Collaborative Ranking and Profiling: Exploiting the Wisdom of Crowds in Tailored Web Search

Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, Etienne Rivière

Collaborative Ranking and Profiling: Exploiting the Wisdom of Crowds in Tailored Web Search*

Pascal Felber[1], Peter Kropf[1], Lorenzo Leonini[1],
Toan Luu[2], Martin Rajman[2], Etienne Rivière[1]

[1] University of Neuchâtel, Switzerland, first.last@unine.ch.
[2] EPFL, Switzerland, first.last@epfl.ch.

Abstract. Popular search engines essentially rely on information about the *structure* of the graph of linked elements to find the most relevant results for a given query. While this approach is satisfactory for popular interest domains or when the user expectations follow the main trend, it is very sensitive to the case of ambiguous queries, where queries can have answers over several different domains. Elements pertaining to an implicitly targeted interest domain with low popularity are usually ranked lower than expected by the user. This is a consequence of the poor usage of user-centric information in search engines. Leveraging semantic information can help avoid such situations by proposing complementary results that are carefully tailored to match user interests. This paper proposes a collaborative search companion system, CoFeed, that collects user search queries and accesses feedback to build user- and document-centric profiling information. Over time, the system constructs ranked collections of elements that maintain the required *information diversity* and enhance the user search experience by presenting additional results tailored to the user interest space. This collaborative search companion requires a supporting architecture adapted to large user populations generating high request loads. To that end, it integrates mechanisms for ensuring scalability and load balancing of the service under varying loads and user interest distributions. Experiments with a deployed prototype highlight the efficiency of the system by analyzing improvement in search relevance, computational cost, scalability and load balance.

1 Introduction

Search engines certainly play the most significant role in today's Web usage. Leading search engines rely on the observation of the structure of linked elements [7] (i.e., the graph formed by hyperlinks between pages and data items), which is used in conjunction with the keywords forming a query to decide on the most relevant elements, or for advanced approaches with user-centric search options and hints (e.g., when using Google's SearchWiki [1]). These search engines do not leverage the *collective knowledge* that is created by the users as part of their navigation choices. Instead, the bulk of the score used to decide on this relevance depends on the links pointing to the element, that is, scores are mostly based on *structural* information. While efficient for retrieving the most relevant elements when the implicit semantic search area (i.e., *interest domain*) is the most popular one, there exist many situations where the elements that are the most cited, or belong to the most renowned sites are not those expected by the user.

* This work is partially funded by the Hasler fundation and SNF project 102819.

For instance, a Web search for the query term "Java" returns a list of elements that overwhelmingly focus on the programming language. This is obviously a result of the predominance of computers-related resources on the Web. Nonetheless, a user looking for information on the Indonesian island of "Java" will be dissatisfied by not finding any relevant information (from her point of view) before the items of rank 6 and 16.[1] The solution for avoiding such a situation and obtaining better-tailored results is to pair the structural information used by the search engine with some *semantic* information about the expectations of a particular user. Concretely, information about which items were deemed interesting by other users with similar interests can be leveraged to avoid search domain inadequacies. As a result, the *information diversity*, which is not well captured by solely monitoring the structure of the information graph, can be achieved by taking into account the diversity of expectations from querying users and using the *wisdom of crowds*, learned from past accesses, to determine relevant content for one particular user.

Information about one user's interest can be derived from the set of elements that she accessed as a results of her previous queries (*feedback information*), and from the keywords forming these past queries themselves. Similarly, the set of elements that are deemed interesting by users of some semantic interest profile can be derived from the elements they accessed after a Web search, that is, relevant elements can be extracted by correlating user accesses and extracted interests.

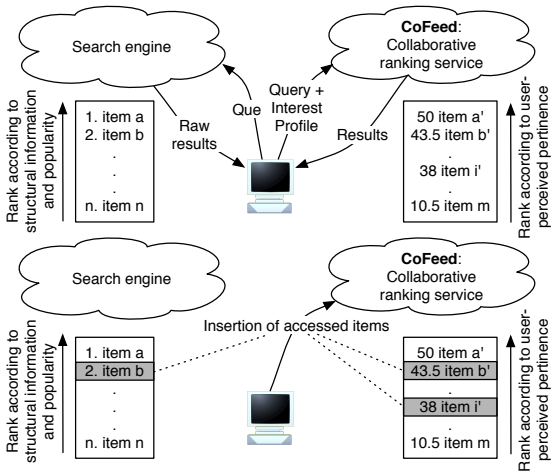


Fig. 1. Usage of a companion search service.

We believe that the best approach for proposing such a service is to build a *companion service* to complement search engines, instead of creating a new stand-alone search mechanism. Indeed, despite many research efforts invested so far to propose collaborative search engines (*e.g.*, Faroo, YaCy, Wowd[2]), no system has been able to reach a sufficient level of quality and efficiency to truly compete with its centralized counterparts. This is a direct consequence of the "bootstrap problem" [9]: *added value* of a new collaborative search engine becomes perceivable only when the system has attracted enough users to fully sustain its specific functionalities.

Figure 1 presents a general vision of the companion service: the user sends her request to a keyword-based search engine, which returns results based on structural information.

Meanwhile, the same query is sent to the collaboratively built companion semantic search service. Note that the latter request is paired with some semantic profile, which is a representation of the user's interest field. The companion service then returns a set of elements tailored to the user requirements on the basis of her semantic profile, and ranked according to their relevance to her interest domains. The additional results

[1] On `http://www.google.com` at the time of writing.

[2] `http://www.faroo.com`, `http://YaCy.net`, `http://www.wowd.com/`.

can then be presented together with the results from the traditional search engine used, in a similar manner that context-sensitive ads are presented as suggestions to the user for a query on most current centralized search engines' results. This simple presentation is also used by [15]. Although more elaborate presentations of the results to the user can be devised, we consider this to be a research task on its own, and not the focus of this paper. Information about subsequent accesses (i.e., which item is accessed for some query and in which order) are sent to the semantic ranking service and used for building, for each request, a set of items that preserves information diversity.

Building such a system poses a set of challenging research issues related to information management (Section 2). First, how to accurately capture the semantic information associated with user activities (profiling interests, using actual accesses to construct a representation of some user's interests)? Second, how to process the feedback information to maintain sets of relevant elements that capture information diversity? Third, how to efficiently construct from these sets a tailored ranked list of results to answer user requests?

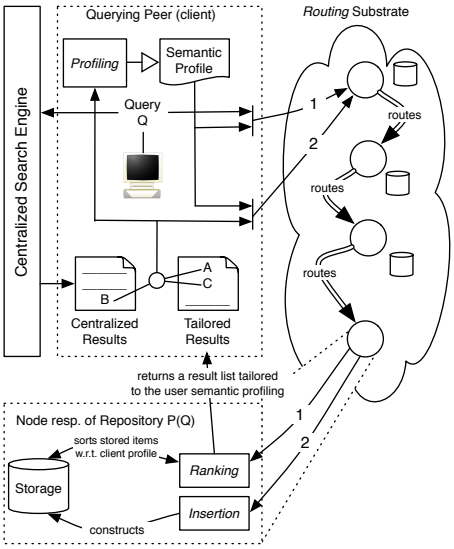


Fig. 2. Components & information flow.

Another challenging question, which this paper answers in detail (Section 3), concerns an appropriate infrastructure for supporting such a service. A centralized approach is easy to implement but scalability in number of users comes at a prohibitively high cost, especially if the service also has to tolerate failures. Moreover, it poses again a *bootstrap problem*, with many resources necessary before being able to serve a reasonably sized set of users. On the other hand, a distributed (*collaborative*) architecture has a much lower cost of bootstrap, and as the number of users increases, the number of servers also increases.[3] Last, beside relieving the bootstrap and scalability issues, distributed architectures are known to be better candidates for implementing fault tolerance and for balancing the load of serving clients over a large set of collaborative machines. **Overall Architecture.** Before describing in detail the components and algorithms of CoFeed, we start by a general overview of its architecture, as depicted in Figure 2. CoFeed consists of a software component on the client computer (typically a browser plugin) and a distributed infrastructure that implements the collaborative ranking. The distributed infrastructure is composed of a possibly large number of nodes that collectively store and update repositories of items and associated relevance feedback information.

Queries from a client are sent to some existing search engine. At the same time, they are sent to CoFeed together with user-specific *interest profile* information. The *routing* substrate is in charge of delivering the query and the profile to the appropriate node, responsible for the target query's repository (see arrows labeled 1 in Figure 2).

[3] Note that end users are not necessarily acting as servers as in a *pure* peer-to-peer model. Instead, institutions can dedicate one or a few servers for provisioning the system as the popularity of the service increases—hence the collaborative aspect.

Based on the query terms and the profiling information, the *ranking* module on that node produces a ranked result list tailored for the user. The client can then combine the lists obtained from the search engine and CoFeed to improve the overall quality of the results presented to the user.

Relevance information is gathered on the user's machine by observing accesses to elements returned by any of the search methods (documents A, B, and C in the figure). This information is used by the *profiling* module to consolidate the local interest profile of the user. It is also sent to the *insertion* module on the node that is in charge of the query repository, and along with the profile of the user, to update the relevance tracking information for this query (see arrows labeled 2 in Figure 2).

2 Profiling, Storing and Ranking

This section describes how our system gathers profiling information, processes user queries, and stores and ranks relevant documents. The set of information associated to one query and stored in CoFeed is called a *repository*. We use the notations and terminologies denoted in Table 1.

Profiling user interests. The accesses of users to documents form the basis for constructing their local user interest profile (UP). Each document from the result list is associated with a snippet, which contains a larger set of keywords (or *tags*) representing the content of the document. These keywords are used to form the interest profile (UP) of the user, which is used in turn to construct document profiles (DP) maintained in the distributed repositories. Keywords are normalized in the system by classical means (stemming, noise-word list, alphabetical sort, duplicate words elimination). As storing all keywords for all accesses is obviously not possible, CoFeed represents profiles using *Bloom filters* [6], which are space-efficient probabilistic data structures allowing fast and false-negative-free inclusion tests over a set of elements. Moreover, Bloom filters have the added advantage of preserving privacy, as users may not want their set of accessed elements to be sent in plain over the network.

A Bloom filter maps elements from an unbounded set to a bounded set of k bits in a bit array of medium size (8,192 bits in our prototype) by using k different uniform hash functions (we use 3 hash functions in CoFeed). Elements (keywords from snippets and queries) are inserted in the profiles (UP) by setting the k bits

Q	Query (a set of keywords, normalized by stemming, stop words removal, etc.)
$P(Q)$	Node in charge of the repository for query Q
$RFitem$	A relevance feedback item (composed of Q, D, UP, $Snippet$)
D	URL of a feedback item
DP	Document profile (Bloom filter)
UP	Interest profile of the user (Bloom filter)
$Snippet$	Summary of the document (title & synopsis with some/all query terms)
$Freq$	Frequency of a feedback item for a query Q as managed by node $P(Q)$ (calculated as a moving average)
T_{first}	First arrival time of a feedback item for a given query Q on $P(Q)$
T_{last}	Last arrival time of a feedback item for a given query Q on $P(Q)$

Table 1. Notations.

corresponding to these hash functions in the associated filter. The inclusion is tested by checking the bits corresponding to each of the k hash functions, and can yield some false positives. This is not much of a concern in CoFeed, as Bloom filters are not used for inclusion tests but for estimating union and intersection sizes of two sets. This is done by counting respectively the number of bits set in the logical OR, or the logical AND of the two corresponding bloom filters. In CoFeed, we compare two profiles S_1 and S_2 by using the Jaccard similarity: $\frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$. This similarity metric between an UP and a DP represents the *adequacy* to the user interest domain of a document. The same metric between two DPs represents their *semantic distance*.

In order to avoid the saturation of bloom filters over time as new queries are performed and as more feedback is inserted in CoFeed, we use for both document and user profiles a variant of bloom filters called *time-decaying bloom filters* [8]. In this variant, bits that are set are associated to decaying timers. Newer elements have a higher weight and older information gradually disappears over time. The larger memory required for each bit is compensated by the frequent removal of elements (and thus the clearance of some bits) from the set. Using this structure allows CoFeed to spontaneously adapt to variations in the popularity of queries and users to receive a feedback that is more relevant to their ongoing search session.

Collecting interest feedback. When a user browses the result list for a query, the title, document reference, and snippet help her select the most relevant documents w.r.t. her query and her interests. The action of accessing some document following a query produces a feedback information item. It represents an implicit *vote* for a document that the user, given her implicit expectations (as summarized by her user profile UP), deemed interesting for the query. The following information is tracked and forms an $RFitem$: (1) the original query Q, (2) the document reference D, e.g., a URL, (3) the local interest profile UP of the user after it has been updated with keywords from Q and the snippet, and (4) the snippet of the document, when available. Elements that are not accessed are simply ignored.

Managing repositories. The repository for a query Q is maintained by a specific node $P(Q)$ in the system. Section 3 explains how this node is reached and how the load for popular queries is dynamically shared amongst several nodes. Managing a repository for some query Q consists of two operations: (1) the management of the relevance feedback information received for Q, and (2) the generation of the results to be sent to a user submitting a request for Q.

We maintain one entry per tuple (Q, D) in the storage. The entries contain additional information $(DP, Snippet, Freq, T_{\text{first}}, T_{\text{last}})$, which are used for various tasks: sorting query results, storage management and garbage collection. Upon arrival of a new RFitem $(Q, D, UP, Snippet)$ at time t (see arrows labeled 2 in Figure 2), if an item (Q, D) already exists, it is updated by computing the union of the DP and UP bloom filters, updating the frequency, and setting T_{last} to t; otherwise, a new item is created and initialized using the content of the new RFitem.

Item ranking. When the node $P(Q)$ receives a request under the format (Q, UP) (see arrows labeled 1 in Figure 2), the storage manager extracts RFitems from the list associated with query Q and sorts them according to the similarity score w.r.t. the user profile (i.e., $Sim(UP, DP)$) and to the frequency. The resulting ranked list of document descriptors $(URL, Snippet)$ is then sent back to the user.

To ensure that a user profile UP provides sufficiently meaningful information to rank search results according to the user's interests, we use on the client side a threshold that specifies the minimum number of distinct documents from the ongoing search session that must be embedded in the user profile for it to be sent along with the query. This helps ensure a minimum level of quality in the results returned by CoFeed and avoids spending bandwidth and resources when no gain can be expected from the ranking information.

Garbage collection. Clients are continuously inserting new feedback information in the system. The storage on each node may be limited. A garbage collection mechanism allow to reclaim periodically some storage space while making sure that the most important information is preserved. Whenever a predefined limit for storage size has been reached (or no resources are available), a set of rules based on (with decreasing order of priority): (1) frequency of items updates and last update time; (2) popularity thresholds; (3) utility of items for constructing results list. We omit further details of the garbage collection mechanisms for the sake of brevity.

3 Distributed Storage System

This section presents the design rationale of CoFeed's distributed storage system for managing repositories and allowing efficient processing of ranking and feedback insertion requests. We describe the resulting architecture and focus specifically on its two key features, routing and load balancing mechanisms.

As previously mentioned, our objective in the design of CoFeed is to support large populations of clients, each submitting many requests. To avoid the prohibitive cost of *scalable* centralized solutions (e.g., high traffic server farms), we propose a decentralized approach in which a set of nodes cooperates to provide the service. These nodes may be provided by ISPs or participating institutions (e.g., universities) that collectively share the processing load. The growth of the numbers of these nodes will follow the number of clients and allows solving the bootstrap problem from a resource provisioning perspective. The repository associated with a query is under the responsibility of a specific node in the system, but high loads are shared amongst several nodes. This node is located by using an efficient key-based routing protocol, which is described below.

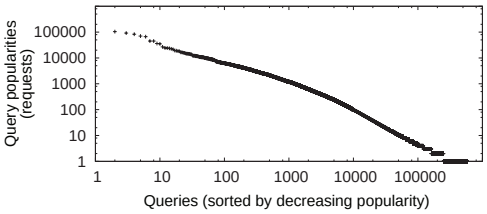


Fig. 3. Popularity distribution of queries in a dataset from AOL [16] follows a typical Zip-like [2] distribution.

A challenging aspect when designing the CoFeed distributed infrastructure is that the popularity distribution of queries is typically very sparse (that is, distributed according to a power law). This means that a small subset of the queries is requested extremely often while the vast majority is only rarely requested. This can be observed in Figure 3, where the popularity of requests from a representative query dataset from AOL [16] is plotted in decreasing order (note the logarithmic scales). Given the high skew in the distribution, one must ensure that popular queries do not overload specific nodes in the infrastructure. To protect against

such scenarios, we have designed adaptive load balancing mechanisms to dynamically offload nodes experiencing too much incoming load. These mechanisms rely neither on fixed load threshold parameters nor manual tuning (see Section 3).

Routing. Each query Q is associated with a node $P(Q)$. This node stores the repository associated to Q: documents references, relevance tracking and interest profiling information. Our overall design is a specialized form of a distributed hash table (DHT). It associates a key-based routing layer (KBR) and a storage layer. The role of the KBR layer is to locate the node responsible for some query based on its key. To that end, it relies on a structured overlay (e.g., an augmented ring), where each node is assigned a unique identifier and the responsibility of a range of data items identifiers. In our case, each query Q has an identifier determined by hashing its terms to a key $h(Q)$. The node $P(Q)$ whose range covers $h(Q)$ is responsible for maintaining Q's repository and for providing the appropriate sorted set of document references when asked to by some remote node. During the routing process, on each routing step towards the destination, the storage layer can be notified by a `Transit` call that a message is transiting *via* the local node. It can in turn modify the content of this message, or even answer the request on behalf of $P(Q)$. This mechanism is used in our design to implement load balancing.

A typical DHT provides a *raw* put/get interface to the application. Elements are stored as *blocks* on the node responsible for their key, and also retrieved as blocks. Our design differs in the important following point: our storage layer does not store information *blindly*, but provides an interface and functionalities that are *specific to the storage and processing of ranking and feedback information*. This has a strong impact on the design of fault-tolerance and load balancing mechanisms.

We based our system on the routing layer of Pastry [18], known for its stability and its performance (small number of hops, usage of network distance for choosing neighbors, etc.). In Pastry, nodes are organized in an augmented ring and maintain routing tables of size $O(log_b N)$, where b is a system parameter (keys are expressed in base b). Greedy routing succeeds in at most $O(log_b N)$ steps. When routing a request to its destination, each intermediary node selects as the next hop a node from its routing table with an identifier that has a longer common prefix with the target key than itself. As each routing step "resolves" at least one digit, at most $d = O(\log_b N)$ routing steps are required. An interesting property of such a greedy routing strategy is that routing paths towards a destination converge to the same set of nodes, and do so with an increasing probability as they get closer to the destination: the more digits have been resolved, the less nodes remain that have a longer common prefix with the target key. Routes from all nodes to some key in the network collide in the last hops. The *path convergence* property is particularly useful for the design of load balancing mechanisms [17, 21], as described next.

Load balancing. CoFeed has to be able to manage large numbers of users simultaneously and support the storage and access to repositories in a scalable manner. The sparseness of query popularities is the main problem, as nodes responsible for storing most popular queries may receive unbearable amounts of traffic.

When some node $P(Q)$ gets overloaded by requests to a popular query Q, it replicates its responsibility for managing information and answering requests related to Q. A wide range of techniques has been proposed for balancing load in structured overlays (e.g., [13, 17, 19, 21]). All these proposals however target scenarios where the

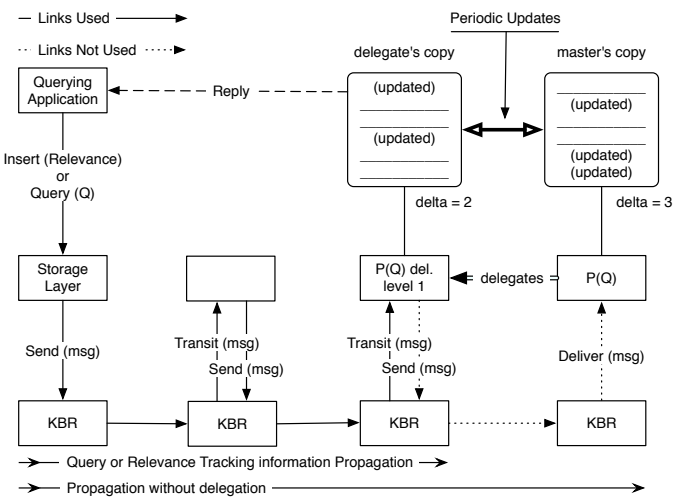


Fig. 4. Delegation mechanism for load balancing.

number of accesses is much greater than the number of updates to the data. These systems support access to non-mutable data by placing replicas on nodes that lie on the path towards its key.

Our system requirements are different. First, the amount of writes (insertion of interest tracking information) and the amount of reads (queries) are of the same order. Caching only read accesses is thus not possible: routing every insertion for a query Q to the node $P(Q)$ would involve notifying all copies, resulting in a load similar to the one avoided by caching access requests. It is thus necessary to also cache insertions, that is, to allow copies of information about a query to be modified *independently* from the "master" copy. We call such a copy a *delegate*: a replica onto which modifications are possible with only loose synchronization to its master copy. Second, queries are very dynamic by nature (e.g., a little-known personality can suddenly become famous and trigger millions of searches). Therefore, load balancing needs to be *reactive*, i.e., be able to initiate and cancel delegation dynamically as a function of the actual load.

Figure 4 presents the principle of delegation: a request (either for ranking or for insertion) is sent by the node on the left side and is routed towards the node $P(Q)$ on the right side. As the next to last node on the path is a delegate of $P(Q)$ for Q, it notices that a request for Q is going through its KBR layer and intercepts it. It replies on behalf of $P(Q)$ or inserts the information in its local copy. Periodic synchronization takes place between the delegates and their delegator (which may itself be a delegate).

Delegates are chosen according to the auditing Algorithm 1, which is run periodically by each node to evaluate its need for delegation. Table 2 gives the default parameter values used by the algorithm, as well as the notations used in the pseudocode.

The periodic auditing of the local load for deciding on a new delegation works as follows. $P(Q)$ keeps a counter $p.\text{in}_x$ of the number of requests received on each of its

SET: $cand \leftarrow \{c \in p.\text{in} \mid p.\text{in}_c \geq \sum_{x \in p.\text{in}} p.\text{in}_x / |p.\text{in}|\}$
foreach $c \in cand$ *(in parallel)* **do**
  retrieve $p.\text{load}_x$ from c
if $p.\text{load}_p > \gamma_{\text{del}} \times \sum_{c \in cand} p.\text{load}_c / |cand|$ **then**
  `// Details of logging ommited for brevity`
  during time Δ_{log}, log requests from nodes $cand$
  foreach $c \in cand$ **do**
    $c.q, c.q_{\text{load}} \leftarrow$ most frequent query from c, and its associated load
  $p.\text{load}_{\text{avg}} \leftarrow \left(p.\text{load}_p + \sum_{c \in cand} p.\text{load}_c\right) / \left(|cand| + 1\right)$
  choose $d \in cand$ that yields the minimal $|p.\text{load}_d(d.q \rightarrow d) - p.\text{load}_{\text{avg}}|$
  if $d.q_{load} > \sum_{x \in p.in} p.\text{in}_x \times \xi_{del}$ **then**
    send a copy of the repository for query $d.q$ to d
    delegate $d.q$ to q

Algorithm 1: Node p's periodic (Δ_{del}) auditing of incoming links for delegation.

Constants		**Value**
Δ_{del}	Auditing period	15mn
Δ_{log}	Request logging period	5mn
γ_{del}	Imbalance tolerance before delegating	180%
ξ_{del}	Minimum relative gain for delegation decision	10%

Notations	
p.in	All "last-hop" nodes that sent some request to p during last period Δ_{del}
$p.\text{in}_x$	Number of requests p received from x during last period Δ_{del}
$p.\text{load}_x$	Incoming request load at x as known to p

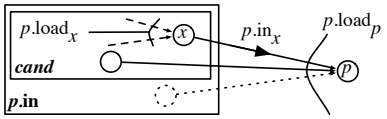


Table 2. Delegation: constants and notations.

incoming links p.in, labeled by the previous hop x. Note that p does not maintain information about which query was targeted, as the role of this lightweight *passive monitoring* is only to detect load imbalance and not to spot their origin. All nodes in p.in that sent more than the average load received on all p's incoming links are asked for their own incoming load, normalized to the period Δ_{del}. This information is stored in $p.\text{load}_x$ for node x.

The auditing of nodes for delegation is done only if sufficient imbalance is detected between the incoming load on node p and the load experienced by nodes in the *cand* (candidates) set. The imbalance threshold is γ_{del}: a value of 180% indicates that p has to handle more than 80% more requests than the average of *cand* nodes being investigated for possible delegation. If some imbalance is detected, the node enters a logging phase (*active monitoring*) in which the requests received from *cand* are recorded. This phase does not have to be as long as the passive monitoring phase, as only the most requested queries are of interest to p for deciding on a delegation, and those are likely to occur in great quantity even in a short period. Then, the most popular query received from each node $c \in cand$ is evaluated as a potential target for delegation. Basically, we select as delegate a node such that, when ignoring the most popular set of request for the same query coming from that node, the difference

between the load experienced by p and the average load experienced by the nodes in *cand* is minimal. Said differently, the goal is less to unload p than to evenly distribute the processing load on all nodes. Moreover, in order to prevent oscillations of delegations and un-delegations, p requires that at least ξ_{del} percent of its load will be handled by the new delegate.

When the delegation of a query by node d is decided, p sends a copy of the repository it has for the delegated query and instructs d to handle requests on its behalf. The cost of sending a delegation depends only on the size of the repository, which is typically small (in the order of a few kilobytes).

Delegates can in turn use this mechanism for *redelegating* Q: the master copy on $P(Q)$ and its delegates form a tree. Synchronization between the copies is performed periodically when the number of changes, denoted *delta* in Figure 4, reaches a configurable threshold. Pair-wise synchronization is used to aggregate the two copies in a new list, either by inserting "new" elements in the master list or by re-ranking the union of the two lists and keeping the k highest items. This list is then forwarded along the tree, resetting all *delta*s to 0.

Delegations are revoked by similar mechanisms: a node can revoke a delegation, based on the observation of requests load, either if it receives notably more requests than the other node for which it is a delegate, or if the revocation of the delegation helps balancing the load between a delegate and its delegator (i.e., the mean incoming load for both nodes gets closer to the average load observed by the delegate). This process uses hysteresis-based threshold values to avoid oscillations: the threshold for triggering delegation is higher than the threshold used for revoking one. We omit the detailed algorithm for brevity.

4 Evaluation

In this section, we evaluate CoFeed using two methods. Both use an actual implementation of the system. First, we assess the validity of interest profiling by running it against user behavior models. Second, we evaluate the performance and effectiveness of the infrastructure itself by observing the peak performance on a single node and the scalability in terms of managed elements, as well as distributed aspects: performance of routing, load balancing and reactiveness to dynamically changing loads.

Experiments were conducted on a cluster of 11 dual-core computers, each with 2 GB of main memory and running GNU/Linux. In experiments that do involve large number of nodes but no time-based performance measurements, each machine of the cluster executes *multiple processes* that represent different nodes. Naturally, for experiments that evaluate the performance of a single node *w.r.t.* time or peak performance, machines are used exclusively by one process. The implementation is based on a combination of C and Lua deployed using the SPLAY infrastructure [11].

User-centric ranking effectiveness. We first evaluate the effectiveness of interest-based profiling and ranking to actually report better tailored results to the user, especially in the case where this user issues request for ambiguous query terms. To that extent, we developed both a synthetic data distribution model and a user behavior model. We do not consider distributed system aspects in this first part of the evaluation and assume that one node replies to all requests coming for one particular query (i.e., there is no use of load balancing). Our evaluation metrics are the ranks of

elements of interest for the user, given her interest domain, with and without interest profiling.

We consider a set of U users $u_1, u_2, \ldots$, interested in a set of queries $Q = q_1, q_2, \ldots$ (e.g., "java", "jaguar", etc.). All these terms are ambiguous, and are associated to a set of documents (or elements) belonging to two or more *interest domains* chosen amongst $D = d_1, d_1, \ldots$. The actual number of domains $dom(q_i)$ for one query q_i is determined randomly using a power-law distribution: $dom(q_i) = 1 + extra(q_i)$, with $\Pr[extra(q_i)] \propto extra(q_i)^{-\alpha_{\text{dom/query}}}$. This means that most queries are associated with documents along 2 domains, a smaller set with documents over 3 domains, an even smaller with 4. No query is associated to more than 4 domains, and the parameter $\alpha_{\text{dom/query}}$ determines the skewness of this distribution. Each domain has a popularity, which is also determined using a power-law distribution: $Pr[d_i \in D] \propto i^{-\alpha_{\text{dompop}}}$. For each query q_i, the $dom(q_i)$ domains are selected according to this *domain popularity* distribution. Each user is interested in one single domain, also selected according to the same *domain popularity* distribution, and issues requests for elements related to this domain only.

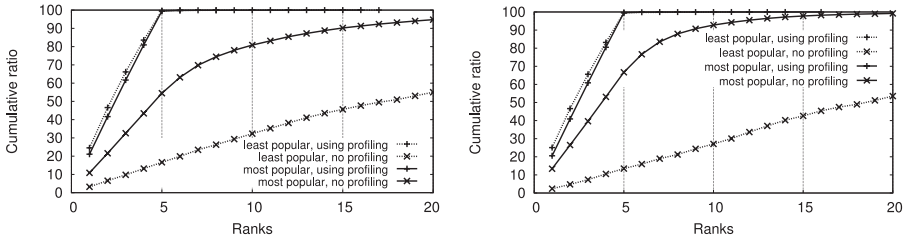
We consider a set of documents (or elements) $E = e_1, e_2 \ldots$, each of which is associated with one single interest domain chosen according to the domains' popularity distribution. For each domain d_i we create a list of documents $E(d_i)$, which is used as follows to generate a set of elements at each repository. Each query q_i is associated with a sorted set of 100 documents $E(q_i)$ representing the repository's content. Each element in this set is dedicated to one of the domains for which q_i is associated, chosen according the domain popularity distribution. The elements of the set are then filled by using a randomly picked and shuffled subset of $E(d_i)$. We use the values in Table 3 for the parameters of the workload.

Each document is associated with some text that represents the *content* of the document. This text is composed of a random number of keywords (between 15 and 30) chosen among queries from the domains associated with the document. To simulate the fact that the snippet returned by a centralized search engine for a given document will vary according to the search keywords (e.g., it highlights the sentences that surround the occurrence of the keywords in the original document), the snippet is generated as a random subset of 5 to 7 keywords forming the document content. One such snippet is generated initially for each query a document is attached to.

Name	**Value**	**Role**
$\|U\|$	500	Number of users
$\|Q\|$	2,000	Number of queries
$\|D\|$	20	Number of interest domains
$\|E\|/\|D\|$	400	Number of documents/elements per domain
$\alpha_{\text{dom/query}}$	1	Distribution of the number of extra domains per query
α_{dompop}	0.8	Distribution of the popularity of interest domains

Table 3. Workload parameters.

The search and access behavior of users is modeled using two phases. In a first phase, each user issues requests for queries that are attached to her interest domain and receives the list of elements as it is stored in the repository (i.e., without using



(a) 25% least and most popular domains (b) 5% least and most popular domains

Fig. 5. Impact of interest-based profiling.

interest-based ranking). This process continues until the user has sent at least 100 interest feedback items to the system (by simulated clicks on some of the returned results). This first phase helps construct the user and document profiles.

We simulate the behavior of a user interested in the domain d receiving a list of items for some query q as follows. The user favors elements that are (1) higher up in the list, and (2) related to domain d. To model this behavior, we choose accessed elements according to a power-law distribution of the ranks in the list, with $Pr[accessing\ i^{th}\ element] \propto i^{-0.8}$, and we drop links that are not in d with probability 80%. In other words, there is a 20% chance that a user accesses some links that are not in her interest domain. This accounts for some "pollution" in the user and document profiles that is representative of real users' behaviors.

In a second phase, we compare the impact on the lists received by the users for their queries, of the use of the profiles and interest-based ranking. This allows us to evaluate whether the user profiling helps in leveraging links interesting to the user by ranking them higher.

For our evaluation, we consider two sets of domains: the 25% most popular ones (ranked 1 to 5) and the 25% least popular ones (ranked 16 to 20). We consider all the requests made by users that are interested in any of the domains of each set. For each such request, we examine the ranks of items that belong to the corresponding interest domain. We consider the ranks of the first 5 elements in the returned lists that are of the correct domain: the higher these 5 elements are in the list, the more effective the search mechanism is from the user point of view. We compare the distribution of these ranks both when using the direct result from the simulated search engine, and when using CoFeed.

Obviously, there are more users interested in the 25% more popular interest domains than in the 25% least popular ones, and elements that are in the latter are ranked lower in the list returned by the centralized search engine model (being attached to an ambiguous query, they compete for positions in the list with at least one more popular domain). The goal of CoFeed is to promote links that are related to the user's domain of interest toward the first positions of her tailored list.

Figure 5(a) shows the cumulative distribution of the rank in the returned list for these first 5 elements, both when CoFeed interest-based ranking is used and when it is not, considering the 25% most/least popular elements. We observe that elements for the popular domains are already ranked higher than elements for the least popular domains: the median of the ranks of elements for popular domains is 5, while it is

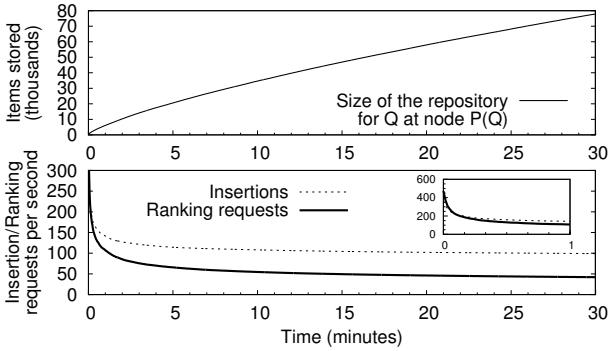


Fig. 6. Performance, single repository: max. possible load vs. repository size.

about 20 for unpopular domains. It follows that for both sets, CoFeed's ability to promote in the list the elements that are really of interest to the user is real, as in these representative sets, a vast majority of such elements appears in the first 5 ranks of the list. Figure 5(b) presents a similar plot, but when considering the 5% most/least popular set of the interest domains. We observe similar results, with unpopular domains ranked much higher in the list for the users who need it.

Repository peak performance. Next, we observe the performance of our prototype by running a single repository $P(Q)$ on a single machine (Core 2 Duo processor at 2.4 GHz with 2 GB memory) submitting synthetic request loads in a synchronous manner: we therefore achieve the highest possible throughput of requests that can be handled by one node in the system. We do not limit the size of the repository, as we want to highlight the relative cost of inserting feedback information and ranking elements as a function of the number of stored elements.

Figure 6 presents the maximal throughput evolution for insertions and ranking requests submitted alternatively. We observe that for reasonable repository sizes (up to 8,000 elements, which we expect to be the common case in practice), the throughput is consistently higher than 100 requests served per second. Note that the costs for one single request increase logarithmically in the size of the repository. The throughput still achieves as many as 50 ranking and 100 insertion requests per second with 30,000 items in the repository.

Routing layer. We measured the distribution of route lengths at the KBR layer for various system side. As expected [18], the distribution of route lengths is balanced around a low average route size (3.7 for 128 nodes, 5.7 for 4,096 nodes, 6.5 for 16,3984 nodes), which grows logarithmically in the system size.

Delegation-based load balancing: efficiency and reactiveness. Figure 7 shows a 3-days experiment using real request load from AOL [16].[4] The experiment evaluates two aspects: a bootstrap phase with no *dramatic* change in the user interest, showing the balancing process with stable popularity distributions, and a second phase with a previously unknown query Q_{pop} (artificially added to the AOL data set) suddenly

[4] Unfortunately we could not use this data set for our evaluation of the profiling and ranking effectiveness because it lacks the necessary feedback information.

generating a massive load in the system followed by a massive loss of popularity. During this time period, the associated $P(Q_{\text{pop}})$ has to efficiently tackle the massive and sudden load imbalance.

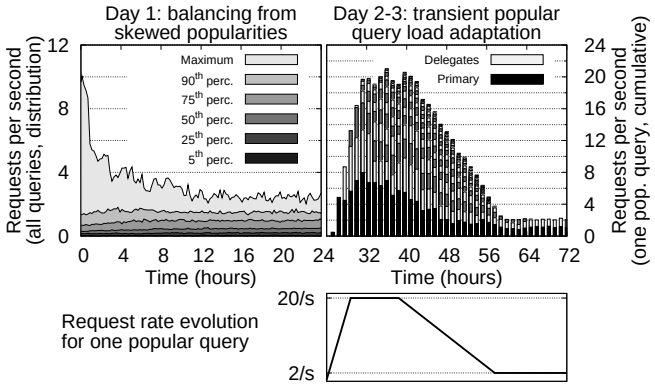


Fig. 7. Evaluation of the delegation mechanism reactiveness and efficiency.

The first day (on the left) presents the evolution of the distribution of the request load on all 1024 nodes, as delegation progressively takes place. The system starts up without bootstrap at time $t_0 = 0$ hours and initially no delegation is made. The evolution of the load distribution is presented by stacking up *percentiles*: the median load is thus represented by the 50^{th} percentile and the maximal load by the lightest shade of gray. We observe that, from an initial high imbalance (where some nodes receive 10 times more load than 50% of all nodes), the system quickly converges to a reasonable imbalance (the most loaded node receives approximately twice as many requests than 50% of the nodes). Further balancing could probably be achieved by modifying γ_{del} and ξ_{del} (see Section 3), but the small extra gain would likely not compensate for the additional synchronization messages necessary to more evenly balance the load.

The two last days (on the right) present the reactiveness of the system *for one single and suddenly popular query* Q_{pop} (while the first day presented results for *all* queries). At time t_s=24 hours, randomly chosen nodes in the system start issuing requests for Q_{pop}. The rate (shown in the bottom-right graph) reaches 20 requests per second in 4.8 hours, i.e., 10 times more than the median overall load at each node; it then remains constant for 9.6 more hours, before decreasing during 19.2 hours. The upper graph presents the load for Q_{pop} at $P(Q_{\text{pop}})$ (black bars) and its delegates (gray bars). Each bar represents the load and number of delegates at the end of a 70-minutes observation period. We observe that the number of delegates follows the popularity trend closely, in both directions (gain and loss). Furthermore, the load at $P(Q_{\text{pop}})$ experiences a small increase in the beginning but remains very low and stable when delegation is active. While some delegates may have only a very small portion of the load, they are still serving 1 or 2 queries per second, i.e., about the median load at all nodes. This is due to delegation decisions being made not based on fixed threshold but on the comparison of the loads of several nodes. Similarly,

some delegates have a higher load than others but the imbalance remains within the limits imposed by the γ_{del} and ξ_{del} parameters.

5 Related Work

Many of the research efforts on P2P Web search focus on decreasing the bandwidth consumption as compared to a centralized approach [5, 12, 14, 22]. However, none of these P2P systems has yet succeeded in gaining sufficient popularity as they all suffer from the bootstrapping problem. CoFeed avoids this problem by leveraging existing search engines and providing added value to the user.

The personalization of search results for a user based on her interest profile was studied by [23,24] but not exploited in the context where knowledge is collaboratively built and aggregated. The use of social annotations (e.g., from bookmarking platforms such as *del.icio.us*) to improve Web search has been recently explored [4,20]. Another example is the *PeerSpective* system [15], which leverages implicit interest between communities of users based on the posting of links from one page to the other on social networks (e.g., FaceBook, MySpace, etc.). Such services operate in a centralized way and require intervention from the user to bookmark and annotate accessed items, which restricts them to a small subset of *power users*.

Our approach is more similar to the Chora [9] and Sixearch [3] systems, which also use decentralized architectures for sharing and leveraging user search experiences. CoFeed differs from these systems in several ways, notably they do not use interest profiling nor do they target information diversity.

A decentralized storage specifically designed for P2P Web search has been proposed in [10] for term frequency-inverse document frequency (TF-IDF). Unlike CoFeed, this system does not provide any mechanism for handling the skew in the popularity of queries, and it does not deal with the terms extraction nor use user-centric information to answer the queries.

Lopes *et al.* have proposed in [13] a storage architecture for large data on top of a DHT, using B+-trees to balance the storage load over several nodes. This architecture was designed for TF-IDF and only supports non-mutable data. Several other systems use the inverse routing paths convergence property, notably for load balancing [21] and or for replication and performance [17].

6 Conclusion

We have presented the architecture and building blocks of a novel *collaborative ranking service*, CoFeed, that can efficiently complement existing search engines. CoFeed leverages user-centric information such as interest profiling and relevance tracking in order to return search result lists tailored to the user interests. Collaborative ranking allows us to present tailored results to users, which can be more relevant especially when the user expectations do not follow the main trend. CoFeed combines methods for interest profiling and mechanisms to maintain *information diversity*. It builds on a support distributed P2P systems that combines classical key-based routing with an application specific storage layer. This layer proposes novel load balancing mechanisms based on the application needs and characteristics.

References

1. http://googleblog.blogspot.com/2008/11/searchwiki-make-search-your-own.html
2. L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.
3. R. Akavipat, L.-S. Wu, F. Menczer, and A. Maguitman. Emerging semantic communities in peer web search. In *P2PIR'06*.
4. S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *WWW'07*.
5. M. Bender, S. Michel, G. Weikum, and C. Zimmer. The Minerva project: Database selection in the context of P2P search. *Datenbanksysteme in Business, Technologie und Web*, 65:125–144, 2005.
6. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
7. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
8. K. Cheng, L. Xiang, M. Iwaihara, H. Xu, and M. M. Mohania. Time-decaying bloom filters for data streams with skewed distributions. In *RIDE-SDMA'05*.
9. H. Gylfason, O. Khan, and G. Schoenebeck. Chora: Expert-based p2p web search. In *AAMAS'06*.
10. F. Klemm and K. Aberer. Aggregation of a term vocabulary for peer-to-peer information retrieval: a DHT stress test. In *DBISP2P'05*.
11. L. Leonini, E. Rivière, and P. Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI'09*.
12. J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. The feasibility of peer-to-peer web indexing and search. In *IPTPS'03*.
13. N. Lopes and C. Baquero. Taming hot-spots in dht inverted indexes. In *LSDS-IR'07*.
14. T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. Alvis peers: A scalable full-text peer-to-peer retrieval engine. In *Proc of P2PIR'06*.
15. A. Mislove, K. P. Gummadi, and P. Druschel. Exploiting social networks for internet search. In *HotNets'06*.
16. G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *InfoScale '06*, New York, NY, USA.
17. V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI'04*.
18. A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware'01*.
19. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP'01*.
20. R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR'08*.
21. S. Serbu, S. Bianchi, P. Kropf, and P. Felber. Dynamic load sharing in peer-to-peer systems: When some peers are more equal than others. *IEEE Internet Computing, Special Issue on Resource Allocation*, 11(4):53–61, 2007.
22. T. Suel, C. Mathur, J.-W. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB'03*.
23. B. Tan, X. Shen, and C. Zhai. Mining long-term search history to improve search accuracy. In *SIGKDD'06*.
24. J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In *SIGIR-IR'05*.