DBease: Making Databases User-friendly and Easily Accessible

Guoliang Li Ju Fan Hao Wu Jiannan Wang Jianhua Feng
Department of Computer Science, Tsinghua University, Beijing 100084, China
{liguoliang, fengjh}@tsinghua.edu.cn; {fan-j07, haowu06, wjn08}@mails.thu.edu.cn

ABSTRACT

Structured query language (SQL) is a classical way to access relational databases. Although SQL is powerful to query relational databases, it is rather hard for inexperienced users to pose SQL queries, as they are required to be familiar with SQL syntax and have a thorough understanding of the underlying schema. To provide an alternative search paradigm, keyword search and form-based search are proposed, which only need users to type in keywords in single or multiple input boxes and return answers after users submit a query with *complete* keywords. However 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 towards solving this problem. In this paper, we propose a new search method DBEASE to make databases user-friendly and easily accessible. DBEASE allows users to explore data "on the fly" as they type in keywords, even in the presence of minor errors. DBEASE has the following unique features. Firstly, DBEASE can find answers as users type in keywords in single or multiple input boxes. Secondly, DBEASE can tolerate errors and inconsistencies between query keywords and the data. Thirdly, DBEASE can suggest SQL queries based on limited query keywords. We study research challenges in this framework for large amounts of data. We have deployed several real prototypes, which have been used regularly and well accepted by users due to its friendly interface and high efficiency.

1. INTRODUCTION

Structured query language (SQL) is a database language for managing data in relational database management systems (RDBMS). SQL supports schema creation and modification, data insertion, deletion and update, and data access control. Although SQL is powerful, it has a limitation that it requires users to be familiar with the SQL syntax and have a thorough understanding of the underlying schema. Thus SQL is rather hard for inexperienced users to pose queries.

To provide an easy way to query databases, keyword search and form-based search are proposed. Keyword search only needs users to type in query keywords in a single input box and the system returns answers that contain keywords in any attributes. Although single-input-box interfaces for keyword search are easy to use, users may need an interface that allows them to specify keyword conditions more precisely. For example, a keyword may appear in different attributes, and multiple keywords may appear in the same attribute. If a user has a clear idea about the underlying semantics, it is more user-friendly to use a form-based interface with multiple input boxes to formulate queries.

However existing keyword-search-based systems and form-based systems require users to compose 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. Many systems are introducing various features to solve this problem. One of the commonly used methods is *autocomplete*, which predicts a word or phrase that the user may type in based on the partial string the user has typed. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords.

One limitation of autocomplete is that the system treats a query with multiple keywords as a single string, and it does not allow these keywords to appear at different places. For instance, consider the search box on 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 October 2010), simply because these two keywords are not adjacent in the record.

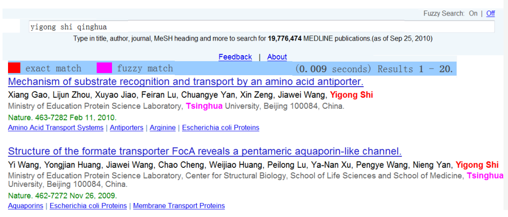
To address this problem, recently search-as-you-type [22, 17, 16, 13, 6, 3, 1, 2, 8] is proposed in which a user types in keywords letter by letter, and the system finds answers that include these keywords (possibly at different places). For instance, if a user types in a query "`cidr database sear`" with a partial keyword "`sear`," search-as-you-type finds answers that contain complete keywords "`cidr`" and "`database`," and a keyword with the partial keyword "`sear`" as a prefix, such as "`search`." Note that the keywords may appear at different places (possibly in different attributes).

In this paper, to improve user experience of querying databases and make <u>d</u>ata<u>b</u>ases user-fri<u>e</u>ndly and e<u>a</u>sily acces<u>s</u>ibl<u>e</u>, we propose a new search method, called "DBEASE,"[1] to improve keyword-search, form-based search, and SQL-based

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11)
January 9-12, 2011, Asilomar, California, USA.

[1] http://dbease.cs.tsinghua.edu.cn


Fuzzy Search: On | Off

yigong shi qinghua

Type in title, author, journal, MeSH heading and more to search for **19,776,474** MEDLINE publications.(as of Sep 25, 2010)

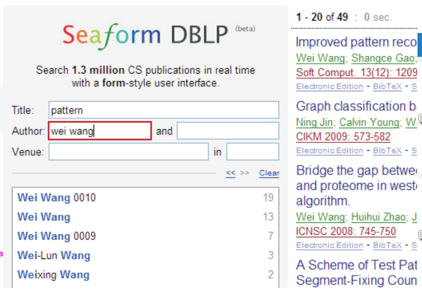
Feedback | About

🟥 exact match 🟪 fuzzy match (0.009 seconds) Results 1 - 20.

Mechanism of substrate recognition and transport by an amino acid antiporter.
Xiang Gao, Lijun Zhou, Xuyao Jiao, Feiran Lu, Chuangye Yan, Xin Zeng, Jiawei Wang, Yigong Shi
Ministry of Education Protein Science Laboratory, Tsinghua University, Beijing 100084, China.
Nature. 463-7282 Feb 11, 2010.
Amino Acid Transport Systems | Antiporters | Arginine | Escherichia coli Proteins

Structure of the formate transporter FocA reveals a pentameric aquaporin-like channel.
Yi Wang, Yongjian Huang, Jiawei Wang, Chao Cheng, Weijiao Huang, Peilong Lu, Ya-Nan Xu, Pengye Wang, Nieng Yan, Yigong Shi
Ministry of Education Protein Science Laboratory, Center for Structural Biology, School of Life Sciences and School of Medicine, Tsinghua University, Beijing 100084, China.
Nature. 462-7272 Nov 26, 2009.
Aquaporins | Escherichia coli Proteins | Membrane Transport Proteins

(a) Keyword Search (PubMed)


Seaform DBLP (beta)

Search **1.3 million** CS publications in real time with a **form**-style user interface.

Title: pattern
Author: wei wang and
Venue: in

<< >> Clear

Wei Wang 0010	19
Wei Wang	13
Wei Wang 0009	7
Wei-Lun Wang	3
Weixing Wang	2

1 - 20 of 49 : 0 sec.

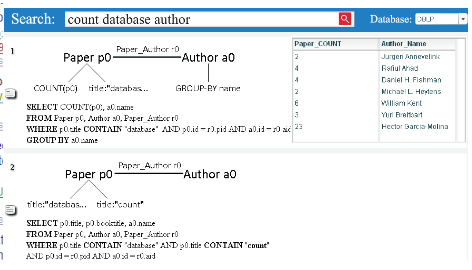
Improved pattern reco
Wei Wang; Shangce Gao;
Soft Comput. 13(12): 1209
Electronic Edition - BibTeX - S

Graph classification b
Ning Jin; Calvin Young; W
CIKM 2009: 573-582
Electronic Edition - BibTeX - S

Bridge the gap betwe
and proteome in west
algorithm.
Wei Wang; Huihui Zhao; J
ICNSC 2008: 745-750
Electronic Edition - BibTeX - S

A Scheme of Test Pat
Segment-Fixing Coun

(b) Form-based Search (DBLP)



Search: count database author Database: DBLP

1

```
SELECT COUNT(p0), a0.name
FROM Paper p0, Author a0, Paper_Author r0
WHERE p0.title CONTAIN "database"  AND p0.id = r0.pid AND a0.id = r0.aid
GROUP BY a0.name
```

Paper_COUNT	Author_Name
2	Jurgen Annevelink
4	Rafiul Ahad
4	Daniel H. Fishman
2	Michael L. Heytens
6	William Kent
3	Yuri Breitbart
23	Hector Garcia-Molina

2

```
SELECT p0.title, p0.booktitle, a0.name
FROM Paper p0, Author a0, Paper_Author r0
WHERE p0.title CONTAIN "database" AND p0.title CONTAIN "count"
AND p0.id = r0.pid AND a0.id = r0.aid
```

(c) SQL-based Search (DBLP)

Figure 1: Screenshots of prototypes implemented using our techniques (http://dbease.cs.tsinghua.edu.cn).

search by supporting *search-as-you-type* and *tolerating minor errors* between query keywords and the underlying data. DBEASE has the following unique features. Firstly, DBEASE searches the underlying data "on the fly" as users type in keywords. Secondly, DBEASE can find relevant answers as users type in keywords in a single input box or multiple input boxes. Thirdly, DBEASE can tolerate inconsistencies between queries and the underlying data. Fourthly, DBEASE can suggest SQL queries from limited keywords.

We study research challenges in this framework for large amounts of data. The first challenge is search efficiency to meet the high interactive-speed requirement. Each keystroke from the user can invoke a query on the backend server. 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. In order 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. The second challenge is to provide "on-the-fly join" for form-based search, as it is rather expensive to "join" keywords in multiple attributes. The third challenge is to infer users' query intent, including structures and aggregations, from limited keywords. To achieve a high speed for search-as-you-type, we develop novel index structures, caching techniques, search algorithms, and ranking mancinism. For effective SQL suggestion, we propose *queryable templates* to model the structures of promising SQL queries and a probabilistic model to evaluate the relevance between a template and a keyword query. We generate SQL queries from templates by matching keywords to attributes. We devise an effective ranking model and top-k algorithms to efficiently suggest the best SQL queries. We have deployed several real prototypes using our techniques, which have been used regularly and well accepted by users due to its friendly interface and high efficiency. Figure 1 gives three prototypes implemented using our techniques, which are available at http://dbease.cs.tsinghua.edu.cn.

1.1 Related Work

There have been many studies on predicting queries and user actions [19, 14, 9, 21, 20] 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[2] 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. 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 simply traversing a trie index (Section 2.1). Instead, the backend intersection (or "join") operation of multiple lists requires more efficient indexes and algorithms.

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) [2, 3, 1, 5]. Different from CompleteSearch[3], we propose trie-based index structures and incremental search algorithms to achieve a high interactive speed. Chaudhuri et al. [6] also studied how to extend autocompletion to tolerate errors. Different from [6], we support answering multi-keyword queries.

In addition, there have been some studies on keyword search in relational databases [12, 11, 10, 18]. However they cannot support search on-the-fly.

2. IMPROVING KEYWORD SEARCH

This section improves keyword search by supporting *search-as-you-type* and *tolerating errors* between query keywords and the underlying data. We first give an example to show how search-as-you-type works for queries with multiple keywords in a relational table. Our method can be extended to support search-as-you-type on documents [13], XML data [15], and multiple relational tables [16]. Assume a relational table resides on a server. A user accesses and searches the data through a Web browser. Each keystroke that the user types invokes a query, which includes the current string the user has typed in. The browser sends the query string to the server, which computes and returns to the user the best answers ranked by their relevancy to the query. We treat every query keyword as a *partial (prefix)* keyword[3].

Formally consider a set of records R. 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".

Different from exact search-as-you-type, the query answer of *fuzzy* search-as-you-type is a set of records r in R such

[2]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.

[3]Clearly our techniques can be used to answer queries when only the last keyword is treated as a partial keyword, and the other keywords are treated as completed keywords.

Table 1: A set of records

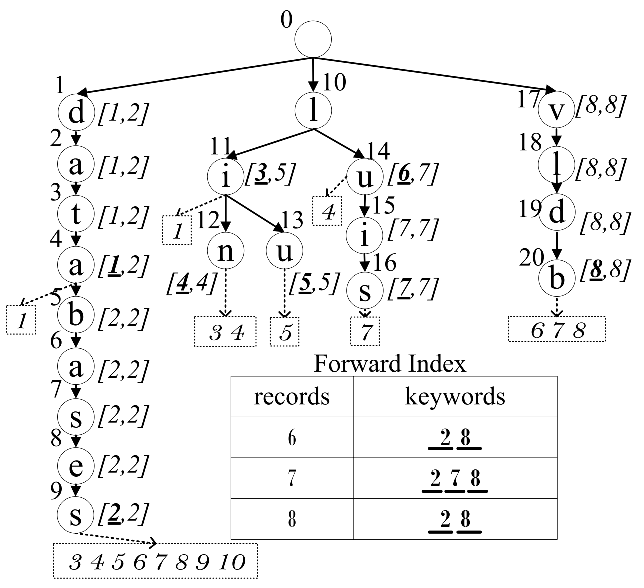
ID	Record
1	EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data. Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, Lizhu Zhou. SIGMOD, 2008.
2	BLINKS: Ranked Keyword Searches on Graphs. Hao He, Haixun Wang, Jun Yang, Philip S. Yu. SIGMOD, 2007.
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.

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. The *edit distance* between two strings s_1 and s_2, denoted by $\mathsf{ed}(s_1, s_2)$, is the minimum number of edit operations (i.e., insertion, deletion, and substitution) 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.[4] 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" (i.e., their edit distance is 1, which is within the threshold $\tau = 1$).

A search-as-you-type based system works as follows. 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 server. The sever answers queries based on the following components. The Indexer component indexes the data as a trie structure with inverted lists on leaf nodes (Section 2.1). For each query, Searcher checks whether the query can be answered using the cached results (Section 2.2). If not, Searcher answers the query using the cache and indexes, and caches the results for answering future queries. Ranker ranks results to return the best answers (Section 2.2).

2.1 Indexer

We use a trie to index the words in the data. Each word w 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 word ID for the corresponding word. The word ID is assigned in the pre-order. Each node maintains the range of word IDs in its subtree: $[minKeyID, maxKeyID]$, and the word IDs of leaf nodes under this node must be in $[minKeyID, maxKeyID]$, and vice versa. For each leaf node, we store an inverted list of record IDs that contain the corresponding word[5]. In order to improve search performance, optionally we can also maintain a forward index for the records. For each record, the forward index keeps the sorted word IDs in the record. For instance, Figure 2 shows a partial index structure for publication records in Table 1. The word "`luis`" has a node ID of 16. Its word ID is 7 and its inverted list includes record **7**. The word ID of node 11 is 3. The word range of node 11 is [`3,5`]. That is the IDs of words starting with "`li`" must be in [`3,5`]. The forward list of record 7 includes word IDs 2, 7, and 8.



records	keywords
6	2 8
7	2 7 8
8	2 8

Figure 2: The trie structure and the forward index.

[4]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.

[5]In the literature a common "trick" to make sure each leaf node on a trie corresponds to a word and vice versa is to add a special mark to the end of each word. For simplicity we do not use this trick in the figure, and a leaf node refers to a word in the data.

2.2 Searcher

2.2.1 *Exact Search*

Consider a single-keyword query $c_1c_2 \ldots c_x$, in which each c_j is a character. Let $p_i = c_1c_2 \ldots c_i$ be a prefix query ($1 \leq i \leq x$). Suppose n_i is the trie node corresponding to p_i. After the user types in a prefix query p_i, we store node n_i for p_i. For each keystroke the user types, for simplicity, we assume that the user types in a new character c_{x+1} at the end of the previous query string.[6] To incrementally answer the new query, we first check whether node n_x that has been kept for p_x has a child with a label of c_{x+1}. If so, we locate the leaf descendants of node n_{x+1}, and retrieve the corresponding complete words. Finally we compute the union of inverted lists of complete words using a heap-based merge algorithm. The union is called the *union list* of this keyword. Obviously the union list is exactly the answer. For instance, assuming a user has typed "`l`" and types in a character "`i`," we check whether node 10 ("`l`") has a child with label "`i`." We find node 11, retrieve complete words ("`li`, `lin`, `liu`"), and compute answers (records `1`, `3`, `4`, `5`).

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 $p = c_1c_2 \ldots c_id_{i+1} \ldots d_y$. 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.

For a multi-keyword query, we first compute the union list of each keyword, and then intersect these union lists to compute the results. We can use two methods to compute the results. The first one is to use a merge-join algorithm to intersect the (pre-sorted) lists. Another method is to check whether each record on the shortest lists appears on other lists by doing a binary search. The latter method has been shown to achieve a higher performance in our experiments.

2.2.2 *Fuzzy Search*

In the case of exact search, there exists only one trie node corresponding to a partial keyword k_j. However, to support fuzzy search, there may be *multiple* prefixes similar to the keyword. We call the nodes of these similar prefixes the *active nodes* for keyword k_j. Thus for a single-keyword query, we first compute its active nodes, and then locate the leaf descendants of the active nodes. Finally we compute the *union list* of this keyword by computing union of inverted lists of all such leaf descendants. Obviously the union list is exactly the answer of this keyword. For example, consider the trie in Figure 2. Suppose $\tau = 1$ and a user types in a keyword "li." The words "li," "lin," "liu," "lu," and "lui" are all similar to the keyword, since their edit distances to "li" are within a threshold $\tau = 1$. Thus nodes 11, 12, 13, 14, and 15 are active nodes. We find the leaf descendants of the active nodes as the similar complete words ("li," "lin," "liu," "lu," and "luis"). We compute the union of inverted lists of these complete words as answers (records 1, 3, 4, 5, and 7).

Next we study how to incrementally compute active nodes for a keyword as the user types in letters. Given a keyword k_j, different from exact search which keeps only one trie node, we store a set of active nodes. We compute k_j's active-node set based on that of its prefix. The idea behind our method is to use the prefix pruning. That is, when the user types in one more letter after k_j, only the descendants of the active nodes of k_j could be active nodes of the new query, and we need not consider other trie nodes. We use this property to incrementally compute the active-node set of a new query, and refer to [13] for more details.

For a multi-keyword query, we first compute union list of each keyword, and then intersect the union lists to compute the answers. Note that as a keyword may have many active nodes and large numbers of complete words, if the sizes of these lists are large, it is computationally expensive to compute these union lists. Various algorithms can be adopted here. Specifically, we can use two methods. The first one is to use a merge-join algorithm to intersect the lists, assuming these lists are pre-sorted. Another method is to check whether each record on the short lists appears on other long lists by doing a binary search. The second method has been shown to achieve a high performance [13]. Figure 3 (a) illustrates an example in which we want to answer query "`li database vld`" using the first-union-then-intersection method.

To improve the performance, we propose a forward-index based method, which only computes the union list for a single keyword. We choose the keyword with the shortest union list, and only compute its union list. We use the forward index to check whether each candidate record on the shortest union list contains similar prefixes of other keywords. If so, this record is an answer. For each of other keywords, for the word range of each of its active node, for example $[l, u]$, we check whether the candidate record contains words in $[l, u]$. We use a binary-search method to find the word ID in the corresponding forward list, and get the smallest word ID on the list that is larger than or equal to l. Then we check whether the word ID is smaller than u. If so, this candidate contains a word in $[l, u]$, that is the record contains a prefix similar to the keyword. Thus we can use this method to compute the answer. Figure 3 (b) illustrates the forward-list-based method to answer query "`li database vld`".

2.3 Ranker

In order to compute high-quality results, we need to devise a good ranking function to find the best answers. The function should consider various factors such as the edit distance between an active node and its corresponding query keyword, the length of the query keyword, the weight of each attribute, and inverted document frequencies. If edit distance dominates the other parameters, we want to compute the answer with smaller edit distances. If there are no enough top answers with edit distance τ, we then compute answers with an edit distance $\tau + 1$, and so on. Thus, when computing the union lists, we always first compute those of the active nodes with smaller edit distances. If there are enough top answers in the intersection list of such union lists,

[6]In the general case where the user can modify the current query arbitrarily, we find the cached keyword that has the longest prefix with the input keyword, and use the same method to incrementally compute the answers.

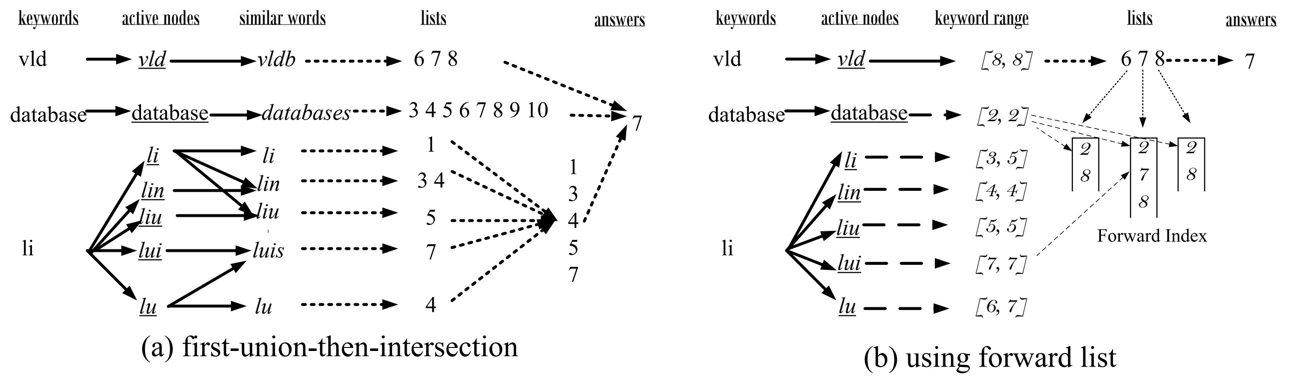


Figure 3: Two methods for answering a keyword query "`li database vld`"

we can do an early termination. We can also develop Fagin algorithms [7] to efficiently compute the top-k answers.

2.4 Additional Features

We have implemented two prototypes for keyword search on PubMed and DBLP. Figure 1(a) shows a screenshot on PubMed. In addition to the features of search-as-you-type and tolerating errors, we demonstrate the following features.

Highlighting Similar Prefix: We show how to highlight a prefix in the results that best matches a keyword. Highlighting is straightforward for the case of exact matching, since each keyword must be a prefix of the matching word. For the case of fuzzy matching, a query keyword may not be an exact prefix of a similar word. Instead, the query keyword is just similar to some prefixes of the complete word. Thus, there can be multiple similar words to highlight. 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 "`luis`," such as "**lu**is" or "**luis**." We highlight the longest matched one ("**luis**").

Using Synonyms: We can utilize a-priori knowledge about synonyms to find relevant records. For example, in the domain of person names, "`William` = `Bill`" is a synonym. Suppose in the underlying data, there is a person called "`William Kropp`." If a user types in "`Bill Cropp`," we can also find this person. To this end, on the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`," and vise versa. When a user types in "`Bill`," in addition to retrieving the relevant records for "`Bill`," we also identify those of "`William`" following the link. In this way, our method can be easily extended to utilize synonyms.

3. IMPROVING FORM-BASED SEARCH

Note that keyword search cannot support aggregation queries, and form-based search can address this problem. This section improves form-based search by supporting *search-as-you-type* and *faceted search*. To allow users to search on different attributes, we partition the original table into several *local tables*. A local table stores the distinct values of an attribute. Each record in a local table is called a *local record*, and is assigned with a *local id*. Accordingly, the original table is called the *global table*, in which each record is called a *global record* and is assigned with a *global id*. We associate each local table with one or more input boxes in the form. For each query triggered by a keystroke in an input box, the system returns to the user not only the global ids (called the *global results*), but also the matched local ids in the corresponding local table (called the *local results*). For example, in Figure 11(b), if we type in keywords "`wei wang`" in the Author input box, the system returns the names of matched authors below the form (local results), such as `Wei Wang` and `Weixing Wang`, and their publications on the right side (global results). Next we extend the architecture for search-as-you-type to support form-based search.

3.1 Indexer

We first read the global table stored in the disk and split it into local tables. For each local table, we tokenize each record into words, and build the following index structures.

1. A trie structure with inverted lists on the leaf nodes. In the trie structure, a path from the root to a leaf corresponds to a word. The local ids for the word are added to the inverted list of the corresponding leaf node. These structures are used to efficiently retrieve the local-id lists according to query keywords.
2. A local-global mapping table. This table is used to map a local id to its corresponding global ids, so that we can retrieve the global results based on the local results. The ℓ-th row of the mapping table stores the ids of all the global records containing the ℓ-th local record. Given a set of local ids, we can obtain the corresponding global ids using this table. Take Figure 11(b) as an example. The local result "`Wei Wang`" has a local id. Its corresponding global records are the first and third publications. These two global results can be retrieved using the local-global mapping table.
3. A global-local mapping table. This table is used to map a global id to its local ids, so that we can get the local results based on the global results. The g-th row of the table stores the ids of all local records contained in the g-th global record. This table is used for the *synchronization* operations which are necessary as the local results are also affected by other attributes. For example, when the focus of input boxes is changed, we need to retrieve the correct local results of the focus based on the current

global results. For instance, in Figure 11(b), when the focus is changed to Title from Author, we need to update the local results of Title based on the global results using this mapping table.

3.2 Searcher

A query of a form-based interface can be segmented into a set of *fields*, each of which contains the query string of the corresponding input box. When a query is submitted, the system first checks whether the query can be answered from the cached results. If the query can be obtained by extending a field of a cached query with one or more letters, then we have a cache hit. We call this cached query the *base query* and cached results the *base results*. The Searcher performs an incremental search based on *base results* if there is a cache hit. Otherwise, we do a basic search as follows.

Basic search. When we cannot find cached results to answer the query, we split the query into a sequence of sub-queries, in which each query appends a word to the previous query. Thus the sequence starts from an empty query and ends with the issued query. The final results can be correctly calculated if we use each of these queries one by one as the input of the *incremental search* algorithm (described below). For example, if a user inputs "`jiawei han`" in the Author input box and none prefix of the query is cached, we split it into three sub-queries, ϕ, `jiawei`, `jiawei han`. We send them one by one to the incremental-search algorithm.

Incremental search. This type of search uses previously cached results to answer a query.

Step 1. Identify the difference between the base query and the new query. We use f_i to denote the currently edited field (the i-th field), and use w to denote the newly appended keyword.

Step 2. Calculate the local ids of f_i based on the query string in f_i, by first merging the id lists of all leaf descendants of the trie node corresponding to keyword w and then intersecting the merged list with the local base results of f_i.

Step 3. Compute the global results, by first calculating the set of global ids corresponding to the local results of f_i (Step 2) using the local-global mappings and then intersecting the set with the global base results.

Step 4. Calculate the local results of f_i, called "synchronization," by first calculating the set of local ids corresponding to the global results using the global-local mapping table and then intersecting it with the local base results of f_i.

Note that we need to calculate the aggregations of each local result. Using the local-global mapping table, we can easily calculate the number of occurrences of a local result in the global result. For example, in Figure 11(b), the number "13" on the right of the entry "`Wei Wang`" means that "`Wei Wang`" appears 13 times in the global results. Next we discuss how to rank the local results and global results so as to return top-k answers. The design of the ranking function depends on the application. For example, we can rank the results according to the values of an attribute, e.g., Year or Rating. Ranking functions will be added in the final paper.

3.3 Improvements

In step 3, to obtain the global results, we map the local ids calculated in step 2 to lists of global ids, merge these lists, and then intersect the merged list with the global base results. The number of lists to be merged is equal to the number of local ids. If there are many local ids, the merge operation could be very time consuming. To address this problem, we propose another index *dual-list tries* by attaching an inverted list of global ids to each of the corresponding trie leaf nodes. In this way, given a keyword prefix, we can identify the global record that contain the keyword without any mapping operation. In addition, the number of lists to be merged is the number of complete words, which is often much smaller than the number of local ids. A smaller number of lists leads to faster merge operations. Using dual-list tries, the overall search time can be reduced compared with using original tries (called *single-list tries*).

Since we can identify the global ids using dual-list tries, those local-global mapping tables are no longer needed. In addition, we propose an alternative method to calculate the aggregations without using those local-global mapping tables. Given both local results and global results, we assign each local result a counter, initialized as 0. We first map each global id back to a list of local ids using the global-local mapping table. Then, if a local result appears in the mapped list, its corresponding counter, i.e., its occurrence number, is increased by 1. Using this method to compute the aggregations, the local-global mapping tables are not needed and we can remove them to reduce the index size.

3.4 Additional Features

We have implemented two prototypes for form-based search on DBLP and IMDB datasets. Figure 1(b) shows a screenshot of a system on the DBLP dataset. We describe the following main features of our form-based search systems.

Precise search paradigm: Suppose a user wants to find papers written by `Wei Wang` whose titles contain the word `pattern`. If she types in "`wei wang pattern`" in keyword-search system, many returned results are not very relevant. In contrast, if she types in `wei wang` and `pattern` in different input boxes in our system, she can find high-quality results.

Search-as-you-type: Suppose the user wants to find the movie titled `The Godfather` made in 1972 using the IMDB Power Search interface. She is not sure if there is a space between the word `god` and the word `father`, so she fills in the Title input box with `god father`. Unfortunately, she can not get relevant result. So she has to try several new queries. In contrast, in our system, she can modify the query and easily find the results.

Faceted search and aggregation:. Suppose a user has limited prior knowledge about the KDD conference and wants to know more about it. At first, she wants to know how many papers were published in this conference each year. She types in `kdd` in the Venue input box and then changes the editing focus to the Year input box. The listed local results show the years sorted by the number of published papers. Next, she wants to know the number of published papers of each author in KDD 2009. She chooses the year 2009 by clicking on the list, and changes the focus to the Author input box. The list below the form shows the authors, and she can see the most *active* authors. After several rounds of typing and clicking, she can get a deeper understanding about the KDD conference.

4. IMPROVING SQL-BASED SEARCH

SQL-based method is more powerful than keyword search and form-based search, and in this section we improve SQL-based search to combine the user-friendly interface of keyword search and the power of SQL. As users type in keywords, we on-the-fly suggest the top-k relevant SQL queries based on the keywords, and users can select SQL queries to retrieve the corresponding answers. For example, consider a database with tables "`paper, author, write`" in Table 2, where "`paper`" contains paper information (e.g., title), "`author`" contains author information (e.g., author name), and "`write`" contains paper-author information (e.g., which authors write which papers). Suppose an SQL programmer types in a query "`count database author`." We can suggest the following SQL queries.

```
1)  SELECT      COUNT( P.title ), A.name
    FROM        PAPER AS P, WRITE AS W, AUTHOR AS A
    WHERE       P.title CONTAIN "database" AND
                P.id = W.pid AND A.id = W.aid
    GROUP BY    A.name

2)  SELECT      P.title, A.name
    FROM        PAPER AS P, WRITE AS W, AUTHOR AS A
    WHERE       P.title CONTAIN "database" AND
                P.title CONTAIN "count"
                P.id = W.pid AND A.id = W.aid
```

where CONTAIN is a user-defined function (UDF) which can be implemented using an inverted index. The first SQL is to group the number of papers by authors, and the second one is to find a paper as well as its author such that the title contains the keywords "`database`" and "`count`" . We can provide graphical representation and example results to each suggested SQL as shown in Figure 1(c). The user can refine keywords as well as the suggested queries to obtain the desired results interactively.

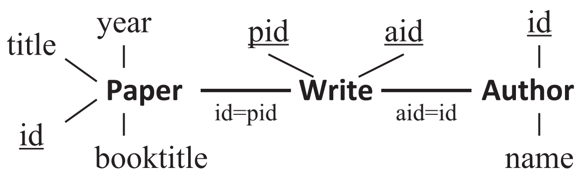
Compared with Candidate Networks (CNs) [18, 4, 12, 10] which only generate SPJ (Selection-Projection-Join) queries, our method has the following advantages. (1) We can not only suggest SPJ queries, but also support aggregate functions. (2) We can group the results by their underlying query structures, rather than mixing all results together. Our SQL-suggestion method has the following unique features. Firstly, it helps various users (e.g., administrators, SQL programmers) formulate (even complicated) structured queries based on limited keywords. It can not only reduce the burden of posing queries, but also boost SQL coding productivity significantly. Secondly, it helps users express their query intent more precisely than keyword search and form-based search, especially for complex queries. Thirdly, compared with keyword search, our method has performance superiority as we only need to first generate SQL queries and then return answers after users select an SQL query.

4.1 Overview

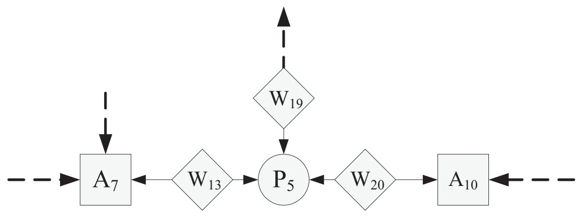
Our work focuses on suggesting SQL queries for a relational database $\mathcal{D}$ with a set of relation tables, $R_1, R_2, \ldots, R_n$, and each table R_i has a set of attributes, $A_1^i, A_2^i, \ldots, A_m^i$. To represent the schema and underlying data of $\mathcal{D}$, we define the schema graph and the data graph respectively.

To capture the foreign key to primary key relationships in the database schema, we define the *schema graph* as an undirected graph $G_S = (V_S, E_S)$ with node set V_S and edge set E_S: 1) each node is either a relation node corresponding to a relation table, or an attribute node corresponding to an attribute; 2) an edge between a relation node and an attribute node represents the membership of the attribute to the relation; 3) an edge between two relation nodes represents the foreign key to primary key relationship between the two relation tables.

Similarly, we define the *data graph* to represent the data instances in the database. The data graph is a directed graph, $G_D = (V_D, E_D)$ with node set V_D and edge set E_D, where nodes in V_D are data instances (i.e., tuples). There exists an edge from node v_1 to node v_2 if their corresponding relation tables have a foreign key (v_1) to primary key (v_2), and the foreign key of v_1 equals to the primary key of v_2.



(a) Schema graph.



(b) Data graph.

Figure 4: Schema graph and data graph [8].

Table 2 provides an example database containing a set of publication records. The database has three relation tables, PAPER, AUTHOR, and WRITE, which are respectively abbreviated to P, A and W in the rest of the paper for simplicity. Figure 4(a) shows the schema graph of the example database. In the graph, the relation PAPER has four attributes (i.e., id, title, booktitle, and year), and has an edge to another relation WRITE. Figure 4(b) shows the data graph. In this graph, an instance of PAPER (i.e., P_5) is referred by three instances of WRITE (i.e., W_{13}, W_{19}, and W_{20}).

We focus on suggesting a ranked list of SQL queries from limited keyword queries. Note that the query keywords can be very flexible that they may refer to either data instances, the meta-data (e.g., names of relation tables or attributes), or aggregate functions (e.g., the function COUNT). Formally, Given a keyword query $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$, the answer of Q is a list of SQL queries, each of which contains all keywords in its clauses, e.g., the WHERE clause, the FROM clause, the SELECT clause, or the GROUP-BY clause, etc. Since there may be many SQL queries corresponding to a keyword query, we propose to rank SQL queries by their relevance to the keyword query. For example, consider the example database in Table 2 and a keyword query "`count database author`" . We can suggest two SQL queries as follows.

Table 2: An example database (Join Conditions: Paper.id = Write.pid and Author.id = Write.aid).

(a) PAPER

id	title	booktitle	year
P_1	database ir	tois	2009
P_2	xml name count	tois	2009
P_3	evaluation	database theory	2008
P_4	database ir	database theory	2008
P_5	database ir xml	ir research	2008

(b) AUTHOR

id	name
A_6	lucy
A_7	john ir
A_8	tom
A_9	jim
A_{10}	gracy

(c) WRITE

id	pid	aid	id	pid	aid
W_{11}	P_2	A_6	W_{16}	P_3	A_6
W_{12}	P_1	A_7	W_{17}	P_4	A_7
W_{13}	P_5	A_7	W_{18}	P_4	A_8
W_{14}	P_2	A_9	W_{19}	P_5	A_9
W_{15}	P_3	A_{10}	W_{20}	P_5	A_{10}

```
1)  SELECT      COUNT( P.id ), A.name
    FROM        P, W, A
    WHERE       P.title CONTAIN “database” AND
                P.id = W.pid AND A.id = W.aid
    GROUP BY    A.id

2)  SELECT      P.title, P.booktitle, A.name
    FROM        P, W, A
    WHERE       P.title CONTAIN “database” AND
                P.title CONTAIN “count” AND
                P.id = W.pid AND A.id = W.aid
```

Observed from the above SQL queries, the first one is to group the number of papers with titles containing “`database`” by authors, and the second one is to find a paper as well as its author such that the title contains the keywords, “`database`” and “`count`”.

As query keywords may refer to either data instances of different relation tables, relation or attribute names, or aggregate functions, there could be large numbers of possibly-relevant SQL queries. Thus it is a challenge to suggest SQLs based on keywords. To suggest the best SQL queries, we propose a two-step method: (1) Suggest query structures, called *queryable templates* (“template” for short), ranked by their relevance to the keyword query; (2) Suggest SQL queries from each suggested template, ranked by the degree of matchings between keywords and attributes in templates.

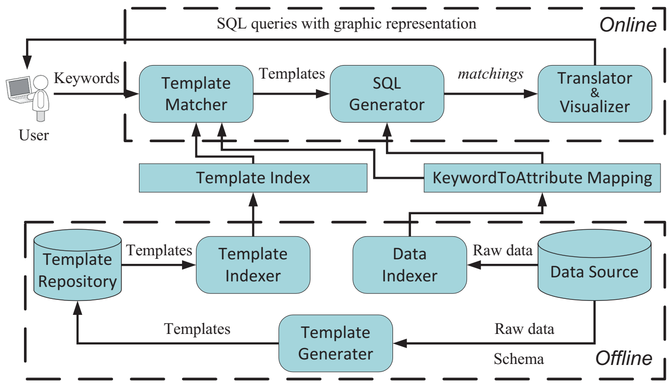
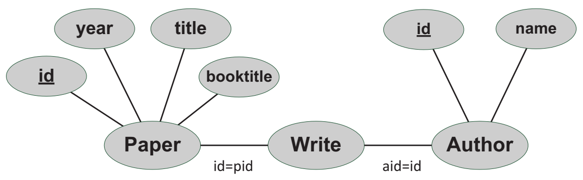
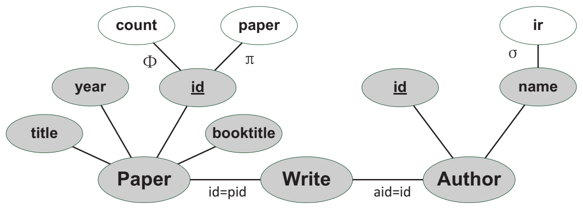


Figure 5: Architecture of SQL Suggestion [8].

Figure 5 shows the architecture of an SQL-suggestion-based system. Template Generator generates templates from Data Source and stores them in Template Repository. As there may be many templates, especially for complex schema, Template Indexer constructs Template Index for efficient on-line template suggestion. Since keywords may refer to data instances, meta-data, functions, etc., Data Indexer builds KeywordToAttribute Mapping for mapping keywords to attributes. Given a keyword query, Template Matcher suggests relevant templates based on the index structures. Then SQL Generator generates SQL queries from suggested templates by matching keywords to attributes. Translator & Visualizer shows SQL statements with graphical representation.



(a) A suggested template.



(b) A matching between keywords and attributes.

Figure 6: An example for query “`count paper ir`”.

4.2 Template Suggestion

Template. The template is used to capture the structures of SQL queries, e.g., which entities are involved in SQL queries and how the entities are joined together. A template is an undirected graph, where nodes represent entities (e.g., `Paper`) or attributes (e.g., `year`). An edge between entities represents a foreign-primary relationship (e.g., `Paper.id=Write.pid`), and an edge between an entity and an attribute represents that the entity has the attribute. Formally, a template is defined as follows.

DEFINITION 1 (QUERYABLE TEMPLATE). *A template is an undirected graph, $G_T(V_T, E_T)$ with the node set V_T and the edge set E_T, where nodes in V_T are:*

- entity nodes: *relation tables, or*
- attribute nodes: *attributes of entities,*

and edges in E_T are:

- membership edges: *edges between an entity node and its attribute nodes, or*
- foreign-key edges: *edges between entity nodes with foreign keys and the entity nodes referred by them.*

*In particular, templates with only one entity node are called **atomic-templates**.*

For example, Figure 6(a) shows a template which involves three entities (`Paper`, `Author` and `Write`) and their attributes. Compared with CNs (trees of joined entities) [12, 11], the templates can be generated and indexed offline for fast template suggestion, and can effectively capture the relevance between keyword queries and structures.

Template Generation. Given a database, there could be a huge amount of templates that capture various query structures. We design a method to generate them using the schema graph. The basic idea of the method is to *expand* templates to generate new templates. To this end, the algorithm firstly generates atomic-templates, and takes them as bases of template generation. Then, we use an *expansion rule* to generate new templates: A template can be expanded to a new template if it has an entity node that can be connected to a new entity node via a foreign-key edge. For example, consider an atomic template, P. It can be expanded to P−W according to the expansion rule. Since a template may be expanded from more than one template, we eliminate duplicated templates. Furthermore, we examine the relationship between relation tables. For example, since the two relation tables, P and W, have a *1-to-n* relationship, i.e., an instance of W has at most one instance of P. Hence, the template, P−W−P is invalid.

Table 3: Templates for our example database (γ=5).

Size	ID	Template
1	T_1	P
	T_2	A
	T_3	W
2	T_4	P − W
	T_5	A − W
3	T_6	P−W−A
	T_7	W−P−W
	T_8	W−A−W
4	T_9	P−(W,W,W)
	T_{10}	A−(W,W,W)
	T_{11}	W−P−W−A
	T_{12}	W−A−W−P
5	T_{13}	P−W−A−W−P
	T_{14}	A−W−P−W−A
	...	...

Apparently, the above-mentioned expansion rule may lead to a combinatorial explosion of templates. To address this problem, we employ a parameter γ to restrict the maximal size of templates (i.e., the number of entities in a template), which is also used in CN-based keyword-search approaches [12, 11]. The motivation here is that templates with too many entities are meaningless, since the corresponding query structures are not of interests to users. Table 3 provides the templates generated for our example database under $\gamma = 5$, where P−(W,W,W) represents that P is connected with three W entities. Even if we restrict the maximal size of templates, there still are many templates due to complex relationships between tables. We can devise an efficient top-k template ranking algorithm to avoid exploring the entire space of searching templates.

Template Suggestion Model. We propose a scoring function to measure the relevance between keywords and templates, which considers two factors: 1) the relevance between a keyword q and the entity R in a template T, denoted by $Pr(q, R)$, and 2) the importance of the entity R, denoted by $I(R)$. $Pr(q, R)$ can be taken as the probability that the entity R contains the keyword q. We use the relative frequency that R contains q to estimate this probability. We treat the entity R as a *virtual* document and estimate the likelihood of sampling keyword q from the "document" using term frequency and inverse document frequency. In particular, the virtual document R does not only have words in tuples, but also contains words in meta-data (e.g., attribute names, entity names, etc.). We use the number of tuples to estimate $Pr(q, R)$, if q refers to the meta-data of R. The importance of an entity R in a template T, $I(R)$, is computed using a graph model. We compute the *PageRank* of tuples on the data graph [18], and take the average value as template importance. Given a query Q and a template T, we combine the two factors to evaluate its score $S_1(Q, T)$:

$$S_1(Q, T) = \sum_{q \in Q} \sum_{R \in T} I(R) \cdot Pr(q, R). \qquad (1)$$

Top-k Ranking Algorithm. As the number of templates could be very large for complex schemas, it is expensive to enumerate the templates. A straightforward way to address this problem is to calculate the ranking score for every template according to our template ranking model. Unfortunately, since the number of templates is exponential, this approach becomes impractical for real-world databases. Therefore, an efficient top-k ranking algorithm is rather necessary to avoid exploring all possible templates. To address this problem, we devise a threshold algorithm (TA) [7] to compute top-k templates efficiently.

Our basic idea is to scan multiple lists that present different rankings of templates for an entity, and aggregate scores of multiple lists to obtain scores for each template ($Pr(q, R)$). For early termination, the algorithm maintains an upper bound for the scores of all unscanned templates. If there are k scanned templates whose scores are larger than the upper bound, the top-k templates have been found. Interested readers are referred to [8] for more details.

4.3 SQL Generation from Templates

Given a query and a template, to generate SQLs, we need to map each keyword to an attribute in the template. This section proposes an SQL-suggestion model to suggest SQLs.

SQL Generation Model. To map keywords to a template, we construct a set of keyword-to-attribute mappings between keywords and attributes. A *matching* is a set of mappings which covers all keywords. Then we evaluate the score of each matching, find the best matching with the highest score, and generate SQL queries based on the matching.

SQL Ranking. We measure the score of each matching by considering two factors: 1) the degree of each keyword-to-attribute mapping in the matching; and 2) the importance of the mapped attributes. The degree of a mapping from a keyword q to an attribute A with a type t, denoted as $\rho_t(q, A)$, is used to determine whether q is relevant to A. We consider three types of keyword-to-attribute mappings, i.e., 1) the *selection* type (denoted as σ): q refers to data instances of A; 2) the *projection* type (denoted as π): q refers to the name of A; 3) the *aggregation* type (denoted as ϕ): q refers to an aggregate function of A. We use the relative frequency that A contains q to estimate $\rho_\sigma(q, A)$ and $\rho_\pi(q, A)$. $\rho_\phi(q, A) = 1$ if q is a function name (e.g., COUNT, MAX).

We employ the *entropy* to compute the importance of an attribute A, $I(A)$. Let $V = \{v_1, v_2, \ldots, v_x\}$ denote distinct values of A and let f_i denote the relative frequency that A has value v_i, and $I(A) = -\sum_{i=1}^{x} f_i \cdot \log f_i$.

Combining the two factors, we present the scoring function as follows. Consider a keyword query $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ and a set of attributes $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ in a suggested template T. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_{|\mathcal{M}|}\}$ be a matching between Q and T, where $M \in \mathcal{M}$ is a keyword-to-attribute mapping of keyword q_M and attribute A_M. The score is

$$S_2(\mathcal{M}) = \sum_{M \in \mathcal{M}} I(A_M) \cdot \rho_t(q_M, A_M). \tag{2}$$

Thus we first find the best matching (i.e., an SQL query) from a template, which can be formulated as a *weighted set-covering problem* and can be solved using a polynomial-time greedy approximation algorithm. Then, we extend the algorithm to generate top-k SQL queries by finding k best matchings with highest scores in a greedy manner. Interested readers are referred to [8] for more details.

In summary, given a keyword query, we first find relevant templates for the keywords using the threshold-based algorithms. Then for each template, we generate corresponding SQL queries using the keyword-to-attribute mappings.

4.4 Additional Features

We have implemented two prototypes for SQL-based search on DBLP and DBLife datasets. For example, Figure 1(c) shows a screenshot of a system on the DBLP dataset. We describe the main features as follows.

Predicting the query intent from limited keywords: Unlike existing SQL assistant tools and keyword-search methods, DBEASE focuses on *predicting* the query intent from keywords and *translating* them into structured queries. Suppose a user wants to find a paper with *title* containing "`data`" and *booktitle* containing "`icde`" which is written by an author "`wang`." After she issues a keyword query "`wang data icde`" to the system, the system returns SQL queries, user-friendly representation, and sample answers instantly. This search paradigm is more expressive and powerful than the conventional keyword search.

Assisting users to formulate SQL queries efficiently: To reduce the burden of writing SQL queries without the loss of expressiveness, we propose an effective ranking mechanism to suggest the most relevant SQL queries to users. Suppose that a user issues a query "`count person stanford`," she would be presented with the following SQL queries: (*i*) Find the number of people whose organization is `Stanford`; (*ii*) Find the number of people who give talks to `Stanford`; (*iii*) Find the number of people who are related to `Stanford`. The above SQL queries are useful for users to navigate the underlying data, and can reduce the burden of posing queries from several complex SQL statements to 3 keywords.

Improving the effectiveness of data browsing: Users can browse the data by typing keywords and selecting suggested SQL queries. In our systems, the answers are naturally organized based on their corresponding SQL queries. Each SQL query represents a group of records with the same structure, which could help users find similar and relevant records in one group and browse other potentially relevant records in other groups.

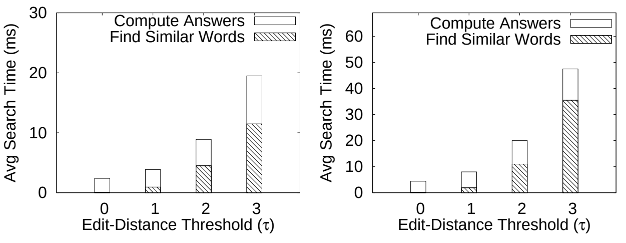
5. EXPERIMENTAL RESULTS

We have implemented our techniques on several real datasets, PubMed[7], DBLP[8], IMDB[9], and DBLife[10].

All the codes were implemented in C++. We conducted the experiment on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz CPU and 4 GB RAM.

5.1 Improving Keyword Search

We evaluated the performance of search-as-you-type by varying the edit-distance threshold τ. We used two datasets: DBLP and PubMed. We selected 1000 queries from query logs from our deployed systems. 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 7 shows the results. Our methods could answer a query within 50 ms. The variant of our method was about 8 ms.



(a) DBLP (b) MEDLINE

Figure 7: Efficiency of search-as-you-type.

5.2 Improving Form-based Search

We evaluated the performance of form-based search. We used two datasets: DBLP and IMDB. We used a workload of 40 thousand queries collected from our deployed system. Figure 8 shows the comparison of average search time per query of four algorithms: (1) SL-BF, which uses Single-List tries and Brute-Force synchronization (do synchronization for each keystroke), (2) SL-OD, which uses Single-List tries and On-Demand synchronization (do synchronization when search focus is changed), (3) DL-BF, which uses Dual-List tries and Brute-Force synchronization, and (4) DL-OD, which uses Dual-List tries and On-Demand synchronization.

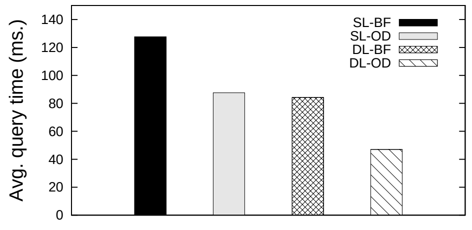


Figure 8: Efficiency of form-based search (IMDB).

We can see that both the dual-list tries and on-demand synchronization can improve the search speed. If we use these two together, the DL-OD algorithm can answer a query within 50 milliseconds. Moreover, compared with search-as-you-type, this method supports on-the-fly faceted search.

[7]http://www.ncbi.nlm.nih.gov/pubmed
[8]http://dblp.uni-trier.de/xml/
[9]http://www.imdb.com/interfaces
[10]http://dblife.cs.wisc.edu/

5.3 Improving SQL-based Search

We evaluated effectiveness and efficiency of our method (SQLSugg) and compared with Discover-II [11]. We used two datasets: DBLP and DBLife. We selected ten queries for each dataset [8]. We evaluated the precision of template suggestion (Figure 9(a)) and that of returned answers (Figure 9(b)) on the DBLife dataset. We see that our method outperforms Discover-II significantly. For example, the precision of our method is much better than that of Discover-II. The improvement of our method is due to the following reasons. Firstly, compared with Discover-II, our method allows users to search the meta-data (i.e., names of relation tables, or attributes), while Discover-II only supports full-text search. Secondly, we group the answers based on structures and employ a more effective ranking function.

We examined the efficiency of our SQL suggestion methods, and compared the query time with Discover-II. Figure 9(c) shows the experiment results on the DBLife dataset. The results show that our methods outperform Discover-II significantly. For example, consider the query time for a keyword query with length 6 in Figure 9(c). Our method outperforms Discover-II by an order of magnitude. Moreover, the query time of our method is very stable, and is always smaller than 100 milliseconds. It indicates that our method can suggest SQL queries in real time.

The improvement of our methods is mainly attributed to the top-k template ranking algorithm. Discover-II exploits a keyword-to-attribute index to find tuple sets that may contain query keywords, and on-the-fly generates candidate networks. Since the amount of candidate networks could be large, especially for complex schemas, Discover-II is inefficient to generate all candidate networks. For example, the query time of Discover-II is hundreds of milliseconds on the DBLife dataset with 14 tables. In contrast, our method focuses on suggesting top-k templates according to our ranking model. The result shows that the algorithm is very efficient and can suggest template in real time.

5.4 Scalability

We tested the scalability of our methods for different search paradigm. Figure 10 shows the scalability of efficiency of our methods. We see the search time increased linearly as the dataset increased. All average search time is less than 60 milliseconds and the variant is about 10 milliseconds. The index size also increased linearly as the dataset increased, as shown in Figure 11.

6. CONCLUSION

In this paper, we have studied a new search method DBease to make databases user-friendly and easily accessible. We developed various techniques to improve keyword search, form-based search, and SQL-based search for enhancing user experiences. Search-as-you-type can help users on-the-fly explore the underlying data. Form-based search can provide on-the-fly faceted search. SQL suggestion can help various users to formulate SQLs based on limited keywords.

We believe this study on making databases user-friendly and easily accessible opens many new interesting and challenging problems that need further research investigation. Here we give some new open research problems.

Supporting Ranking Queries Efficiently: Our proposed techniques need to first compute all candidates and then rank them to return the top-k answers. If there are larger numbers of answers, these methods are expensive. To address this problem, it calls for new techniques to support ranking queries efficiently. Different from traditional search paradigm, in search-as-you-type, there are multiple corresponding complete keywords and inverted lists for each prefix keyword. Existing threshold-based methods [7] need to scan the elements in each inverted list to compute the top-k answers and they are expensive if there are large numbers of inverted lists. We have an observation that some inverted lists can be pruned if the corresponding complete keywords are not very relevant. Especially, we can prune the inverted lists of complete keywords with larger edit distances. Thus we have an opportunity to prune some inverted lists and thus improve the search performance for ranking queries.

Supporting Non-string Data Types: Our proposed techniques only support string data types and do not support other data types, such as Integer and Time. Consider the case where we have a publication database, and a user types in a keyword "2001". Based on edit distance, we may find a record with a keyword "2011". On the other hand, if we knew this keyword is about the year of a publication, then we may relax the condition in a different way with edit distance, e.g., by finding publications in a year between 1999 and 2002. This kind of relaxation and matching depends on the data type and semantics of the corresponding attribute, and requires new techniques to do indexing and searching. Similarly, to support approximate search, we also want to support other similarity functions, such as Jaccard.

Supporting Personalized Search: Different users may have different search intentions, and it is challenging to support personalized search for various users. We can utilize user query logs and mine search interests for different users so as to support personalized search.

Client Caching: We can not only support server caching, but also use client caching to improve search performance. We can cache results of users' previous queries and use the cached results to answer subsequent queries. However it is very challenging to support client cache, since it is hard to predicate subsequent queries of users. We need to study how to cache previous queries and corresponding results.

7. ACKNOWLEDGEMENT

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, and National S&T Major Project of China.

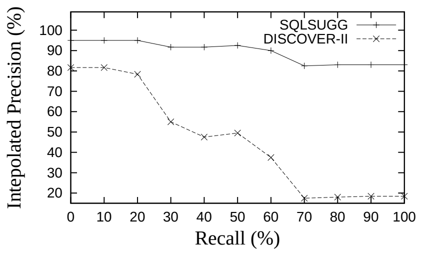
8. REFERENCES

[1] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.

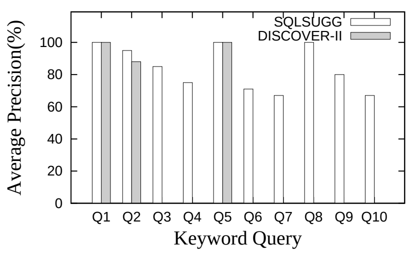
[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

[3] H. Bast and I. Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, 2007.

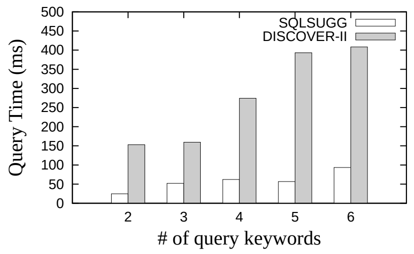
[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.



(a) Effectiveness of template suggestion.

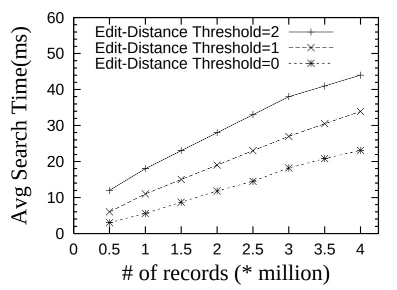


(b) Effectiveness of record retrieval.

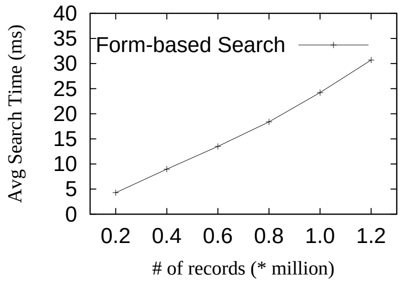


(c) Efficiency comparison.

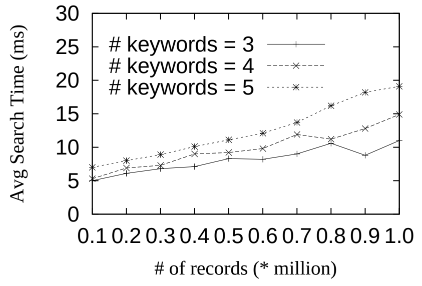
Figure 9: Evaluation of SQL-based Search on the DBLife data set.



(a) Keyword Search (PubMed)

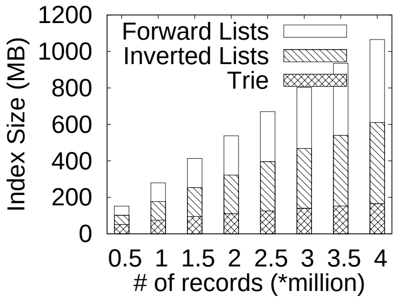


(b) Form-based Search (DBLP)

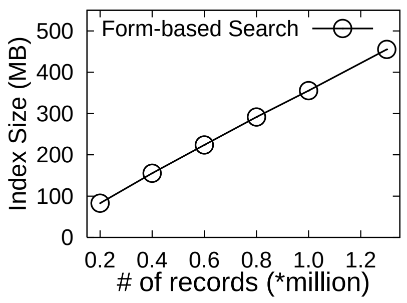


(c) SQL-based Search (DBLP)

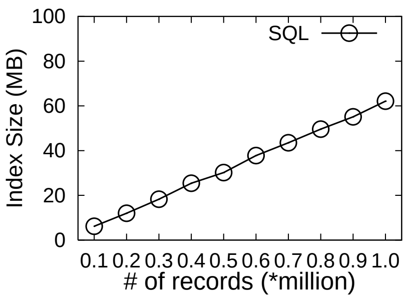
Figure 10: Scalability of search performance.



(a) Keyword Search (PubMed)



(b) Form-based Search (DBLP)



(c) SQL-based Search (DBLP)

Figure 11: Scalability of index size.

[5] M. Celikik and H. Bast. Fast error-tolerant search on very large texts. In *SAC*, pages 1724–1731, 2009.

[6] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[8] J. Fan, G. Li, and L. Zhou. Interactive sql query suggestion: Making databases user-friendly. In *ICDE*, 2011.

[9] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, 2007.

[11] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[12] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[13] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[14] K. Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[15] G. Li, J. Feng, and L. Zhou. Interactive search in xml data. In *WWW*, pages 1063–1064, 2009.

[16] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.

[17] G. Li, S. Ji, C. Li, J. Wang, and J. Feng. Efficient fuzzy type-ahead search in tastier. In *ICDE*, pages 1105–1108, 2010.

[18] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD Conference*, pages 903–914, 2008.

[19] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.

[20] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.

[21] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[22] H. Wu, G. Li, C. Li, and L. Zhou. Seaform: Search-as-you-type in forms. *PVLDB*, 3(2):1565–1568, 2010.