Mining Concept Sequences from Large-Scale Search Logs for Context-Aware Query Suggestion

ZHEN LIAO, Nankai University
DAXIN JIANG, Microsoft Research Asia
ENHONG CHEN, University of Science and Technology of China
JIAN PEI, Simon Fraser University
HUANHUAN CAO, University of Science and Technology of China
HANG LI, Microsoft Research Asia

Query suggestion plays an important role in improving usability of search engines. Although some recently proposed methods provide query suggestions by mining query patterns from search logs, none of them models the immediately preceding queries as context systematically, and uses context information effectively in query suggestions. Context-aware query suggestion is challenging in both modeling context and scaling up query suggestion using context. In this article, we propose a novel context-aware query suggestion approach. To tackle the challenges, our approach consists of two stages. In the first, *offline model-learning stage*, to address data sparseness, queries are summarized into concepts by clustering a click-through bipartite. A *concept sequence suffix tree* is then constructed from session data as a context-aware query suggestion model. In the second, *online query suggestion stage*, a user's search context is captured by mapping the query sequence submitted by the user to a sequence of concepts. By looking up the context in the concept sequence suffix tree, we suggest to the user context-aware queries. We test our approach on large-scale search logs of a commercial search engine containing 4.0 billion Web queries, 5.9 billion clicks, and 1.87 billion search sessions. The experimental results clearly show that our approach outperforms three baseline methods in both coverage and quality of suggestions.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Query suggestion, context-aware, click-through data, session data

ACM Reference Format:
Liao, Z., Jiang, D., Chen, E., Pei, J., Cao, H., and Li, H. 2011. Mining concept sequences from large-scale search logs for context-aware query suggestion. ACM Trans. Intell. Syst. Technol. 3, 1, Article 17 (October 2011), 40 pages.
DOI = 10.1145/2036264.2036281 http://doi.acm.org/10.1145/2036264.2036281

The research of E. Chen is supported in part by Natural Science Foundation of China (grant no. 61073110), and Research Fund for the Doctoral Program of Higher Education of China (20093402110017). The research of J. Pei is supported in part by Natural Sciences and Engineer Research Council of Canada (NSERC) through an NSERC Discovery Grant, and by BCFRST Foundation Natural Resources and Applied Sciences (NRAS) Endowment Research Team Program. All opinions, findings, conclusions and recommendations in this article are those of the authors and do not necessarily reflect the views of the funding agencies.

Authors' addresses: Z. Liao, College of Information Technical Science, Nankai University, Tianjin, China; D. Jiang (corresponding author), Microsoft Research Asia, Beijing, China; email: djiang@microsoft.com; E. Chen, College of Computer Science, University of Science and Technology of China; J. Pei, School of Computer Science, Simon Fraser University; H. Cao, College of Computer Science, University of Science and Technology of China; H. Li, Microsoft Research Asia, Beijing, China.

1. INTRODUCTION

The effectiveness of a user's information retrieval from the Web largely depends on whether the user can raise queries properly describing the information need to search engines. Writing queries is never easy, partially because queries are typically expressed in a very small number of words (two or three words on average) [Jansen et al. 1998] and many words are ambiguous [Cui et al. 2002]. To make the problem even more complicated, different search engines may respond differently to the same query. Therefore, there is no "standard" or "optimal" way to raise queries to all search engines, and it is well recognized that query formulation is a bottleneck issue in the usability of search engines. Recently, most commercial search engines such as Google, Yahoo!, and Bing provide *query suggestions* to improve usability. That is, by guessing a user's search intent, a search engine can suggest queries which may better reflect the user's information need. A commonly used query suggestion method [Baeza-Yates et al. 2004; Beeferman and Berger 2000; Wen et al. 2001] is to find similar queries in search logs and use those queries as suggestions for each other. Another approach [Huang et al. 2003; Jensen et al. 2006] mines pairs of queries which are adjacent or co-occur in the same query sessions.

Although the existing methods may suggest good queries in some scenarios, none of them models the immediately preceding queries as context systematically, and uses context information effectively in query suggestions. Context-aware query suggestion is challenging in both modeling context and scaling up query suggestion using context.

Example 1.1 (*Search Intent and Context*). Suppose a user raises a query "*gladiator*". It is hard to determine the user's search intent, that is, whether the user is interested in the history of gladiator, famous gladiators, or the film *Gladiator*. Without looking at the context of search, the existing methods often suggest many queries for various possible intents, and thus result in a low accuracy in query suggestion.

If we find that the user submits a query "*beautiful mind*" before "*gladiator*", it is very likely that the user is interested in the film *Gladiator*. Moreover, the user is probably searching the films played by Russell Crowe. The *query context* which consists of the search intent expressed by the user's recent queries can help to better understand the user's search intent and make more meaningful suggestions.

In this article, we propose a novel context-aware approach for query suggestion by mining concept sequences from click-through data and session data. When a user submits a query q, our context-aware approach first captures the context of q, which is reflected by a short sequence of queries issued by the same user immediately before q. The historical data are then checked to find out what queries many users often ask after q in the same context. Those queries become the candidates of suggestion.

There are two critical issues in the context-aware approach. First, how should we model and capture contexts well? Users may raise various queries to describe the same information need. For example, to search for Microsoft Research Asia, queries "*Microsoft Research Asia*", "*MSRA*" or "*MS Research Beijing*" may be formulated. Directly using individual queries to describe context cannot capture contexts concisely and accurately.

To tackle this problem, we propose summarizing individual queries into *concepts*, where a concept consists of a small set of queries that are similar to each other. Using concepts to describe contexts, we can address the sparseness of queries and interpret users' search intent more accurately.

To mine concepts from queries, we use the URLs clicked by users as the features of the corresponding queries. In other words, we mine concepts by clustering queries in a click-through bipartite. Moreover, to cover new queries not in the click-through

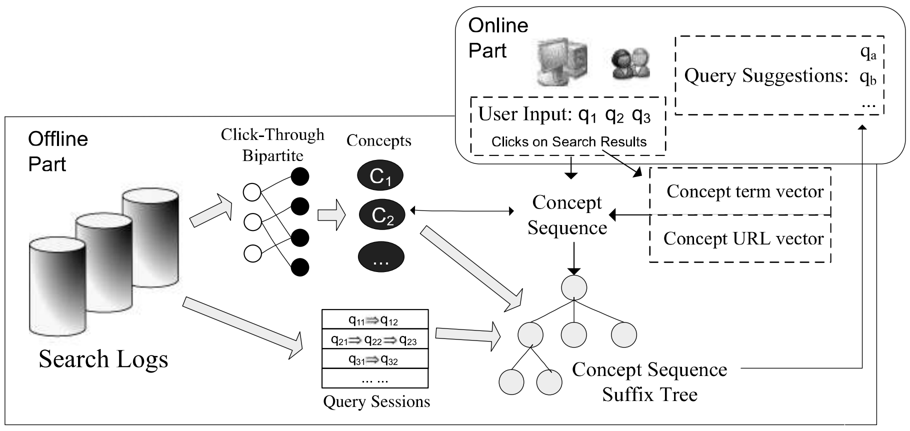

Online Part
Query Suggestions: q_a q_b ...
User Input: q_1 q_2 q_3
Clicks on Search Results
Offline Part
Click-Through Bipartite
Concepts
C_1
C_2
...
Concept Sequence
Concept term vector
Concept URL vector
Search Logs
$q_{11} \Rightarrow q_{12}$
$q_{21} \Rightarrow q_{22} \Rightarrow q_{23}$
$q_{31} \Rightarrow q_{32}$
... ...
Query Sessions
Concept Sequence Suffix Tree

Fig. 1. The framework of our approach.

bipartite, we represent each concept by a URL feature vector and a term feature vector. We will describe how to mine concepts of queries and build feature vectors for concepts in Section 3.

With the help of concepts, a context can be represented by a short sequence of concepts corresponding to the queries asked by a user in the current session. The next issue is that, given a particular context, what queries many users often ask in the following.

It is infeasible to search a huge search log online for a given context. We propose a context mining method which mines frequent contexts from historical sessions in search logs. The frequent contexts mined are organized into a concept sequence suffix tree structure which can be searched quickly. The previous mining process is conducted offline. In the online stage, when a user's input is received, we map the sequence of queries into a sequence of concepts as the user's search context. We then look up the context in the concept sequence suffix tree to find out the concepts to which the user's next query most likely belongs, and suggest the most popular queries in those concepts to the user. The details about mining sessions, building a concept sequence suffix tree, and making query suggestions are discussed in Section 4.

Figure 1 shows the framework of our context-aware approach, which consists of two stages. The offline model-learning stage mines concepts from a click-through bipartite constructed from search logs, creates feature vectors for the concepts, and builds a concept sequence suffix tree from the sessions in the logs. The online query suggestion stage maps user query sequences into a concept sequence, looks up the concept sequence against the concept sequence suffix tree, finds the concepts that the user's next query may belong to, and suggests the most popular queries in the concepts. The major contributions of this work are summarized as follows.

First, instead of mining patterns of individual queries which may be sparse, we summarize queries into concepts. A concept is a group of similar queries. Although mining concepts of queries can be reduced to a clustering problem on a bipartite graph, the very large data size and the "curse of dimensionality" pose great challenges. To tackle these challenges, we develop a novel and effective clustering algorithm in linear time complexity. We further increase the scalability of the clustering algorithm through two approaches.

Second, there are often a huge number of patterns that can be used for query suggestion. Mining those patterns and organizing them properly for online query suggestion is far from trivial. We develop a novel *concept sequence suffix tree* structure to address this challenge.

Third, users raise new queries to search engines everyday. Most log-based query suggestion methods cannot handle novel queries that do not appear in the historical data. To tackle this challenge, we represent each concept derived from the log data by a *URL feature vector* and a *term feature vector*. New queries can thus be recognized as belonging to some concepts through the two types of feature vectors.

Fourth, we conduct extensive studies on a large-scale search log dataset which contains 4.0 billion Web queries, 5.9 billion clicks, and 1.87 billion query sessions. We explore several interesting properties of the click-through bipartite and illustrate several important statistics of the session data. The data set in this study is several magnitudes larger than those reported in previous work.

Last, we test our approach on the search log data. The experimental results clearly show that our approach outperforms three baseline methods in both coverage and quality of suggestions.

The rest of the article is organized as follows. We first review the related work in Section 2. The clustering algorithm and the query suggestion method are described in Sections 3 and 4, respectively. We report an empirical study in Section 5. The article is concluded in Section 6.

2. RELATED WORK

A great challenge for search engines is to understand users' search intent behind queries. Traditional approaches to query understanding focus on exploiting information such as users' explicit feedbacks (e.g., Magennis and van Rijsbergen [1997]), implicit feedbacks (e.g., Terra and Clarkem [2004]), user profiles (e.g., Chirita et al. [2007]), thesaurus (e.g., Liu et al. [2004]), snippets (e.g., Sahami and Heilman [2006]), and anchor texts (e.g., Kraft and Zien [2004]). Several recent studies have used search logs to mine "the wisdom of crowds" for query suggestions. In general, those methods can be divided into *session-based* approaches and *document-click-based* approaches.

In session-based approaches, query pairs which are often adjacent or co-occurring in the same sessions are mined as candidates for query suggestion. For example, Huang et al. [2003] mined co-occurring query pairs from session data and ranked the candidates based on their frequency of co-occurrence with the user input queries. Jensen et al. [2006] considered not only the co-occurrence frequency of the candidates, but also their mutual information with the user input queries. To address the sparseness of the data, the authors treated query suggestions at phrase level instead of query level. Moreover, to further improve the coverage of the query suggestion method, the authors manually mapped query terms to topics and then aggregated the co-occurrence patterns at topic level.

Boldi et al. [2008] built a *query-flow graph* where each node represents a distinct query and a directed edge from query q_i to query q_j means that at least one user submitted query q_j immediately after submitting q_i in the same session. An edge (q_i, q_j) is also associated with some weight to indicate how likely a user moves from q_i to q_j. To generate query suggestions, the edge weights were estimated by the frequency of observed transitions from q_i to q_j in search logs, and a straightforward method is to return the queries q_j which have the largest weights of edges (q_i, q_j). Other methods conduct random walks starting from either the given query q_i or the last k queries visited before q_i in the same session.

Several studies extended the work in Boldi et al. [2008] along various directions. For example, the study in Boldi et al. [2009] suggested labeling the edges in a query-flow graph into four categories, namely, generalization, specialization, error correction, and parallel move, and only using the edges labeled as specialization for query suggestion.

Anagnostopoulos et al. [2010] argued that providing query suggestions to users may change user behavior. They thus modeled query suggestions as shortcut links on a query-flow graph and considered the resulted graph as a perturbed version of the original one. Then the problem of query suggestion was formalized as maximizing the utility function of the paths on the perturbed query-flow graph. Sadikov et al. [2010] extended the query-flow graph by introducing the clicked documents for each query. The queries q_j following a given query q_i are clustered together if they share many clicked documents.

Unlike the preceding session-based methods which only focus on query pairs, we model various-length context of the current query, and provide context-aware suggestions. Although Boldi et al. [2008] and Huang et al. [2003] used the preceding queries to prune the candidate suggestions, they did not model the sequential relationship between the preceding queries in a systematic way. Moreover, most existing session-based methods focused on individual queries in context modeling and suggestion generation. However, as mentioned in Section 1 (Introduction), user queries are typically sparse. Consequently, the generated query suggestions could be very similar to each other. For example, it is possible in a query-flow graph that two very similar queries q_{j1} and q_{j2} are both connected to query q_i with high weights. When q_{j1} and q_{j2} are both presented to users as suggestions for q_i, the user-perceived information would be redundant. Although the method by Sadikov et al. [2010] grouped similar queries in candidate suggestions, it cannot handle the sparseness of context. Our approach summarizes similar queries into concepts and uses concepts in both context modeling and suggestion generation. Therefore, it is more effective to address the sparseness of queries.

The document-click-based approaches focus on mining similar queries from a click-through bipartite constructed from search logs. The basic assumption is that two queries are similar to each other if they share a large number of clicked URLs. For example, Mei et al. [2008] performed a random walk starting from a given query q on the click-through bipartite to find queries similar to q. Each similar query q_i is labeled with a "hitting time," which is essentially the expected number of random walk steps to reach q_i starting from q. The queries with the smallest hitting time were selected as the query suggestions. Different from our method, the "hitting time" approach does not summarize similar queries into concepts. Moreover, it does not consider the context information when generating query suggestions. Other methods applied various clustering algorithms to the click-through bipartite. After the clustering process, the queries within the same cluster are used as suggestions for each other. For example, Beeferman and Berger [2000] applied a hierarchical agglomerative method to obtain similar queries in an iterative way. Wen et al. [2001] combined query content information and click-through information and applied a density-based method, DBSCAN [Ester et al. 1996], to cluster queries. These two approaches are effective to group similar queries. However, both methods have high computational cost and cannot scale up to large data. Baeza-Yates et al. [2004] used the efficient k-means algorithm to derive similar queries. However, the k-means algorithm requires a user to specify the number of clusters, which is difficult for clustering search logs.

There are some other efficient clustering methods such as BIRCH [Zhang et al. 1996] though they have not been adopted in query clustering. In BIRCH, the algorithm constructs a Clustering Feature (*CF* for short) vector for each cluster. The CF vector consists of the number N of data objects in the cluster, the linear sum $\overrightarrow{LS}$ of the N data objects, and the squared sum SS of the N data points. The algorithm then scans the data set once and builds a hierarchical CF tree to index the clusters. Although the BIRCH algorithm is very efficient, it may not handle high dimensionality well. As shown in previous studies (e.g., Hinneburg and Keim [1999]), when the dimensionality increases, BIRCH tends to compress the whole dataset into a single data item. In this article, we borrow the CF vector from BIRCH. However, to address the "curse of dimensionality" caused by the large number of URLs used as dimensions for queries in logs, we do not build CF trees. Instead, we develop the novel dimension array to leverage the characteristics of search log data.

The approach developed in this article also clusters similar queries using the click-through bipartite. However, different from the previous document-click-based approaches which suggest queries from the same cluster of the current query, our approach suggests queries that a user may ask in next step, which are more interesting than queries simply replaceable to the current query.

To a broader extent, our method for query suggestion is also related to the methods for *query expansion* and *query substitution*, both of which also target at helping search engine users to formulate good queries.

Query expansion involves adding new terms to the original query. Traditional IR methods often select candidate expansion terms or phrases from pseudorelevance feedbacks. For example, Lavrenko and Croft [2001] created language models from the pseudorelevance documents and estimated the joint distribution $P(t, q_1, \ldots, q_l)$, where t is a candidate term for expansion and $q_1, \ldots, q_l$ are the terms in the given query. Several other studies used machine learning approaches to integrate richer features in addition to term distributions. For example, Metzler and Croft [2007] applied a Markov Random Field model and Cao et al. [2008] employed a Support Vector Machine. Both approaches considered the co-occurrence as well as the proximity between query terms and candidate terms. In recent years, several studies mined search logs for query expansion. For example, Cui et al. [2002] showed that queries and documents may use different terms to refer to the same information. They built correlations between query terms and document terms using the click-through information. When the system receives a query q, all the document terms are ordered by their correlation to the terms in q, and the top terms are used for query expansion. Fonseca et al. [2005] proposed a method for concept-based interactive query expansion. For a query q, all the queries which are often adjacent with q in the same sessions are mined as the candidates for query expansion.

Query substitution alters the original query into a better form, for example, correcting spelling errors in the original query. In Guo et al. [2008], Jones et al. [2006], Lau and Horvitz [1999], Rieh and Xie [2001], and Silverstein et al. [1999], the authors studied the patterns how users refine queries in sessions and explored how to use those patterns for query substitution. In Rieh and Xie 2001, the authors categorized session patterns into three facets, that is, content, format, and resource. For each facet, they further defined several subfacets. For example, within the content facet, the subfacets include specification, generalization, replacement with synonyms, and parallel movement. In Jones et al. [2006], the authors extracted candidates for query substitution from session data and built a regression model to calculate the confidence score for each candidate. Guo et al. [2008] associated the query refinement patterns with particular operations. For example, for spelling correction, possible operations include deletion, insertion, substitution, and transposition. The authors then treated the task of query substitution as a structure prediction problem and trained a conditional random field model from session data. Although many previous studies for query expansion and query substitution are related to our work in the sense that they also use click-through bipartite and session data, those methods have the following two essential differences with our method. First, the methods for query expansion and query substitution aim at finding better formulation of the current query. In other words, they try to find queries replaceable to the current one. On the contrary, our query suggestion method may not necessarily provide candidate queries which carry the same meaning with the current one. Instead, it may provide suggestions that a user intends to ask in the next step of the search process. Second, almost all the existing methods for query expansion and query substitution only focus on the current query pairs, while our methods provide query suggestions depending on the context of the query. While conducting context-aware query expansion and substitution is an interesting research problem, we focus on on the problem of context-aware query suggestion in this article.

Another line of related work explored the effectiveness of using context information for predicting user interests. For example, White et al. [2010] assigned the topics from the taxonomy created by the Open Directory Project[1] to three types of contexts. The first type considered the preceding queries only, while the second and third types added clicked and browsed documents by the user. The authors confirmed that user interests are largely consistent within a session, and thus context information has good potential to predict the users short-term interests. In White et al. [2009], the authors explored various sources of contexts in browsing logs and evaluated their effectiveness for the prediction of user interests. For example, besides the preceding pages browsed within the current session, the authors also considered the pages browsed in a long history, the pages browsed by other users with the same interests, and so on. They found a combination of multiple contexts performed better than a single source. Mei et al. [2009] proposed using query sequences in sessions for four types of tasks, including sequence classification, sequence labeling, sequence prediction, and sequence similarity. They found that many tasks, such as segmenting queries in sessions according to use interests, can benefit from context information. Although those previous studies showed the effectiveness of context information, none of them targeted at the particular problem of query suggestion. Therefore, the techniques in those studies cannot be applied or directly extended for our problem.

3. MINING QUERY CONCEPTS

In this section, we summarize queries into concepts. We first describe how to form a click-through bipartite from search logs in Section 3.1. To address the sparseness of the click-through bipartite, we perform a random walk on the graph. We then present an efficient algorithm in Section 3.2, which clusters the bipartite with only one scan of the data. We further increase the scalability of the clustering algorithm by two approaches in Section 3.3. To improve the quality of clusters, we develop some postprocessing techniques in Section 3.4. Finally, in Section 3.5, we derive concepts from clusters and construct a URL feature vector and a term feature vector for each concept.

3.1. Click-Through Bipartite

To group similar queries into a concept, we need to measure the similarity between two queries. When a user raises a query to a search engine, a set of URLs will be returned as the answer. The URLs clicked by the user, called the *clicked URL set* of the query, can be used to approximate the information need described by the query. We can use the clicked URL set of a query as the features of that query. The information about queries and their clicked URL sets is available in search logs.

A search log can be regarded as a sequence of query and click events. Table I shows an example of a search log. In general, each row in a search log contains several fields to record a query or click event. From example, from Table I, we can read that an anonymous user 1 submitted query *“KDD 08”* at 11:08:43 on Dec. 5th, 2007, and then clicked on URL `www.kdd2008.com` after two seconds.

[1]http://www.dmoz.org/

Table I. A search Log as a Stream of Query and Click Events

User ID	Time Stamp	Event Type	Event Value
User 1	20071205110843	QUERY	KDD 08
User 2	20071205110843	CLICK	www.aaa.com
User 1	20071205110845	CLICK	www.kdd2008.com
...	...	...	...

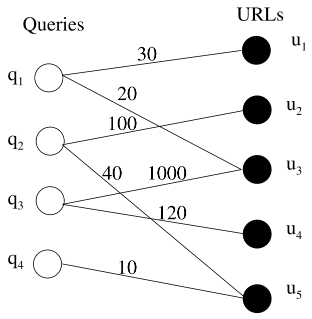


Fig. 2. An example of click-through bipartites.

From the raw search log, we can construct a *click-through bipartite* as follows. A *query node* is created for each unique query in the log. Similarly, a *URL node* is created for each unique URL in the log. An *edge* e_{ix} is created between query node q_i and URL node u_x if u_x is a clicked URL of q_i. The *click count* cc_{ix} associated with edge e_{ix} is the total number of times when u_x is a click of q_i aggregated over the whole log. Figure 2 shows an example of click-through bipartites.

The click-through bipartite can help us to find similar queries. The basic idea is that if two queries share many clicked URLs, they are similar to each other [Baeza-Yates et al. 2004; Beeferman and Berger 2000; Wen et al. 2001]. However, a click-through bipartite is typically sparse. In our raw experiment data, a query node is connected with an average of 1.57 URL nodes. Consequently, many similar queries do no share any clicked URLs. To address this challenge, several previous studies [Crasell and Szummer 2007; Gao et al. 2009] apply the random walk technique to densify the graph. In the following, we also adopt this technique as a preprocessing step before we derive the concepts.

To evaluate the similarity between queries, we first estimate the transition probabilities between queries and URLs. To be specific, let $p(u_x|q_i)$ be the probability of reaching URL u_x from query q_i among all the URLs connected to q_i. Reversely, $p(q_i|u_x)$ denotes the probability to reach q_i from u_x among all the queries connected to u_x. Let Q and U be the sets of query nodes and URL nodes of a click-through bipartite, the transition probabilities $p(u_x|q_i)$ and $p(q_i|u_x)$ are estimated by

$$p(u_x|q_i) = \frac{cc_{ix}}{\sum_{u_{x'} \in U} cc_{ix'}}, \tag{1}$$

$$p(q_i|u_x) = \frac{cc_{ix}}{\sum_{q_{i'} \in Q} cc_{i'x}}. \tag{2}$$

The transition probabilities form two matrices, that is, the query-to-URL transition matrix $P_{q2u} = [p(u_x|q_i)]_{ix}$ and the URL-to-query transition matrix $P_{u2q} = [p(q_i|u_x)]_{xi}$. The random walk on the click-through bipartite can be performed by

$$P^{(s)}_{q2u} = (P_{q2u}P_{u2q})^s P_{q2u}, \tag{3}$$

where s is the number of steps of the random walk. When the number of s increases, more query-URL pairs will be assigned nonzero transition probabilities. In other words, the click-through bipartite becomes denser. However, a too large s may introduce noise and connect irrelevant query-URL pairs. In Gao et al. [2009], the authors conducted an empirical study on different values of s and suggested a small value of 1 to achieve high relevance between the connected query-URL pairs after random walk. We made consistent observations in our empirical study; we found more steps of random walk may bring in many small-weight edges between unrelated query-URL pairs. Therefore, we adopt the empirical value in Gao et al. [2009] and set s to a small value of 1.

After the random walk, each query q_i is represented as an L_2-normalized vector, where each dimension corresponds to one URL in the bipartite. To be specific, the x-th element of the feature vector of a query $q_i \in Q$ is

$$\overrightarrow{q_i}^{URL}[x] = norm(w_{ix}) = \frac{w_{ix}}{\sqrt{\sum_{u_{x'} \in U} w_{ix'}^2}}, \tag{4}$$

where $norm(\cdot)$ is the L_2 normalization function, and the weight w_{ix} of edge e_{ix} is defined as the transition probability from q_i to u_x after the random walk. If $w_{ix} > 0$, an edge will be added between query q_i and URL u_x if it does not exist before the random walk.

The distance between two queries q_i and q_j is measured by the Euclidean distance between their normalized feature vectors. That is,

$$distance^{URL}(q_i, q_j) = \sqrt{\sum_{u_x \in U} (\overrightarrow{q_i}^{URL}[x] - \overrightarrow{q_j}^{URL}[x])^2}. \tag{5}$$

Please note that we have to handle two types of data sparseness. First, the click-through bipartite is usually sparse in the sense that each query node is connected with a small number of related URLs, and vice versa. To address this problem, we apply the random walk technique. Second, the user queries are also sparse since different users may refer to the same search intent using different queries. We handle this problem by summarizing queries into concepts in the following subsection.

3.2. Clustering Method

Now the problem is how to cluster queries effectively and efficiently in a click-through bipartite. There are several challenges. First, a click-through bipartite from a search log is often huge. For example, the raw log data in our experiments consist of more than 28.3 million unique queries. Therefore, the clustering algorithm has to be efficient and scalable to handle large datasets. Second, the number of clusters is unknown. The clustering algorithm should be able to automatically determine the number of clusters. Third, since each distinct URL is treated as a dimension in a query vector, the dataset is of extremely high dimensionality. For example, the dataset used in our experiments includes more than 40.9 million unique URLs. Therefore, the clustering algorithm has to tackle the “curse of dimensionality”. To the best of our knowledge, no existing methods can address all the preceding challenges simultaneously. We develop a new method called the *Query Stream Clustering* (QSC) algorithm (see Algorithm 1).

In the QSC algorithm, a cluster C is a set of queries. The *centroid* of cluster C is

$$\overrightarrow{C}^{URL}[x] = \frac{\sum_{q_i \in C} \overrightarrow{q_i}^{URL}[x]}{|C|}, \tag{6}$$

ALGORITHM 1: Query Stream Clustering (QSC).

```
Input: the set of queries $Q$ and the diameter threshold $D_{max}$;
Output: the set of clusters $\Theta$ ;
Initialization: dim_array[d] = $\emptyset$ for each dimension $d$;
 1: for each query $q_i \in Q$ do
 2:    $C$-Set = $\emptyset$;
 3:    for each non-zero dimension $d$ of $\overrightarrow{q_i}^{URL}$ do
 4:       $C$-Set $\cup$= dim_array[d];
 5:    end for
 6:    $C = \arg\min_{C' \in C\text{-Set}} distance(q_i, C')$;
 7:    if $diamter(C \cup \{q_i\}) \leq D_{max}$ then
 8:       $C \cup= \{q_i\}$; update the centroid and diameter of $C$;
 9:    end if
10:    else $C$ = new cluster($\{q_i\}$); $\Theta \cup= C$;
11:    for each non-zero dimension $d$ of $\overrightarrow{q_i}^{URL}$ do
12:       if $C \notin$ dim_array[d] then dim_array[d] $\cup = \{C\}$;
13:    end for
14: end for
15: return $\Theta$;
```

where $|C|$ is the number of queries in C. The distance between a query q and a cluster C is given by

$$distance^{URL}(q, C) = \sqrt{\sum_{u_x \in U} (\overrightarrow{q}^{URL}[x] - \overrightarrow{C}^{URL}[x])^2}. \tag{7}$$

We adopt the diameter measure in Zhang et al. [1996] to evaluate the compactness of a cluster, that is,

$$D = \left(\frac{\sum_{i=1}^{|C|} \sum_{j=1}^{|C|} (\overrightarrow{q_i}^{URL} - \overrightarrow{q_j}^{URL})^2}{|C|(|C|-1)}\right)^{\frac{1}{2}}. \tag{8}$$

To control the granularity of clusters, we set a diameter parameter D_{max}, that is, the diameter of every cluster should be smaller than or equal to D_{max}.

The QSC algorithm considers the set of queries as a *query stream* and scans the stream only once. The query clusters are derived during the scanning process. Intuitively, each cluster is initialized by a single query and then expanded gradually by similar queries. The expansion process stops when inserting more queries will make the diameter of the cluster exceed the threshold D_{max}. To be more specific, for each query q, we first find the closest cluster C to q among the clusters obtained so far, and then test the diameter of $C \cup \{q\}$. If the diameter is not larger than D_{max}, q is assigned to C and C is updated to $C \cup \{q\}$. Otherwise, a new cluster containing only q is created.

The potential major cost in our method is from finding the closest cluster for each query since the number of clusters can be very large. One may suggest to build a tree structure such as the CF-Tree in BIRCH [Zhang et al. 1996]. Unfortunately, as shown in previous studies (e.g., Hinneburg and Keim [1999]), the CF-Tree structure may not handle high dimensionality well: when the dimensionality increases, BIRCH tends to compress the whole dataset into a single data item.

How can we overcome the "curse of dimensionality" and find the closest cluster fast? We observe that the queries in the click-through bipartite are very sparse in dimensionality. For example, in our experimental data, a query is connected with an average number of 3.1 URLs after random walk, while the average degree of URL nodes is only

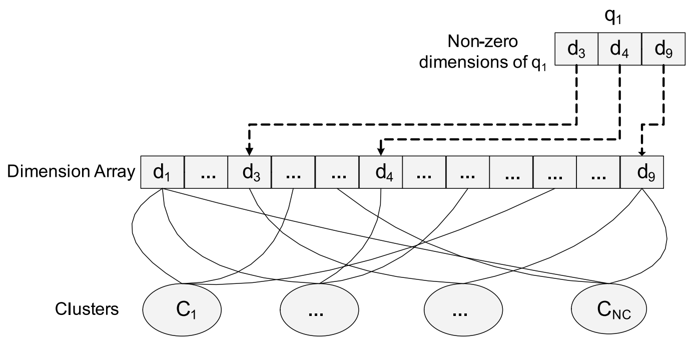


Fig. 3. The data structure for clustering.

3.7. Therefore, for a query q, the average size of Q_q, the set of queries which share at least one URL with q, is only $3.1 \cdot (3.7 - 1) = 8.37$. Intuitively, for any cluster C, if $C \cap Q_q = \emptyset$, C cannot be close to q since the distance of any member of C to q is $\sqrt{2}$, which is the farthest distance calculated according to Eq. (5) (please note the feature vectors of queries are normalized). In other words, to find out the closest cluster to q, we only need to check the clusters which contain at least one query in Q_q. Since each query belongs to only one cluster in the QSC algorithm, the average number of clusters to be checked is not larger than 8.37.

Based on the previous idea, we use a *dimension array* data structure (Figure 3) to facilitate the clustering procedure. Each entry of the array corresponds to one dimension d_x and links to a set of clusters Θ_x, where each cluster $C \in \Theta_x$ contains at least one member query q_i such that $\overrightarrow{q_i}^{URL}[x] \neq 0$. Given a query q, suppose the nonzero dimensions of $\overrightarrow{q}^{URL}$ are d_3, d_6, and d_9. To find the closest cluster to q, we only need to union the cluster sets Θ_3, Θ_6, and Θ_9, which are linked by the third, sixth, and ninth entries of the dimension array, respectively. Suppose Θ_3 contains cluster C_2, Θ_6 contains clusters C_5 and C_7, and Θ_9 contains cluster C_{10}. According to the preceding discussion, if q can be inserted into any existing cluster C_a, that is, the diameter of $C_a \cup \{q\}$ does not exceed D_{max}, then C_a must belong to the union of Θ_3, Θ_6, and Θ_9. Therefore, we only need to check whether q can be inserted into C_2, C_5, C_7, and C_{10}. Suppose q can be inserted C_2 and C_7, and the centroid of C_7 is closer to q than that of C_2, we will insert q to C_7. On the other hand, if q cannot be inserted into any cluster of C_2, C_5, C_7, and C_{10}, we will initialized a new cluster with q.

The QSC algorithm is very efficient since it scans the dataset only once. For each query q_i, the number of clusters to be accessed is at most $\sum_{d_x \in ND_i} |Q_x|$, where ND_i is the set of dimensions d_x such that $\overrightarrow{q_i}^{URL}[x] \neq 0$ and $|Q_x|$ is the number of queries q_j such that $\overrightarrow{q_j}^{URL}[x] \neq 0$. As explained before, since the queries are sparse on dimensions in practice, the average sizes of both ND_i and Q_x are small. Therefore, the practical cost for each query is constant, and the complexity of the whole algorithm is $O(N_q)$, where N_q is the number of queries.

Our method needs to store the dimension array and the set of clusters. Since the centroid and diameter of a cluster may be updated based on the feature vectors of the member queries during the clustering process, a naïve method would hold the feature vectors of the queries in clusters. In this case, the space complexity is $O(N_u \cdot N_q)$, where N_u and N_q are the numbers of URLs and queries, respectively.

To save space, we summarize a cluster C_a using a 3-tuple cluster feature [Zhang et al. 1996] $(N_{q_a}, \overrightarrow{LS_a}, SS_a)$, where N_{q_a} is the number of objects in the cluster,

$\overrightarrow{LS_a}$ is the linear sum of the N_{q_a} objects, and SS_a is the squared sum of the N_{q_a} objects. It is easy to show that the update of the centroid and diameter can be accomplished by referring only to the cluster feature. In this way, the total space is reduced from $O(N_u \cdot N_q)$ to $O(N_u + N_q)$, where $O(N_u)$ space is for dimension array and $O(N_q)$ space is for the cluster feature vectors.

One might wonder that since the click-through bipartite is sparse, is it possible to derive the clusters by finding the connected components from the bipartite? To be specific, two queries q_i and q_j are *connected* if there exists a query-URL path q_i-u_{x_1}-q_{i_1}-u_{x_2}-...-q_j where adjacent query and URL in the path are connected by an edge. A cluster of queries can be defined as a maximal set of connected queries. An advantage of this method is that it does not need a specified parameter D_{max}.

However, in our experiments, we find that although the bipartite is sparse, it is highly connected. In other words, a large percentage (up to 50%) of queries, no matter similar or not, are included within a single connected component. Moreover, the path between dissimilar queries cannot be broken by simply removing a few "hubs" of query or URL nodes (please refer to Figure 12). This suggests the cluster structure of the click-through bipartite is quite complex and we may have to use some parameters to control the granularity of desired clusters.

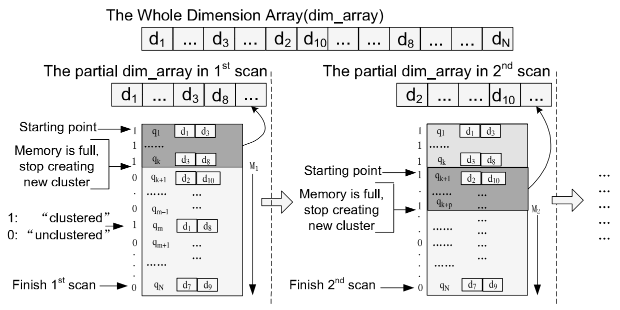
Although Algorithm 1 is very efficient, the time and space cost can still be very large. Can we prune the queries and URLs without degrading the quality of clusters? We observe that edges with low weights (either absolute or relative) are likely to be formed due to users' random clicks. Such edges should be removed to reduce noise. To be specific, let e_{ix} be the edge connecting query q_i and u_x, cc_{ix} be the click count of e_{ix}, and w_{ix} be the weight of e_{ix} after the random walk. We can prune an edge e_{ix} if $cc_{ix} \leq \tau_{abs}$ or $w_{ix} \leq \tau_{rel}$, where τ_{abs} and τ_{rel} are user-specified thresholds. After pruning low-weight edges, we can further remove the query and URL nodes whose degrees become zero. In our experiments, we empirically set $\tau_{abs} = 5$ and $\tau_{rel} = 0.05$.

3.3. Increasing the Scalability of QSC

With the support of the dimension array, the QSC algorithm only needs to scan the data once. However, the algorithm requires the dimension array to be held in main memory during the whole clustering process. In practice, a search log may contain tens of millions unique URLs even after the pruning process. Holding the complete dimension array in main memory becomes a bottleneck to scale up the algorithm. To address this challenge, we develop two approaches. The first approach scans the data iteratively on a single machine. During each scan, only a part of the dimension array needs to be held in the main memory. The second approach applies distributed computation under the master-slave programming model, where each slave machine holds a part of the dimension array in main memory.

3.3.1. Iterative Scanning Approach. In the QSC algorithm (Algorithm 1), each query is either inserted into an existing cluster if the diameter of the cluster does not exceed the threshold D_{max} after insertion, or assigned to a newly created cluster otherwise. The first case does not increase the memory consumption since we only need to update the value of the cluster feature of the existing cluster. However, the second case requires extra memory to record the cluster feature for the new cluster. Therefore, a critical point to control the memory usage of the QSC algorithm is to constrain the creation of new clusters.

Our idea is to adopt a divide-and-conquer approach and scan the query dataset in multiple runs (see Figure 4). During each scan, only a part of the dimension array is held in main memory. At the beginning, we scan the query dataset and process the queries in the same way as in Algorithm 1. When the memory consumption reaches



The Whole Dimension Array(dim_array)
The partial dim_array in 1st scan
The partial dim_array in 2nd scan
Starting point
Memory is full, stop creating new cluster
1: "clustered"
0: "unclustered"
Finish 1st scan
Finish 2nd scan

Fig. 4. The idea of the iterative scanning approach.

the total size of the available machine memory, we stop creating new clusters. For the remaining queries in the dataset, we only try to insert them into the existing clusters. If a query cannot be inserted into any existing cluster, it will not be processed but tagged as "unclustered." After all queries in the dataset have been scanned, the algorithm will output the clusters and release the memory for the current part of the dimension array.

Suppose the first scanning process stops creating new clusters at the M_1-th query, that is, the M_1-th query cannot be inserted into any existing cluster and the memory consumption has reached the limit. The second run will continue with that M_1-th query and only process those unclustered queries. Again the second run of the scanning process stops creating new clusters when the memory limit is reached and only allows insertion operation for the remaining queries. This process continues until all the unclustered queries are processed. This method is called *Query Stream Clustering with Iterative Scanning* (QSC-IS for short). The pseudocode is shown in in Algorithm 2.

ALGORITHM 2: Query Stream Clustering with Iterative Scanning (QSC-IS).

Input: the set of queries Q, the diameter threshold D_{max} and the memory limit L;
Output: the set of clusters Θ on disk;
Initialization: the position to start scanning M=0;

```
1:  while file end not reached do
2:     $\Theta$ =$\emptyset$; dim_array[d] = $\emptyset$ for each dimension $d$;
3:     seek to the $M$-th query;
4:     while (memory cost $< L$) && (file end not reached) do
5:        // process query as in Algorithm 1: until memory limit is reached or file ends
6:        read a query $q$ into memory; M++;
7:        if $q$.state == "unclustered" then
8:           perform steps 3-10 of Algorithm 1:; $q$.state = "clustered";
9:        end if
10:    end while
11:    while file end not reached do
12:       // continue scanning the file, but no new clusters are allowed to be created
13:       read a query $q$ into memory;
14:       if $q$.state == "unclustered" then
15:          perform steps 3-7,9,10 of Algorithm 1:; $q$.state = "clustered";
16:       end if
17:    end while
18:    Output $\Theta$ into disk;
19: end while
```

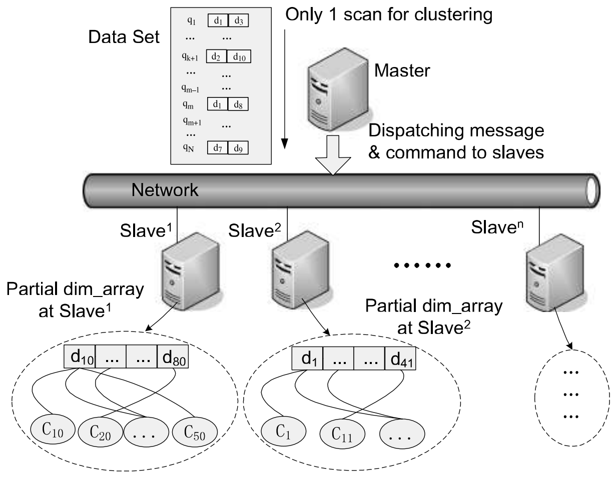

Data Set
Only 1 scan for clustering
Master
Dispatching message & command to slaves
Network
Slave1
Slave2
Slaven
Partial dim_array at Slave1
Partial dim_array at Slave2

Fig. 5. The idea of the master-slave approach.

The main time cost of the QSC-IS algorithm is on multiple scans of the query stream. The number of scans depends on the memory limit. The larger the machine memory, the fewer runs needed. Let N_q be the number of queries and $\mathcal{L}$ be the total number of scans. Suppose the ι-th $(1 \leq \iota \leq t-1)$ scanning process stops creating new clusters at the M_ι-th query, then the QSC-IS algorithm needs to scan a total number of $\mathcal{L}N_q - \sum_{\iota=1}^{\mathcal{L}-1}(M_\iota - 1)$ queries. In our experiment, only 11 scans are needed by a 2G memory machine for a dataset with 13.87 million queries (please refer to Section 5.2 for details). Therefore, the complexity of QSC-IS is still considered as $O(\mathcal{L} \cdot N_q)$ where $\mathcal{L} \ll N_q$.

The QSC-IS algorithm may generate different clustering results from those by the QSC algorithm (Algorithm 1). Suppose q can be inserted into both clusters C_a and C_b, which are created in the first and second scan of the data, respectively. If C_b is closer to q, the QSC algorithm will insert q into C_b, while the QSC-IS algorithm will insert q into C_a, since C_b is not allowed to be created in the first scan of the data due to memory limit. As we will discuss in Section 3.4, this problem can be addressed by some postprocessing techniques.

3.3.2. Master-Slave Approach. When we have multiple machines, we can hold the dimension array distributively under the master-slave programming model. As shown in Figure 5, each slave machine only holds a part of the dimension array and the master machine determines on which slave machines a query should be processed. This method is called *Query Stream Clustering in Master-Slave Model* (or QSC-MS for short). The pseudocode is presented in Algorithm 3.

Suppose there are $\mathcal{M}$ slave machines, we distribute the dimension array by the following rule: the x-th entry of the dimension array is allocated to the $(\omega + 1)$-th slave machine if $x \bmod \mathcal{M} = \omega$. Under such a rule, the master machine can easily determine where to find the dimension array entry for any URL u_x by a simple mode operation on $\mathcal{M}$.

In the clustering process, the master machine reads each query q from the input query stream and identifies the set of URLs U_q with nonzero weights for q. For each URL $u \in U_q$, the master determines the slave machine which stores the dimension array entry for u by the mode operation and dispatches (q, u) to that slave machine with command "TEST". The slave machine will look up the entry corresponding to u

ALGORITHM 3: Query Stream Clustering with Master-Slave Model (QSC-MS).

```
Input: the set of queries $Q$, the number of slave machines $M$;
Output: the set of clusters $\Theta$ ;
Master Side: Host Program
 1: for each query $q \in Q$ do
 2:   for each non-zero dimension $u$ of $\overrightarrow{q_i}^{URL}$ do
 3:     let $k = u \bmod M$;
 4:     send message $(q, u)$ with "TEST" to the $k$-th slave machine;
 5:   end for
 6:   receive a set of $\langle cid, dia \rangle$ tuples from the slave machines;
 7:   if the smallest diameter $dia_{min}$ in the received tuples is -1 then
 8:     let $k$= id of slave machine with maximal free memory;
 9:     send message $(q)$ with "CREATE" to the $k$-th slave machine;
10:     receive the cluster-id $cid$ from $k$-th slave machine;
11:     send message $(q, cid)$ with "UPDATE" to the other slave machines;
12:   else
13:     find the cluster-id $cid_{min}$ with the smallest diameter $dia_{min}$;
14:     let $k$ = id of slave machine which reports $dia_{min}$;
15:     send message $(q, cid_{min})$ with "INSERT" to $k$-th slave machine;
16:     send message $(q, cid_{min})$ with "UPDATE" to the other slave machines;
17:   end if
18: end for
19: send command "EXIT" to all slave machines.

Slave Side: Daemon Program
Input: the diameter threshold $D_{max}$;
Initialization: dim_array[d] = $\emptyset$;
 1: while true do
 2:   if command == "TEST" then
 3:     receive message $(q, u)$;
 4:     $C_u \leftarrow$ the set of clusters in the entry corresponding to $u$ in dim_array;
 5:     if $C_u \neq \emptyset$ then
 6:       get the minimum diameter $dia_{min}$ from cluster $cid_{min} \in C_u$;
 7:       if $dia_{min} < D_{max}$ then
 8:         send message $(cid_{min}, dia_{min})$ to the master machine;
 9:       else
10:         send message $(-1, -1)$ to the master machine;
11:       end if
12:     else
13:       send message $(-1, -1)$ to the master machine;
14:     end if
15:   end if
16:   if command == "INSERT" then
17:     received message $(q, cid)$;
18:     add query $q$ into cluster $cid$;
19:     update the cluster feature of cluster $cid$;
20:   end if
21:   if command == "CREATE" then
22:     received message $(q)$;
23:     $cid \leftarrow cid + M$;
24:     initialize new cluster $cid$ with query $q$ and send $cid$ to master machine;
25:   end if
26:   if commend == "UPDATE" then
27:     received message $(q, cid)$;
28:     for each nonzero dimension $d$ of $\overrightarrow{q}$ do
29:       if $d$ falls in this slave machine's dim_array then
30:         link $cid$ to the corresponding entry;
31:       end if
32:     end for
33:   end if
34:   if command == "EXIT" then output clusters and exit the loop;
35: end while
```

in its local dimension array, and retrieves a list of clusters $\mathcal{C}_u$ linked to the entry. For each cluster $C \in \mathcal{C}_u$, the slave machine will test whether the insertion of q will make the diameter of C exceed the threshold D_{max}. If q can be inserted into at least one cluster

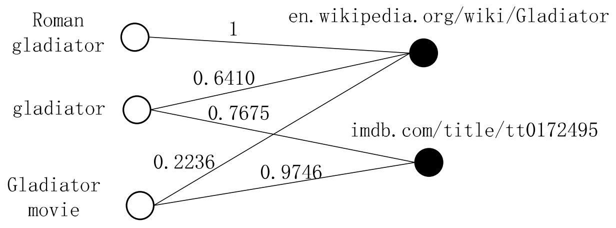


Fig. 6. The click-through bipartite for Example 3.1.

$C \in \mathcal{C}_u$, the slave machine will pick up the closest cluster to q and send a response (cid, dia) to the master machine, where *cid* is the ID of the closest cluster and *dia* is the corresponding diameter of the cluster if q is inserted. Otherwise, the response will be $(-1, -1)$.

After the master machine receives all the replies from the slave machines, it will handle two cases. In one case, all replies are $(-1, -1)$. This means the current query q cannot be added into any existing cluster. For this case, the master machine identifies the slave machine with the maximal free memory and initializes a new cluster C with query q on that slave machine by a message (q) with command "CREATE".

In the other case, the current query q can be inserted into the closest existing cluster C with id *cid*. Suppose the cluster C locates on the k-th slave machine, the master machine will dispatch a message (q, cid) with command "INSERT" to the target slave machine. The slave machine will update the cluster feature by incorporating the query q into the cluster. Finally, the master machine will find out the nonzero URLs U_q and send a message (u, cid) with command "UPDATE" for each $u \in U_q$ to the corresponding slave machine. Each recipient slave machine will check the dimension array entry of u and link *cid* to it if the link does not exist.

The major time expense of the QSC-MS algorithm is on network communication. Since network speed is usually far slower than disk scanning speed, the QSC-MS algorithm with ten slave machines is still slower than the QSC-IS algorithm on a single machine when they are compared on an experimental data with 13, 872, 005 queries (please refer to Section 5.2 and Figure 13). However, the QSC-MS algorithm requires much less memory for an individual machine than the QSC-IS algorithm (please refer to Section 5.2 and Figure 13).

3.4. Postprocessing of the Clusters

With the techniques presented in Section 3.3, the clustering algorithm can scale up to very large amounts of data. In this section, we target at improving the quality of the clusters. First, the QSC algorithm, as well as its extensions QSC-IS and QSC-MS, are order sensitive in that the clustering results may change with respect to the order of the queries in the input stream. Such order sensitivity may result in low-quality clusters. Second, the QSC family algorithms generate a hard clustering, which means a query can only belong to one cluster. This requirement is not reasonable for multi-intent queries such as *"gladiator"* in Example 1.1. To better understand the these aforesaid two problems, let us consider an example.

Example 3.1 (*Order Sensitivity*). Figure 6 shows a piece of click-through bipartite from a search log. In the search log, users who raised query *"Roman gladiators"* uniformly clicked on `en.wikipedia.org/wiki/Gladiator`. Consequently, the query *"Roman gladiators"* has a L_2 normalized weight 1.0 on URL `en.wikipedia.org/wiki/Gladiator`. Moreover, users who raised query *"Gladiator movie"* mainly clicked on URL `www.imdb.com/title/tt0172495`, which is the IMDB site of the movie. At

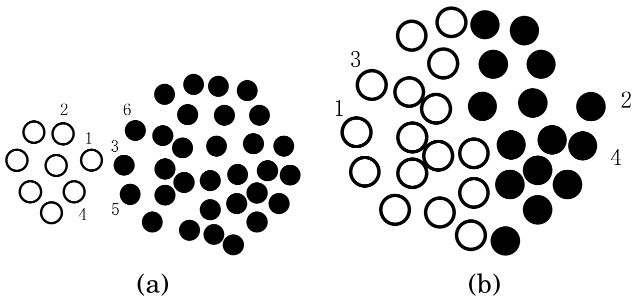


Fig. 7. Two problems caused by the order sensitivity problem.

the same time, a few users might further explore some information about the gladiators in the history, and they clicked on URL `en.wikipedia.org/wiki/Gladiator`. As a result, the query *"Gladiator movie"* has a normalized weight 0.9746 on URL `www.imdb.com/title/tt0172495` as well as a normalized weight 0.2236 on `en.wikipedia.org/wiki/Gladiator`. The third query *"gladiator"* bears mixed intents. On the one hand, some users raised this query to find the information of Roman gladiators, which resulted in a weight 0.6410 on `en.wikipedia.org/wiki/Gladiator`. On the other hand, some users intended to find the film *Gladiator*, which caused a weight 0.7675 on the URL `www.imdb.com/title/tt0172495`.

Now suppose we set the diameter threshold $D_{max} = 1$. If the order of the three queries in the input data is *"Roman gladiators"* $\succ$ *"Gladiator movie"* $\succ$ *"gladiator"*, where "$q_1 \succ q_2$" indicates query q_1 appears before q_2 in the input stream, then the clustering results will be C_1 = {*"Roman gladiators"*} and C_2 = {*"gladiator", "gladiator movie"*}. However, if the order becomes *"gladiator"* $\succ$ *"Roman gladiators"* $\succ$ *"Gladiator movie"*, all the three queries will be assigned to the same cluster.

Ideally, *"Roman gladiators"* and *"Gladiator movie"* should be assigned to separate clusters while *"gladiator"* is assigned to both of them. However, the clustering results of the QSC family algorithms are dependent on the order of queries. In some orders, such as *"gladiator"* $\succ$ *"Roman gladiators"* $\succ$ *"Gladiator movie"*, all the queries are grouped in the same cluster. Even in the orders when *"Roman gladiators"* and *"Gladiator movie"* are assigned to separate clusters, since the algorithms generate a hard clustering, *"gladiator"* can only belong to one of them.

In general, the order sensitivity problem may cause two situations as shown in Figures 7(a) and 7(b). In Figure 7(a), the points on the left side (represented by unfilled circles) of point 1 belong to one cluster, and those on the right side (represented by filled circles) of point 1 belong to a second cluster. However, if the points are processed in the interleaved order between the two clusters, such as the order labeled by 1, 2, 3, 4, 5, and so on in Figure 7(a), all the points are grouped in one cluster (recall the example of *"gladiator"* $\succ$ *"Roman gladiators"* $\succ$ *"Gladiator movie"*). In Figure 7(b), all the points are closely related and should be assigned to the same cluster. However, if the point labeled as 1 in the figure comes in first, followed by the point labeled as 2, the QSC algorithm may put them in two clusters. Consequently, the whole set of points in Figure 7(b) may be split into two clusters (marked as filled and unfilled circles in the figure).

To tackle the preceding problems, we apply a split-merge process to the clusters derived by the QSC family algorithms. Our purpose is that no matter in which order the points are processed, the split process will divide the points in Figure 7(a) into two clusters, while the merge process will group all the points in Figure 7(b) into a single cluster.

In the split-merge process, we apply the *Cluster Affinity Search Technique* Algorithm (CAST) [Ben-Dor et al. 1999] to each cluster derived by the QSC family algorithms. In other words, we first apply the QSC family algorithm to the full query dataset and derive the query clusters. Then, we apply the CAST algorithm to each derived cluster as a postprocessing step.

The basic idea of the CAST algorithm is to start with an empty cluster and gradually grow the cluster each time with the object (i.e., query) that has the largest average similarity to the current cluster. The growing process stops until the largest average similarity to the current cluster is below a threshold σ_{min}. At this point, the algorithm will pick up the point in the current cluster which has the lowest average similarity σ with the current cluster and remove this point if $\sigma < \sigma_{min}$. The algorithm iterates between the growing and removing processes until no object can be added or removed. Then, the algorithm will output the current cluster and start to grow the next one. This process continues until all the objects have been processed.

The CAST algorithm, on the one hand, is quite similar to the QSC family algorithms because both approaches use the average similarity or distance to control the granularity of clusters. One can easily verify that when the weights of queries $\overrightarrow{q}^{URL}$ is L_2 normalized, the thresholds adopted by the two approaches have the relationship $D_{max} = \sqrt{2(1 - \sigma_{min})}$.

On the other hand, the CAST algorithm has several critical differences with the QSC family algorithms. First, compared with the QSC family algorithms where the objects are processed in the fixed order by the input stream, the CAST algorithm determines the sequence of objects to added into a cluster based on their similarity to the cluster. Second, the CAST algorithm may adjust the current cluster by moving out some objects, while the QSC family algorithms have no chance to correct their previous clustering decisions. Due to the preceding two differences, the CAST algorithm may improve the quality of the clusters derived by the QSC family algorithms. Please note the quality improvement is obtained at the cost of higher computation complexity. As described in Ben-Dor et al. [1999], the computation complexity of the CAST algorithm is $O(N^2)$, while the complexity of the QSC family algorithms is only $O(N)$, where N is the number of objects to be clustered. Clearly, it is impractical to apply the CAST algorithm to the whole query stream. However, in the postprocessing stage, we only apply the CAST algorithm to the clusters derived by the QSC family algorithms. In practice, the number of queries of the same concept is usually small. For example, in our experiment data, the largest clusters derived from the QSC family algorithms contain 6,684 queries. Therefore, we can load each cluster derived by the QSC family algorithms into the main memory and apply the CAST algorithm efficiently.

After the split process, we merge two clusters if the diameter of the merged cluster does not exceed D_{max}. In fact, the merge process can be considered as a second-order clustering process, in which the query clusters instead of the individual queries are clustered. Naturally, we can reuse the dimension array structure and perform the merge process at $O(N_c)$ time, where N_c is the number of clusters.

To allow a multi-intent query such as *"gladiator"* belong to two clusters, we apply a reassignment process. To be specific, we check for each query q in cluster C_a whether it can be inserted to another cluster C_b without making the diameter of C_b exceed D_{max}. If so, we call q is *reassignable* to C_b. Again, we maintain the dimension array structure and only check those clusters having common nonzero dimensions with q. Let Q_b be the set of reassignable queries to cluster C_b. We first sort the queries in Q_b in the ascending order of their similarity to the centroid of C_b and then insert the queries into C_b one by one. The insertion process stops when: (a) all the queries in Q_b

have been inserted; or (b) the insertion of a query q in the sorted list makes the diameter of C_b exceed D_{max}. In the latter case, the last query q will be removed from C_b. In our empirical study, among all the clusters with at least one reassignable queries, 60% have exactly one reassignable query. According to the definition of reassignable queries, those 60% cases trivially fall into the preceding category (a). In more general cases, 98% of the clusters with at least one reassignable queries fall into category (a). Therefore, our reassignment process is robust to the order of queries. To be more specific, our reassignment process is independent of the order of queries in the input data stream, since we specify a fixed order (similarity to the centroid of clusters) for them. We may choose other orders to insert the reassignable queries. However, different orders will only affect 2% of the clusters with at least one reassignable queries.

Let us review Example 3.1 of queries *"Roman gladiators", "gladiator"*, and *"Gladiator movie"*. We can verify no matter in which order these three queries appear, the split and merge process will put *"Roman gladiators"* and *"Gladiator movie"* separately in two clusters. After the reassignment process, we get two clusters: {*"Roman gladiators", "gladiator"*} and {*"Gladiator movie", "gladiator"*}.

3.5. Building Concept Features

Now we have derived a set of high-quality clusters from the click-through bipartite, where each cluster represents a concept. In the online stage of query suggestion, users may raise new queries which are not covered by the log data. To handle such new queries, we create two feature vectors for each cluster. Let C be a cluster, and c be the corresponding concept. The *URL feature vector* for c, denoted by $\overrightarrow{c}^{URL}$, is simply the centroid of C defined in the URL space (Eq. (6)) , that is,

$$\overrightarrow{c}^{URL}[x] = \overrightarrow{C}^{URL}[x] = \frac{\sum_{q_i \in C} \overrightarrow{q_i}^{URL}[x]}{|C|}, \tag{9}$$

where $|C|$ is the number of queries in C. Analogously, we create a *term feature vector* for c based on the terms of the queries in C. To be specific, for each query $q_i \in C$, we can represent q_i by its terms with the following formula

$$\overrightarrow{q_i}^{term}[t] = norm(tf(t, q_i) \cdot icf(t)), \tag{10}$$

where $norm(\cdot)$ is the L_2 normalization function, $tf(t, q_i)$ is the frequency of term t in q_i, $icf(t) = \log \frac{N_c}{N_c(t)}$ is the *inverse cluster frequency* of t, N_c is number of clusters and $N_c(t)$ is number of clusters which contain queries with t. Then the term feature vector for concept c is defined by

$$\overrightarrow{c}^{term}[t] = \frac{\sum_{q_i \in C} \overrightarrow{q_i}^{term}[t]}{|C|}. \tag{11}$$

We will describe how to use these two features vectors to handle new queries at the online stage in Section 4.3.

4. GENERATING QUERY SUGGESTIONS

In this section, we first introduce how to derive session data from search logs. We then develop a novel structure, *concept sequence suffix tree*, to summarize the patterns mined from session data. Finally, we present the query suggestion method based on the mined patterns.

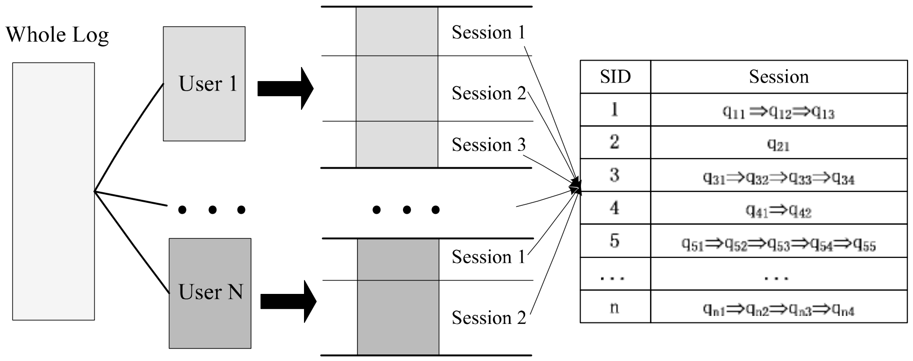

Whole Log
User 1
User N
Session 1
Session 2
Session 3
Session 1
Session 2
SID
Session
1 $q_{11} \Rightarrow q_{12} \Rightarrow q_{13}$
2 q_{21}
3 $q_{31} \Rightarrow q_{32} \Rightarrow q_{33} \Rightarrow q_{34}$
4 $q_{41} \Rightarrow q_{42}$
5 $q_{51} \Rightarrow q_{52} \Rightarrow q_{53} \Rightarrow q_{54} \Rightarrow q_{55}$
... ...
n $q_{n1} \Rightarrow q_{n2} \Rightarrow q_{n3} \Rightarrow q_{n4}$

Fig. 8. The procedure of building search and query session data.

Table II. Examples of Query Sessions and Relationship between Queries in Sessions

Query Relation	Query Session
Spelling correction	"*MSN messnger*" ⇒ "*MSN messenger*"
Peer queries	"*SMTP*" ⇒ "*POP3*"
Acronym	"*BAMC*" ⇒ "*Brooke Army Medical Center*"
Generalization	"*Washington mutual home loans*" ⇒ "*home loans*
Specialization	"*Nokia N73*" ⇒ "*Nokia N73 themes*" ⇒ "*free themes Nokia N73*"

4.1. Session Extraction

As explained in Section 1, the context of a user query consists of the immediately preceding queries issued by the same user. To learn a context-aware query suggestion model, we need to collect query contexts from the user query session data.

We construct session data as follows. First, we extract each user's behavior data from the whole search log as a separate stream of query/click events. Second, we segment each user's stream into *search sessions* based on a widely-used rule [White et al. 2007]: two queries are split into two sessions if the time interval between them exceeds 30 minutes. To obtain training data for query suggestion, we further derive *query sessions* by discarding the click events and only keeping the sequence of queries in each session. The process of building query session data is shown in Figure 8.

Table II shows some real query sessions as well as the relationship between the queries in the sessions. We can see that a user may refine the queries or explore related information about his or her search intent in sessions. As an illustrating example, from the last session in Table II, we can derive three training examples, that is, "*Nokia N73 themes*" is a candidate suggestion for "*Nokia N73*", and "*free themes Nokia N73*" is a candidate suggestion for both single query "*Nokia N73 themes*" and query sequence "*Nokia N73*" ⇒ "*Nokia N73 themes*".

4.2. Concept Sequence Suffix Tree

Queries in the same session are often related. However, since users may formulate different queries to describe the same search intent, mining patterns of individual queries may miss interesting patterns. To address this problem, we map each query session $qs = q_1 q_2 \cdots q_l$ in the training data into a sequence of concepts $cs = c_1 c_2 \cdots c_{l'}$, where a concept c_a $(1 \le a \le l')$ is represented by a cluster C_a derived in Section 3 and a query q_i is mapped to c_a if $q_i \in C_a$. If two consecutive queries belong to the same concept, we record the concept only once in the sequence.

A special case in the mapping process is that some queries may be assigned to multiple concepts. Enumerating all possible concept sequences for those queries may cause some false positive patterns.

Example 4.1 (*Multiconcept Queries*). Query *"jaguar"* belongs to two concepts. The first concept c_1 consists of queries *"jaguar animal"* and *"jaguar"*, while the second concept c_2 consists of queries *"jaguar car"* and *"jaguar"*. Suppose we observe a query session qs_1 *"jaguar"* $\Rightarrow$ *"Audi"* in the training data. Moreover, suppose query *"Audi"* belongs to concept c_3. We may generate two concepts sequences: $cs_1 = c_1c_3$ and $cs_2 = c_2c_3$. If we adopt both sequences, for query *"jaguar animal"*, which belongs to concept c_1, we may apply sequence cs_1 and find concept c_3 following c_1. Consequently, we may generate an irrelevant suggestion *"Audi"* to *"jaguar animal"*.

To avoid false concept sequence such as cs_1, if a query session qs contains a multiconcept query q_i, we may leverage the click information of q_i to identify the concept it belongs to in the particular session qs. In the previous example, we find *"jaguar"* in session qs_1 is a multiconcept query. Then we will refer to the search session corresponding to the query session qs_1 *"jaguar"* $\Rightarrow$ *"Audi"*. If we find in the search session that the user clicks on URLs `www.jaguar.com` and `www.jaguarusa.com` for *"jaguar"*, we can build a URL feature vector $\overrightarrow{q}_i^{URL}$ for *"jaguar"* based on Eq. (4). Then we compare $\overrightarrow{q}_i^{URL}$ with the URL feature vectors of concepts c_1 and c_2, respectively. Since c_1 refers to the jaguar animal, while c_2 refers to jaguar car, the URL feature vector $\overrightarrow{q}_i^{URL}$ for *"jaguar"* in session qs_1 *"jaguar"* $\Rightarrow$ *"Audi"* must be closer to that of c_2. In this way, we can tell the query *"jaguar"* in qs_1 belongs to concept c_2 instead of c_1. Consequently, we only generate the concept sequence cs_2 for session qs_1. For sessions where no click information is available for multiconcept queries, we simply discard them to avoid generating false concept sequences. In our experiments, we discarded about 20% sessions among those with multiconcept queries.

In the following, we mine patterns from concept sequences. First, we find all frequent sequences from session data. Second, for each frequent sequence $cs = c_1 \ldots c_l$, we use c_l as a candidate concept for $cs' = c_1 \ldots c_{l-1}$. We then build a ranked list of candidate concepts c for cs' based on their occurrences following cs' in the same sessions; the more occurrences of c, the higher c is ranked. For each candidate concept c, we choose the member query which receives the largest number of clicks in the log data as the representative of c. In practice, for each sequence cs', we only keep the representative queries of the top K (e.g., $K = 5$) candidate concepts. These representative queries are called the *candidate suggestions* for sequence cs' and will be used for query suggestion when cs' is observed online.

The major cost in the preceding method is from computing the frequent sequences. Traditional sequential pattern mining algorithms such as GSP [Srikant and Agrawal 1996] and PrefixSpan [Pei et al. 2001] can be very expensive, since the number of concepts (items) and the number of sessions (sequences) are both very large. We tackle this challenge with a new strategy based on the following observations. First, since the concepts co-occurring in the same sessions are often correlated in semantics, the actual number of concept sequences in session data is far less than the number of possible combinations of concepts. Second, given the concept sequence $cs = c_1 \ldots c_l$ of a session, since we are interested in extracting the patterns for query suggestions, we only need to consider the subsequences with lengths from 2 to l. To be specific, a *subsequence* of the concept sequence cs is a sequence $c_{m+1}, \ldots, c_{m+l'}$, where $m \geq 0$ and $m + l' \leq l$. Therefore, the number of subsequences to be considered for cs is only

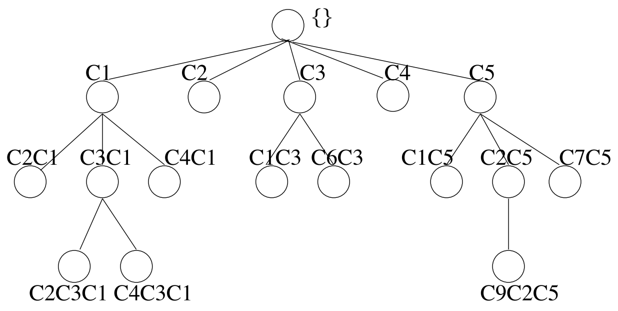


Fig. 9. A concept sequence suffix tree.

$\frac{l \cdot (l-1)}{2}$. Finally, the average number of concepts in a session is usually small. Based on these observations, we do not enumerate the combinations of concepts; instead, we enumerate the subsequences of sessions.

Technically, we implement the mining of frequent concept sequences with a distributed system under the map-reduce programming model [Dean and Ghemawat 2004]. In the map operation, each machine (called a *process node*) receives a subset of sessions as input. For the concept sequence cs of each session, the process node outputs a key-value pair $(cs', 1)$ to a bucket for each subsequence cs' with a length greater than 1. In the reduce operation, the process nodes aggregate the counts for cs' from all buckets and output a key-value pair $(cs', freq)$ where $freq$ is the frequency of cs'. A concept sequence cs' is pruned if its frequency is smaller than a threshold.

Once we get the frequent concept sequences, we organize them with a *concept sequence suffix tree* structure (see Figure 9). To be formal, a *suffix* of a concept sequence $cs = c_1 \dots c_l$ is an empty sequence or a sequence $cs' = c_{l-m+1} \dots c_l$, where $m \leq l$. In particular, cs' is a *proper suffix* of cs if cs' is a suffix of cs and $cs' \neq cs$. On the concept sequence suffix tree, each node corresponds to a frequent concept sequence cs. Given two nodes cs_1 and cs_2, cs_1 is the parent node of cs_2 if cs_1 is the longest proper suffix of cs_2. Except the root node, which corresponds to the empty sequence, each node on the tree is associated with a list of candidate query suggestions and URL recommendations.

Algorithm 4 describes the process of building a concept sequence suffix tree. Basically, the algorithm starts from the root node and scans the set of frequent concept sequences once. For each frequent sequence $cs = c_1 \dots c_l$, the algorithm first finds the node cn corresponding to $cs' = c_1 \dots c_{l-1}$. If cn does not exist, the algorithm creates a new node for cs' recursively. Finally, the algorithm updates the list of candidate concepts of if c_l is among the top K candidates observed so far. Figure 10 shows a running example to illustrate the process of building a concept suffix tree.

In Algorithm 4, the major cost for each sequence is from the recursive function *findNode*, which looks up the node cn corresponding to $c_1 \dots c_{l-1}$. Clearly, the recursion executes for $l-1$ levels, and at each level, the potential costly operation is the access of the child node cn from the parent node pn (the last statement in line 2 of Method *findNode*). We use a heap structure to support the dynamic insertion and access of the child nodes. In practice, only the root node has a large number of children, which is upper bounded by the number of concepts N_C; while the number of children of other nodes is usually small. Therefore, the recursion takes $O(\log N_C)$ time and the whole algorithm takes $O(N_{cs} \cdot \log N_C)$ time, where N_{cs} is the number of frequent concept sequences.

ALGORITHM 4: Building the concept sequence suffix tree.

```
Input: the set of frequent concept sequences $CS$ and the number $K$ of candidate suggestions;
Output: the suffix concept tree $T$ ;
Initialization: $T$.root=∅;
1: for each frequent concept sequence $cs = c_1 \dots c_l$ do
2:     $cn$ = findNode($c_1 \dots c_{l-1}$, $T$);
3:     $minc$ = argmin$_{c \in cn.candlist} c.freq$;
4:     if ($cs$.freq $>$ $minc$.freq) or ($|cn$.candlist$| < K$) then
5:         add $c_l$ into $cn$.candlist; $c_l$.freq= $cs$.freq;
6:         if $|cn$.candlist$| > K$ then remove $minc$ from $cn$.candlist;
7:     end if
8: end for
9: return $T$;

Method: findNode($cs = c_1 \dots c_l$, $T$);
1: if $|cs| = 0$ then return $T$.root;
2: $cs' = c_2 \dots c_l$; $pn$ = findNode($cs'$, $T$); $cn$ = $pn$.childlist[$c_1$];
3: if $cn$ == null then
4:     $cn$ = new node ($cs$); $cn$.candlist=∅; $pn$.childlist[$c_1$]= $cn$;
5: end if
6: return $cn$;
```

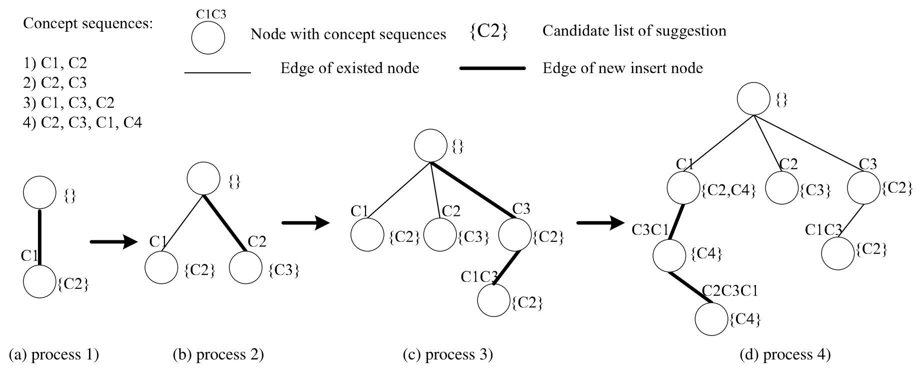


Fig. 10. An example of constructing a concept sequence suffix tree.

4.3. Suggestion Generation

The previous sections focus on the offline part of the system which learns a suggestion model from search logs. In this subsection, we discuss the online part of the system which generates query suggestions based on the learned model.

When the system receives a sequence of user input queries $q_1 \cdots q_l$, similar to the procedure of building training examples, the query sequence is also mapped into a concept sequence. Again, we need to handle the cases when a query belongs to multiple concepts. However, unlike the offline procedure of building training examples, when we provide online query suggestions, we cannot discard any sessions. Moreover, we have to handle new queries which do not appear in the training data. In the following, we will address these two challenges.

As described in Section 4.2, if a query q_i in an input query sequence qs belongs to multiple concepts, we may leverage the click information of q_i to identify the concept to which it should be mapped for the current sequence qs. However, if the user does not make any clicks for q_i, this method will not work. In the procedure of building training examples, such sessions are simply discarded. However, at online stage, we cannot discard user inputs. To handle this problem, we can check whether the user inputs any query before or after the multiconcept query q_i in the current input sequence. Then we may tell q_i's concept through the adjacent queries.

For example, suppose at online stage, the system receives a query sequence qs_1 "*jaguar*" ⇒ "*Audi*". As in Example 4.1, suppose query *"jaguar"* belongs to two concepts: the first concept c_1 consists of queries *"jaguar animal"* and *"jaguar"*, while the second concept c_2 consists of queries *"jaguar car"* and *"jaguar"*. Moreover, suppose query "*Audi*" belongs to concept c_3. We may generate two concepts sequences: $cs_1 = c_1c_3$ and $cs_2 = c_2c_3$. However, from our method of building training examples in Section 4.2, the false sequence cs_1 can be effectively avoided. Consequently, at the online stage, the only choice to map qs_1 is cs_2. In other words, we can make the correct mapping for multiconcept queries at online stage by matching the adjacent queries with the patterns mined from the training data.

In the last case, if the multiconcept query q_i is the only query in the current input sequence, we can map q_i to all the concepts it belongs to and generate query suggestions accordingly. For example, if the user inputs a single query "*jaguar*", since we have no context available, it is reasonable to suggest queries such as "*cheetah*" and "*Audi*" at the same time.

To handle new queries, our idea is to assign them to existing concepts by the URL and term feature vectors. To be specific, if the user clicks on some URLs for a new query q_i in the online session, we can build a URL feature vector $\overrightarrow{q}_i^{URL}$ for q_i based on the clicked URLs by Eq. (4). Otherwise, we may use the top ten search results returned by the search engine to create $\overrightarrow{q}_i^{URL}$. Besides the URL feature vector, we can merge the snippets of the top ten search results of q_i and create a term feature vector $\overrightarrow{q}_i^{term}$ for q_i by Eq. (10). Then we can calculate the distance between q_i and a concept c by

$$distance(q_i, c) = \min(||\overrightarrow{q}_i^{URL} - \overrightarrow{c}^{URL}||, ||\overrightarrow{q}_i^{term} - \overrightarrow{c}^{term}||), \tag{12}$$

where $||\cdot||$ is the L_2 norm, and $\overrightarrow{c}^{URL}$ and $\overrightarrow{c}^{term}$ are defined by Eqs. (9) and (11), respectively. To facilitate the online computation of Eq. (12), we also create a dimension array for terms, which is similar to the one for URLs as shown in Figure 3. For each new query q_i, we only need to calculate the distance between q_i and the clusters which have at least one overlapping nonzero weight URL or term with q_i. Finally, we pick up the concept c which is the closest to q_i and map q to c if the diameter of c does not exceed D_{max} after inserting q_i into c.

After the mapping procedure, we start from the last concept in the sequence and search the concept sequence suffix tree from the root node. The process is shown in Algorithm 5. Basically, we maintain two pointers: *curC* is the current concept in the sequence and *curN* is the current node on the suffix tree. We check whether the current node *curN* has a child node *chN* whose first concept is exactly *curC*. If so, we move to the previous concept (if exists) of *curC* and visit the child node *chN* of *curN*. If no previous concept exists, or no child node *chN* of *curN* matches *curC*, the search process stops, and the candidate suggestions of the current node *curN* are used for query suggestion.

The mapping of a query sequence *qs* into a concept sequence *cs* (line 1) takes $O(|qs|)$ time. The aim of the while loop (lines 3–8) is to find the node which matches the suffix of *cs* as much as possible. As explained in Section 4.2, the cost of this operation is $O(\log N_C)$. In fact, when generating suggestions online, we do not need to maintain the dynamic heap structure as during the building process of the tree. Instead, we can serialize the children of the root node into a static array structure. In this case,

ALGORITHM 5: Query suggestion.

```
Input: the concept sequence suffix tree $T$ and user input query sequence $qs$;
Output: the ranked list of query suggestions $S\text{-}Set$
Initialization: $curN = T$.root; $S\text{-}Set = \emptyset$;
1: map $qs$ into $cs$;
2: $curC$ = the last concept in $cs$;
3: while true do
4:     $chN$ = $curN$'s child node whose first concept is $curC$;
5:     if ($chN$ == null) then break;
6:     $curN = chN$;
7:     $curC$ = the previous concept of $curC$ in $cs$;
8:     if ($curC$ == null) then break;
9: end while
10: if $curN$ != $T$.root then
11:     $S\text{-}Set$ = $curN$'s candidate query suggestions;
12: end if
13: return $S\text{-}Set$;
```

Table III. The Size of the Click-Through Bipartite Before and After Pruning

	Original Graph	Pruned Graph
# Query Nodes	28, 354, 317	13, 872, 005
# URL Nodes	40, 909, 657	11, 399, 944
# Edges	44, 540, 794	27, 711, 168
# Query Occurrences	3, 957, 125, 520	2, 977, 054, 437
# Clicks	5, 918, 834, 722	4, 682, 875, 167

the search cost can be reduced to $O(1)$. To sum up, the time for our query suggestion process is $O(|qs|)$, which meets the requirement of online process well.

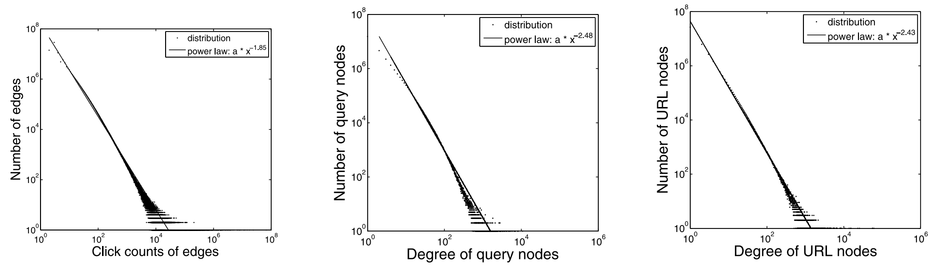
5. EXPERIMENTS

We extract a large-scale search log from a commercial search engine as the training data for query suggestion. To facilitate the interpretation of the experimental results, we only focus on the Web searches in English from the US market. The dataset contains 3,957,125,520 search queries, 5,918,834,722 clicks, and 1,872,279,317 search sessions, which involves 28,354,317 unique queries and 40,909,657 unique URLs.

5.1. Characteristics of the Click-Through Bipartite

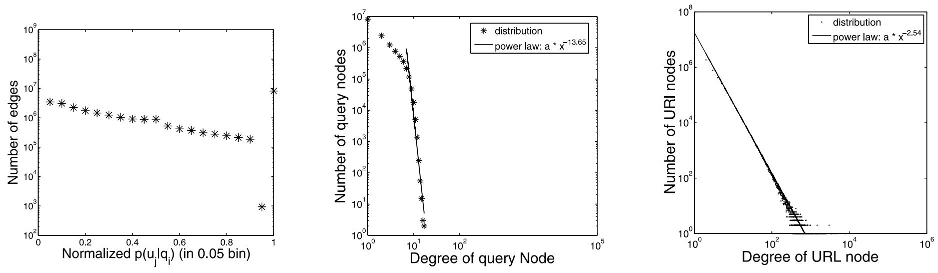
We build a click-through bipartite to derive concepts. As described in Section 3.2, we set $\tau_{abs} = 5$ and $\tau_{rel} = 0.05$ to prune low-weight edges. Table III shows the sizes of the click-through bipartite before and after the pruning process. Please note the pruned graph is more than seven times larger than the one in our previous study [Cao et al. 2008] in terms of the number of query nodes. Such a large set helps us better evaluate the scalability of the clustering algorithms.

It has been shown in previous work (e.g., Baeza-Yates and Tiberim [2007]) that the occurrences of queries and the clicks of URLs exhibit power-law distributions. However, the properties of the click-through bipartite have not been well explored. In this experiment, we first investigate the distributions of: (1) the click counts of edges, (2) the degrees of query nodes, and (3) the degrees of URL nodes. We then explore the



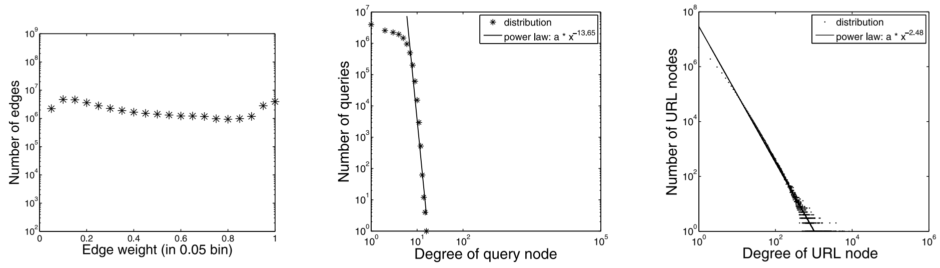
(1a) Click counts of edges (1b) Query node degrees (1c) Url node degrees

(1) The distributions before pruning.



(2a) Normalized $p(u_j|q_i)$ (2b) Query node degrees (2c) Url node degrees

(2) The distributions after pruning with ($\tau_{abs} = 5$ and $\tau_{rel} = 0.05$).



(3a) Edge weights (3b) Query node degrees (3c) Url node degrees

(3) The distributions after one step random walk.

Fig. 11. The distributions of: (a) click counts/normalized transition probabilities/weights of edges; (b) query node degrees; (c) URL node degrees of the click-through bipartite (1) before pruning, (2) after pruning, and (3) after one step random walk.

connectivity of the bipartite. Moreover, we also compare the graph statistics before and after the random walk algorithm.

Figure 11(1a) shows the distribution of the click counts of edges. Please recall the click count cc_{ix} of an edge between query q_i and URL u_x is the number of clicks on u_x for q_i. Please also note the x- and y-axes in Figure 11(1a) are in log scale. We can see the distribution follows the power-law distribution. Figures 11(1b) and (1c) show the number of query nodes and the number of URL nodes with respect to the degree of nodes, respectively. Both figures follow the power-law distribution. That means, most of the query and URL nodes have low degrees (e.g, smaller than 10), but a small number of nodes have large degrees (e.g., greater than 1000).

Figure 11(2a) shows the distribution of normalized transition probability $p(u_x|q_i)$ after pruning. As explained in Section 3.2, edges with low transition probabilities are likely to be formed due to users' random clicks, and should be removed to reduce

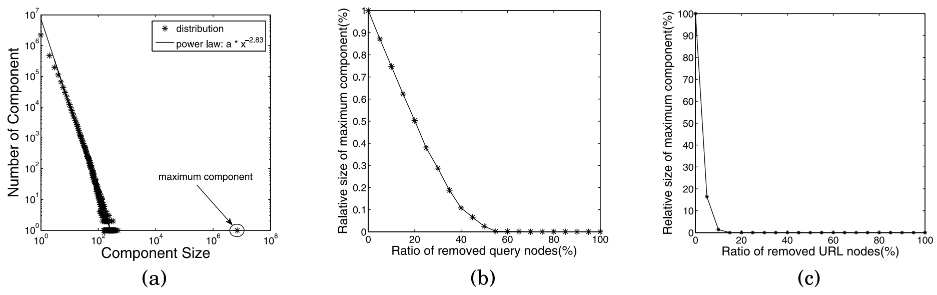


Fig. 12. (a) The distribution of component sizes. The relative size of the the largest component after removing top degree (b) query nodes and (c) URL nodes.

noise. Since we set the τ at 0.05, the thick head in Figure 11(1a) was pruned and the remaining edges show a flat distribution.

Figure 11(2b) and (2c) show the distributions of the degrees of query and URL nodes after pruning, respectively. Since each unpruned edge has a transition probability $p(u_x|q_i)$ no smaller than 0.05, a query node can only be connected with at most 20 URLs. However, the distribution of the URL node degrees remains similar to that before pruning. Please note we did not prune edges symmetrically by $p(q_i|u_x)$, because the roles of queries and URLs are not symmetric in the clustering algorithms: queries are considered as the objects to be clustered, while the URLs are treated as the features to describe queries. After the pruning process, the average degree of query nodes is 2.0, and the average degree of URL nodes is 2.4. In other words, the click-through bipartite is very sparse. This motivates us to perform a random walk on the click-through bipartite.

Figure 11(3a)–(3c) show the distributions of edge weight, degree of query nodes, and degree of URL nodes after one step of random walk, respectively. The trends of the three curves are similar to those in Figure 11(2a)–(2c), but the numbers increase substantially: the number of edges increases from 27,711,167 to 42,384,884, the average degree of query nodes increases from 2.0 to 3.1, and the degree of URL nodes increases from 2.4 to 3.7. Although the random walk alleviates the sparseness of the bipartite, the average degrees of query nodes and URL nodes are still small. This suggests why the clustering algorithms in Section 3 are efficient: since the average degrees of query and URL nodes are both low, the average number of clusters to be checked for each query is small.

We then explore the connectivity of the click-through bipartite after random walk. To be specific, we find the connected components in the bipartite and plot the number of connected components versus the number of queries in the components (see Figure 12(a)). We can see the bipartite consists of a single large connected component (including about 50% queries) and many small connected components (with sizes from 1 to 489).

We further test whether the large connected component can be broken by removing a few "hubs", that is, nodes with high degrees. To do this, we keep removing the top 5%, 10%, 15%, ..., 95% query nodes with the largest degrees and measure the percentage of the size of the largest component over the total number of remaining query nodes. Figure 12(b) shows the effect of removing top degree query nodes. We can see the percentage of the queries held by the largest component gradually drops when more top degree query nodes are removed. However, even when 20% of the query nodes are removed, the largest component still holds about half of the remaining query nodes. This suggests the click-through bipartite is highly connected, and the cluster structure cannot be obtained by simply removing a few "hubs." Figure 12(c) shows the

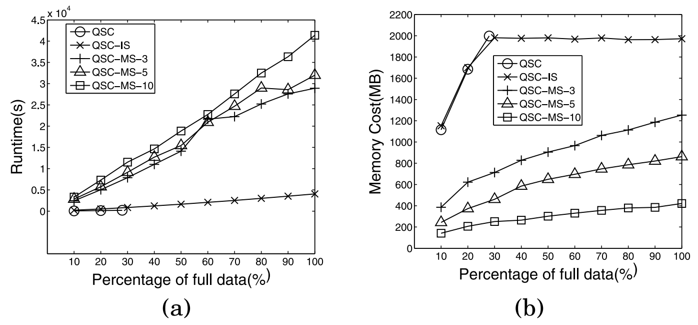


Fig. 13. The (a) time and (b) memory cost of the QSC family algorithms.

Table IV. The Clustering Results of the QSC-IS and QSC-MS Algorithms

Method	# Clusters	# Non-Singleton Clusters	# Queries in Non-Singleton Clusters
QSC-IS	4,665,871	1,903,453	11,109,587 (80.08%)
QSC-MS	4,660,757	1,958,101	11,169,349 (80.52%)

effect of removing the top degree URL nodes. We can see removing top degree URL nodes can break the largest connected component faster than removing the top degree query nodes. However, removing URL nodes loses the correlation between queries since URLs are considered as the features of queries.

5.2. Clustering the Click-through Bipartite

We set D_{max} = 1 and perform the clustering algorithms in Section 3 on the click-through bipartite. Figures 13(a) and (b) show the runtime and memory cost of the QSC, QSC-IS, and QSC-MS algorithms, respectively. We run all the three algorithms on PCs with 2G memory and Intel Xeon(R) 2-core CPUs. For the QSC-MS algorithm, we use 3, 5, and 10 slave machines.

From Figure 13(a), we can observe that the QSC and the QSC-IS algorithms have almost the same efficiency before the memory limit is reached. However, when the size of the dataset is larger than 28% of the full data, the QSC algorithm cannot hold the whole dimension array into the main memory and thus reports an out-of-memory error. Moreover, the QSC-IS algorithm is more efficient than the QSC-MS algorithm on our experiment data. This is because the speed of disk scanning is faster than network communication.

From Figure 13(b), we can see the memory consumption of the QSC algorithm increases sharply with respect to the size of the data. It reports an out-of-memory error at 28% of the full data. The memory usage of the QSC-IS algorithm keeps relatively stable. The algorithm makes almost full use of the available main memory during the iterative scanning of the data. The memory consumption of the QSC-MS algorithm is much less on each master/slave machine than that of the QSC-IS algorithm. The more slave machines used, the less memory consumption for each machine. Moreover, the size of the memory usage on each slave machine grows linearly with respect to the size of the data. Please note we do not show the memory cost of the master machine of the QSC-MS algorithm, since the master has little memory expense.

Table IV shows the number of clusters, number of nonsingleton clusters, number and percentage of unique queries in nonsingleton clusters by the QSC-IS and QSC-MS algorithms derived from the full dataset with D_{max} = 1. We do not list the results of the QSC algorithm, since it reports an out-of-memory error for the full dataset. However,

Table V. Examples of Query Clusters

Example Cluster 1	Example Cluster 2
bothell wa	catcountry
city of bothell	cat country
bothell washington	cat country radio
city of bothell wa	catcountryradio.com
city bothell washington	cat country radio station
city of bothell washington	

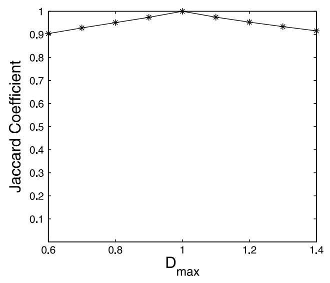


Fig. 14. The Jaccard coefficient between the set of clusters derived under D_{max} = 1 and that derived under different D_{max} values.

it is easy to see that the QSC algorithm and the QSC-MS algorithm must derive exactly the same clustering results. On the other hand, as mentioned in Section 3.3, the QSC-IS algorithm may generate different clustering results from those by the QSC algorithm. This is because the QSC-IS restrains the creation of new clusters when the memory limit is reached. As we can see from Table IV, such restraint does not influence the clustering results much.

Table V lists two examples of clusters commonly derived by both QSC-IS and QSC-MS algorithms with $D_{max} = 1$. We can see the queries in the same cluster are similar to each other and represent a common concept.

Next we test the robustness of the clustering algorithms with respect to the diameter threshold D_{max}. Two commonly used metrics to compare the clustering results are *Rand statistic* and *Jaccard coefficient* [Tan et al. 2005]. To be specific, given two clustering results $\mathcal{C}_1$ and $\mathcal{C}_2$ on the same set of data objects, let n_{11} be the number of object pairs which belong to the same clusters in $\mathcal{C}_1$ and $\mathcal{C}_2$, n_{10} be the number of object pairs which belong to the same cluster in $\mathcal{C}_1$ but not in $\mathcal{C}_2$, n_{01} be the number of object pairs which belong to the same cluster in $\mathcal{C}_2$ but not in $\mathcal{C}_1$, and n_{00} be the number of object pairs which do not belong to the same cluster in either $\mathcal{C}_1$ or $\mathcal{C}_2$, the Rand statistic and the Jaccard coefficient are defined by

$$Rand\ statistic = \frac{n_{11} + n_{00}}{n_{11} + n_{10} + n_{01} + n_{00}}, \tag{13}$$

$$Jaccard\ coefficient = \frac{n_{11}}{n_{11} + n_{10} + n_{01}}. \tag{14}$$

Figure 14 shows the Jaccard coefficients when we compare the clusters derived from the QCS-MS method under $D_{max} = 1$ with those under a wide range of D_{max} settings. We do not show the Rand statistics since most queries do not belong to the same cluster and the statistics are dominated by n_{00}. From the figure, we can see the QCS-MS clustering algorithm is very robust to the threshold D_{max}. The figure for the clusters derived from the QCS-IS method shows a similar trend.

Table VI. The Multiple Clusters that Query "*webster*" was Assigned to After the Reassignment Process

Cluster 1	Cluster 2	Cluster 3
webster	webster	webster
w-m dictionary	webster bank	webster university
merriam web dictionary	websteronline.com	webster.edu
	webster bank online	webster university st louis

After the clustering process, we conduct two postprocessing steps, that is, the split-merge of clusters and the reassignment of queries on the set of clusters derived under $D_{max} = 1$. As mentioned in Section 3.4, setting the split parameter $\sigma = 0.5$ is equivalent to set $D_{max} = 1$. Therefore, we use $\sigma = 0.5$ in the split process and keep $D_{max} = 1$ in the merge process. After the split-merge process, the number of clusters reduces from 4,665,871 (in case of QCS-IS) and 4,660,757 (in case of QCS-MS) to 4,637,669 and 4,638,701, respectively. Please note that after the split-merge process, the two clustering methods result in more consistent consistent clusters. After the split-merge process, we test whether a query can be inserted into multiple clusters as described in Section 3.4. A total number of $416,259$ (3%) queries are assigned to multiple clusters.

To evaluate the effectiveness of the split-merge process, we compare the *silhouette coefficient* [Rousseeuw 1987] of the clusters before and after the split-merge process. To be specific, for a dataset $\mathcal{D}$ of N objects, suppose $\mathcal{D}$ is partitioned into n clusters $C_1, \ldots, C_n$. Then, for each object $o \in \mathcal{D}$, $\alpha(o)$ is the average distance between o and all other objects in the cluster that o belongs to, and $\beta(o)$ is the minimum average distance from o to all clusters that o does not belong to. Formally, suppose $o \in C_a$ $(1 \le a \le n)$, then $\alpha(o) = \frac{\sum_{o' \in C_a, o \neq o'} dist(o,o')}{|C_a|-1}$, and $\beta(o) = \min_{C_b : 1 \le b \le n, b \neq a} \frac{\sum_{o' \in C_b} dist(o,o')}{|C_b|}$. The silhouette coefficient of o is defined as $\gamma(o) = \frac{\beta(o)-\alpha(o)}{\max\{\alpha(o),\beta(o)\}}$. Clearly, the value of the silhouette coefficient is between -1 and 1, and the larger the value, the better the clustering results. To compare the clusters before and after the split-merge process, we only focus on the objects whose cluster membership change after the process. In our experiments, the average silhouette coefficient increases from 0.4666 to 0.5118 for the QCS-IS method and increases from 0.5146 to 0.5280 for the QCS-MS method. This indicates that our split-merge process is effective to improve the quality of clusters.

To evaluate the accuracy of the reassignment process, we arbitrarily sample 500 queries which are assigned to multiple clusters and manually check whether the assignment was accurate. To be specific, for each query q and the cluster C_a it was assigned to, we ask three judges to label whether q is semantically related to the other queries in C_a. We consider q is correctly assigned to C_a if at least two judges agreed that q is semantically related to C_a. Not surprisingly, almost all the cases were considered correctly assigned. This is because the reassignment process requires the newly inserted query should not make the diameter of the cluster exceed the threshold D_{max}. Table VI shows an example, where query "*webster*" is assigned to three clusters representing the concepts about Webster dictionary, Webster bank, and Webster University, respectively.

5.3. Building the Concept Sequence Suffix Tree

After clustering queries, we extract session data to build the concept sequence suffix tree as our query suggestion model. Figure 15(a) shows the distribution of session lengths. We can see it is a prevalent scenario that users submit more than one query for a search intent. That means in many cases, the context information is available for query suggestion.

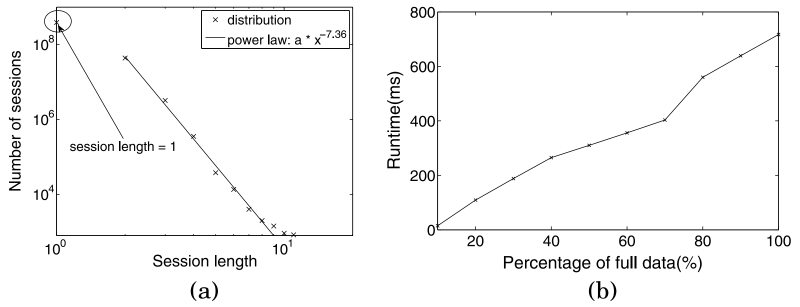


Fig. 15. (a) The distribution of session lengths. (b) The scalability of building concept sequence suffix tree.

Table VII. The Number of Nodes on the Concept Sequence Suffix Tree at Different Levels

Level	Num of Nodes	Level	Num of Nodes
1	360,963	3	14,857
2	90,539	4	2,790

We then construct the concept sequence suffix tree as described in Section 4.2. Each frequent concept sequence has to occur at least 6 times in the session data. Table VII shows the number of nodes at each level of the tree. Please note that we prune the nodes (349 in total) containing more than four concepts since we find those long patterns are not meaningful and are likely to be derived from query sequences issued by robots. Figure 15(b) shows the scalability of building the concept sequence suffix tree (Algorithm 4). We can see the time complexity of the tree construction algorithm is almost linear.

5.4. Evaluation of Query Suggestions

In this subsection, we compare the coverage and quality of the query suggestions generated by our approaches with the following baselines.

—*Adjacency*. Given a sequence of queries $q_1 \ldots q_l$, this method ranks all queries by their frequencies immediately following the query q_l in the training sessions and outputs top queries as suggestions.
—*N-Gram*. Given a sequence of queries $qs = q_1 \ldots q_l$, this method ranks all queries by their frequencies of immediately following the query sequence qs in training sessions and outputs top queries as suggestions.
—*Co-occurrence*. Given a sequence of queries $qs = q_1 \ldots q_l$, this method ranks all queries by their frequency of co-occurrence with respect to all the queries $q_1, \ldots, q_l$ as in Huang et al. [2003].

All the three baselines are session-based methods. As shown in Table V, queries in the same cluster are very similar and may not be meaningful as suggestions for each other. Therefore, we do not compare our approach with cluster-based methods. We will compare the performance of the aforesaid baselines with the approach proposed in Section 4. To evaluate the effect of the mapping method for new queries (see Section 4.3), we compare the performances when the mapping method is switched on and off, respectively. We use *CACB* to denote the the basic context-aware concept-based approach (without mapping new queries to existing concepts), and *CACB-M* to denote the one with the mapping method switched on.

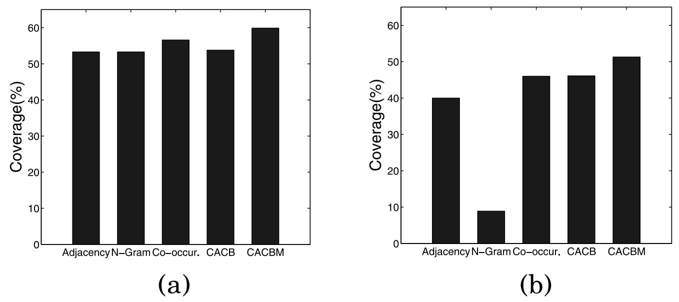
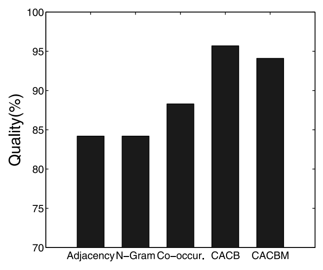


Fig. 16. The coverage of all methods on (a) Test-0 and (b) Test-1.

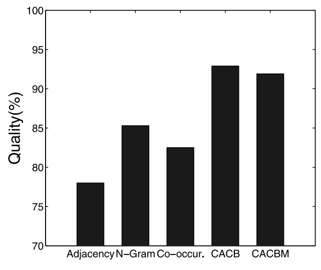
We extract 2,000 test cases from query sessions other than those serve as training data. To better illustrate the effect of context for query suggestion, we form two test sets: Test-0 contains 1,000 randomly selected single-query cases while Test-1 contains 1,000 randomly selected multiquery cases.

The *coverage* of a query suggestion method is measured by the number of test cases for which the method is able to provide suggestions over the total number of test cases. Figures 16(a) and (b) show the coverage of all methods on Test-0 and Test-1, respectively. For the single-query cases (Figure 16(a)), the N-Gram method actually reduces to the 1-Gram method, and are thus equivalent to the Adjacent method. The coverage of the CACB method is slightly higher than that of the N-Gram method and the Adjacent method, and the coverage of the Co-occurrence method is even higher. Finally, the CACB-M method has the highest coverage, increasing that of the CACB method by 11.3 percent. For the multiquery cases (Figure 16(b)), the CACB method and the Co-occurrence method have comparable coverages, which are much higher than those of the N-Gram method and the Adjacent method. In particular, the N-Gram method has the lowest coverage, while the CACB-M method has the highest coverage, improving that of the CACB method by 11.2 percent.

Given a test case $qs = q_1 \dots q_l$, the N-Gram method is able to provide suggestions only if there exists a session $qs_1 = \dots q_1 \dots q_l q_{l+1} \dots q_{l'_1}$ $(l'_1 > l)$ in the training data. The Adjacency method is more relaxed; it provides suggestions if there exists a session $qs_2 = \dots q_l q_{l+1} \dots q_{l'_2}$ $(l'_2 > l)$ in the training data. Clearly, qs_1 is a special case of qs_2. The Co-occurrence method is even more relaxed; it provides suggestions if there exists a session qs_3 where q_l co-occur in the session. We can see qs_2 is a special case of qs_3. The CACB method is also more relaxed than the Adjacency method. Suppose no sessions such as qs_2 exist in the training data, then the Adjacency method cannot provide suggestions. However, as long as there exists any sequence $qs'_2 = \dots q'_l q_{l+1} \dots q_{l'_3}$ $(l'_3 > l)$ in the training data such that q_l and q'_l belong to the same concept, the CACB method can still provide suggestions. However, if q_l does not appear in the training data, none of the N-Gram, Adjacency, Co-occurrence, and CACB methods can provide suggestions. In such cases, the CACB-M method may provide suggestions if q_l can be assigned to some existing concepts by the URL and term feature vectors. Another trend in Figure 16(a) and (b) is that for each method, the coverage drops on Test-1, where the test cases contain various lengths of context. The reason is that the longer the context, the more queries a user submits, and the more likely a session ends. Therefore, the training data available for test cases in Test-1 are not as sufficient as those in Test-0. In particular, we can see the coverage of the N-Gram method drops drastically on Test-1, while the other four methods are relatively robust. This is because the N-Gram method is most strict with training examples.



(a)



(b)

Fig. 17. The quality of all methods on (a) Test-0 and (b) Test-1.

We then evaluate the quality of suggestions generated by our approach and the baseline methods. For each test case, we mix the suggestions ranked up to top five by individual methods into a single set. We then ask human judges to label for each suggestion whether it is meaningful or not. To be more specific, each test case has the form of $\langle (q_1, U_1), \ldots, (q_l, U_l), q \rangle$, where q is the current query to which suggestions will be provided, and q_k and U_k $(1 \leq k \leq l)$ are the preceding queries and the corresponding clicked documents. Please note that in Test-0, there are no preceding queries or clicked documents available. The judges are asked to infer the user's search goal based on the current query q as well as the preceding queries and clicked documents (if available), and then tell whether clicking on a query suggestion will help the user to achieve the inferred search goal. To reduce the bias of judges, we ask ten judges with or without computer science background. Each suggestion is reviewed by at least three judges and labeled as "meaningful" only if the majority of the judges indicate that it is helpful to achieve the inferred search goal.

If one suggestion provided by a method is judged as meaningful, that method gets one point; otherwise, it gets zero point. Moreover, if two suggestions provided by a method are both labeled as meaningful, but they are near-duplicate to each other, then the method gets only one point. The overall score of a method for a particular query is the total points it gets divided by the number of suggestions it generates. If a method does not generate any suggestion for a test case, we skip that case for the method. The average score of a method over a test set is then the total score of that method divided by the number of cases counted for that method. Figure 17(a) and (b) show the average scores of all methods over the two test sets, respectively.

From Figure 17(a), we can see that in case of single-query cases (no context information available), the suggestions generated by the Adjacency and N-Gram methods have the same quality, since the N-Gram method is equivalent to the Adjacency method in this case. The quality of the Co-occurrence method is better than that of the Adjacency and N-Gram methods. This is because the Co-occurrence method in Huang et al. [2003] removes the queries which have similar co-occurrence patterns with the current one. Thus, some queries which are very similar to the current one are removed in the candidate list. For example, in the first example in Table VIII, both the Adjacency method and the N-Gram method provide "*www.att.com*" as a suggestion to the current query "*www.at&t.com*". However, this suggestion is removed by the Co-occurrence method in Huang et al. [2003], since it has similar co-occurrence pattern with that of the current query. Moreover, the method in Huang et al. [2003] groups queries by their co-occurrence patterns. In some cases, such a method can remove nearly synonym suggestions in the candidate list. For example, in the second example in Table VIII, both the Adjacency method and the N-Gram method provide suggestions "*cnn news*" and "*cnn*" at the same time to the current query "*msn news*". Since these

Table VIII. Examples of Query Suggestions Provided by the Three Methods

	Methods			
Test Case	Adjacency	N-Gram	Co-Occur	CACB(-M)
www.at&t.com	at&t www.att.com cingular www.cingular.com att net	at&t www.att.com cingular www.cingular.com att net	att wireless cingular att net bellsouth at&t	att wireless cingular bellsouth verizon tilt phone
msn news	cnn news fox news cnn msn	cnn news fox news cnn msn	cnn news msnnews msnbc news ksl news yahoo news	cnn news fox news abc news cbs news bbc news
www.chevrolet.com ⇒www.gmc.com	www.chevy.com www.chevrolet.com www.dodge.com www.pontiac.com	<null>	www.chevy.com www.dodge.com www.pontiac.com	ford toyota dodge pontiac
circuit city ⇒best buy	circuit city walmart target best buy stores sears	walmart target sears office depot	walmart staples office depot dell amazon	radio shack walmart target sears staples

two queries have similar co-occurrence patterns, only "*cnn news*" is suggested by the Co-occurrence method.

However, the quality of the Co-occurrence method is still not as good as that of the CACB and CACB-M methods. The reason is that the method in Huang et al. [2003] considers the similarity of queries by their co-occurrence patterns. However, queries with highly coherent co-occurrence patterns are not necessarily synonyms to each other. Consequently, this method may remove some meaningful suggestions. For example, in the first example in Table VIII, query "*verizon*" is pruned from the suggestion list since it has similar co-occurrence pattern with that of query "*cingular*". Moreover, by the method in Huang et al. [2003], some queries which are nearly synonyms to the current query may not necessarily have highly coherent co-occurrence patterns with that of the current one. As a result, some queries which are nearly synonyms to the current one cannot be removed from the candidate list. For example, in the second example in Table VIII, both "*msnnews*" and "*msnbc news*" have the same meaning with the current query "*msn news*". In contrast, the CACB and CACB-M methods do not use the co-occurrence patterns to measure the similarity between queries. Instead, two queries are considered similar only if they share similar clicked URLs. Therefore, the queries within the same concept by the CACB and CACB-M methods are nearly synonyms. Those queries will not be suggested at the same time to the users (see the first two examples in Table VIII). Please note that in Figure 17, the CACB-M method has comparable accuracy with the CACB method. This suggests the URL and term features can well represent the intent of a new query.

From Figure 17(b), we can see that in cases when context queries are available, the Adjacency method does not perform as well as the other methods. This is because the N-Gram, Co-occurrence, CACB, and CACB-M methods are context-aware and understand users' search intent better. Moreover, the CACB and CACB-M methods generate even better suggestions than the N-Gram and the Co-occurrence method. For example, in the third example in Table VIII, the Co-occurrence method in Huang et al. [2003] provides "*www.chevy.com*" as a suggestion. In fact, this suggestion has similar meaning with the query "*www.chevrolet.com*" which was raised by the user in the context. In the fourth example in Table VIII, the CACB and CACB-M methods

Table IX. Examples of Query Suggestions for Ambiguous or Multi-intent Queries when Context Information is Available and Not

No Context Available	Context Available	
comcast:	ebay⇒comcast:	cable⇒comcast:
myspace ebay aol comcast email login craigslit	myspace aol comcast email login craiglist	verizon at&t dish network quest t-mobile
mq:	games⇒mq	websphere⇒mq:
games dragonfable miniclip runescape adventure quest	dragonfable adventure quest runescape miniclip tribal wars	mq client mq document mq training
webster:	online dictionary ⇒webster:	citibank ⇒webster:
dictionary encarta thesaurus free dictionary bank of america	encarta thesaurus free dictionary oxford dictionary spanish dictionary	bank of america american express peoples bank citizens chase
ctc:	tenax ⇒ctc:	child tax⇒ctc:
central texas college transcript child tax benefit tarleton state university goarmyed tax rebate	central texas college transcript goarmyed tarleton state university university of maryland temple college	child tax benefit tax rebate working tax credit tax credits irs

generate a suggestion "*radio shack*", which is not suggested by any other methods. This is because the CACB and CACB-M methods model the context as a concept sequence instead of a query sequence. Therefore, these two methods can better capture the users' search intent from the context information and provide more meaningful suggestions.

We further list several examples of ambiguous or multi-intent queries in Table IX to highlight the advantage of using context. In Table VIII, the bad suggestions provided by the baseline methods mainly derive from two different reasons. One reason is that the suggestions are near-duplicates to the current query or to each other. The other reason is that the suggestions do not consider the context information. To highlight the advantage of using context, in Table IX, we compare the suggestions generated by CACB for the same query when context is available and not. Please note that the CACB method does not generate near-duplicate suggestions. Table IX contains four examples. The first query "comcast" may either refer to the Comcast Internet cable company (see `comcast.com`), or a portal Web site `comcast.net`. Without context information, the suggestions are all about the portal site. However, when a user inputs query *"cable"* before *"comcast"*, our method can provide other Internet service providers as suggestions. Similarly, the second query "mq" may be an abbreviation of an online game or a product of IBM, the third query "webster" may refer to either the online dictionary or the Webster Bank, and the last query "ctc" may refer to either the Central Texas college or child tax credit. In all examples, our method can provide meaningful suggestions according to the context information.

5.5. Evaluation on Runtime and Memory Cost

We first compare the memory cost by different query suggestion methods in Figure 18(a). For the Adjacency and Co-occurrence methods, the main memory holds a lookup table where the key is a query and the value is the set of candidate query suggestions for that query. Since the set of query pairs extracted from sessions by the Co-occurrence method is much larger than that by the Adjacency method, the memory cost by the Co-occurrence method is larger than that by the Adjacency method. For the N-gram method, the data structure to be held in the main memory is a query-sequence suffix tree, which is similar to a concept-sequence suffix tree as described in Section 4.2. The only difference is that in a query-sequence suffix tree, each node represents a query instead of a concept. Since the N-gram method indexes not only adjacent query pairs but also query sequences with more than two queries, the memory cost of the N-gram method is larger than that of the Adjacency method. On the other hand, the query sequences indexed by the N-gram method keep the original order of queries in sessions. Therefore, the memory cost of the N-gram method is still smaller than that of the Co-occurrence method, where the indexed query pairs do not need to be adjacent in the sessions. The CACB method is supported by a concept-sequence suffix tree as well as a mapping table which maps a query to a concept. The size of the concept-sequence suffix tree is smaller than that of the query-sequence suffix tree by the N-gram method, since each concept node includes a set of similar queries. However, the extra cost of the mapping table makes the total memory cost of the CACB method larger than that of the N-gram method. Please note that some concepts have no following concepts in sessions and thus cannot be used for query suggestion. Therefore, those concepts are not indexed in the concept-sequence suffix tree and it is not necessary to hold their mapping entries in the main memory. Finally, compared with the CACB method, the CACB-M method needs additional memory to hold the term and URL feature vectors. Therefore, the memory cost of the CACB-M method is the largest. Again, we only need to hold the feature vectors for those concepts indexed in the concept-sequence suffix tree. Although the CACB-M method costs the most memory, it requires less than 2G memory for the full dataset. Such a requirement is acceptable for a modern server.

We then compare the runtime for different methods. Basically, the runtime of all the methods include two parts. One is the offline mining time, and the other is the online query suggestion time. The offline mining time for our method consists of two parts, that is, (a) the time for clustering the click-through bipartite to generate concepts, and (b) the time to mine frequent concept sequences from session data and construct the concept sequence suffix tree. For all the baseline methods, since they do not generate concepts, the offline mining time only includes (b'), which is the time to mine frequent query pairs or sequences from session data and construct the query suggestion lookup table or the query sequence suffix tree. Since we have reported the clustering time of our method at different scales in Figure 13, in the following, we only compare (b) and (b') of different methods.

Figure 18(b) shows the runtime of mining frequent patterns for all the methods on different percentages of the full data. Recall that in Section 4.2 we develop a method to distributively mine frequent patterns under the map-reduce programming model. In this experiment, we mine the query pairs and sequences for all the baseline methods in a similar way. In Figure 18(b), the three baseline methods have different runtime because they mine different frequent patterns. For example, given a session $qs = q_1q_2q_3$, the Adjacency method generates two query pairs at the map step, that is, q_1q_2 and q_2q_3; the N-Gram method generates two subsequences q_1q_2 and $q_1q_2q_3$; and the Co-occurrence method generates six query pairs, including q_1q_2, q_1q_3, q_2q_1, q_2q_3, q_3q_1, and q_3q_2. Since the Adjacency method only mines length-2 patterns and restricts the queries should appear adjacent in sessions, the runtime is the shortest. The N-gram method mines various length patterns. Thus, its runtime is longer than that of the Adjacency method. The Co-occurrence method does not constrain the order of queries

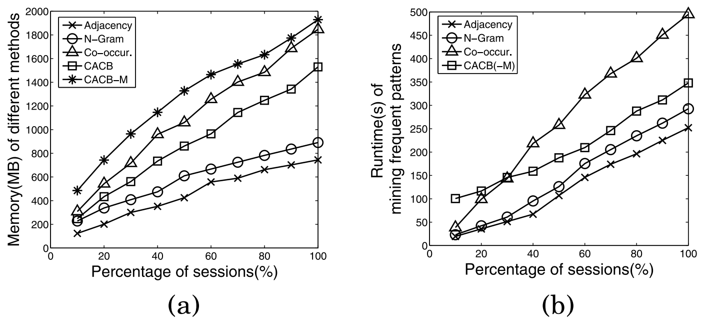


Fig. 18. (a) The memory cost of different query suggestion methods. (b) The runtime of mining and indexing candidate suggestions.

and generates the largest number of patterns. Consequently, it needs more time than that by the Adjacency and the N-Gram methods. The CACB(-M) method needs the overhead to convert a query sequence into a concept sequence before it mines frequent patterns. From Figure 18(b), we can see such overhead makes it more costly than the Adjacency and the N-gram methods. When comparing CACB(-M) with Co-occurrence, we can see that the overhead of the former makes it more expensive than the latter when the dataset is small, that is, smaller than 30% of the full data. However, when the dataset grows larger, the runtime of CACB(-M) is smaller than that of the Co-occurrence method since CACB(-M) restricts the order of concepts and thus generates a smaller number of frequent patterns.

Finally, we conduct a series of experiments to thoroughly evaluate the online query suggestion time for all the methods. In the first experimental setting, we randomly pick up 10,000 single-query sessions as test cases and calculate the average time for different methods to provide query suggestions. Therefore, in this setting, the N-gram method have the same with the Adjacency method. Moreover, in this setting, we do not use the term and URL feature vectors to capture novel queries. In other words, the CACB method is the equivalent with the CACB-M method in this setting. As Figure 19(a) shows, the Adjacency and the N-gram method has comparable performance, while the Co-occurrence method is slower. This is because the Co-occurrence method has to search a larger lookup table. The online response time of the CACB(-M) is the slowest, since it needs to map the input query into the corresponding concept.

In the second experimental setting, we randomly pick up 10,000 sessions with at least two queries as test cases. Again, we shut off the term and URL feature vectors in this setting. As indicated by Figure 19(b), when context is available, both context-aware methods, that is, the N-Gram method and the CACB(-M) method, cost more than those noncontext-aware methods. This is because the context-aware methods need to search deep in the suffix tree to match the context information.

In the last experimental setting, we randomly pick up 10,000 sessions where at least one query in the session is not covered by the CACB method. Then we compare the runtime when the term and URL features are switched on and off. From Figure 19(c), we can see that using the term and URL features to cover novel queries is quite expensive, which costs almost ten times the response time when the features are shut off. However, even when the term and URL features are turned on to capture novel queries, the total response time is still small, that is, about 0.3 millisecond. Such response time is feasible for a commercial search engine, since the query suggestion process can be conducted in parallel with search results ranking.

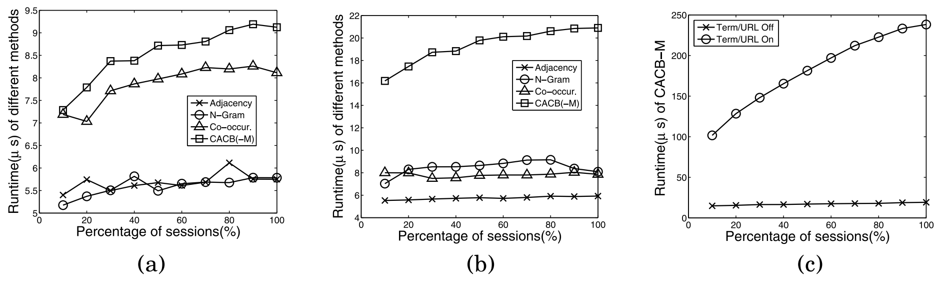


Fig. 19. The online runtime of different methods when: (a) context is not available and (b) context is available. (c) The online runtime of CACB-M when the term and URL feature vectors are turned on and off. The y-axis is in 10^{-6} seconds.

6. CONCLUSION

In this article, we proposed a novel approach to query suggestion using click-through and session data. Unlike previous methods, our approach groups similar queries into concepts and models context information as sequences of concepts. The experimental results on a large-scale dataset containing billions of queries and URLs clearly show our approach outperforms three baselines in both coverage and quality.

In the future, we will extend our context-aware approach to other search applications, such as query expansion, query substitution, query categorization, and document ranking. Moreover, we will explore a uniform framework which summarizes context patterns and support various context-aware applications at the same time.

REFERENCES

Anagnostopoulos, A., Becchetti, L., Castillo, C., and Gionis, A. 2010. An optimization framework for query recommendation. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM'10)*. ACM, New York, 161–170.

Baeza-Yates, R. and Tiberim A. 2007. Extracting semantic relations from query logs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'07)*. ACM, 76–85.

Baeza-Yates, R. A., Hurtado, C. A., and Mendoza, M. 2004. Query recommendation using query logs in search engines. In *Proceedings of the EDBT Workshop on Clustering Information over the Web*. Springer, 588–596.

Beeferman, D. and Berger, A. 2000. Agglomerative clustering of a search engine query log. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)*. ACM, 407–416.

Ben-Dor, A., Shamir, R., and Yakhini, Z. 1999. Clustering gene expression patterns. *J. Comput. Biol. 6,* 3/4, 281–297.

Boldi, P., Bonchi, F., Castillo, C., Donato, D., and Vigna, S. 2009. Query suggestions using query-flow graphs. In *Proceedings of the Workshop on Web Search Click Data (WSCD'09)*. ACM, 56–63.

Boldi, P., Bonchi, F., Castillo, C., Donato, D., Gionis, A., and Vigna, S. 2008. The query-flow graph: model and applications. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*. 609–618.

Cao, G., Nie, J.-Y., Gao, J., and Robertson, S. 2008. Selecting good expansion terms for pseudo-relevance feedback. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (SIGIR'08)*. ACM, 243–250.

Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., and Li, H. 2008. Context-Aware query suggestion by mining click-through and session data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08)*. ACM, 875–883.

Chirita, P. A., Firan, C. S., and Nejdl, W. 2007. Personalized query expansion for the web. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Rtrieval (SIGIR'07)*. 7–14.

CRASELL, N. AND SZUMMER, M. 2007. Random walks on the click graph. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 239–246.

CUI, H., WEN, J., NIE, J., AND MA, W. 2002. Probabilistic query expansion using query logs. In *Proceedings of the 11th International Conference on World Wide Web (WWW'02)*. 325–332.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*. 137–150.

ESTER, M., KRIEGEL, H., SANDER, J., AND XU, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on KDD*. 226–231.

FONSECA, B. M., GOLGHER, P., PÔSSAS, B., RIBEIRO-NETO, B., AND ZIVIANI, N. 2005. Concept-Based interactive query expansion. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*. 696–703.

GAO, J., YUAN, W., LI, X., DENG, K., AND NIE, J.-Y. 2009. Smoothing clickthrough data for web search ranking. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09)*. 355–362.

GUO, J., XU, G. AND LI, H., AND CHENG, X. 2008. A unified and discriminative model for query refinement. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*. 379–386.

HINNEBURG A. AND KEIM, D. A. 1999. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*. 506–517.

HUANG, C., CHIEN, L., AND OYANG, Y. 2003. Relevant term suggestion in interactive web search based on contextual information in query session logs. *J. Amer. Soc. Inf. Sci. Technol. 54,* 7, 638–649.

JANSEN, B.J., SPINK, A., BATEMAN, J., AND SARACEVIC, T. 1998. Real life information retrieval: A study of user queries on the web. *SIGIR Forum*, 5–17.

JENSEN, E. C., BEITZEL, S., CHOWDHURY, A., AND FRIDER, O. . 2006. Query phrase suggestion from topically tagged session logs. In *Proceedings of the 7th International Conference on Flexible Query Answering Systems (FQAS'06)*. 185–196.

JONES, R., REY, B. MADANI, O. AND GREINER, W. 2006. Generating query substitutions. In *Proceedings of the 15th International Conference on World Wide Web (WWW'06)*. 387–396.

KRAFT, R. AND ZIEN, J. 2004. Mining anchor text for query refinement. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*. 666–674.

LAU, T. AND HORVITZ, E. 1999. Patterns of search: Analyzing and modeling web query refinement. In *Proceedings of the 7th International Conference on User Modeling*. 119–128.

LAVRENKO, V. AND CROFT, W.B. 2001. Relevance based language models. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'01)*. 120–127.

LIU, S., LIU, F., YU, C., AND MENG, W. 2004. An effective approach to document retrieval via utilizing wordnet and recognizing phrases. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'04)*. ACM, 266–272.

MAGENNIS, M. AND VAN RIJSBERGEN, C.J. 1997. The potential and actual effectiveness of interactive query expansion. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'97)*. 324–332.

MEI, Q., KLINKNER, K., KUMAR, R., AND TOMKINS, A. 2009. An analysis framework for search sequences. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*. 1991–1996.

MEI, Q., ZHOU, D., AND CHURCH, K. 2008. Query suggestion using hitting time. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*. 469–478.

METZLER, D. AND CROFT, W.B. 2007. Latent concept expansion using markov random fields. In *In Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 311–318.

PEI, J., HAN, J., MORTAZAVI-ASL, B., PINTO, H., CHEN, Q., DAYAL, U., AND HSU, M.-C. 2001. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the International Conference on Data Engineering (ICDE'01)*. 215–224.

RIEH, S. Y. AND XIE, H. 2001. Patterns and sequences of multiple query reformulations in web searching: A preliminary study. In *Proceedings of ASIS&T Annual Meeting*. 246–255.

ROUSSEEUW, P. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math. 20*, 53–65.

SADIKOV, E., MADHAVAN, J., WANG, L., AND HALEVY, A. 2010. Clustering query refinements by user intent. In *Proceedings of the International World Wide Web Conference (WWW'10)*. 841–850.

SAHAMI, M. AND HEILMAN, T.D. 2006. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th International Conference on World Wide Web (WWW'06)*. 377–386.

SILVERSTEIN, C., MARAIS, H., HENZINGER, M., AND MORICZ, M. 1999. Analysis of a very large web search engine query log. *ACM SIGIR Forum*, 6–12.

SRIKANT, R. AND AGRAWAL, R. 1996. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference of Extending Database Technology (EDBT'96)*. 3–17.

TAN, P. N., STEINBACH, M., AND KUMAR, V. 2005. *Introduction to Data Mining*, 1st Ed. Addison-Wesley Longman Publishing Co., Inc.

TERRA, E. AND CLARKEM, C.L.A. 2004. Scoring missing terms in information retrieval tasks. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM'04)*. 50–58.

WEN, J., NIE, J., AND ZHANG, H. 2001. Clustering user queries of a search engine. In *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*. 162–168.

WHITE, R. W., BAILEY, P., AND CHEN, L. 2009. Predicting user interests from contextual information. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09)*. 363–370.

WHITE, R. W., BENNETT, P.N., AND DUMAIS, S.T. 2010. Predicting short-term interests using activity-based search context. In *Proceedings of 19th International Conference on Information and Knowledge Management (CIKM'10)*. 1009–1018.

WHITE, R. W., BILENKO, M., AND CUCERZAN, S. 2007. Studying the use of popular destinations to enhance web search interaction. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*. 159–166.

ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 103–114.

Received October 2010; revised February 2011; accepted April 2011