Incremental Algorithms for Effective and Efficient Query Recommendation

Daniele Broccolo[1], Ophir Frieder[2], Franco Maria Nardini[1], Raffaele Perego[1], and Fabrizio Silvestri[1]

[1] ISTI-CNR, Pisa, Italy
[2] Department of Computer Science, Georgetown University, Washington DC, USA

Abstract. Query recommender systems give users hints on possible *interesting queries* relative to their information needs. Most query recommenders are based on static knowledge models built on the basis of past user behaviors recorded in query logs. These models should be periodically updated, or rebuilt from scratch, to keep up with the possible variations in the interests of users. We study query recommender algorithms that generate suggestions on the basis of models that are updated continuously, each time a new query is submitted. We extend two state-of-the-art query recommendation algorithms and evaluate the effects of continuous model updates on their effectiveness and efficiency. Tests conducted on an actual query log show that contrasting model aging by continuously updating the recommendation model is a viable and effective solution.

1 Introduction

A key challenge for web search engines is improving user satisfaction. Therefore, search engine companies exert significant effort to develop means that correctly "guess" what is the real hidden intent behind a submitted query.

In the latest years, web search engines have started to provide users with query recommendations to help them refine queries and to quickly satisfy their needs. Query suggestions are generated according to a model built on the basis of the knowledge extracted from query logs. The model usually contains information on relationships between queries that are used to generate suggestions. Since the model is built on a previously collected snapshot of a query stream, its effectiveness decreases due to interest shifts [9]. To reduce the effect of aging, query recommendation models must be periodically re-built or updated.

We propose two novel incremental algorithms, based on previously proposed, state-of-the-art query recommendation solutions, that update their model continuously on the basis of each new query processed. Designing an effective method to update a recommendation model poses interesting challenges due to: i) *Limited memory availability* – queries are potentially infinite, and we should keep in memory only those queries "really" useful for recommendation purposes,

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 13–24, 2010.

ii) *Low response time* – recommendations and updates must be performed efficiently without degrading user experience.

Some of the approaches considered in related works are not suitable for continuous updates because modifying a portion of the model requires, in general, the modification of the whole structure. Therefore, the update operation would be too expensive to be of practical relevance. Other solutions exploit models which can be built incrementally. The two algorithms we propose use two different approaches to generate recommendations. The first uses association rules for generating recommendations, and it is based on the *static* query suggestion algorithm proposed in [11], while the second uses click-through data, and its *static* version is described in [2].

We named the new class of query recommender algorithms proposed here "*incrementally updating*" query recommender systems to point out that this kind of systems update the model on which recommendations are drawn without the need for rebuilding it from scratch. We conducted multiple tests on a large real-world query log to evaluate the effects of continuous model updates on the effectiveness and the efficiency of the query recommendation process. Result assessment used an evaluation methodology that measures the effectiveness of query recommendation algorithms by means of different metrics. Experiments show the superiority of incrementally updating algorithms with respect to their static counterparts. Moreover, the tests conducted demonstrated that our solution to update the model each time a new query is processed has a limited impact on system response time.

The main contributions presented in this work are: i) a novel class of query recommendation algorithms whose models are continuously updated as user queries are processed, ii) two new metrics to evaluate the quality of the recommendations computed, iii) an analysis of the effect of time on the quality and coverage of the suggestions provided by the algorithms presented and by their static counterparts.

2 Related Work

The wisdom of the crowds, i.e., the behavior of many individuals is smarter than the behavior of few intelligent people, is the key to query recommenders and to many other web 2.0 applications. We now present a review of state-of-the-art techniques for query recommendation.

Document-based. Baeza-Yates *et al.* in [1] propose to compute groups of related queries by running a clustering algorithm over the queries and their associated information recorded in the logs. Semantically similar queries may even not share query-terms if they share relevant terms in the documents clicked by users. Query suggestions are ranked according to two principles: i) the similarity of the queries to the input query, and ii) the support, which measures how much the answers of the query have attracted the attention of users. The solution is evaluated by using a query log containing 6,042 unique queries from the TodoCL search engine.

Click-through-based. Beeferman and Berger in [4] apply a hierarchical agglomerative clustering technique to click-through data to find clusters of similar queries and similar URLs in a Lycos log. A bipartite graph is created from queries and related URLs which is iteratively clustered by choosing at each iteration the two pairs of most similar queries and URLs. The experimental evaluation shows that the proposed solution is able to enhance the quality of the Lycos's query recommender which was used as baseline.

Cao *et al.* propose a query suggestion approach based on contexts [8]. A query context consists of recent queries issued by a user. The query suggestion process is structured according to two steps. An offline phase summarizes user queries into concepts (i.e., a small set of similar queries) by clustering a click-through bipartite graph, and an on-line step finds the context of the submitted query, and its related concepts suggesting associated queries to the user. This solution was experimented with a large query log containing $1,812$ millions of queries.

Session-based. Boldi *et al.* introduce the concept of *Query Flow Graph* [5] (QFG). Authors define a QFG as a directed graph in which nodes are queries, and edges are weighted by the probability $w(q_i, q_j)$ of being traversed. Authors highlight the utility of the model in two concrete applications, namely, *finding logical sessions* and *query recommendation*. Boldi *et al.* refine the previous study in [6], [7] proposing a query suggestion scheme based on a random walk with restart model. The query recommendation process is based on reformulations of search missions. Baraglia *et al.* showed that the QFG model ages [3] and propose strategies for updating it efficiently.

Fonseca *et al.* use an association rule mining algorithm to devise query patterns frequently co-occurring in user sessions, and a query relations graph including all the extracted patterns is built [10]. A click-through bipartite graph is then used to identify the concepts (synonym, specialization, generalization, etc.) used to expand the original query.

3 Incremental Algorithms for Query Recommendation

Our hypothesis is that continuously updating the query recommendation model is feasible and useful. As validation, we consider two well-known query recommendation algorithms and modify them to continuously update the model on which recommendations are computed. It is worth mentioning that not all query recommendation algorithms can be redesigned to update their model on-line. For example, some of the approaches presented in Section 2 are based on indexing terms of documents selected by users, clustering click-through data, or extracting knowledge from users' sessions. Such operations are very expensive to perform on-line and their high computational costs would compromise the efficiency of the recommender system.

The two algorithms considered use different approaches for generating recommendations. The first uses association rules [11] (henceforth *AssociationRules*), while the second exploits click-through data [2] (henceforth *CoverGraph*). Hereinafter, we will refer to the original formulations of the two algorithms as "*static*",

as opposed to their relative incremental versions which will be called "*incremental*".

3.1 Static solutions

Static solutions work by preprocessing historical data (represented by past users' activities on query logs), building an on-line recommendation module that is used to provide suggestions to users.

AssociationRules. Fonseca *et al.* uses association rules as a basis for generating recommendations [11]. The algorithm is based on two main phases. The first uses query log analysis for session extraction, and the second basically extracts association rules and identifies highly related queries. Each session is identified by all queries sent by an user in a specific time interval ($t = 10$ minutes). Let $I = I_1, \ldots, I_m$ be the set of queries and T the set of user sessions t. A session $t \in T$ is represented as a vector where $t_k = 1$ if session t contains query $k \in [1, \ldots, m]$, 0 otherwise.

Let X be a subset of I. A session t satisfies X, if for all items I_k in X, $t_k = 1$.

Association rules are implications of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds with i) a *confidence* factor of c if $c\%$ of the transactions in T that contains X also contains Y, and ii) a *support* s if $s\%$ of the sessions in T contains $X \cup Y$. The problem of mining associations is to generate all the rules having a support greater than a specified minimum threshold (*minsup*). The rationale is that distinct queries are considered related if they occurs in many user sessions.

Suggestions for a query q are simply computed by accessing the list of rules of the form $q \Rightarrow q'$ and by suggesting the q''s corresponding to rules with the highest support values.

CoverGraph. Baeza-Yates et al. use click-through data as a way to provide recommendations [2]. The method is based on the concept of *cover graph*. A *cover graph* is a bipartite graph of queries and URLs, where a query and a URL are connected if the URL was returned as a result for the query and a user clicked on it.

To catch the relations between queries, a graph is built out of a vectorial representation for queries. In such a vector-space, queries are points in a high-dimensional space where each dimension corresponds to a unique URL u that was, at some point, clicked by some user. Each component of the vector is weighted according to the number of times the corresponding URL has been clicked when returned for that query. For instance, suppose we have five different URLs, namely, $u_1, u_2, \ldots, u_5$, suppose also that for query q users have clicked three times URL u_2 and four times URL u_4, the corresponding vector is $(0, 3, 0, 4, 0)$. Queries are then arranged as a graph with two queries being connected by an edge if and only if the two queries share a non-zero entry, that is, if for two different queries the same URL received at least one click. Furthermore, edges are weighted according to the cosine similarity of the queries they connect. More formally, the weight of an edge $e = (q, q')$ is computed according

to Equation 1. In the formula, D is the number of dimensions, i.e., the number of distinct clicked URLs, of the space.

$$W(q, q') = \frac{q \cdot q'}{|q| \cdot |q'|} = \frac{\sum_{i \leq D} q_i \cdot q'_i}{\sqrt{\sum_{i \leq D} q_i^2} \sqrt{\sum_{i \leq D} {q'_i}^2}} \quad (1)$$

Suggestions for a query q are obtained by accessing the corresponding node in the cover graph and extracting the queries at the end of the top scoring edges.

3.2 Incremental algorithms

The interests of search-engine users change over time, and new topics may become popular. Consequently, the knowledge extracted from query logs can suffer from an aging effect, and the models used for recommendations rapidly become unable to generate useful and interesting suggestions [3]. Furthermore, the presence of "*bursty*" [12] topics could require frequent model updates independent of the model used.

The algorithms proposed in Section 3.1 use a statically built model to compute recommendations. *Incremental* algorithms are radically different from static methods for the way they build and use recommendation models. While static algorithms need an off-line preprocessing phase to build the model from scratch every time an update of the knowledge base is needed, incremental algorithms consist of a single online module integrating the two functionalities: i) *updating* the model, and ii) providing suggestions for each query.

Starting from the two algorithms presented above, we design two new query recommender methods continuously updating their models as queries are issued. Algorithms 1 and 2 formalize the structure of the two proposed incremental algorithms that are detailed in the following. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. Both algorithms exploit *LRU* caches and *Hash* tables to store and retrieve efficiently queries and links during the model update phase.

Our two incremental algorithms are inspired by the *Data Stream Model* [13] in which a stream of queries are processed by a database system. Queries consist modifications of values associated with a set of data. When the dataset fits completely in memory, satisfying queries is straightforward. Turns out that the entire set of data cannot be contained in memory. Therefore, an algorithm in the data stream model must decide, at each time step, which subset of the set of data is worthwhile to maintain in memory. The goal is to attain an approximation of the results we would have had in the case of the non-streaming model. We make a first step towards a data stream model algorithmic framework aimed at building query recommendations. We are aware that there is significant room for improvement, especially in the formalization of the problem in the streaming model. Nonetheless, we show empirically that an incremental formulation of two popular query recommender maintains the high accuracy of suggestions.

IAssociationRules. Algorithm 1 specifies the operations performed by *IAssociationRules*, the incremental version of AssociationRules.

Algorithm 1. IAssociationRules

```
1: loop
2:    (u, q) ← GetNextQuery() {Get the query q and the user u who submitted it.}
3:    ComputeSuggestions (q, σ) {Compute suggestions for query q over σ.}
4:    if ∃LastQuery (u) then
5:       q' ← LastQuery (u)
6:       LastQuery (u) ← q {Update the last query submitted by u.}
7:       if ∃σ_{q',q} then
8:          ++σ_{q',q} {Increment Support for q' ⇒ q.}
9:       else
10:         LRUInsert (σ, (q', q)) {Insert an entry for (q', q) in σ. If σ is full, remove
            an entry according to an LRU policy.}
11:      end if
12:   else
13:      LRUInsert (u, q, LastQuery) {Insert an entry for (u, q) in LastQuery. If
         LastQuery is full, remove an entry according to an LRU policy.}
14:   end if
15: end loop
```

The data structures storing the model are updated at each iteration. We use the LastQuery auxiliary data structure to record the last query submitted by u. Since the model and the size of LastQuery could grow indefinitely, whenever they are full, the LRUInsert function is performed to keep in both structures only the most recently used entries.

Claim. Keeping up-to-date the AssociationRule-based model is $O(1)$.

The proof of the claim is straightforward. The loop at line 3 of Algorithm 1 is made up of constant-cost operations (whenever we use hash structures for both LastQuery and σ). LRUInsert has been introduced to maintain the most recently submitted queries in the model.

ICoverGraph. The incremental version of CoverGraph adopts a solution similar to that used by IAssociationRules. It uses a combination of LRU structures and associative arrays to incrementally update the (LRU managed) structure σ. Algorithm 2 shows the description of the algorithm. The hash table queryHasAClickOn is used to retrieve the list of queries having c among their clicked URLs. This data structure is stored in a fixed amount of memory, and whenever its size exceeds the allocated capacity, an entry is removed on the basis of a LRU policy (this justifies the conditional statement at line 6).

Claim. Keeping up-to-dated a CoverGraph-based model is $O(1)$.

Actually, the cost depends on the degree of each query/node in the cover graph. As shown in [2], i) the degree of nodes in the cover graph follows a power-law

Algorithm 2. ICoverGraph

```
1: Input: A threshold τ.
2: loop
3:    (u, q) ← GetNextQuery() {Get the query q and the user u who submitted it.}
4:    ComputeSuggestions (q, σ) {Compute suggestions for query q over σ.}
5:    c = GetClicks (u, q)
6:    if ∃queryHasAClickOn (c) then
7:       queryHasAClickOn (c) ← q
8:    else
9:       LRUInsert (queryHasAClickOn, c)
10:   end if
11:   for all q' ≠ q ∈ queryHasAClickOn (c) s.t. W ((q, q')) > τ do
12:      if w > τ then
13:         if ∃σ_{q',q} then
14:            σ_{q,q'} = w
15:         else
16:            LRUInsert (σ, (q', q) , w)
17:         end if
18:      end if
19:   end for
20: end loop
```

distribution, and ii) the maximal number of URLs between two queries/nodes is constant, on average. The number of iterations needed in the loop at line 11 can be thus considered constant.

From the above methods, it is clear that to effectively produce recommendations, a continuous updating algorithm should have the following characteristics:

- The algorithm must cope with an undefined number of queries. LRU caches can be used to allow the algorithm to effectively keep in memory only the most relevant items for which it is important to produce recommendations.
- The lookup structures used to generate suggestions and maintain the models must be efficient, possibly constant in time. Random-walks on graph-based structures, or distance functions based on comparing portions of texts, etc., are not suitable for our purpose.
- A modification of an item in the model must not involve a modification of the entire model. Otherwise, update operations take too much time and jeopardize the efficiency of the method.

4 Quality Metrics

Assessing the effectiveness of recommender systems is a though problem that can be addressed either through *user-studies* or via automatic evaluation mechanisms.

We opted for an automatic evaluation methodology conducted by means of two novel metrics based on the analysis of users' traces contained in query logs. Both metrics measure the overlap between queries actually submitted by the

users and recorded in the tails of users' sessions and suggestions generated starting from the first queries in the same sessions. The more users actually submitted queries suggested, the more the recommender system is considered effective.

To focus the evaluation on the most recently submitted queries in the user session, we introduce the *QueryOverlap* and the *LinkOverlap* metrics defined as follows. Let $\mathbb{S}$ be the set of all users' sessions in the query log, and let $S = \{q_1, \ldots, q_n\}$ be a user session of length n. We define $S_1 = \{q_1, \ldots, q_{\lfloor \frac{n}{2} \rfloor}\}$ to be the set of queries in the first half of the session, and let $R_j = \{r_1, \ldots, r_m\}$ be the set of top-m query recommendations returned for the query $q_j \in S_1$[1]. For each q_j, we now define $S_2 = \{q_{j+1}, \ldots, q_n\}$ to be the $n - j$ most recently submitted queries in the session, and

$$QueryOverlap = \frac{1}{K} \sum_{\substack{r_i \in R_j \\ s_k \in S_2}} [r_i = s_k] f(k) \tag{2}$$

$$LinkOverlap = \frac{1}{K} \sum_{\substack{r_i \in \mathrm{findClk}(R_j) \\ s_k \in \mathrm{clk}(S_2)}} [r_i = s_k] f(k) \tag{3}$$

where $[expr]$ is a boolean function whose result is 1 if $expr$ is *true* or 0 otherwise, $clk(S_2)$ is a function returning the set of clicked URLs by the user for the queries in S_2, $findClk(R_j)$ is a function returning the set of clicked URLs by other users for the queries in R_j, and $f(k)$ is a weighting function allowing us to differentiate the importance of each recommendation depending on the position it occupies in the second part of the session. The value of K is defined as $\sum_{k=1}^{m} f(k)$, where $m = |S_2|$ for the *QueryOverlap*, and $m = |clk(S_2)|$ for the *LinkOverlap* metric. K normalizes the values in the range $[0, 1]$. Finally, the *Coverage* of a recommendation model is defined as the fraction of queries for which a recommendation can be computed.

5 Experiments

5.1 Experimental Setup

We conducted our experiments on a collection consisting of the first $3,200,000$ queries from the AOL query log [14]. The AOL data-set contains about 20 million queries issued by about $650,000$ different users, submitted to the AOL search engine over a period of three months from 1st March, 2006 to 31st May, 2006.

5.2 Results

First, we analyze the effect of time on the static models (Section 3.1) showing that this type of models age as time passes.

[1] In our experiments we use $m = 5$.

The plots reported in Figures 1, 2, and 3, show the effectiveness of query suggestions on a per time window basis for both the static and incremental algorithms. We use a "timeline" composed of 10 days of the query log. The "timeline" is divided into ten intervals, each corresponding to one day of queries stored in the query log (about 400,000 queries). The queries in the first time interval were used to train the models used by the algorithms. While static models are trained only on the first interval, the incremental counterparts update their model on the basis of the queries submitted during the entire timeline considered. Effectiveness of recommendations generated by the different algorithms during the remaining nine days considered is measured by means of the *LinkOverlap*, *QueryOverlap*, and *Coverage* metrics.

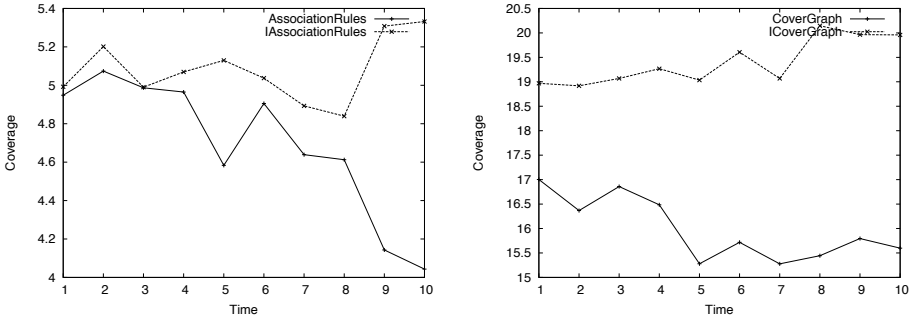

Fig. 1. Coverage for AssociationRules, IAssociationRules, CoverGraph, and ICoverGraph as a function of the time

Our first finding is illustrated in Figure 1, where coverage, i.e., the percentage of queries for which the algorithms are able to generate recommendations, is plotted as a function of time. In both plots, the coverage measured for the static versions of the recommendation algorithms decreases as time passes. In particular, at the end of the observed period, AssociationRules and CoverGraph lose 20%, and 9% of their initial coverage, respectively. Even if CoverGraph appears to be more robust than AssociationRules, both algorithms suffer an aging effect on their models. On the other hand, the coverage measured for the two incremental algorithms is always greater than the one measured for their respective static versions. In particular, at the end of the observed period, the IAssociationRules algorithm covers 23.5% more queries with respect to its static version, while ICoverGraph covers 22% more queries with respect to CoverGraph. This is due to the inclusion in the model of new and "fresh" data.

Figures 2 and 3 illustrate the effectiveness of recommendations produced by the static and incremental versions of the AssociationRules and CoverGraph algorithms as a function of the time. Both QueryOverlap and LinkOverlap in Figures 2 and 3 are measured in the above described setting.

From the plots we can see that the two static algorithms behave in a slightly different way. CoverGraph seems to suffer more than AssociationRules for the aging of its recommendation model.

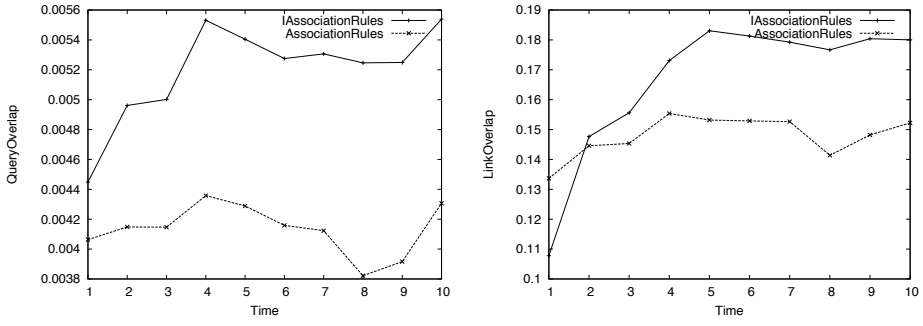

Fig. 2. QueryOverlap, and LinkOverlap for AssociationRules, and IAssociationRules as a function of the time

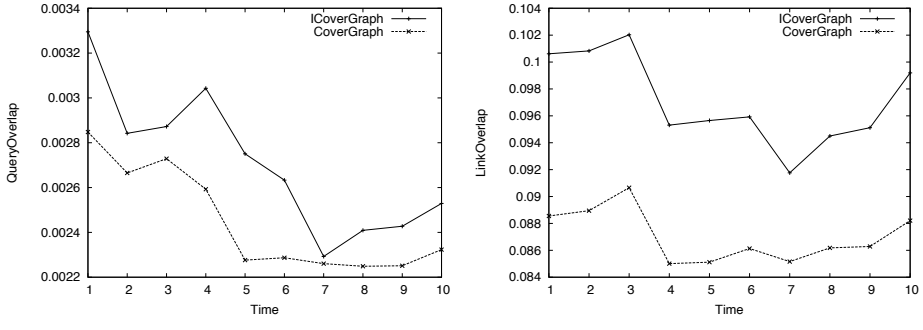

Fig. 3. QueryOverlap, and LinkOverlap for CoverGraph, and ICoverGraph as a function of the time

By considering both QueryOverlap and LinkOverlap metrics, AssociationRules is able to return better quality recommendations than CoverGraph, but, as Figure 1 shows, the coverage of queries for which suggestions can be generated is lower. In particular, CoverGraph is able to give suggestions to a number of queries which is three times larger than the one measured with AssociationRules.

We argue that incremental algorithms for query recommendation can provide better recommendations because they do not suffer for model aging, and can rapidly cover also *bursty* topics. From the figures, it is evident that the effectiveness of recommendations provided by both static and incremental models eventually stabilize. Indeed, the proposed incremental algorithms IAssociationRules and ICoverGraph produce better recommendations. With the exception of the initialization phase (see Figure 2, LinkOverlap) in which the model warms up, the percentage of effective suggestions generated by the two incremental algorithms during the entire period observed is larger than those provided by their static counterparts.

5.3 Efficiency Evaluation

The feasibility of an incremental update of the recommendation model is an important point of our work. The update operations must run in parallel with the query processor; thus, those operations must not constitute a bottleneck for the entire system. As analyzed in Section 3.2, we propose a method for keeping up-to-date two algorithms in constant time. The payoff in terms of processing time is, thus, constant. Furthermore, in the incremental algorithms we use efficient data structures, and an optimized implementation of the model update algorithm. We measure the performances of the two incremental algorithms in terms of mean response time. For each new query, our algorithms are able to update the model, and to produce suggestions in, on-the-order-of, a few tenth of a second. Such response times guarantee the feasibility of the approach on a real-world search engine where the query recommender and the query processor run in parallel.

6 Conclusions

We studied the effects of incremental model updates on the effectiveness of two query suggestion algorithms. As the interests of search-engine users change over time and new topics become popular, the knowledge extracted from historical usage data can suffer an aging effect. Consequently, the models used for recommendations may rapidly become unable to generate high-quality and interesting suggestions.

We introduced a new class of query recommender algorithms that update "*incrementally*" the model on which recommendations are drawn. Starting from two state-of-the-art algorithms, we designed two new query recommender systems that continuously update their models as queries are issued. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. In addition, we proposed an automatic evaluation mechanism based on two new metrics to assess the effectiveness of query recommendation algorithms.

The experimental evaluation conducted by using a large real-world query log shows that the incremental update strategy for the recommendation model yields better results for both coverage (more than 20% queries covered by both IAssociationRules, and ICoverGraph) and effectiveness due to the "fresh" data that are added to the recommendation models. Furthermore, this improved effectiveness is accomplished without compromising the efficiency of the query suggestion process.

References

1. Baeza-Yates, R., Hurtado, C., Mendoza, M.: Query recommendation using query logs in search engines. In: Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) EDBT 2004. LNCS, vol. 3268, pp. 588–596. Springer, Heidelberg (2004)
2. Baeza-Yates, R., Tiberi, A.: Extracting semantic relations from query logs. In: KDD 2007, pp. 76–85. ACM, New York (2007)

3. Baraglia, R., Castillo, C., Donato, D., Nardini, F.M., Perego, R., Silvestri, F.: The effects of time on query flow graph-based models for query suggestion. In: Proc. RIAO 2010 (2010)
4. Beeferman, D., Berger, A.: Agglomerative clustering of a search engine query log. In: Proc. KDD 2000, pp. 407–416. ACM, New York (2000)
5. Boldi, P., Bonchi, F., Castillo, C., Donato, D., Gionis, A., Vigna, S.: The query-flow graph: model and applications. In: Proc. CIKM 2008, pp. 609–618 (2008)
6. Boldi, P., Bonchi, F., Castillo, C., Donato, D., Vigna, S.: Query suggestions using query-flow graphs. In: Proc. WSCD 2009, pp. 56–63. ACM, New York (2009)
7. Boldi, P., Bonchi, F., Castillo, C., Vigna, S.: From 'dango' to 'japanese cakes': Query reformulation models and patterns. In: Proc. WI 2009. IEEE CS, Los Alamitos (2009)
8. Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., Li, H.: Context-aware query suggestion by mining click-through and session data. In: Proc. KDD 2008, pp. 875–883 (2008)
9. Cayci, A., Sumengen, S., Turkay, C., Balcisoy, S., Saygin, Y.: Temporal dynamics of user interests in web search queries. ICAINAW 0, 762–767 (2009)
10. Fonseca, B.M., Golgher, P., Pôssas, B., Ribeiro-Neto, B., Ziviani, N.: Concept-based interactive query expansion. In: Proc. CIKM 2005. ACM, New York (2005)
11. Fonseca, B.M., Golgher, P.B., de Moura, E.S., Ziviani, N.: Using association rules to discover search engines related queries. In: LA-WEB 2003. IEEE CS, Los Alamitos (2003)
12. Kleinberg, J.: Bursty and hierarchical structure in streams. In: Proc. KDD 2002. ACM, New York (2002)
13. Muthukrishnan, S.: Data streams: algorithms and applications. Found. Trends Theor. Comput. Sci. 1(2), 117–236 (2005)
14. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proc. INFOSCALE 2006. ACM, New York (2006)