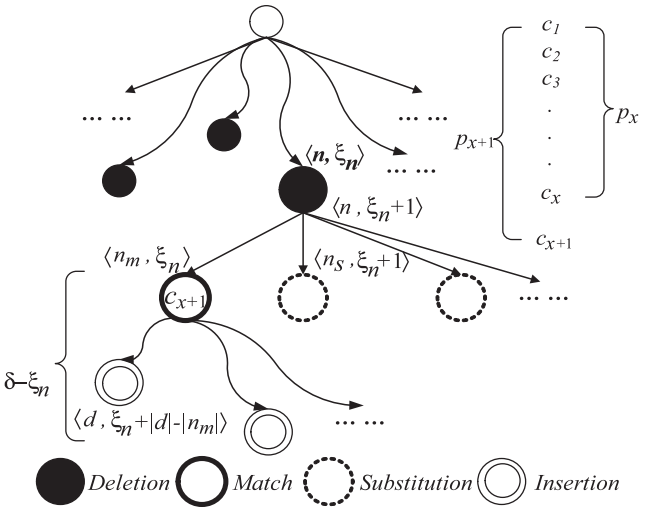

Fig. 4. Incrementally computing active nodes.

2.3 Index structure

We use a trie to index the words in the table. Each word w in the table corresponds to a unique path from the root of the trie to a leaf node. Each node on the path has a label of a character in w. The nodes with the same parent are sorted by the node label in their alphabetical order. Each leaf node has a unique keyword ID for the corresponding word. The keyword ID is assigned in the pre-order. Each node maintains the range of keyword IDs in its subtree: [minKeyID, maxKeyID]. For each leaf node, we store an inverted list of record IDs that contain the corresponding word. To improve search performance, we can also maintain a forward index for the records. For each record, the forward index keeps the sorted keyword IDs in the record. Consider the publication relation in Table 1. Its trie for the tokenized words is shown in Figure 3. The word 'luis' has a node ID of 16, and its inverted list includes record $r4$. The keyword ID of leaf node 11 is 3. The keyword range of node 11 is [3, 5]. The forward list of record $r4$ includes keyword IDs 2, 7 and 9.

2.4 Search algorithm

We tokenize each query string to keywords. Our search algorithm consists of the following three steps: (i) Finding the predicted words of each keyword and the list of records that contain the predicted words; (ii) identifying the predicted records by computing the intersection of the lists corresponding to different query keywords; and (iii) ranking the answers. Next, we describe these three steps.

2.4.1 Incrementally identifying predicted words of each keyword For each input keyword, we incrementally identify the predicted words based on its prefixes. In the case of exact search, there exists only one trie node that match a partial keyword, therefore finding the predicted words is relatively easy and can be done by traversing the descendants of the trie node. However, to support fuzzy search, we need to predict multiple prefixes that are similar to the partial keyword. We call the nodes of these similar prefixes the active nodes of the input keyword (Fig. 4). We will need to locate the leaf descendants of all active nodes, and identify the predicted records of these leaf nodes. For example, consider the trie in Figure 3. Suppose $\delta=1$, and a user types in a partial keyword 'li'. The words 'li', 'lin', 'liu', 'lu' and 'lui' are all similar to the input keyword, since their edit distances to 'li' are within a threshold $\delta=1$. Thus, nodes 11, 12, 13, 14 and 15 are active nodes.

Given an input keyword p, we store the set of active nodes $\Phi_p=\{<n,\ \xi_n>\}$, where n is an active node for p, and $\xi_n=\mathrm{ed}(p,\ n)\leq\delta$. (For the simplicity of notation, we will use n to denote both the trie node and its corresponding string). We call Φ_p the 'active-node set' for keyword p (together with the edit-distance information for each active node). The main idea behind our method is to use the prefix-filtering. That is, when the user types in one more letter after p, only the descendants of the active nodes of p can be the active nodes of the new query and need to be examined. We use this property to *incrementally* compute the active-node set of a new query, taking advantage of the *cached* active-node sets Φ_p.

Suppose a user is typing in a query string $c_1c_2\ldots c_x$ letter by letter. After the user types in a prefix query $p_i=c_1c_2\ldots c_i$ $(i\leq x)$, we keep an active-node set Φ_{pi} for p_i. When the user types in a new character c_{x+1} and submits a new query p_{x+1}, we compute the active-node set Φ_{px+1} for p_{x+1} making use of Φ_{px} as follows.

We start by initializing an active node set corresponding to the empty keyword ε, i.e. $\Phi p_0=\Phi_\varepsilon=\{<n,\ \xi_n>|\xi_n=|n|\leq\delta\}$. That is, it includes all trie nodes n whose corresponding string has a length $|n|$ within the edit-distance threshold δ. These nodes are active nodes for the empty string since their edit distances to ε are within δ.

For each $<n,\ \xi_n>$ in Φ_{px}, we consider whether the descendants of n are active nodes for p_{x+1}. If $\xi_n+1\leq\delta$, then n is an active node for p_{x+1}, so we add $<n,\ \xi_n+1>$ to Φ_{px+1}. This case corresponds to deleting the last character c_{x+1} from the new query string p_{x+1}. Note that even if $\xi_n+1\leq\delta$ does not hold, node n can still potentially become an active node of the new query string, due to operations described below on other active nodes in Φ_{px}. For each child n_c of node n, we consider two possible cases.

In the first case, the child node n_c has a character different from c_{x+1}. Suppose node n_s is such a child node. We have $\mathrm{ed}(n_s, p_{x+1})\leq\mathrm{ed}(n, p_x)+1=\xi_n+1$. If $\xi_n+1\leq\delta$, n_s is an active node for the new string, and thus $<n_s,\ \xi_n+1>$ will be added to Φ_{px+1}. This case corresponds to substituting the label of n_s for the letter c_{x+1}.

In the second case, the child node n_c has a label c_{x+1}. Suppose node n_m is such a child node. In this case, we have $\mathrm{ed}(n_m, p_{x+1})\leq\mathrm{ed}(n,\ p_x)=\xi_n\leq\delta$. Therefore, n_m is always an active node of the new string, so we add $<n_m,\ \xi_n>$ to Φ_{px+1}. This case corresponds to the match between the character c_{x+1} and the label of n_m. One subtlety here is that, if the distance for the node n_m is smaller than δ, i.e., $\xi_n<\delta$, we need to consider additional nodes: for each descendant d of n_m that is at most $\delta-\xi_n$ letters away from n_m, we also need to add $<d,\ \xi_d>$ to Φ_{px+1}, where $\xi_d=\xi_n+|d|-|n_m|$. This operation corresponds to inserting letters after node n_m (for node n_s, we do not need to consider its descendants for insertions; because if these descendants are active nodes, they must be in Φ_{px} and thus will still be considered).

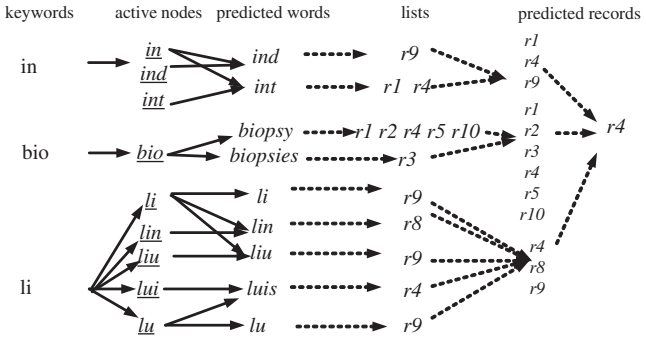

Fig. 5. Intersecting lists for answering a keyword query 'in bio li'.

Note that during the update of Φ_{px+1}, the above procedure may result in the addition of multiple sets corresponding to the same node, in which case we only keep the one with the shortest edit distance to the query string p_{x+1}.

2.4.2 Finding predicted records Given a query $Q=\{p_1, p_2,\ldots, p_l\}$, suppose $\{k_{i1}, k_{i2},\ldots\}$ is the set of keywords that are similar to the prefix p_i. Let L_{ij} denote the inverted list of k_{ij}, and $U_i=\cup_j L_{ij}$ be the union of the lists for p_i. Our goal is to find $\cap_i U_i$, the intersection of different prefix union lists. Figure 5 illustrates an example in which we want to answer query 'in bio li'.

To find the intersection, we first find the prefix with the shortest union list. We call each record in this list candidate record. Then we use the forward index to check whether each candidate record contains similar prefixes of every other query keyword. If so, this record is an answer. Each active state of other query keywords has a keyword range $[s, l]$, and we check whether the candidate record contains a keyword in the range $[s, l]$ using the following steps: (i) use a binary search method to find the candidate record ID in the forward index; (ii) find the smallest keyword ID on the candidate record's forward list that is larger than or equal to s; and (iii) check whether this keyword ID is smaller than l.

2.4.3 Ranking We consider the following several factors when designing a metric for ranking the search answers: (i) *Matching prefixes*: we consider the similarity between a query keyword and its best matching prefix. The more similar a record's matching keywords are to the query keywords, the higher this record should be ranked. The similarity is also related to keyword length. Exact matches on the query have a higher priority than fuzzy matches. For example, consider the trie in Figure 3. If a user types in 'liu', the record *r9* could be ranked higher than *r8*; since the record *r9* has an exact keyword match when *r8* has a fuzzy keyword match 'lin'. (ii) *Record weights*: different records could have different weights. For example, a newly added publication record could be ranked higher than older publications.

To combine these factors, we use the following scoring function. Suppose the query is $Q=\{p_1, p_2\ldots, p_m\}, p_i'$ is the best matching prefix for p_i. The score of a record r for Q is defined as:

$$\text{score}(r,Q)=\sum_{i=1,\ 2,\ldots,m}[\psi(r)/(\alpha\times \text{ed}(p_i,p_i')^2+\beta)],$$

where α and β are weights used to adjust the effect of edit distance. We use $\alpha=10$, $\beta=1$ in our system. $\psi(r)$ is the score of record r, which is defined as:

$$\psi(r)=r[\text{year}]-1900+10^{-9}\times r[\text{pmid}],$$

where r[year] and r[pmid] are the corresponding fields of the record r. These fields are used in the ranking function to give a higher priority to recent publications. Since many records have the same year and *pmid*s are given to the records in an increasing order, we also used the *pmid* field to be able to rank the records within the same year.

2.5 Caching algorithms

Results of earlier computations are cached to speed up later queries. After finding the answers of a query, we cache the active states for prefixes of each input keyword. We then incrementally answer the subsequent keywords using the cached active states. For the query with multiple keywords, we also cache the predicted records (intersection of union lists). If the user types another keyword, we use the cached records to answer the query by checking whether the cached records contain the new keyword using the forward index. If there are too many predicted records, we just cache the highly relevant ones. For each subsequent keyword, we first use the cached records to compute the answer. If there are not enough top answers, we continue to compute more answers for the previous query and store the results in the cache. This 'on-demand' caching method makes sure that each query is answered efficiently, and we cache results of a query only if they are needed.

Results in the client are cached to reduce communication cost. This optimization is especially important in scenarios where the user has a limited network bandwidth, such as mobile networks. The main idea is that the client browser caches the results of previous queries. To send to the client the answers to a subsequent query, the server just sends the identifiers of those already in the earlier results, in addition to the additional records. In this way, only the ids of the earlier results need to be transferred over the network.

2.6 Incremental updates

Since many new records are added to the MEDLINE database on a daily basis, updating our dataset timely becomes very crucial. We download the provided update files from an NLM FTP server every day and use incremental-update techniques to maintain the trie structure, inverted lists and forward index. This allows us to process inserted, revised and deleted records without reconstructing the whole structure from scratch.

The MEDLINE database is maintained via insertions, deletions and revisions. For each revision, we delete the existing record first, and then insert the new record. Therefore, we will focus on insertions and deletions. We store the trie, inverted lists, forward lists and the original data in memory. We also keep a copy of the data shard on the disk to be able to rebuild the structures in case of a system failure. Next, we discuss how these structures change in the presence of an insertion or deletion.

Deletion: Assume a record r is deleted. First, we delete it from the copy on disk. For the in-memory copy, we mark the record r as invalid, but do not delete its keywords from the trie, because other records may contain these keywords. We do not modify the inverted index nor the forward index, since they are kept sorted and could be large. In this scenario, if the record r is found in the answers to a query, the system will not return the record r to the user since it is marked as invalid.

Insertion: Let r be an inserted record. First, we insert the record into the data on disk. Then, we tokenize r to keywords and insert each of its keywords into the trie. If there is a leaf node for the keyword, we can just add the record r into the inverted list of this leaf node. Since the inverted list of this keyword is sorted and might be huge, it could be expensive to insert r directly into the list. For this reason, for each leaf node, we keep a primary list and a secondary inverted list. We use the primary inverted list when building the structure, and use the secondary inverted list for storing updates. This method can reduce the time to insert a record to the inverted list, since the number of records in the secondary inverted list tends to be smaller than the primary one. These two lists can be merged into the primary list periodically to be able to keep the secondary inverted list small.

If a keyword is seen for the first time, it should be added to the trie. To be able to use the forward index with the updated trie, we want to preserve the order of the assigned ids of the trie nodes. If the keyword ids on the trie are assigned consecutively, we may not be able to assign new unique ordered IDs for the new keywords. To solve this problem, we reserve some extra keyword ids on the trie to use in case the updated dataset contains new keywords. In the rare case where the reserved space is not enough for new keywords, we can rebuild the index structures.

After inserting all the keywords of record r into the trie and the record id of r into their corresponding inverted lists, we can simply append the record id of r with its corresponding keyword ids into the forward index. In this scenario, for a query, if we reach a leaf node in the trie, we need to consider both its primary and secondary inverted lists. The rest of the search process will be the same as before.

3 RESULTS

3.1 System implementation

The iPubMed web server was set up using Apache2 on a Linux machine. The web server has a broker which receives a query from a user, and sends the query to the FastCgi Servers in the cluster. In order to process queries over 19 million records, the current iPubMed system at Tsinghua University is using a cluster of two slave machines, each with four Intel Xeon E5420 (2.5 GHz) CPUs and 16G DDR2-800 memory. The system at UCI is using a cluster of four slave machines, each with two AMD Opteron 248 (2.2 GHz) CPUs and 8G DDR2-800 memory. In the rest of the article, we focus on the cluster at UCI and run our experiments in this cluster. Each slave at UCI has two FastCgi Servers and each server builds its local index on its local data (about 2.4 million records). The data are partitioned through these eight processes by round-robin partitioning to do the load balancing. The backend was implemented as a FastCGI server process, written in C++, compiled with a GNU compiler. Indexes were constructed on six most commonly searched attributes: authors, their affiliations, article title, journal name, journal issue and MESH headings. Table 2 shows the size of the dataset, index size and index-construction time. These numbers are the sum of the sizes across eight processes. In the future, if the total size of the index structures in one processor exceeds the memory limit; we can add more machines to the cluster.

Table 2. Total index size and construction time for four slaves each with two processes

Record number	19 million
Total size of indexed attributes	5.8 GB
Number of distinct keywords	3.07 million
Index construction time (for each process)	320 s
Trie size	1.82 GB
Inverted-list size	2.65 GB
Forward-index size	2.65 GB

3.2 User interface

The iPubMed interface is designed to show the query results in a user friendly way. Figure 1 demonstrates an example of a query results. In this interface, a user can specify whether to use fuzzy search feature. If this feature is disabled, only the exact matches to a query will be displayed. Furthermore, the fuzzy matches and exact matches are highlighted with different colors to make them more distinguishable. The interface has also a pagination feature for navigation through the different pages of the results.

3.3 Query performance

We evaluated the query performance as the number of keywords increased. Two types of queries were generated: the first one consisting of keywords randomly chosen from the dataset, and the second one consisting of modified queries from the first type by adding 1 edit error to each keyword. Each query asked for 10 best records. The average query response time for a query is shown in Figure 6a, which demonstrates that the algorithms can answer a single-keyword query very efficiently (within 20 ms) for both types of queries. The processing time for multiple-keyword queries is typically longer; however, it is still within a millisecond range. Our algorithm caches the earlier results and uses them to calculate the new result set incrementally. It intersects the earlier results with the results of the new query keyword. Thus, the average search time may also decrease if the last query keywords are very restrictive. We see such a behavior in the results of exact match for four-keyword queries.

We also measured the query time as the number of characters increased in the query keyword. We generated single-keyword queries that asked for 10 best records incrementally starting from the third keystroke to the tenth keystroke. The average query response time for each keystroke is shown in Figure 6b. Since all the queries have single keyword, the time does not include intersecting any inverted list. So the time to retrieve the best 10 records is expected to be very similar no matter how many characters the keyword has. However, the figure shows that our algorithms can speedup the later queries by caching the former results.

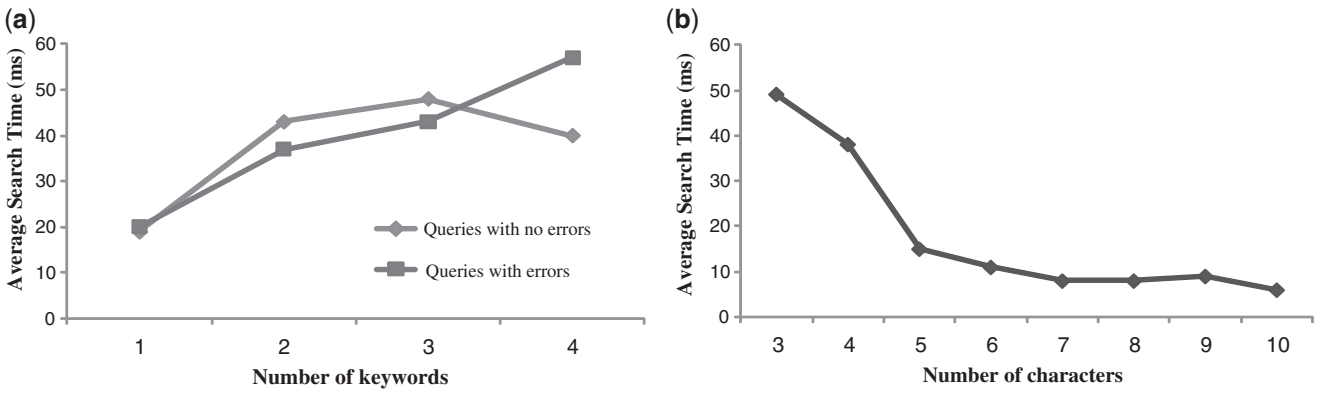

Fig. 6. (**a**) Average search time for queries with different numbers of keywords. (**b**) Average search time for queries with different numbers of characters.

3.4 Incremental updates

MEDLINE is a highly dynamic database with thousands of publication records added or revised each day. Therefore, it is important for iPubMed to be able to keep up with these daily changes and update the internal data structures quickly and efficiently. In our current implementation, we download the update files from the NLM FTP server every day and use incremental-update techniques to maintain the trie structure, inverted lists and forward index (see Section 2.6). This allows us to process inserted, revised, and deleted records without reconstructing the whole structure from scratch. Instead of spending 320 s to reconstruct the index structures, we incrementally update the structures around 15 s in average.

4 DISCUSSION

We described a new system for searching the MEDLINE database, implemented in a fully functional server called iPubMed. Comparing with the most widely used PubMed system at NCBI, the iPubMed system contains two unique features: (i) being interactive, returning search results on the fly and allowing users to change queries adaptively, and (ii) allowing approximate search.

We emphasize that our goal is not to replace the PubMed system, which contains a number of useful features not implemented in iPubMed, such as limiting search within different fields, allowing boolean operations and so on. If a user knows exactly the authors and the title of the paper he or she wants to find, the PubMed system is sufficient for the task. Instead, iPubMed is targeting at a different category of searches, in which the users have uncertain or partial information regarding the publication records that they would like to find as showed in Figure 1. Through interactive search, iPubMed allows users to refine and/or modify queries on the fly without the need of issuing separate, independent queries as in PubMed.

Although iPubMed is fully functional in its current form, there is a lot of room for further improvement. Currently, iPubMed does not search abstracts of articles due to computational constraints. In the future, we plan to increase the scalability of the system by utilizing parallel computing and expanding system hardware. We also plan to increase the functionality of iPubMed in several other directions, such as limiting search in different fields and allowing boolean operations. Our goal is to make iPubMed a truly practical and useful tool for biomedical researchers.

Funding: Google and Microsoft; the National Natural Science Foundation of China (60873065); the National High Technology Development 863 Program of China (2007AA01Z152, 2009AA011906); the National Grand Fundamental Research 973 Program of China (2006CB303103) in part.

Conflict of Interest: the authors declare financial interest in Bimaple Technologies Inc., which is commercializing the algorithms described in this publication.

REFERENCES

Bast,H. and Weber,I. (2006) Type less, find more: fast autocompletion search with a succinct index. *Proc. ACM SIGIR 2006*, ACM, pp. 364–371.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA.

Ji,S. *et al*. (2009) Efficient interactive fuzzy keyword search. *Proceedings of the 18th International Conference on World Wide Web, 2009*, ACM, pp. 371–380.

Lewis,J. *et al*. (2006) Text similarity: an alternative way to search MEDLINE. *Bioinformatics*, **22**, 2298–2304.

Navarro,G. (2001) A guided tour to approximate string matching. *ACM Comput. Surv. Arch.*, **33**, 31–88.

Wildemuth,B.M. and Moore,M.E. (1995) End-user search behaviors and their relationship to search effectiveness. *Bull. Med. Libr. Assoc*., **83**, 294–304.