FAST: Friends Augmented Search Techniques
— System Design & Data-Management Issues

Christian von der Weth
DERI, National University of Ireland, Galway
Email: christian.vonderweth@deri.org

Anwitaman Datta
Nanyang Technological University (NTU), Singapore
Email: anwitaman@ntu.edu.sg

Abstract—Improving web search solely based on algorithmic refinements has reached a plateau. The emerging generation of searching techniques tries to harness the “wisdom of crowds”, using inputs from users in the spirit of Web 2.0. In this paper, we introduce a framework facilitating friends augmented search techniques (FAST). To that end, we present a browser add-on as front end for collaborative browsing and searching, supporting synchronous and asynchronous collaboration between users. We then describe the back end, a distributed key-value store for efficient information retrieval in the presence of an evolving knowledge base. The mechanisms we explore in supporting efficient query processing for FAST are applicable for many other recent Web 2.0 applications that rely on similar key-value stores. The specific collaborative search tool we present is expected to be an useful utility in its own right and spur further research on friends augmented search techniques, while the data-management techniques we developed are of general interest and applicability.

Keywords: social search, collaboration, browser add-on, key-value store, distributed information retrieval

I. INTRODUCTION

We believe that the systematic exploitation of social networks holds the key to the next generation of search techniques, complementing the traditional algorithmic solutions that power current web search engines. Specifically, we envision friends augmented search techniques which leverage on both explicitly declared social networks as well as implicit ones, determined based on shared interest or expertise, and support (a-)synchronous collaboration. Several existing systems leverage on information sharing and user collaboration. For instance, Q&A systems (e.g., YAHOO! ANSWERS), bookmark sharing sites (e.g., DELICIOUS) and the recent integration of BING search with FACEBOOK for search result personalization. Their existence demonstrate the potential and need for moving towards more social approaches to information sharing and search. While similar in spirit, we propose mechanisms following a different paradigm, and independently from service providers. Instead of expecting users to visit a specific site to look for information, we use a browser add-on to facilitate information search and sharing based on whichever sites the user visits.

This work has been done while Christian von der Weth was working at the Nanyang Technological University (NTU), Singapore.

The demand for quickly finding the right information on the web has spurred (and still spurs) the success of search engines like GOOGLE and BING. Search engines apply sophisticated algorithms to crawl the web and to analyze the content of web pages as well as the link structure of the web. Although traditional search engines are and will be an important mechanism for finding information, algorithm-driven improvements have reached a relative saturation. They are also inadequate to deal with some apparent challenges: (a) While the returned results are correct in terms of matching the keywords, the content of the result pages may be wrong, outdated or of limited relevance for the user. (b) Using search engines is a non-trivial task. Particularly for a novice user, identifying meaningful keywords and quickly assessing the relevance of result pages is difficult [1], [2]. (c) Even with the same keywords, a result page can have a different relevance for different users. Approaches towards personalization exploit additional information like user profiles or user behavior; [3] gives a good overview. However, the access to such additional information is restricted to the specific search engine. (d) Search results are determined and ranked using algorithms which often do not account for correctness or credibility of the content.

EXAMPLE 1: Consider a user searching/browsing for a medical treatment for a sore throat. Already the first 20 results of popular search engines range from various homespun remedies posted in online forums, Q&A systems, personal websites or even YOUTUBE, to exact but varying medications on professional(-looking) websites, including intentional or honestly mistaken wrong advices. For users without a medical background knowledge, it is difficult to assess the relevance and correctness of the advices.

In Section III we describe COBS (collaborative browsing and search), the front end to facilitate friends augmented search, along with the technical challenges in realizing it. We have demonstrated its usability and explored some of the human computer interface issues [4]; here we focus on the technical challenges in realizing COBS and the directly associated (tag) data management issues. Like many recent Web 2.0 applications, FAST principally relies on a custom-made distributed key-value store for managing a bulk of
the data. While various aspects of key-value stores have been widely investigated over the last decade, we focus on the description of novel techniques employed for efficient query processing of tagged data using a key-value store, taking into account the peculiarities of the workloads for FAST and other typical Web 2.0 applications; see Section IV. Our approach is of universal relevance, beyond FAST. In Section V we evaluate the front end COBS, as well as the back end data store, thus making a holistic evaluation of our framework for friends augmented search techniques.

II. RELATED WORK

Web 2.0 platforms. At a high level, the differentiating aspects of FAST with respect to current Web 2.0 services such as DELICIOUS, REDDIT, YAHOO! ANSWERS or the recent tie up between MICROSOFT’S BING and FACEBOOK are as follows: User interactions and the provided information are determined based on where users naturally are online, instead of restricting or requiring them to visit a specific site; and the knowledge base driving FAST is open to be used and refined by one and all, and independent of service providers. Thus, FAST naturally complements (rather than competes with) such existing services.

Collaborative browsing and searching. Browsing and searching the internet is primarily a single-user task. Collaboration among users is typically based on mechanisms like e-mail or instant messenger. Several approaches that enhance browser capabilities towards collaboration exist. [5], [6] are systems for co-located collaboration, i.e. several users gathering around one computer. [7], [8] focus on the collaboration between users that already know each other, e.g. family members or co-workers. [9] supports synchronous communication, i.e. not storing permanent data. A recent initiative, the GOSSPLE [10] project, has similar overall aims as FAST, but the way of achieving the same are very different. GOSSPLE relies on gossip based information aggregation techniques. In FAST, the basic premise of collaboration is based on a shared location (i.e., users visiting the same website) on the Web, be it for a presence-driven synchronous collaboration, or using meta-information about the location shared among users asynchronously.

Distributed information retrieval. Distributed inverted indices map terms, e.g. tags, to documents, e.g. web pages, containing that term in order to facilitate information retrieval. Multi-term queries – which represents the majority of user queries – are evaluated by merging the corresponding list of documents for each query term [11], [12]. Although techniques to reduce the bandwidth consumption, like Bloom Filter [11], exist, the costs for multi-term searches using single-term inverted indexes are generally very high [13]. As a result, approaches utilizing multi-term inverted indexes have been proposed, e.g. [14], [15]. Given a document with \( n \) terms, the number of possible term combinations is in \( O(2^n) \). Thus, the limitation to a meaningful subset of term combinations is a crucial part of the proposed systems. Existing approaches assume static documents, i.e. the set of terms for an document does not alter. However, in Web 2.0 applications in general, and also in FAST, the set of tags of a resource changes over time. With that, not only the size of the index but also the bandwidth needed to propagate updates to the index needs to be taken into account.

III. COBS: COLLABORATIVE BROWSING AND SEARCH

FAST has two logical components, the front end (COBS), implemented as a browser add-on\(^1\), and the back end information system. Being implemented as a browser add-on, FAST is independent from features of specific websites, and also available for the "old web". The add-on comprises a toolbar and a sidebar, providing different functionalities.

Current features. In the following we outline the current features of the COBS add-on and how user may benefit in the context of our motivating Example 1:

Page rating. Users can rate each web page, uniquely identified by its URL, by means of a 5-star rating scheme. A second 5-star scale displays the average of rating of a page. This allows user to quickly assess the quality of the content of a page, e.g. about the effectiveness of a home remedy.

Page tagging. Using a small text box in the toolbar, a user can add tags to a web page. In addition, each tag has a rating value, initially set to 0. Now other users can upvote (+1) or downvote (−1) a tag to increase or decrease its overall rating. The intuition is that highly rated tags describe a web page or its content better that tags with a low rating.

Creating links. The toolbar maintains a history holding the last 20 visited pages. Users can create links between visited pages by drag&drop between the entries of the history, e.g., a link from the website of a medicament to a page that addresses the side effects of the product. User-generated links can serve as input for network analysis techniques. Again, users can up- or downvote the links of others.

Discussion board. The discussion board is a forum-like commenting system where users can post comments and reply to them. Each board is identified by a URL’s domain and path. Thus, all users visiting the same website, e.g. looking for a treatment, have access to the same board. A discussion board is “in situ”, i.e., the discussion is tied to the discussed content of a page. The communication between users is primarily asynchronous: users can visit a same page regularly to access the corresponding board.

Online chat. The online chat allows users to communicate with each other in real-time. Like for discussion boards, each chat is assigned to a specific domain and path of a URL, showing all visitors of the same site, and thus tied to the discussed content. The intuition is, in analogy with meeting like-minded people at a physical space (e.g., museum, concert, etc.), that people with an apparently similar interest or expertise “meet” on the same web page.

\(^1\)Available for download at http://code.google.com/p/socialcobs/.
**Guided browsing.** The concept of guided browsing is particularly interesting for collaboration between a novice user and an expert, e.g., a user with a medical background, where the expert actively guides the novice while browsing for information. For this, a user sends a request to another user to act as guide. If the guide accepts, the URL of each page the guide visits is displayed in the follower’s browser. By our definition, (a) users can follow only one guide, (b) a guide can have several followers, and (c) a follower can be a guide for others. The latter may result in long forwarding chains, incl. cycles which, however, the add-on detects.

**Web of trust.** Users can maintain a list of users they deem trustworthy collaborators, e.g., for making good contributions or for helping others. The web of trust helps users to filter additional information about a web page and serves as input for various analyzing techniques, e.g., social network analysis or collaborative filtering to recommend interesting web pages or new potentially trustworthy collaborators.

**Challenges in major design decisions.** While most features of the COBS add-on are rather well-known, their integrated implementation to be independent from a specific website poses new challenges.

**Appearance and usability.** Existing discussion boards, commenting systems, etc., are designed according to the layout of the containing website. To be independent from specific sites, such an integrated layout is not possible. Although our focus is less on the UI design of the add-on, to make it an accepted tool requires some consideration in this respect. In order not to clash with the layout of websites, we favor a plain and neutral design. Further, we stick to best practice techniques like a 5-star rating scheme for web pages or the simple up/downvote scheme for contributions.

**Incentives to collaborate.** The effectiveness of COBS depends on the willingness of the users to collaborate. From an economic perspective users are willing to contribute if their perceived benefit in doing so outweighs their perceived costs. To quantify the benefit is difficult, since it is highly subjective and includes emotional aspects. This particularly holds for indirect collaboration where users do not immediately but in the long run benefit from their contributions. At this stage of our work, we focus on minimizing the costs, i.e., the effort to make contributions. Therefore, adding ratings, tags and links can be performed in a matter of seconds.

**Handling query strings.** Often the content of a web page is identified by a query string composed of a series of field-value pairs, e.g., the field `v` holds the identifier of the shown video on www.youtube.com. But the query string may contain fields that do not affect the content, e.g., the `fmt` field in YOUTUBE URLs specifies the quality of videos. Since both the usage and names of fields in query strings are not standardized, it is impossible to reliably identify the relevant fields which specify the content of a web page. Thus, it is not obvious how to assign a user contribution (tag, rating, etc.) to a page or how to decide when two users have access to the same chat. Our current approach is as follows: We assign user contributions to pages identified by their complete URL, including the query string. Adding contributions can be done very quickly, compensating the risk that, e.g., a rating does depend on a non-relevant field in the query string. The discussion board or online chat require a sufficient number of users. Here the risk of considering non-relevant parameters is much more pronounced. Thus, we group visitors of web pages with the same domain and path to the members of a discussion board or online chat.

**Integration with distributed back end.** The handling of URL query strings directly affects the integration of the add-on front end with the back end based on a key-value store. In FAST, the most important request is to get all related information about a page. We therefore group information into data objects according to their assignment to a web page. The first group contains all user contributions. Thus, we merge all tags, ratings, etc. about a page into one data object and use the complete URL (incl. query string) as key. The second group contains the comments and chat messages, and the path and domain of the URL as key to store the data. However, in FAST various informations needs refer more than one specific web page, e.g. a keyword-based search to get all pages featuring a specific set of tags. To evaluate non-URL-centric queries efficiently, we deploy the concept of a distributed inverted index. Since multi-term queries represent the majority of user queries (cf. Section V), we favor a multi-term inverted index, i.e. storing combinations of terms/tags as key in the index. Figure 1 illustrates the approach. Given a page with \( n \) tags, the index potentially grows in \( O(2^n) \). Since in FAST the set of tags of a page may change over time, not only the size of the index but also the bandwidth needed to propagate updates to the index is an issue.

**IV. Data Management Over a Key-Value Store**

FAST relies on a custom-made distributed key-value store. To support keyword-based searches, FAST features a multi-term inverted index. Our approach to cope with exponential growth of the index is two-fold. Firstly, our analysis of a real-world query log (see Section V-B) shows that the average number of query terms is 2.43. Thus, storing large tag sets is not meaningful since they would very rarely be queried. We therefore limit the maximum size of tag sets, denoted by \( s_{max} \). Let \( t_p \) be the number of tags of
A page $p$, then the number of tag combinations for $p$ is in $O(t_p^{s_{max}})$, where, in practice, the value for $s_{max}$ tends to be small. Secondly, we also evaluated the distribution of term combinations of various sizes in the query log, see Figure 2. All sizes yield power law relationships, i.e. most term combinations are very rarely queried while only a few combination are very frequently queried. We therefore aim for a query-driven optimization, storing only frequent term combinations (see Section IV-B). We distinguish between single-term keys derived from single terms, and multi-term keys derived from a combination of terms. We assume that all single-term keys are available in the index. A more thorough and focused discussion on the distributed key-value store and our multi-term indexing techniques can be found at [16].

### A. Query Processing

The query processing algorithm exploits the current state of the inverted index. In general, if a query comprises two terms or more there are several ways to process the query.

**Example 2**: Let $q = \{t_1, t_2, t_3, t_4\}$ be a query containing four terms. With $s_{max} = 3$ the following set keys can be derived from $q$: gray marked keys are available in the index:

- $|k| = 3: \{t_1t_2t_3\}, \{t_1t_2t_4\}, \{t_1t_3t_4\}, \{t_2t_3t_4\}$
- $|k| = 2: \{t_1t_2\}, \{t_1t_3\}, \{t_1t_4\}, \{t_2t_3\}, \{t_2t_4\}, \{t_3t_4\}$
- $|k| = 1: \{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}$

Possible subsets of available keys to answer query $q$ are, e.g., $\{\{t_1, t_2\}, \{t_1, t_3, t_4\}\}$ or $\{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$.

**Algorithm.** Algorithm 1 shows the steps for processing a query $q$. If $q$ contains $\leq s_{max}$ terms we access the inverted index using $q$ as key (Line 1-4). If not available or the number of query terms is larger than $s_{max}$ we compute all relevant keys (derived from all possible term combinations up to size $s_{max}$) for $q$ (Line 5). Next, we retrieve for each available key the size of its inverted list (Line 7-10). Only if no partial result size is 0, we proceed; otherwise we return an empty result (Line 11-12). From the set of available keys $K_q^{avail}$ we derive the ordered list of keys that specifies the order of index accesses (Line 15; described in next paragraph). Finally, we access the index to retrieve the actual data and to compute the intersection for the final result.

**Algorithm 1: processQuery(q)**

<table>
<thead>
<tr>
<th>Input: query $q = {t_1, t_2, ..., t_4}$, Output: query result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $n \leq s$ then</td>
</tr>
<tr>
<td>2 $\text{result} \leftarrow \text{getResultFromCache}(q)$</td>
</tr>
<tr>
<td>3 if $\text{result} \neq \text{null}$ then</td>
</tr>
<tr>
<td>4 return result</td>
</tr>
<tr>
<td>5 $K_q \leftarrow \text{computeSubsetKeys}(q)$</td>
</tr>
<tr>
<td>6 $K_q^{avail} \leftarrow \emptyset$</td>
</tr>
<tr>
<td>7 foreach $k \in K_q$ do</td>
</tr>
<tr>
<td>8 $\text{sizes}[k] \leftarrow \text{getResultSize}(k)$</td>
</tr>
<tr>
<td>9 if $\text{sizes}[k] \neq \text{null}$ then</td>
</tr>
<tr>
<td>10 $K_q^{avail} = K_q^{avail} \cup k$</td>
</tr>
<tr>
<td>11 if $\exists k \in K_q^{avail} : \text{size}[k] = 0$ then</td>
</tr>
<tr>
<td>12 return $\emptyset$</td>
</tr>
<tr>
<td>13 $L_{access} \leftarrow \text{getKeyAccessList}(q, K_q^{avail}) ; \text{result} \leftarrow \emptyset$</td>
</tr>
<tr>
<td>14 foreach $k \in L_{access}$ do</td>
</tr>
<tr>
<td>15 $\text{result} \leftarrow \text{result} \cap \text{getResult}(k)$</td>
</tr>
<tr>
<td>16 return result</td>
</tr>
</tbody>
</table>

**Algorithm 2: getKeyAccessList(q, K)**

<table>
<thead>
<tr>
<th>Input: set of keys $K$, Output: list $L_k$ of keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $K \leftarrow K \setminus {k_1 \in K \mid \exists k \in K : k \subset k_1}$</td>
</tr>
<tr>
<td>2 $L \leftarrow \emptyset ; L.\text{add}(\arg \min_{k \in K} \text{size}[k])$</td>
</tr>
<tr>
<td>3 while $\bigcup_{k \in L} k \neq K$ do</td>
</tr>
<tr>
<td>4 $K_{max_{coverage}} \leftarrow \arg \max_{k \in K}</td>
</tr>
<tr>
<td>5 $L.\text{add}(\arg \min_{k \in K_{max_{coverage}}} \text{size}[k])$</td>
</tr>
<tr>
<td>6 return $L$</td>
</tr>
</tbody>
</table>

The optimization goal is to minimize size of transferred data. To find the optimal subset of keys and their ordering would require complete knowledge, particularly about the expected size of the intersection of two or more partial results. This would require the costly maintenance of statistics over the data in the inverted index, which are typically not available in distributed systems. We therefore favor a heuristic to determine the set and order of keys to access the index; see Algorithm 2. Firstly, we remove redundant keys from $K_q^{avail}$ (Line 1); a key $k_i \in K_q^{avail}$ is redundant if there is a key $k_j \in K_q^{avail}$ and $k_i \subset k_j$; e.g., if $K_q^{avail} = \{k_1 = \{t_1, t_4\}, k_2 = \{t_1, t_3, t_4\}\}$ we can remove $k_1$ since $k_2$ already covers all terms of $k_1$. We then generate $L$ as list of keys to access the index. We initialize $L$ with the key having the shortest inverted list (Line 2). We then iteratively add keys to $L$ that maximize $L$’s coverage of $q$ until $L$ covers $q$, i.e. all terms in $q$ are represented in at least one key in $L$ (Line 3-5). If several keys maximize the coverage, we add the one with the shortest inverted list.

**Cost analysis.** The retrieval algorithm considers all relevant keys up to size $s_{max}$ for a given query $q$ to access the index. Thus the algorithm performs $\left(\sum_{j=1}^{s_{max}} \binom{q}{j}\right)$ accesses to the index. In practice, however, this polynomial growth has only a limited impact on the performance. Firstly, as our analysis shows, the value for $|q|$ is rather small ($\sim 2.4$ on average) and a reasonable value for $s_{max}$ is with 3 or 4 also small.
Secondly, the \( O(|q|^*_{\text{max}}) \) index accesses are only required to retrieve the length of inverted lists, and not the lists themselves. The actual size of the data transferred, e.g. in terms of required bandwidth, is thus very small.

### B. Index Maintenance

The maintenance of the inverted index comprises two major tasks: (a) suspending and resuming of keys depending on their popularity, (b) handling of updates on the tag data.

**Suspending and resuming keys.** The inverted index stores only the inverted lists of popular keys, where the popularity of a key \( k \) is derived by the frequency how often \( k \) is requested during query processing. If a key \( k \) becomes unpopular, we suspend \( k \), i.e. we delete \( k \)'s inverted list and mark \( k \) as unavailable for processing queries. If a suspended or new key \( k \) becomes popular, we resume \( k \). Resuming a key \( k \) involves retrieving its corresponding inverted list which translates to performing a query for \( k \) (cf. Algorithm 1) and storing the result as \( k \)'s inverted list. As last step, we mark \( k \) as available again. To measure the popularity, we provide each key \( k \) with a bit vector \( B_k \) of length \( \ell \). Every time \( k \) is requested, we first set \( B_k := B_k \gg 1 \), i.e. we shift the bit vector for \( k \) one bit to the right, and then set \( B_k := B_k \mid 2^\ell \), where \( \ell \) performs a bitwise inclusive OR operation. To implement the timely decay of a key \( k \)'s popularity, we periodically, after time \( \Delta_{\text{decay}} \), set \( B_k := B_k \gg 1 \). With that, the number of set bits in \( B_k \) represents the popularity of \( k \).

**Example 3:** The following figure shows a bit vector \( B_k \) both after a request for \( k \) and after a periodically shifting.

\[
B_k = 0100011 \quad \rightarrow \quad B_k = 1010011
\]

While each periodic shifting decreases the number of set bits, a request for \( k \) increases the number or keeps it.

Beside vector length \( \ell \) and interval \( \Delta_{\text{decay}} \), further relevant parameters are (a) \( b^{\text{res}} \) as the minimum number of set bits in \( B_k \) to resume \( k \) and (b) \( b^{\text{sep}} \) as the number of set bits in \( B_k \), when falling below, to suspend \( k \). Resuming keys adds to the workload for processing user queries. However, depending on the choice of the values for these four parameters, we expect resuming keys to be a much more infrequent event than evaluating user queries.

**Handling updates of tags.** Updating a resource (here, a web page) by adding or deleting a tag must be propagated to the inverted index. To propagate each update to all relevant keys, would result \( O(t_{\text{max}}^* s_{\text{max}}) \) update messages. While with this approach the index is always up to date, it is not suitable for high update rates like we observed in DELICIOUS and FLICKR. We therefore propose an update mechanism which relaxes the guarantee of the timeliness of the index but resulting in a significant decrease of bandwidth consumption. In a nutshell, we propagate the information about a new or deleted tag only to the corresponding single-tag key in the inverted index. We further update only available multi-term keys periodically. To do so, we propose incremental update queries, where the results only contain the relevant changes, i.e. the tags to be added or to be deleted, for a multi-term key's inverted list. In the following, we present our update mechanism in detail.

**Extensions to the inverted index.** To incrementally update multi-term keys, nodes have to distinguish between resources that have already been propagated to multi-term keys and both newly added and deleted resources. Thus, we cannot delete resources immediately from a single-term key’s inverted list, but mark them as deleted. We assign a timestamp to each resource in the inverted list of single-term keys, indicating when it has either been added or marked as deleted. Secondly, we assign a timestamp to each multi-term key, indicating the time of its last update. Thus, for a multi-term key \( k_m \), we can identify all resources in the inverted lists of all single-term keys \( k_i, \forall i : k_i \subset k_m \), that have been added or deleted after the last update of \( k_m \). Regarding the deletion of marked resources, we define \( \Delta_{\text{update}} \) as the maximum period of time before updating a multi-term key. Thus, after a time of \( \Delta_{\text{update}} \), starting from the time a resource \( r \) has been marked as deleted, we can safely delete \( r \) from the inverted list.

**Incremental updates of keys.** The rationale is to only transfer the latest changes in the inverted lists of single-term keys to evaluate the necessary changes required to update multi-term keys. Latest changes in an inverted list refer to the set of resources added or marked as deleted after the last update of a multi-term key. To further formalize the concept of incremental update queries, let \( R_{k_i}^B \) be the set of resources in the inverted list of key \( k_i \) that are marked as deleted; \( R_{k_i}^C \) contains all resources not marked. Additionally, let \( ts(r) \) be the timestamp when a resource was added or marked as deleted in an inverted list, and \( ts(k) \) the timestamp of the last update of a key \( k \). We then can define \( R_{k_i|k_j}^{BG} = \{ r \in R_{k_i}^B \mid ts(r) > ts(k_j) \} \) as set of added resources in \( k_i \)'s inverted list with a timestamp older than the timestamp of a key \( k_j \); analogously we define \( R_{k_i|k_j}^{CG} = \{ r \in R_{k_i}^C \mid ts(r) > ts(k_j) \} \). With these definitions, Figure 3 shows the involved steps for an incremental update of a 2-term key. Extending the update process for keys of size \( > 2 \) is straightforward. Consider a multi-term key \( k_m \) of size \( s \) and the corresponding single-term keys \( k_1, k_2, ..., k_s \subset k_m \). The basic mechanism is that the changes of each \( k_i \)'s inverted list are successively incorporated into the intermediate results, before being sent to update \( k_m \).

V. EVALUATION

We present our experiences with the COBS browser ad-on, and evaluate the performance of our multi-term inverted index as distributed back end for FAST.
Thus, given a user can be a guide and follower at the same time.

experiences; we installed the COBS add-on on 20 PCs.

guided browsing. In the following we present some of our
contributions (tags, links, comments) and user data (web
of trust) and (b) providing the server component for the
chat protocol. The chat protocol is required the online chat
and for the guided browsing. Evaluating the overall usability
of the add-on is more a HCI (human-computer interaction)
issue, and is thus beyond the scope of this paper [4].

Here, we focus on enabling systems issues. From a systems
perspective, the most interesting aspect is the concept of
guided browsing. In the following we present some of our
experiences; we installed the COBS add-on on 20 PCs.

For guided browsing, one guide can have several followers
and a user can be a guide and follower at the same time .
Thus, given n users (u1, u2, ... , un), two extreme cases of
guiding relationships can occur: (a) a star network, i.e. one
guide and n − 1 followers, (b) all user form a forwarding
chain of length n, i.e. u1 follows u2, u2 follows u3, ... , un−1 follows un. For both cases we evaluated the average
time required for a forwarded URL to reach all followers.
Figure 4 confirm our expectations. For forwarding chains,
the time when the last follower receives the URL is directly
proportional to the length of the chain. For the star network,
the time also increases with the number of followers, but
less pronounced compared to a chain. Once the guide has
forwarded a URL to all followers, loading the URL
in the browsers of all followers is done in parallel.

We also investigated some kind of worst case scenario
in the context of guided browsing. Supporting forwarding
chains may result in forwarding cycles. As a counter-
measure, our add-on stops forwarding a received URL if
this URL is currently loaded in the browser. However, if two
users in a forwarding cycle load a new (different) page at the
same time, both corresponding URLs are forwarded within
the cycle, resulting in a cascade of alternating reloading of
the URLs in all users’ browser windows. While in theory this
may last forever, in practice one URL eventually “overtakes”
the other one and our cycle detection takes effect. Our tests
showed that the time for detecting such cascades is very
hard to predict; over repeated experiments, we encountered
times from several seconds up to some minutes until the
forwarding finally comes to a halt. Even the number of
users in the forwarding chain is not a meaningful parameter
to estimate this time a-priori. Since the add-on cannot
distinguish between a genuine forward of a new URL and a
cycling URL, each mechanism to counter this behavior will
limit the current flexibility of guided browsing. However, we
dem the occurrence of such (large) cycles as a very rare
event, and end-users can also manually intervene.

B. Multi-Term Inverted Index

We next report the performance of our multi-term inverted
index using a key-value store as distributed back end infra-
structure for FAST based on trace-driven experiments.

Used data sets and preliminary steps. We used publicly
available tag data sets from DELICIOUS and FLICKR, both
obtained in 2006. Table I shows their basic characteristics.
Note that number of actions per minute represents just an
estimation for the lower bound, since the data does not
reflect actions like the (repeated) deletion and re-insertion
of tags. Acquiring query logs is challenging. Due to privacy
concerns, service providers do not make their query logs
public. Synthetically created query histories are, in general,
inapt to pattern the frequency, popularity, etc. of queries in
real-world systems over time. We therefore use the AOL
query log [2] which is, to the best of our knowledge, the only
real-world query log of reasonable size, containing mainly
English queries. The log contains ∼28.8 Mio. queries, and
was collected in the period from March to May, 2006.

Assumptions and evaluation method. We assume a
distributed key-value store (like used by GOOGLE [17],
FACEBOOK [18], AMAZON [19]) for managing the inverted
index. We ignore node failures; particularly for single-term
keys, we assume that they are always available. Since we do
not consider locality-preserving data placement strategies,
we assume the worst case, i.e. a sufficiently large number of nodes so that all keys for processing a query or for propagating an update reside on different nodes. We measure three parameters to evaluate the system performance:

Number of contacted keys (CK). Parameter CK represents all single accesses to keys in the inverted index.

Number of invoked keys (IK). As subset of CK, the IK is the number keys whose inverted list is read while performing queries, updates or resuming keys.

Number of transferred resources (TR). The most relevant parameter in terms of total bandwidth required is TR representing the number of resources that are actually transferred for processing queries, updates and resuming keys.

With these parameters and our assumption of a sufficiently large number of nodes, our results are independent from the actual number of nodes in the systems. In other words, adding further nodes would have no impact on the results.

We evaluate our approach using multi-term keys, henceforth denoted by MTK, against the naive one based solely on single-term keys (STK). To make the results comparable between each other, we compute the relative differences between MTK and STK, where we normalize the load for the STK to 100%. Since the processing of single-term queries is identical for STK and MTK, we use only queries with more than one term throughout our experiments. We performed all experiments on the DELICIOUS and FlickR data set, using the corresponding adjusted query logs. While the absolute figures may vary, the quantitative results are very similar for both data sets. Therefore, due to space constraints, we present only the results for the DELICIOUS data set.

Resuming keys. We first consider the suspending and resuming of keys. While suspending keys is bandwidth-neutral, resuming keys adds to the overall workload.

In the first test we vary the minimum of set bits in a bit vector $B_k$ specifying when to resume key $k$. We set $b^{\text{res}} = 0$, i.e., we suspend keys when no bit is set in the corresponding bit vector. Further, we set $\ell = 24$ and $\Delta^{\text{decay}} = 1h$. Thus, each request on a key $k$ is represented as a set bit in $B_k$ for 24h. Figure 5 shows the results for $b^{\text{res}} \in \{1, 2, 4, 8, 16\}$. In this figure we differentiate between the load only induced by processing user queries and the overall load to emphasize on the additional load caused by resuming keys. Processing user queries clearly benefits from smaller values for $b^{\text{res}}$, since the number of available multi-term keys increases, see Table II. However, the frequent resuming of keys adds to the overall load. For increasing values for $b^{\text{res}}$, since less keys are available in the index, the ratio between the load for resuming keys and processing queries shifts toward a higher load for processing queries, while the overall load stays quite equal. If $b^{\text{res}}$ becomes too large, thus the number of available keys too small, the decreasing load for resuming keys can no longer compensate for the increasing load caused by queries, and the overall load increases. The optimal value for $b^{\text{res}}$ is application-specific, i.e. it depends on the actual tag data and query history. This recommends the implementation of self-tuning mechanisms, to adapt $b^{\text{res}}$ according to the current load.

| Resuming keys: various values for $b^{\text{res}}$ |
|------------------|-----|-----|-----|-----|-----|
| 1                | 2   | 4   | 8   | 16  |
| 3.08%            | 1.38%| 0.53%| 0.12%| 0.08%|

Table II

Relative index size compared to optimal index with all relevant keys available (‘Practically Empty’)

In a second test we modify $\Delta^{\text{decay}}$, i.e. the time span a request on a key $k$ is represented by a set bit in $B_k$. Again, $\ell = 24$ and $b^{\text{res}} = 4$. Figure 6 shows the results for $\Delta^{\text{decay}} \in \{400s, 20min, 1h, 3h, 9h\}$, and Table II the resulting index sizes. Here, the load for resuming keys hardly changes for different values of $\Delta^{\text{decay}}$, since $\Delta^{\text{decay}}$ only specifies how long a key is kept in the index and not how soon. The overall performance increases for increasing values for $\Delta^{\text{decay}}$, since more and more keys are kept in the index, see Table II. Thus, since the number of multi-term keys are w.r.t. the storage requirements are still reasonable low, larger values for $\Delta^{\text{decay}}$ are beneficial. However, the more multi-term keys are available in the inverted index the higher the expected overhead to update them.

Handling updates. As a consequence of our incremental update approach, processing queries might yield different results for STK and MTK. To quantify this, we compared the results for both approaches on the inverted index, after various numbers of updates on the inverted lists of single-term keys. We assumed an optimal index, i.e. all relevant multi-term keys are available. Regarding updates, this is the worst-case scenario, since MTK never has to invoke up-to-date single-term keys. Table III shows the results. Naturally, for an increasing number of updates, the average overlap between query results decreases. The degree of deviation which is still acceptable is a system design decision.

| Changes in inverted lists of single-term keys (in %) |
|------------------|-----|-----|-----|-----|-----|
| 0.25%            | 0.5%| 1%  | 2%  | 4%  | 8%  |
| overlap          | 99.1%| 98.6%| 97.6%| 95.7%| 92.3%| 86.4%|

Table III

Average overlap of query results between na"ive and multi-term approach for various rates of updates

For our subsequent experiments we make the following assumptions: Users perform 150 actions per minute, which is more than twice the figure we derived from the DELICIOUS data set (cf. Table I). Further, to ensure an overlap of above 99%, we only tolerate 0.25% of changes in the inverted lists of single-term keys. With that, given the number of $\sim$10.9 Mio. inverted list entries of single-term keys, we
have to update all available multi-term keys at least every \( \Delta^{update} = 3h \). We vary \( \Delta^{decay} \) and keep the other parameters fix (\( \ell = 24, s_{\text{max}} = 3, b^{res} = 4, b^{sus} = 0 \)). STK only requires the propagation of updates to single-term keys; MTK additionally requires incremental updates. Figure 7 shows the result. Since the incremental update of a multi-term key contacts each corresponding single-term key, the number of contacted keys significantly increases for larger values of \( \Delta^{decay} \). Now with updates, the saved number of transferred resources due to MTK does no longer benefit from many available keys.

To sum up, our results indicate the trade-off between the query processing performance and the load for maintaining the index in the presence of updates w.r.t. the number of available multi-term keys in the index. A large index speeds up the evaluation of queries, but causes high maintenance costs, and vice versa. However, the improvements due to MTK regarding the overall bandwidth consumption significantly outweighs the maintenance costs. Despite our worst cases assumptions for the parameter settings, MTK reduces the number of transferred resources to 50% compared to STK. In real-world systems, we expect even better results.

VI. CONCLUSION

We outlined our architecture towards enriching traditional web search by friends augmented search techniques (FAST), i.e., exploiting the expertise, interests, perceptions and social ties of users. As front end, a browser add-on [4] allows users to collaborate with each other in various ways, yielding explicit networks based on a web of trust as well as implicit networks based on expertise or interests. For the application to scale, the knowledge base resulting from user contributions (tags, ratings, etc.), requires an efficient and scalable data management. To this end, we presented FAST’s back end infrastructure based on a distributed key-value store [16]. Our evaluation shows that this approach can cope with the peculiarities of FAST-like applications, particularly with high update rates on the knowledge base. In our ongoing work we emphasize on the user perspective. Relevant questions are, e.g., how do users work with the add-on, which features are most popular (or which are still missing), what is the expected load of the back end per user, and eventually how do users benefit from applications like FAST. To answer these and related questions, and gain deeper insights into friends-augmented search techniques, requires comprehensive empirical studies. To conduct such studies is challenging, and represents a major part of our long-term efforts.

ACKNOWLEDGMENT. This work has been funded partly by A*Star SERC Grant No 072 134 0055, partly by NTU AcRF Tier 1 Grant RG 29/09 and partly by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

REFERENCES