Context-Aware Query Suggestion by Mining Click-Through and Session Data[*]

Huanhuan Cao[1]   Daxin Jiang[2]   Jian Pei[3]   Qi He[4]
Zhen Liao[5]   Enhong Chen[1]   Hang Li[2]

[1]University of Science and Technology of China   [2]Microsoft Research Asia   [3]Simon Fraser University
[4]Nanyang Technological University   [5]Nankai University
[1]{caohuan, cheneh}@ustc.edu.cn   [2]{djiang, hangli}@microsoft.com   [3]jpei@cs.sfu.ca
[4]qihe@pmail.ntu.edu.sg   [5]liaozhen@mail.nankai.edu.cn

ABSTRACT

Query suggestion plays an important role in improving the usability of search engines. Although some recently proposed methods can make meaningful query suggestions by mining query patterns from search logs, none of them are context-aware – they do not take into account the immediately preceding queries as context in query suggestion. In this paper, we propose a novel context-aware query suggestion approach which is in two steps. In the *offline model-learning step*, to address data sparseness, queries are summarized into concepts by clustering a click-through bipartite. Then, from session data a *concept sequence suffix tree* is constructed as the query suggestion model. In the *online query suggestion step*, 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, our approach suggests queries to the user in a context-aware manner. We test our approach on a large-scale search log of a commercial search engine containing 1.8 billion search queries, 2.6 billion clicks, and 840 million query sessions. The experimental results clearly show that our approach outperforms two 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

Keywords

Query suggestion, click-through data, session data

---

1. INTRODUCTION

The effectiveness of information retrieval from the web largely depends on whether users can issue queries to search engines, which properly describe their information needs. Writing queries is never easy, because usually queries are short (one or two words on average) [19] and words are ambiguous [5]. 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 issue queries to 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!, Live Search, Ask, and Baidu provide *query suggestions* to improve usability. That is, by guessing a user's search intent, a search engine suggests queries which may better reflect the user's information need. A commonly used query suggestion method [1, 3, 19] is to find similar queries in search logs and use those queries as suggestions for each other. Another approach [8, 10, 11] 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 cases, none of them are *context-aware* – they do not take into account the immediately preceding queries as context in query suggestion.

EXAMPLE 1   (SEARCH INTENT AND CONTEXT). Suppose a user raises a query "*gladiator*". It is hard to determine the user's search intent, i.e., 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 may have 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 recent queries issued by the user can help to better understand the user's search intent and enable us to make more meaningful suggestions. ∎

In this paper, we propose a novel context-aware query suggestion approach by mining click-through data and session data. We make the following contributions.

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. We may have millions of unique queries involving millions of unique URLs, which may result in hundreds of thousands of concepts. To tackle these challenges, we develop a novel, highly scalable yet effective algorithm.

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

Third, we empirically study a large-scale search log containing 1.8 billion search queries, 2.6 billion clicks, and 840 million 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 query suggestion approach on the search log. The experimental results clearly show that our approach outperforms two baseline methods in both coverage and quality of suggestions.

The rest of the paper is organized as follows. We first present the framework of our approach in Section 2 and review the related work in Section 3. The clustering algorithm and the query suggestion method are described in Sections 4 and 5, respectively. We report an empirical study in Section 6. The paper is concluded in Section 7.

2. FRAMEWORK

When a user submits a query q, our context-aware approach first captures the context of q which is represented by a short sequence of queries issued by the same user immediately before q. We then check the historical data and find what queries many users often ask after q in the same context. Those queries become the candidate suggestions.

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. Using specific queries to describe context directly cannot capture contexts concisely and accurately.

To tackle this problem, we propose summarizing individual queries into *concepts*, where a concept is 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 intents more accurately. To mine concepts from queries, we use the URLs clicked for queries as the features of the queries. In other words, we mine concepts by clustering queries in a click-through bipartite. In Section 4, we will describe how to mine concepts of queries.

With the help of concepts, a context can be represented by a short sequence of concepts about the queries asked by the user in the session. The next issue is how to find the queries that many users often ask in a particular context.

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 log data. The contexts mined are organized into a concept sequence suffix tree which can be searched quickly. The mining process is conducted offline. When a user context is presented, we look up the context in the concept sequence suffix tree to find out the concepts to which the user's next
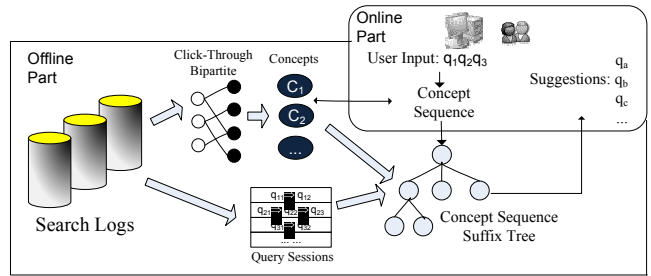

Figure 1: The framework of our approach.

query most likely belongs, and suggest the most popular queries in those concepts to the user. The details about mining sessions, building concept sequence suffix tree, and making query suggestions are discussed in Section 5.

Figure 1 shows the framework of our context-aware approach, which consists of two steps. The *offline model-learning step* mines concepts from a click-through bipartite constructed from search log data, and builds a concept sequence suffix tree from sessions in the data. The *online query suggestion step* matches the current user's 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.

3. RELATED WORK

A great challenge for search engines is to understand users' search intents behind queries. Traditional approaches to query understanding focus on exploiting information such as users' explicit feedbacks (e.g., [14]), implicit feedbacks (e.g., [18]), user profiles (e.g., [4]), thesaurus (e.g., [13]), snippets (e.g., [17]), and anchor texts (e.g., [12]).

Several recent studies use search logs to mine "wisdom of the crowds" for query understanding. For example, Huang *et al.* [10] mined search session data for query pairs frequently co-occurring in the same sessions. The mined query pairs were then used as suggestions for each other. Fonseca *et al.* [8] and Jones *et al.* [11] extracted query pairs which are often adjacent in the same sessions. The extracted adjacent query pairs were utilized for query expansion [8] and query substitution [11]. We call these methods *session-based* approaches.

Some other studies 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. After the clustering process, the queries within the same cluster are used as suggestions for each other. We call these methods *cluster-based* approaches. For example, Beeferman *et al.* [3] applied a hierarchical agglomerative method to obtain similar queries in an iterative way. Wen *et al.* [19] combined query content information and click-through information and applied a density-based method, DBSCAN [7], 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.* [1] used the k-means algorithm to derive similar queries. The k-means algorithm requires the user to specify the number of clusters, which is difficult for clustering search logs.

There are some other clustering methods such as BIRCH [21] though they have not been adopted in query under-

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
...			

Table 1: A search log as a stream of query and click events.

standing. We find, however, those algorithms may not be able to handle the following two challenges. First, many algorithms cannot address the "curse of dimensionality" caused by the large number of URLs in logs. Second, most algorithms cannot support the dynamic update of clusters when new logs are available.

The approach developed in this paper has three critical differences from previous ones. First, unlike the existing session-based methods which only focus on query pairs, we consider variable-length contexts of queries, and provide context-aware suggestions. Second, different from the cluster-based methods, we do not simply use queries in the same cluster as candidate suggestions for each other. Instead, we suggest queries that a user may ask next in the query context, which are more useful than queries simply replaceable to the current query. Finally, instead of using individual queries to capture users' search intents, our suggestion method summarizes queries into concepts.

4. MINING QUERY CONCEPTS

In this section, we summarize queries into concepts. We first describe how to form a click-through bipartite from search log data, and then present an efficient algorithm which can mine from a very large bipartite.

4.1 Click-Through Bipartite

To group similar queries into a concept, we need to measure the similarity between 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 set of features of that query. The information about queries and their clicked URL sets is available in search log data.

A search log can be regarded as a sequence of query and click events. Table 1 shows an example of a search log. 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_{ij} is created between query node q_i and URL node u_j if u_j is a clicked URL of q_i. The *weight* w_{ij} of edge e_{ij} is the total number of times when u_j is a click of q_i aggregated over the whole log. Figure 2 shows an example click-through bipartite.

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 [1, 3, 19]. From the click-through bipartite, we represent each query q_i as an L_2-normalized vector, where each dimension corresponds to one URL in the bipartite. To be specific, given a click-through bipartite, let Q and U be the sets of query nodes and URL nodes, respectively. The j-th element of the feature
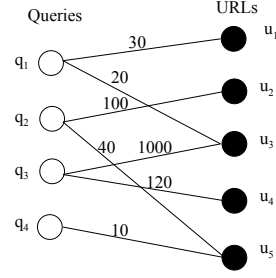

Figure 2: An example of click-through bipartites.

vector of a query $q_i \in Q$ is

$$\overrightarrow{q_i}[j] = \begin{cases} norm(w_{ij}) & \text{if edge } e_{ij} \text{ exists;} \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

where $u_j \in U$ and $norm(w_{ij}) = \frac{w_{ij}}{\sqrt{\sum_{\forall e_{ik}} w_{ik}^2}}$.

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

$$distance(q_i, q_j) = \sqrt{\sum_{u_k \in U} (\overrightarrow{q_i}[k] - \overrightarrow{q_j}[k])^2}. \quad (2)$$

4.2 Clustering Method

There are several challenges in clustering queries effectively and efficiently in a click-through bipartite. First, a click-through bipartite from a search log can be huge. For example, the data set in our experiments consists of more than 151 million unique queries. Therefore, the clustering algorithm must be efficient and scalable to handle large data sets. 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 data set is of extremely high dimensionality. For example, the data set used in our experiments includes more than 114 million unique URLs. Therefore, the clustering algorithm must tackle the "curse of dimensionality". Last, the search logs increase dynamically. Therefore, the clustering needs to be maintained incrementally.

To the best of our knowledge, no existing methods can address all the above challenges simultaneously. We develop a new method as shown in Algorithm 1.

Algorithm 1 Clustering queries.

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

In our method, a cluster C is a set of queries. The *normalized centroid* of the cluster is $\overrightarrow{c} = norm(\frac{\sum_{q_i \in C} \overrightarrow{q_i}}{|C|})$, where

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

$$distance(q, C) = \sqrt{\sum_{u_k \in U} (\overrightarrow{q}[k] - \overrightarrow{c}[k])^2}, \qquad (3)$$

We adopt the diameter measure in [21] to evaluate the compactness of a cluster, i.e.,

$$D = \sqrt{\frac{\sum_{i=1}^{|C|} \sum_{j=1}^{|C|} (\overrightarrow{q_i} - \overrightarrow{q_j})^2}{|C|(|C| - 1)}}. \qquad (4)$$

We use a diameter parameter D_{max} to control the granularity of clusters: every cluster has a diameter at most D_{max}.

Our method only needs one scan of the queries. We create a set of clusters as we scan the queries. 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 [21]. Unfortunately, as shown in previous studies (e.g., [9]), the CF-Tree structure may not handle high dimensionality well: when the dimensionality increases, BIRCH tends to compress the whole data set 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. For example, in our experimental data, a query is connected with an average number of 8.2 URLs. Moreover, each URL is also involved in only a few queries. In our experiments, the average degree of URL nodes is only 1.8. 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 $8.2 \cdot (1.8 - 1) = 6.56$. Intuitively, for any cluster C, if $C \cap Q_q = \phi$, 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 Equation 2 (please note that the feature vectors of queries have been 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 our method, the average number of clusters to be checked is not larger than 6.56.

Based on the above 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_i and links to a set of clusters Θ_i, where each cluster $C \in \Theta_i$ contains at least one member query q_j such that $\overrightarrow{q_j}[i] \neq 0$. As an example, for a query q, suppose the non-zero dimensions of $\overrightarrow{q}$ 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 3rd, the 6th, and the 9th entries of the dimension array, respectively. The closest cluster to q must be in the union.

Since the click-through bipartite is sparse, one might wonder whether it is possible to derive clusters by finding the connected components from the bipartite. To be specific, two queries q_s and q_t are *connected* if there exists a query-URL path $q_s\text{-}u_1\text{-}q_1\text{-}u_2\text{-}\ldots\text{-}q_t$ where a pair of adjacent query and URL in the path are connected by an edge. A clus-
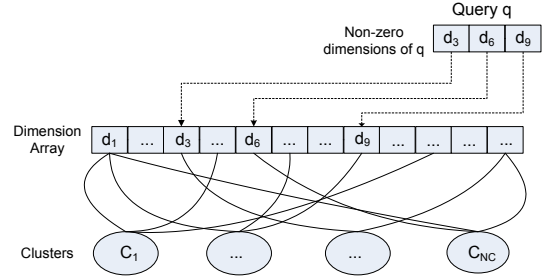

Figure 3: The data structure for clustering.

ter 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 the bipartite is highly connected though sparse. In other words, almost all queries, no matter similar or not, are included in a single connected component. Moreover, the path between dissimilar queries cannot be broken by simply removing a few "hubs" of query or URL nodes as shown in Figure 6. Thus, clusters cannot be derived from connected components straightforwardly.

Although Algorithm 1 is efficient, the computation 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 are likely to be formed due to users' random clicks, and should be removed to reduce noise. To be specific, let e_{ij} be the edge connecting query q_i and u_j, and w_{ij} be the weight of e_{ij}. Moreover, let w_i be the sum of the weights of all the edges where q_i is one endpoint, i.e., $w_i = \sum_j w_{ij}$. We can prune an edge e_{ij} if the absolute weight $w_{ij} \leq \tau_{abs}$ or the relative weight $\frac{w_{ij}}{w_i} \leq \tau_{rel}$, where τ_{abs} and τ_{rel} are user specified thresholds. After pruning low-weight edges, we can further remove the query and the URL nodes whose degrees become zero. In our experiments, we set $\tau_{abs} = 5$ and $\tau_{rel} = 0.1$. After the pruning process, the algorithm can run efficiently on a PC of 2 GB main memory for the experimental data.

5. CONDUCTING QUERY SUGGESTIONS

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

5.1 Query Sessions

As explained in Section 2, 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 user query sessions.

We construct session data in three steps. First, we extract each individual 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 *sessions* based on a widely-used rule [20]: two consecutive events (either query or click) are segmented into two sessions if the time interval between them exceeds 30 minutes. Finally, we discard the click events and only keep the sequence of queries in each session.

Query sessions can be used as training data for query suggestion. For example, Table 2 shows some real sessions as well as the relationship between the queries in the ses-

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

Table 2: Examples of sessions and relationship between queries in sessions.

sions. We can see that a user may refine the queries or explore related information about his or her search intent in a session. As an example, from the last session in Table 2, we can derive three training examples, i.e., "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 $\Rightarrow$ Nokia N73 themes".

5.2 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 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_i is represented by a cluster C_i derived in Section 4.2 and a query q_i is mapped to c_i if $q_i \in C_i$. If two consecutive queries belong to the same concept, we record the concept only once in the sequence.

How can we mine patterns from concept sequences? A straightforward method can first mine all frequent sequences from session data. For each frequent sequence $cs = c_1 \ldots c_l$, we can use c_l as a candidate concept for $cs' = c_1 \ldots c_{l-1}$. We then can build a ranked list of candidate concepts c for cs' based on their occurrences following cs' in the same sessions; the more occurrences c has, the higher c is ranked. For each candidate concept c, we can choose from the corresponding cluster C the member query which has the largest number of clicks as the representative of c. In practice, we only need to 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 can be used for query suggestion when cs' is observed online.

The major cost in the above method is from computing the frequent sequences. Traditional sequential pattern mining algorithms such as GSP [16] and PrefixSpan [15] 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 suggestion, 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 $c_{1+i}, \ldots, c_{m+i}$, where $i \geq 0$ and $m + i \leq l$. Therefore, the number of subsequences to be considered for cs is only $\frac{l \cdot (l-1)}{2}$. Finally, the average num-
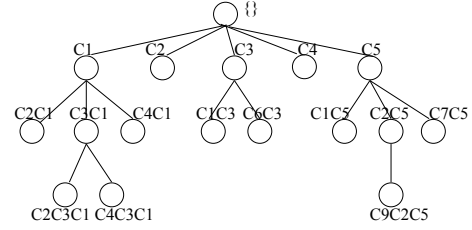

Figure 4: A concept sequence suffix tree.

Algorithm 2 Building the concept sequence suffix tree.

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

Method: findNode($cs = c_1 \ldots c_l$, T);
1: **if** $|cs| = 0$ **then return** $T.\text{root}$;
2: $cs' = c_2 \ldots c_l$; $pn = \text{findNode}(cs', T)$; $cn = pn.\text{childlist}[c_1]$;
3: **if** $cn == $ **null then**
4:     $cn = $ new node (cs); $cn.\text{candlist} = \phi$; $pn.\text{childlist}[c_1] = cn$;
5: **return** cn;

---

ber 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 [6]. In the map operation, each machine (called a *process node*) receives a subset of concept sequences as input. For the concept sequence cs of a 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 the 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 into a *concept sequence suffix tree* (Figure 4). Formally, a *(proper) suffix* of a concept sequence $cs = c_1 \ldots c_l$ is an empty sequence or a sequence $cs' = c_{l-m+1} \ldots c_l$, where $m \leq l$ ($m < l$). In a concept sequence suffix tree, each node corresponds to a frequent concept sequence cs. Given two nodes cs_i and cs_j, cs_i is the parent node of cs_j if cs_i is the longest proper suffix of cs_j. Except the root node which corresponds to the empty sequence, each node on the tree is associated with a list of candidate suggestions.

Algorithm 2 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 $cs = c_1 \ldots c_l$, the algorithm first finds the node cn corresponding to $cs' = c_1 \ldots c_{l-1}$. If cn does not exist, the algorithm creates a new node for cs' recursively. Finally, the algorithm updates the list of candidate concepts of cs if c_l is among the top K candidates observed so far.

Algorithm 3 Query suggestion.

Input: the concept sequence suffix tree T and user input query sequence qs;
Output: the ranked list of suggested queries $S\text{-}Set$;
Initialization: $curN = T.\text{root}$; $S\text{-}Set = \phi$;

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:   $curN = chN$; $curC =$ the previous concept of $curC$ in cs;
7:   **if** ($curC ==$**null**) **then break**;
8: **if** $curN \mathrel{!}= T.\text{root}$ **then**
9:   $S\text{-}Set = curN$'s candidate suggestions;
10: **return** $S\text{-}Set$;

	Original Graph	Pruned Graph
# Query Nodes	151,869,102	1,835,270
# URL Nodes	114,882,486	8,309,988
# Edges	631,590,146	15,043,517
# Query Occurrences	1,812,563,301	926,442,156
# Clicks	2,554,683,191	1,321,589,933

Table 3: The size of the click-through bipartite before and after pruning.

In Algorithm 2, the major cost for each sequence is from the recursive function *findNode*, which looks up the node cn corresponding to $c_1 \ldots c_{l-1}$. Clearly, the recursion executes at $l-1$ levels. 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 cannot exceed 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.

5.3 Online Query Suggestion

Suppose 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. However, unlike the queries in the training examples, an online input query q_i may be new and may not belong to any concept derived from the training data. Moreover, when q_i is a new query, no click-through information is available. In this case, the mapping process stops and the concept sequence corresponding to $q_{i+1} \cdots q_l$ is returned.

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 3. 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 the same as $curC$. 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. A special case is that $curN$ is the root node when the search process stops. This means no match for the last concept in the concept sequence is found on the suffix tree. In this case, the system cannot provide suggested queries according to the current user input.

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 5.2, the cost of this operation is $O(\log N_C)$. In fact, when generating suggested queries 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, 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.

6. 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 log contains 1,812,563,301 search queries, 2,554,683,191 clicks, and 840,356,624 query sessions, which involve 151,869,102 unique queries and 114,882,486 unique URLs.

6.1 Clustering the Click-Through Bipartite

We build a click-through bipartite to derive concepts. As described in Section 4.2, we set $\tau_{abs} = 5$ and $\tau_{rel} = 0.1$ to prune low-weight edges. Table 3 shows the sizes of the click-through bipartite before and after the pruning process.

It has been shown in previous work (e.g., [2]) 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.

Figure 5(a) shows the distribution of the edge weights (please note the x- and y-axes in Figure 5 are in log scale). The distribution follows power law. Therefore, although the pruning process removes 76% edges and 96% nodes, there are still 51.1% of the query occurrences and 51.7% of the URL clicks remained in the pruned data.

Figure 5(b) shows the number of query nodes versus the degree of nodes. The curve can be divided into three intervals. In the first interval (degree ranging from 1 to 10), the number of queries drops sharply from about $300,000$ to $80,000$. This interval consists of the major part (73.8%) of the queries. In other words, most queries are associated with a small number of URLs. The second interval (degree spanning from 11 to 200) approximates a power-law distribution. There are about 26.2% queries falling in this interval. This means a small part of the queries are associated with a moderate number of URLs. The last interval (degree between 201 and 600) includes only 22 queries. It would be interesting to further study how such a degree distribution is formed, though it is beyond the scope of this paper. The average degree of query nodes is 8.2.

Figure 5(c) shows the number of URL nodes versus the degree of nodes. The curve fits a power-law distribution well. The average degree of URL nodes is 1.8. The curves in Figure 5(b) and (c) illustrate why the clustering algorithm in Section 4.2 is 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. First, we find the connected components in the
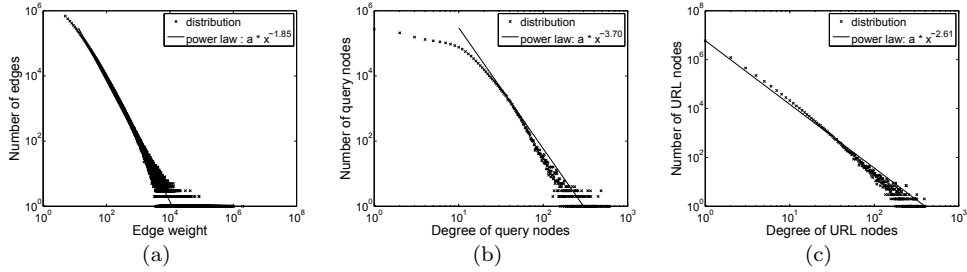
Figure 5: The distributions of (a) edge weights, (b) query node degrees, and (c) URL node degrees.
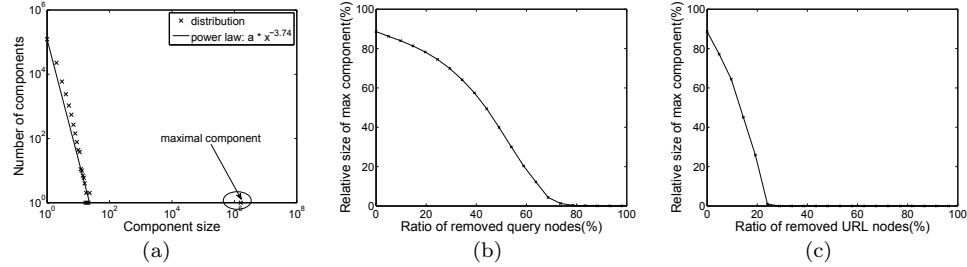

Figure 6: (a) The distribution of component sizes, (b) the relative size of the largest component after removing top degree query nodes, and (c) the relative size of the largest component after removing top degree URL nodes.

bipartite and plot the number of connected components versus the number of queries in the components (Figure 6(a)). The bipartite consists of a single large connected component (about 88.6% of all queries) and many small connected components (with size from 1 to 20). We further test whether the large connected component can be broken by removing a few "hubs", i.e., nodes with high degrees. To do this, we keep removing the top 1%, 2%, ..., 99% query nodes with the largest degrees and measuring the percentage of the size of the largest component over the total number of remaining query nodes. Figure 6(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 half of the query nodes are removed, the largest component still holds about one third of the remaining query nodes. This suggests that the click-through bipartite is highly-connected, and the cluster structure cannot be obtained by simply removing a few "hubs". Figure 6(c) shows the effect of removing the top degree URL nodes. Removing the top degree URL nodes can break the largest connected component faster than removing the top degree query nodes. However, removing the URL nodes loses the correlation between queries since the URLs are considered as the features of queries.

We apply the clustering algorithm on the pruned click-through bipartite with $D_{max} = 1$ and obtain 218,673 clusters with size ≥ 2. These clusters cover 707,797 queries while the remaining queries form singleton clusters.

6.2 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 7 shows the distribution of session lengths. We can see that it is prevalent that users submit more than one query for a search intent. That means in many cases, the context information is available for query suggestion.
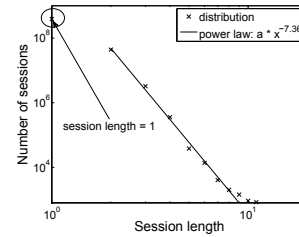

Figure 7: The distribution of session lengths.

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

Table 4: The number of nodes on the concept sequence suffix tree at different levels.

We then construct the concept sequence suffix tree as described in Section 5.2. Each frequent concept sequence has to occur more than 5 times in the session data. Table 4 shows the number of nodes at each level of the tree. Note we prune the nodes containing more than 4 concepts (349 nodes pruned in total), since we find those long patterns are not meaningful and are likely to be derived from query sequences issued by robots.

6.3 Evaluation of Query Suggestions

We compare the coverage and quality of the query suggestions generated by our approach, called the *context-aware concept-based approach* or *CACB* for short, with the following two baselines.

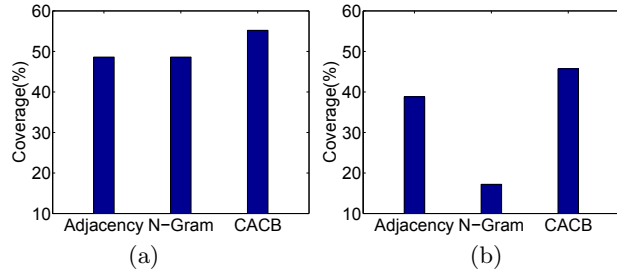**Adjacency**. Given a sequence of queries $q_1 \ldots q_i$, this

Figure 8: The coverage of the three methods on (a) Test-0 and (b) Test-1.

Figure 9: The quality of the three methods on (a) Test-0 and (b) Test-1.

method ranks all queries by their frequencies immediately following q_i in the training sessions and outputs top queries as suggestions.

N-Gram. Given a sequence of queries $qs = q_1 \ldots q_i$, this method ranks all queries by their frequencies of immediately following qs in training sessions and outputs top queries as suggestions.

We extract 2,000 test cases from query sessions other than those serve as training data. To better illustrate the effect of contexts 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 multi-query 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 8(a) and (b) show the coverage of the three methods on Test-0 and Test-1, respectively. The CACB method has a higher coverage than the other two methods on both test sets, and the N-Gram method has the lowest coverage. Given a test case $qs = q_1 \ldots q_i$, the N-Gram method is able to provide suggestions only if there exists a session $qs_1 = q_1 \ldots q_i q_{i+1} \ldots q_l$ in the training data. The Adjacency method is more relaxed; it provides suggestions if there exists a session $qs_2 = \ldots q_i q_{i+1} \ldots q_l$ in the training data. Clearly, qs_1 is a special case of qs_2. The CACB method is the most relaxed. If no session such as qs_2 exists in the training data, then the Adjacency method cannot provide suggestions. However, as long as there exists any sequence $qs_2' = \ldots q_i' q_{i+1} \ldots q_l$ in the training data such that q_i and q_i' belong to the same concept, the CACB method can still provide suggestions.

Another trend in Figures 8(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 is (the more queries a user submits), the more likely a session ends. Therefore, the training data available for test cases with context are not as sufficient as those for test cases without context. In particular, the coverage of the N-Gram method drops drastically on Test-1, while the other two methods are relatively robust because the N-Gram method depends more on training examples.

We then evaluate the quality of suggestions generated by our approach and the two baselines. For each test case, we mix the suggested queries ranked up to top 5 by individual methods into a single set. We then ask human judges to label for each suggested query whether it is meaningful or not. To reduce the bias of judges, we asked 10 judges with or without computer science background. Each suggested query is labeled by at least three judges.
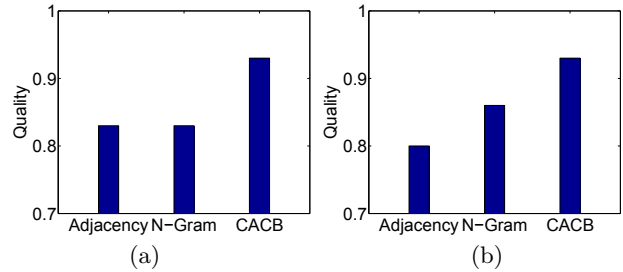
If one suggested query provided by a method is judged as meaningful, that method gets one point; otherwise, it gets zero point. Moreover, if two suggested queries provided by a method are both labeled as meaningful, but they are near-duplicate to each other, then the method gets only one point. The overall score of a method for a particular query is the total points it gets divided by the number of suggested queries it generates. If a method does not generate any suggested query 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. Figures 9(a) and (b) show the average scores of the three methods over the two test sets, respectively.

Figure 9(a) shows that, in case of no context information, the suggestions generated by the Adjacency method and the N-Gram method have the same quality, since the N-Gram method reduces to the Adjacency method in this case. The CACB method shows clear improvement in suggestion quality. This is because the CACB method considers the suggestions at the concept level and recommends queries belonging to related but not exactly the same concept with that of the current query (see the first two examples in Table 5).

Figure 9(b) shows that, in cases when context queries are available, the CACB method and the N-Gram method are better than the Adjacency method. This is because the first two methods are context-aware and understand users' search intents better. Moreover, the CACB method provides even better suggestions than the N-Gram method (see the last two examples in Table 5). This is because the CACB method considers users' search intents at the concept level instead of the detailed query level.

6.4 Robustness and Scalability of Algorithms

We test the robustness and the scalability of our algorithms. We first compare the suggestions generated under various values of parameter D_{max}. To be specific, given a test case, let S_1 and S_2 be the suggestions generated under two parameter values. We define the similarity between S_1 and S_2 by $\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$. Figure 10(a) shows the average similarity between the suggestions generated under default value $D_{max} = 1$ and those under various values. The clustering results do not change much under different parameter settings. Figures 10(b) and (c) show the scalability of the algorithms of clustering queries (Algorithm 1) and building the concept sequence suffix tree (Algorithm 2). We run the algorithms on 10%, 20%, ..., 100% of the entire data to illustrate the trend of scalability. Both algorithms are almost linear to the input size.

Test Case	Methods		
	Adjacency	N-Gram	CACB
www.at&t.com	at&t www.att.com cingular www.cingular.com	at&t www.att.com cingular www.cingular.com	att wireless at&t online billing cingular verizon
msn news	cnn news fox news cnn msn	cnn news fox news cnn msn	cnn news fox news abc news cbs news bbc news
www.chevrolet.com ⇒www.gmc.com	gmc acadia www.chevy.com www.chevrolet.com gmc envoy gmc cars	<null>	ford toyota nissan pontiac
circuit city ⇒best buy	circuit city walmart target best buy stores sears	walmart target sears office depot	walmart target sears radio shack staples

Table 5: Examples of query suggestions provided by the three methods.
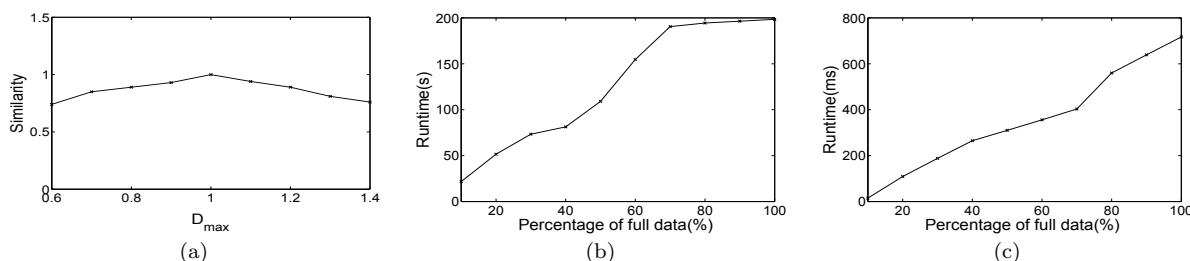

Figure 10: (a) The robustness of the clustering algorithm to parameter D_{max}, (b) the scalability of the algorithm for clustering the click-through bipartite (Algorithm 1), and (c) the scalability of the algorithm for building the concept sequence suffix tree (Algorithm 2).

7. CONCLUSION

In this paper, we proposed a novel approach to query suggestion using click-through and session data. Unlike previous methods, our approach considers not only the current query but also the recent queries in the same session to provide more meaningful suggestions. Moreover, we group similar queries into concepts and provide suggestions based on the concepts. The experimental results on a large-scale data containing billions of queries and URLs clearly show our approach outperforms two baselines in both coverage and quality.

8. REFERENCES

[1] Baeza-Yates, R.A., et al. Query recommendation using query logs in search engines. In *EDBT'04*.
[2] Baeza-Yates, R.A., et al. Extracting semantic relations from query logs. In *KDD'07*.
[3] Beeferman, D., et al. Agglomerative clustering of a search engine query log. In *KDD'00*.
[4] Chirita, P.A., et al. Personalized query expansion for the web. In *SIGIR'07*.
[5] Cui, H., et al. Probabilistic query expansion using query logs. In *WWW'02*.
[6] Dean, J., et al. MapReduce: simplified data processing on large clusters. In *OSDI'04*.
[7] Ester, M., et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD'96*.
[8] Fonseca, B.M., et al. Concept-based interactive query expansion. In *CIKM'05*.
[9] Hinneburg A., et al. Optimal grid-clustering: towards breaking the curse of dimensionality in high-dimensional clustering. In *VLDB'99*.
[10] Huang, C., et al. Relevant term suggestion in interactive web search based on contextual information in query session logs. *Journal of the American Society for Information Science and Technology*, 54(7):638–649, 2003.
[11] Jones, R. R., et al. Generating query substitutions. In *WWW'06*.
[12] Kraft, R., et al. Mining anchor text for query refinement. In *WWW'04*.
[13] Liu, S., et al. An effective approach to document retrieval via utilizing wordnet and recognizing phrases. In *SIGIR'04*.
[14] Magennis, M., et al. The potential and actual effectiveness of interactive query expansion. In *SIGIR'97*.
[15] Pei, J., et al. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01*.
[16] Srikant, R., et al. Mining sequential patterns: Generalizations and performance improvements. In *EDBT'96*.
[17] Sahami, M., et al. A web-based kernel function for measuring the similarity of short text snippets. In *WWW'06*.
[18] Terra, E., et al. Scoring missing terms in information retrieval tasks. In *CIKM'04*.
[19] Wen, J., et al. Clustering user queries of a search engine. In *WWW'01*.
[20] White, R.W., et al. Studying the use of popular destinations to enhance web search interaction. In *SIGIR'07*.
[21] Zhang, T., et al. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD'96*.