Seaform: Search-As-You-Type in Forms

Hao Wu [†] Guoliang Li [‡] Chen Li [§] Lizhu Zhou [‡]

[†‡] Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China

[§]Department of Computer Science, University of California, Irvine 92697, USA

[†] haowu06@mails.tsinghua.edu.cn, [‡]{liguoliang, dcszlz}@tsinghua.edu.cn, [§]chenli@ics.uci.edu

ABSTRACT

Form-style interfaces have been widely used to allow users to access information. In this demonstration paper, we develop a new search paradigm in form-style query interfaces, called *Seaform* (which stands for Search-As-You-Type in Forms), which computes answers on-the-fly as a user types in a query letter by letter and gives the user instant feedback. Seaform provides better user experiences compared with traditional form-based query systems by reducing the efforts for a user to compose a high-quality query to find relevant answers. Seaform can also enhance faceted search and allow users to on-the-fly explore the underlying data. This search paradigm requires high performance to achieve an interactive speed. We develop efficient techniques and use them to implement two systems on real datasets. We demonstrate the features of these systems.

1. INTRODUCTION

Keyword search is an easy way to access structured and semi-structured data [2]. By issuing a query with keywords in an input box, a user can get the results more easily than using traditional, yet complex, query languages, such as SQL for relational databases and XQuery for XML documents.

Although single-input-box interfaces for keyword search are easy to use, often users want an interface that allows them to specify keyword conditions more precisely. Firstly, in a relational database, a single keyword may appear in different attributes. For example, in a publication database of computer science, the word "`pattern`" in a query may appear both in a paper title and in a conference name. Secondly, multiple keywords can be in the value of a single attribute. For instance, suppose we issue a query "`wei wang`" to search for papers written by Wei Wang in a single-input-box interface, such as CompleteSearch[1] [1]. The system may treat each keyword separately and return results with a low relevance. If the user has a clear idea about the underlying semantics of her query, an easy way to formulate the query is to use a form-based interface. A form has multiple input boxes, using which a user can fill in keywords. It can also have drop-down lists for a user to make selections. Forms have been widely used to access databases [3], especially by those "Advanced Search" interfaces, such as the IMDB Power Search[2] and the PubMed Advanced Search[3].

Existing form-based systems require users to compose a complete query. However, if the user knows little information about the underlying data, she has to repeatedly refine the query, submit a new query, and check the returned results. To improve the user experience of form-based interfaces, we propose a new search paradigm, called "Seaform", to enable *search-as-you-type* in form-based interfaces.

Search-as-you-type is a user-friendly search paradigm that can reduce the efforts of users to refine their queries by returning the results instantly as they type in queries character by character. Compared with existing studies [4, 5, 6], which focus on search-as-you-type on single-input-box interfaces, Seaform allows a user to specify her keywords in multiple input boxes on a form and get the results instantly. Besides returning the matched records, Seaform also provide the matched attribute values as well. Seaform incorporates novel index structures and search algorithms to support search-as-you-type in multiple attributes. We have developed two prototype systems based on the proposed techniques to search on two datasets, the DBLP Computer Science Bibliography (DBLP), and the Internet Movie Database (IMDB). For example, Figure 1(a) shows a screenshot of a system on the DBLP dataset.

Another benefit of enabling search-as-you-type in form-based interfaces is improvement on *faceted search*. In Figure 1(a), when the user is currently editing the input box of the Author attribute, the system dynamically groups the results by authors, and lists the authors sorted by the sizes of the corresponding groups. With the help of search-as-you-type, users can explore the underlying data in real time.

2. SYSTEM OVERVIEW

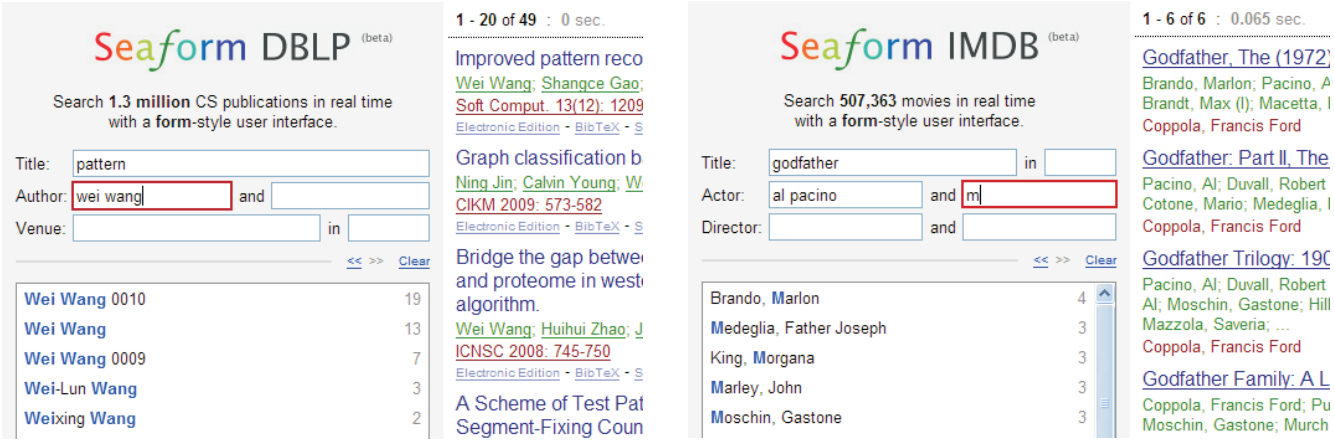
A Seaform system takes a single relational table as its underlying data. To allow users to search on different attributes, we partition the original table into several *local tables*, each for a specified attribute. A local table stores all the distinct values of the attribute. Each record in a local table is called a *local record*, and is assigned with a *local id*.

[1] `http://dblp.mpi-inf.mpg.de/dblp`

[2] `http://www.imdb.com/list`

[3] `http://www.ncbi.nlm.nih.gov/pubmed/advanced`

Proceedings of the VLDB Endowment, Vol. 3, No. 2



(a) Seaform-DBLP (b) Seaform-IMDB

Figure 1: Screenshots of two prototypes using our techniques.

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 1(a), 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-hand side (global results).

A Seaform system uses a client-server architecture. On the client side, each keystroke in any of the input boxes invokes JavaScript code to issue an AJAX query to the server, and the client displays the results returned from the server with query keywords highlighted. The server side has the following components. The Indexer indexes the underlying data. When a query is received, the Searcher searches the index for both the global results and the local results incrementally with the help of the Cache component, which caches the previous queries and their results. The Result Composer ranks the results and sends them to the client. All the components are implemented in a FastCGI module.

2.1 Indexer

When the server starts, the Indexer reads the global table stored in the disk and splits 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 id lists according to input query keywords.
2. A local-global mapping table. 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. These tables are used to map a local id to its corresponding global ids, so that we can retrieve the local results and the global results simultaneously. Take Figure 1(a) as an example. The local result "Wei Wang" has a local id, say ℓ_1. Its corresponding global records are the first and third publications (whose global ids are, say, g_1 and g_2 respectively). These two global results can be retrieved using the local-global mapping table.
3. A global-local mapping table. Similar to the table above, the g-th row of the table stores the ids of all the local records contained in the g-th global record. This table could be used to identify those local ids in a global record efficiently. These tables are used for the synchronization operations (see Section 2.2). The synchronization operations are necessary because the local results of the current editing input box are determined by both the query string of the current input box and the query strings of other input boxes. We can retrieve the correct local results by mapping the final global results back using these tables.

To achieve a high performance, these structures are assumed to be in memory, and the ids on each inverted list are sorted.

2.2 Searcher and Cache

When a query is submitted, the system first checks the Cache to see whether the query can be answered from the cached results. 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. If the new 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 results the base results. The Searcher performs an incremental search based on the base query and base results if there is a cache hit. Otherwise, we do a basic search described as follows.

Basic search. When we cannot find cached results to answer the query, we split the query into a sequence of subqueries, in which each query appends a word to the previous query. Thus the sequence starts from an empty query and ends with the original query. The final results can be cor-

rectly 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 the query into three sub-queries, ϕ, "`jiawei`", and "`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, with the following four steps.

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. This is done by merging the id lists of all leaf nodes on the sub-tree rooted at the node corresponding to w in the trie, and then intersecting the merged list with the local base results of f_i.

Step 3. Calculate the global results. This is done by first calculating the set of global ids corresponding to the local results of f_i calculated in Step 2 using the local-global mappings in the index, and then intersecting it with the global base results.

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

2.3 Result Composer

The Result Composer ranks both the local results and global results, and then returns top-ranked results to the client. The design of the ranking function depends on the application. Specifically, in our proposed systems, we simply rank the results according to the values of a specified attribute, e.g., Year or Rating. Another task of the Result Composer is to calculate the aggregations of each local result. With the help of the local-global mapping table in the index, we can easily calculate the number of occurrences of a local result in the global result list. For example, in Figure 1(a), the number "13" on the right of the entry "Wei Wang" means that "Wei Wang" appears 13 times in the global results listed on the right-hand side.

3. IMPROVEMENTS

Based on our analysis, steps 3 and 4 in the incremental-search algorithm can be computationally expensive. In this section, we present two optimization techniques to improve the above methods.

3.1 Dual-List Trie Structures

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. As a result, if there are many local ids, the merge operation could be very time consuming.

To address this problem, we can modify the original tries to so-called *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. So with the help of dual-list tries, the overall search time can be reduced compared with that of using original tries, which are called *single-list tries*.

Since we can identify the global ids efficiently, those local-global mapping tables are no longer needed by the Searcher. In addition, for the Result Composer, we have 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, which is initiated to be 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 number of occurrences, is increased by 1.

After using the above new method to calculate the aggregations, we find that the local-global mapping tables are not needed by any component of the system. As a result, we can safely remove them to reduce the index size. Meanwhile, since we still need space to store additional inverted lists, the total size of the index is still slightly increased (about 10%, see Section 4).

3.2 On-Demand Synchronization

The synchronization process in step 4 is a brute-force synchronization to keep the local result list of the currently edited field (f_i) up to date. However, the calculation could be unnecessary in some cases. For instance, if the user does not switch the input box to another one, then it is unnecessary to perform the synchronization process. Intuitively, since the values of other fields are not changed, the local results of f_i can be derived directly from the cached local results of the same field, without considering other fields.

Although the local results of f_i are always up to date if the user does not change the input box, we cannot guarantee that the local results of other fields are all up to date. If the user changes her focus to another input box, we must perform a synchronization operation for (and only for) the corresponding field at once. We call this mechanism of synchronization the *on-demand synchronization*. It requires one merge operation and one intersection operation. In contrast, the brute-force synchronization requires one merge operation and one intersection operation whenever the user types in a new letter. Experimental results show that the on-demand synchronization can increase the search performance (see Section 4).

4. EXPERIMENTS

We have developed two prototype systems using the proposed techniques. (1) Seaform-DBLP, which searches 1.3 million computer-science publications on DBLP by Title, Authors, Journal Name, and Year. (2) Seaform-IMDB, which searches over 500 thousand movies on IMDB by Title, Actors/Actresses, Directors, and Year. All the code were implemented in C++ and compiled using Visual Studio 2008. All the experiments were ran on a computer with an Intel Core-2 2.4GHz CPU and 2GB RAM.

We present the results of the Seaform-DBLP system. We used a workload of 45,276 real queries collected from our

deployed system. Figure 2 shows the comparison of average search time per query of four algorithms: (1) SL-BF, which uses Single-List tries and Brute-Force synchronization, (2) SL-OD, which uses Single-List tries and On-Demand synchronization, (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.

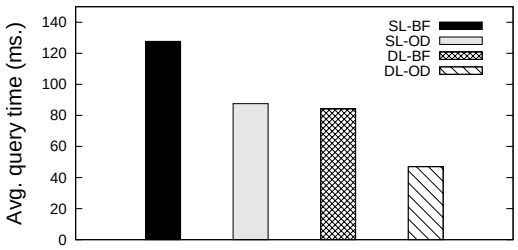
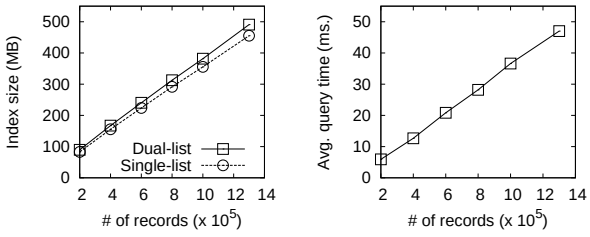


Figure 2: Performances of four algorithms.

The figure shows 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 per query.

Figure 3 shows the scalability of Seaform-DBLP. The search time and index size increased linearly as the dataset increased. The index size increased slightly when we used dual-list tries compared to that of single-list tries (10% larger). At the same time, the search became about 2 times faster.



(a) Index size (b) Performance (DL-OD)

Figure 3: The scalability of Seaform-DBLP.

5. DEMO DESCRIPTION

We describe the main features of Seaform to be demonstrated in four scenarios[4].

Scenario 1 (Precise search paradigm). Suppose a user wants to find papers by Wei Wang whose titles contain the word "`pattern`". If she types in "`wei wang pattern`" in CompleteSearch, many returned results are not very relevant. In contrast, if she types in "`wei wang`" and "`pattern`" in different input boxes in Seaform-DBLP, she can find high-quality results.

Scenario 2 (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, after waiting for several seconds, the user still does not get relevant result. So she has to try a new query. In contrast, in Seaform-IMDB, she can modify the query and see the new results instantaneously.

Scenario 3 (Faceted search). Suppose the user has limited prior knowledge about the KDD conference and wants to know more about it using Seaform-DBLP. 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. To do this, she chooses the year 2009 by clicking on the list, and changes the focus to the first Author input box. The list below the form shows the authors. She can see that the most *active* author. For instance, the author with the most publications is Christos Faloutsos. She then chooses "Christos Faloutsos" and changes the focus to the second Author input box. Then all the co-authors are listed. After several rounds of typing and clicking, the user can get a deeper understanding about the conference.

Scenario 4 (Ranking of results). In this scenario we show the different ranking mechanisms of the two prototypes. For the global results, Seaform-DBLP ranks them by their years of publications, while Seaform-IMDB ranks them by their IMDB rating scores. For the local results, both systems first rank them by the number of occurrences in the global results. If some of the local results have the same number of occurrences, Seaform-DBLP further ranks them by their *activeness* (the average year of publications), while Seaform-IMDB ranks them by their average IMDB rating score. For example, if we input "`godfather`" in the Title input box in Seaform-IMDB, the suggested local results are Godfather Part I, Godfather Part II, etc, which are consistent with their IMDB rating scores.

[4] `http://tastier.cs.tsinghua.edu.cn/seaform`. Due to the copyright policy of IMDB, we cannot publish Seaform-IMDB as an open service on the Internet.

6. ACKNOWLEDGEMENTS

This work is partially supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, and the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103.

7. REFERENCES

[1] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In *CIDR*, pages 88–95, 2007.

[2] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.

[3] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

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

[5] 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.

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